2009 IEEE International Symposium on Parallel and Distributed Processing with Applications

# An Empirically Optimized Radix Sort for GPU

**Bonan Huang, Jinlan Gao and Xiaoming Li**
Electrical and Computer Engineering Department
University of Delaware

## Abstract

Graphics Processing Units (GPUs) that support general purpose program are promising platforms for high performance computing. However, the fundamental architectural difference between GPU and CPU, the complexity of GPU platform and the diversity of GPU specifications have made the generation of highly efficient code for GPU increasingly difficult. Manual code generation is time consuming and the result tends to be difficult to debug and maintain. On the other hand, the code generated by today's GPU compiler often has much lower performance than the best hand-tuned codes. A promising code generation strategy, implemented by systems like ATLAS [14], FFTW [4], SPIRAL [11] and X-Sort [8], uses empirical search to find the parameter values of the implementation, such as the tile size and instruction schedules, that deliver near-optimal performance for a particular machine. However, this approach has only proved successful when applied to CPU where the performance of CPU programs has been relatively better understood. Clearly, empirical search must be extended to general purpose programs on GPU.

In this paper, we propose an empirical optimization technique for one of the most important sorting routines on GPU, the radix sort, that generates highly efficient code for a number of representative NVIDIA GPUs with a wide variety of architectural specifications. Our study has been focused on the algorithmic parameters of radix sort that can be adapted to different environments and the GPU architectural factors that affect the performance of radix sort. We present a powerful empirical optimization approach that is shown to be able to find highly efficient code for different NVIDIA GPUs. Our results show that such an empirical optimization approach is quite effective at taking into account the complex interactions between architectural characteristics and that the resulting code performs significantly better than two radix sort implementations that have been shown outperforming other GPU sort routines with the maximal speedup of 33.4%.

## 1 Introduction

Graphic Processing Units (GPUs) are designed for accelerating graphic operations and have strong computing power. Actually, the latest GPUs such as NVIDIA GTX 280 can achieve the peak performance of almost 1 TFLOPS [9], which rivals the performance of the supercomputers of just a few years back. It makes sense to leverage the computing power of GPUs on general purpose computing. However, programming GPUs is not easy. Because traditional GPU architectures are fully geared towards graphic operations, to exploit the GPU computing power on general purpose computing requires mapping a general computing problem to a graphic problem and implement the mapped graphic problem with graphic APIs such as OpenGL. The indirect mapping and implementation seriously complicate the program development for GPU and block the wide adoption of GPU in general purpose computing.

The introduction of CUDA changes the landscape of GPU programming. CUDA introduces a small set of C programming language extensions that enable programmers to naturally express the parallelization and the data layouts of an algorithm for NVIDIA GPUs. No translation from a computing problem to a graphic problem is necessary in CUDA. Furthermore, the CUDA extensions are relatively easy to understand and have grammars and semantics that may appear familiar to a normal programmer. As a reflection of the wide-spread usage of CUDA in high-performance computing community is that the new standard OpenCL [6] shares many programming structures and architectural abstractions with CUDA. CUDA has solved the problem of "make it work" for general purpose computing on GPUs.

978-0-7695-3747-4/09 $25.00 © 2009 IEEE
DOI 10.1109/ISPA.2009.89

IEEE computer society

However, the problem of *"make it work well"* for GPU computing just start gaining research attentions, and the introduction of CUDA actually complicates the optimization of programs on GPU because the performance modeling of CUDA is less well understood than that of traditional OpenGL programming. Furthermore, GPU hardware evolves fast. In the previous 18 months, there have been 3 major generations of NVIDIA GPU released to public. Architectural features of every new generation of GPU change significantly from those of the previous generation. Programs optimized for one generation can not be expected to perform equally well on the next generation. Manually optimizing a program for a specific GPU architecture could already be time-consuming and tedious. Now the optimization problem is compounded to catch up with every new generation of GPU. We need a systematic method to tune programs for different GPUs.

Table 1 shows the values of several architectural features of 4 different NVIDIA GPUs. The architectural features include the number of cores, the core frequency, the memory size and the memory frequency. Not surprisingly, tuning a program for NVIDIA GPUs needs to consider all those architectural features and the optimal values of tuning parameters need to be adjusted for the different features.

In this paper, we propose an empirical optimization technique for radix sorting algorithm on GPUs. We make two contributions. First we identify the algorithmic parameters of the radix sorting that need to be tuned for different GPU architectural features. Secondly, we empirically optimize the radix sorting to achieve over 30% speedup compared with the best radix sorting implementation currently available for NVIDIA GPUs.

The remaining of this paper is organized as follows. Section 2 will describe the background of GPU programming and optimization and discuss related work. Section 3 will present how different architectural factors affect the determination of the best values of parameters of the radix sorting. A novel empirical tuning technique for the radix sorting is presented in Section 4. Section 5 evaluates our empirically optimized radix sorting with a number of good radix sorting implementations on 4 different NVIDIA GPUs. Finally Section 6 concludes the paper.

## 2 Background and Motivation

In this section, we first overview the NVIDIA CUDA programming framework and the fundamental challenges of optimizing a program for CUDA, and then we discuss the radix sorting algorithm and several implementations of radix sorting on CUDA.

CUDA (Compute Unified Device Architecture) is NVIDIA's programming model that uses GPUs for general purpose computing (GPGPU). It allows the programmer to write programs in C with a few extensions that are designed to allow the programming of the CUDA architecture. These extensions enable programmers to directly access multiple levels of memory hierarchy that are quite different from the common CPU memory/cache model. Compared with general purpose CPU's with cached memory hierarchy, CUDA makes the performance of a program much more sensitive to specific memory layouts used by the program and leaves the full responsibility of taking full advantage of all levels of the CUDA's explicit memory hierarchy to CUDA programmers.

The introduction of the NVIDIA CUDA architecture and the accompanying CUDA driver and C language extension [9] make the computational power of GPUs easier to utilize, in particular, taking GPU programming to a higher level. One that normal programmers feel more familiar with. The CUDA driver and C language extension simplify the usage of the GPU as a co-processing device. However, even though the problem of writing a program that can *work* on a GPU seems to have been solved, the question of how to tune a program to make it *work well* on a GPU is only rudimentary understood and insufficiently investigated. Most notably, the program optimization for GPU faces two major challenges: the radically different organization of GPU memory hierarchy and the re-evaluation of traditional instruction level optimizations in the new context of GPUs.

Optimizing programs for GPU memory hierarchy is extremely challenging mainly because of two factors. First, the GPU memory hierarchy is organized in a radically different way from the organization of the CPU memory hierarchy. The two mostly used layers of the GPU memory hierarchy are the "global memory" and the "shared memory". The global memory is not cached, which invalidates the basic assumptions of classical program optimiza-

tions for the CPU memory hierarchy. When programming the CUDA GPU, the programmer has to manually manage data, copying from the global memory to the shared memory. In this way, the programmer can manipulate the access patterns to increase the applications performance. It's the programmer's responsibility to maximize the throughput by optimizing the memory layout, as well as, removing bank conflicts in shared memory. Second, the GPU runs programs in a SIMD-like manner where each instruction stream is a thread, so that the optimization for GPU memory hierarchy must be tuned for the collection of memory access streams from multiple threads. On the other hand, program optimizations for the CPU memory hierarchy usually assume that a single process occupy all memory levels. The consideration of multiple-thread memory accesses makes the optimization for the GPU memory hierarchy much more complicated than that for the CPU memory hierarchy.

The two most important memory levels on the GPU are the global- and the shared memory. They have completely different organization and memory models and are exposed to different numbers of threads. Particularly for larger data structures, a layout that is beneficial for global memory accesses will ruin performance for shared memory and vice versa. As a result, the *re-thinking* of what memory optimizations are needed for the GPU memory hierarchy is required. Furthermore, instruction level optimizations for the CPU usually assumes a program can occupy all CPU resources such as registers. However, the main focus of instruction level optimization for CUDA programs is to conserve hardware resources to allow for a higher occupancy of the available hardware for all threads. Therefore, traditional instruction level optimizations such as loop unrolling must be re-evaluated in the context of GPU. Loop optimizations are of special interest since most of the algorithms that qualify to be implemented in CUDA are loop based.

The CUDA memory hierarchy is composed of a very large on-board global memory which is used as main storage for the computational data and for synchronization with the host memory. Unfortunately, the global memory is very slow and not cached. To fully utilize the available memory bandwidth between the global memory and the GPU cores, data has to be aligned and accessed consecutively with 128 bit reads. To alleviate the high latencies of the global memory, a shared memory, which could be as fast as a register when accesses to the shared memory are appropriately ordered, is built-in into the CUDA architecture. The shared memory may sound like a kind of cache. However, the shared memory is explicitly managed by CUDA programs and special care has to be taken when the shared memory is used. Since the shared memory is accessed by 16 threads at the same time, each memory bank should be accessed only by one or all threads to guarantee minimum access time. If this is not the case , shared memory access conflicts will occur and reads to the same memory bank will be serialized. Furthermore, two other types of memory, texture- and constant memory, are available. These memories can only be written by the host machine, but not by the GPU.

In addition to the CUDA memory hierarchy, the performance of CUDA programs is also affected by the CUDA tool chain. The CUDA tool chain consists of special GPU drivers, a compiler which is based on the Open64 compiler[1], a debugger, a simulator, a profiler and libraries.

## 3 Factors

GPU architectural features such as memory hierarchy parameters interact with the radix sorting algorithm and determine the best values of the algorithmic parameters. In this section, we first describe the radix sort algorithm that we implement on GPU. Then, we present a detailed analysis of performance impact of those architectural features and how the parameters of the radix sorting should be tuned for those features.

### 3.1 Radix Sort Algorithm

Radix sort is one of the oldest sorting algorithm and is probably the most important non-comparison sorting algorithm. Radix sort work on the bits of a number. Assuming the keys to be sorted are $b-bit$ integers and the width of the digit, that is, the number of bits being worked on by each pass of Radix sort, is $r$, the keys can be viewed as an vector of $d = \lceil \frac{b}{r} \rceil$ digits of r bits each. The radix sort is consist of $d$ passes. The radix sort works from the least significant digit to the most significant digit. Each pass works on just one digit and sorts the input sequence with respect to that digit.

In the $i-th$ pass, the radix sort has to figure out the new positions of all the elements in the input sequence after the $i-th$ digit is sorted. We

call the new position of an element as the $rank$ of that element. The calculation of the $ranks$ of all elements has three steps. In the first step, the algorithm walks through the whole input sequence and counts the number of elements in each of the $2^r$ bucket. The $i-th$ digit of a key with value $x$ will be counted toward the $x-th$ bucket. We call the first step the "counting" step. In the second step, the buckets are accumulated in a way that the new value of the current bucket is the sum of all lower numbered bucket plus the original value of the current bucket. After the second step, the value of the $x-th$ bucket represents the starting position of all elements with the $i-th$ digit being $x$. The rank of an element with the $i-th$ digit being $x$ is the value of the $x-th$ bucket plus the number of elements in front of it that have the same value in the $i-th$ digit. The second step is usually called "prefix sum" or "scan" operation, which is a fundamental computation primitive and is used in many other places [3]. In the third step of the radix sort, all elements of the input sequence are scattered to their final position that is determined by their ranks. Because the scatter operation is extremely inefficient if done in-place, that is, without extra buffer, the radix sort is almost always implemented as a outplace algorithm, that is, the elements are scatter to another array that is equal-sizes as the current array. In the next pass, the roles of the two arrays are reversed. We call the third step the "scattering" step.

The complexity of radix sort is $O(N)$, N being the number of elements to sort. Thus, the main advantage of radix sort is its low instruction count. However, the main disadvantage of radix sort is its poor data locality in the third step, that is, the scattering step, because the elements are generally not written to consecutive memory locations.

### 3.2 Radix Sort Implementation in CUDA

One natural way to implement the radix sort algorithm in CUDA is to parallelize the three steps of the algorithm. All the three steps of radix sort can be parallelized, though in different ways.

The parallelization of the counting step of radix sort is conceptually simple. The input sequence will be blocked and each processor in a multiprocessor machine will work on a different block. The counting on each processor will produce perprocessor buckets. The per-processor buckets can be explicitly combined into a global array of buckets, or the per-processor buckets can be stored in a way that the global offsets of each local bucket can be calculated in one pass of the prefix-sum step. The second approach is more efficient because it enables the computing of prefix sum in the second step in parallel and enables each processor to read the offsets locally in the scattering step. In this paper, the second method is employed in the implementation of the counting step.

The kernel of the second step of radix sort is the prefix sum primitive. The prefix sum primitive is a fundamental data-parallel primitive and can be efficiently on multicore processors [13]. In this paper, we use a publicly available implementation of the prefix sum primitive that is a part of the CUDA Data-Parallel Primitives (CUDPP) library [2]. As a matter of fact, the prefix sum step uses only a small percentage of the total execution of radix sort, because compared with the input sequence, the number of bucket is very small. Even though the operations on each bucket in the prefix sum are more expensive, the overall time spend on the prefix sum is not a main factor in the performance of radix sort on CUDA. The main overhead of radix sort lies in the counting step and the scattering step, in particular, the scattering step is the most expensive step of radix sort.

The counting step and the prefix sum step provide per-block buckets that contain the offsets of all elements in that block. This greatly simplifies the work of the scattering step. The scattering step writes all elements in a block to their final locations in the memory. The final locations are determined wholly in local because all information needed is stored in the per-block buckets. However, this scheme usually makes inefficient use of memory bandwidth on CUDA, because multiple processors scatters elements to widely different locations in the memory. This situation is called un-coalesced memory write in CUDA, and can be up to 10 times slowdown compared with coalesced memory write. The performance tradeoffs of different implementations of the scattering step will be discussed in Section 3.3.1

### 3.3 Algorithmic Parameters and Architectural Factors

In this section we describe the parameters in the implementation of radix sort on CUDA and the architectural features of NVIDIA CUDA GPUs. Our focus is the analysis of the performance impact of dif-

ferent values of the algorithmic parameters, and the interaction between the architectural features and the radix sort algorithm. Usually the relation between the architectural factors and the optimal values of an algorithmic parameter is complex. As discussed in the next section, in this paper we study how to use empirical search to find out the optimal value of the algorithmic parameters, taking into consideration the architectural features. In other words, we try to train the radix sort algorithm on different NVIDIA CUDA GPUs by testing the performance of different shapes of the radix sort and find out the one with the best performance. By measuring execution time, the training identifies the best values for the algorithm parameters that determine the shape of the radix sort algorithm.

### 3.3.1 Algorithmic Parameters

There are two kinds of algorithmic parameters in our implementation of radix sort. The first kind of algorithmic parameters determines the division of workload among all the processors of a GPU. Section 2 describes that threads in CUDA are organized in two levels: thread grid and thread block. Therefore, we can specify the division of workload using a tuple of 3 numbers *(number_of_threadblock, number_of_thread_per_block, number_of_element_per_thread)*. Because the workloads of the three steps of radix sort are different, the counting step and the scattering step working on the input sequence, and the scattering step working on the buckets of a digit, our implementation of radix sort needs two such tuples, or 6 parameters, to describe the division of workload in a specific configuration of implementation.

The second kind of algorithmic parameter describes the number of bits—$r$—that are processed in each pass of the radix sort, that is, the width of a digit. The number of passes of the radix sort is determined by $r$. Previous studies have been using a fixed $r$ for all passes. For example, the radix sort implementation if CUDPP uses $r = 1$ [2], and Satish et.al. use $r = 4$ in their implementation. In our study, we give more flexibility to the selection of $r$ by enabling the usage of different $r$'s in different passes of our radix sort. Therefore, the parameter that describes digits of our radix sort is a vector $\vec{r} = \{r_0, r_1, ..., r_n | \sum_i r_i = b\}$.

In summary, the algorithmic parameters of the radix sort studied in this paper are the tuple *(number_of_threadblock, number_of_thread_per_block, number_of_element_per_thread)* for the counting step and the scattering step, the tuple *(number_of_threadblock, number_of_thread_per_block, number_of_element_per_thread)* for the prefix sum step, and the vector of digit sizes $\vec{r}$.

### 3.3.2 Architectural Factors

In this section we discuss three architectural factors of CUDA, number of cores, global memory bandwidth, and the frequencies of GPU core and global memory, and the effects of those factors on the performance radix sort. Furthermore, we present the relation between the architectural factors and the best values of the algorithmic parameters.

**Number of cores:** The NVIDIA GPUs that support CUDA have very different number of cores, e.g., as few as 8 cores in GeForce G 100 to as many as 240 core in GeForce GTX 280. The number of cores impacts the performance of radix sort mainly through the organization of threads in the CUDA kernel, that is, the number of threadblock ($NB$) and the number of thread per block ($NT$). For a given input sequence, when the number of available cores increases, each thread will handle fewer input elements. Because the execution of thread has a fixed overhead no matter how many elements are processed in the thread, the amortization of the fixed overhead is higher for fewer elements. On the other hand, because the maximum number of thread that can be supported by a core is a constant in CUDA, when the number of core increases, more threads can be spawned. More spawned threads mean that cores are better utilized because it is easier to find threads that are ready to run from a larger pool of thread. As a result, the radix sort may benefit from higher level of concurrency. Therefore, the performance will achieve the best tradeoff with regard to the number of cores when the amortized overhead equals the performance improvement of higher concurrency.

**Global memory bandwidth:** The selection of the digit sizes $\vec{r}$ determines the level of utilization of global memory bandwidth in radix sort. When larger $r_i$ is selected, the whole radix sort needs fewer passes to completely sort a input sequence. Because each additional pass incurs overhead, fewer passes might improve performance. However, on the other hand, larger $r_i$ means the input sequence will be written to more diverse locations in the global memory, because the total number of bucket is $2^{r_i}$ which increases exponentially

with $r_i$. That is, the input sequence will be written to more buckets, and hence it is less likely that elements will be written to consecutive locations. This sacrifices the efficiency of global memory bandwidth because coalesced writes, which are the writes of element to consecutive locations, can improve writing performance by a factor as high as 10 in CUDA. Balancing the two factors, the best point of tradeoff with respect to the utilization of global memory bandwidth is the point when the overhead of one additional pass equals the potential improvements of global memory write performance from writing to a smaller number of buckets.

**Frequencies of core and global memory**: The NVIDIA GPUs have a wide range of core frequencies and memory frequencies. For example, the lowest end NVIDIA GPU that support CUDA, GeForce 8300 GS, has core frequency of 450 MHz and memory frequency of 800 MHz, while the 2 numbers for the fastest NVIDIA GPU, GeForce GTX 285, are 648 MHz and 2484 MHz respectively. Furthermore, GPU cards on the market may not use the standard frequency of its GPU. The GPU chip and the memory might be overclocked by GPU card vendors. This further complicated the tuning of CUDA programs because the tuning must consider more possible GPU configurations. The core frequency and the memory frequency affect the selection of the best value of the number of element processed per thread ($NE$) and the digit size vector $\vec{r}$. When the radix sort runs on a GPU with higher core frequency, the $NE$ should be increased because the core frequency is how fast computation operations can be executed. Faster cores require more computation operations to achieve full utilization. More computation operations means that more elements are needed to saturate the cores. However, more elements in a thread means more elements need to be written to the global memory in a fixed period of time, hence increasing the global memory bandwidth pressure. Additionally, the role of the digit size vector $\vec{r}$ in the performance function is that when the relative speed of the global memory with regard to the core is higher, the radix sort can use big $r_i$ because the un-coalesced writes will lesser a problem for faster memory. In summary, the ideal performance tradeoff with regard to the core frequency and the memory frequency is the point where the time of executing computation operations on GPU cores equals the time of memory operations for each pass of radix sort. speed of

## 4 Empirical Tuning of Radix Sort

We have seen from the discussion of the previous section that there are multiple factors that affect the performance of radix sort on CUDA. The best values of the algorithmic parameters should be a function of those factors. However, it is difficult to calculate the best values of parameter directly from the given GPU architectural factors even that we know the exact values of those architectural factors for a specific GPU, because the relations between the algorithmic parameters and the architectural factors are interdependent with each other. The best value of one algorithmic parameter depends on the selection of the values of other parameters.

In this paper, we use empirical search to determine the best values of algorithmic parameters of radix sort on various GPUs. Exhaustive search is impractical here because the number of algorithmic parameter is large and each parameter can be assigned many different values, therefore the total number of combinations makes it almost impossible to test every possible shape of the radix sort.

On the other hand, exhaustive search is also unnecessary. Even though the global optimal value of a parameter is determined by multiple factors, the relation between the parameter and a single architectural factor is easier to establish as discussed in Section 3 and therefore we can identify the best value of a parameter with respect to one architectural factor using models that we build before. The global optimal value can be found by searching in the neighborhood of the range defined by those single-factor optimals.

Our empirical search leverages the models we build in Section 3 and is much more efficient than exhaustive search. The empirical search is conducted in 2 stages. In the first stage, we identify a range of value that most likely contain the best values of algorithmic parameters. The range is constrained by the best values of the parameters with respect to the number of cores, the global memory bandwidth and frequencies of core and global memory. We expand the range we get by $10\%$ to minimize the possibility that the best value lies outside of the native range, though that rarely happens. The ranges of all algorithmic parameters define the search space of our empirical search. Every point in this search space represents a different shape of the radix sort implementation on GPU, and the search space is believed to contain the best radix

sort implementation. In the second stage, our empirical search engine steps through every point in the search space that is defined from the first stage. For every point, the search engine tests the performance of the radix sort implementation whose shape is determined by the values of algorithm parameters corresponding to that point. The point that delivers the best performance will be selected and its corresponding algorithm parameter values will be used to generate our final radix sort implementation. For every GPU platform, we will conduct the 2-stage empirical search. The output on each platform is the best radix sort implementation for that specific GPU.

# 5 Evaluation

In this section we present the evaluation of our empirically optimized radix sort on CUDA GPUs. In Section 5.1 we describe the environmental setup that we use for the evaluation, and in Section 5.2 we present the performance results.

## 5.1 Environmental Setup

We evaluated our empirically optimized radix sort on four NVIDIA GPUs: GTX 280, 8800 GTX, 9600M GT and 9400M. The selection of the four GPUs is representative of the spectrum of different GPU specifications: from the high-end powerful GTX 280 to the low-end power efficient 9400M. Table 1 lists for each platform the number of cores, the size of global memory, the global memory bandwidth, the frequency of core and the frequency of global memory.

| | GTX 280 | 8800 GTX | 9600M GT | 9400M |
|---|---|---|---|---|
| Num. of cores | 240 | 128 | 32 | 16 |
| Core frequency (MHz) | 602 | 575 | 600 | 450 |
| Mem. size (MB) | 1024 | 768 | 512 | 256 |
| Mem. frequency (MHz) | 2200 | 1800 | 1600 | 1066 |
| Mem. bandwidth (GB/s) | 142 | 86.4 | 25.6 | 21 |

Table 1: Test Platforms.

All experiments sort records with two fields, a 32 bit integer key and a 32 bit pointer. The reason for choosing such records to sort is that for real world applications of sorting such as the sorting in databases, sorting is usually performed on an array of tuples each containing a key and a pointer to the original data record [10]. We assume that this array has been created before our radix sort is invoked.

The training time of our empirical search engine varies depending on the GPUs, but it ranges from 5 minutes in the fastest GTX 280 to about 15 minutes in the slowest 9400M.

## 5.2 Performance Results

We compared out empirically optimized radix sort with two other best known implementations of radix sort on CUDA: the radix sort routine from the CUDPP library [2], and the radix sort from Satish et.al. [12]. The radix sort implementation from Satish has been shown to outperform a number of sorting libraries on GPU including GPU-Sort [5] and a sorting routine proposed in the book GPU Gem3 [7]. We test all the radix sort implementations with data sizes from 4K to 6M records.

Figure 1 shows the performance of our empirically optimized radix sort, the CUDPP radix sort and the Satish radix sort on the four GPU platforms.

The performance results show that our empirical optimization techniques for radix sort is very effective. Our empirically optimized radix sort implementation outperforms the best known radix sort implementation by 17.1%, 15.9%, 15.7% and 23.1% on average on four different NVIDIA GPUs, GTX 280, 8800 GTX, 9600M GT and 9400M. The maximal speedup on the four platforms are 33.4%, 28%, 27.9%, 32.6% respectively

# 6 Conclusion

GPUs that support general purpose programming are promising platforms for high-performance computing. However, the fundamental difference between the architecture of GPU and the architecture of CPU make the tradition program optimization technique not directly applicable on the new GPUs. Furthermore, the vast diversity of GPU specification requires that a program optimization technique developed for GPU must be able to be tuned for different GPU configurations.

In this paper, we propose an efficient empirical optimizing technique to tune the most important sorting routine on GPU, the radix sort, for a variety of GPU platforms. Our empirical optimizing technique identifies the algorithmic parameters of the radix sort and the GPU architectural factors that determine the best values for those parameters, and employs efficient searching method to find the best performing radix sort on four representative NVIDIA GPUs. Our empirical optimizing tech-

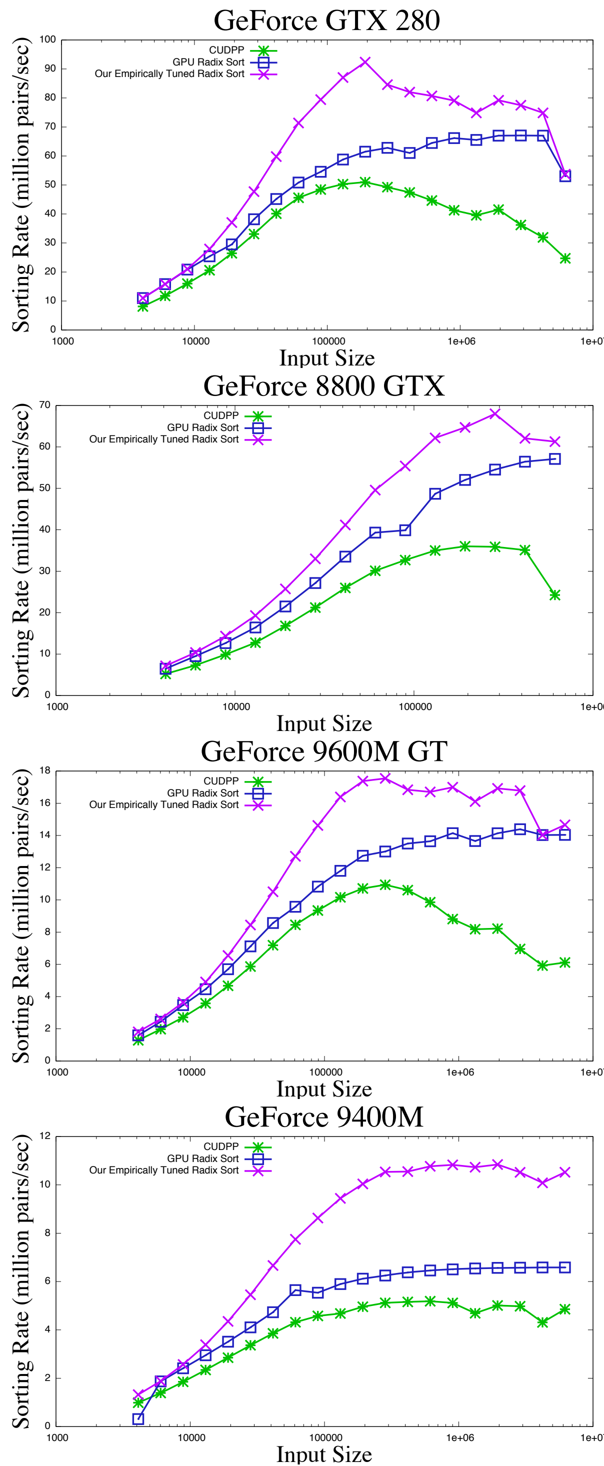


Figure 1: Performance of our empirically tuned radix sort compared with two other radix sort implementations from CUDPP and Satish et.al.

nique is very effective and efficient. On average, our empirically tuned radix sort outperforms the best radix sort implementation by 17.1%, 15.9%, 15.7% and 23.1% with the maximal speed up of 33.4%, 28%, 27.9%, 32.6% on four widely different GPUs.

## References


[1] Open64. http://www.open64.net.

[2] Cudpp: Cuda data parallel primitives library. http://www.gpgpu.org/developer/cudpp/, 2008.

[3] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[4] M. Frigo and S. Johnson. ”the fftw web page.” http://www.fftw.org, 2008.

[5] N. Govindaraju, J. Gray, R. Kumar, and D. 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.

[6] K. Group. Opencl 1.0 specification. http://www.khronos.org/registry/cl, 2008.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems 3*, 2007.

[8] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithm. In *International Symposium on Code Generation and Optimization (CGO)*, pages 99–110, 2005.

[9] C. NVIDIA. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 2007.

[10] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: a risc machine sort. *SIGMOD Rec.*, 23(2):233–242, 1994.

[11] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, 93(2):232– 275, 2005.

[12] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. *Proc. 23rd IEEE Intl Parallel & Distributed Processing Symposium*, 2009.

[13] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[14] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.