Parallel Butterfly Sorting Algorithm on GPU

Conference Paper · March 2013

DOI: 10.2316/P.2013.795-026

5 authors, including:

Bilal Jan
Politecnico di Torino
4 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Carlo Ragusa
Politecnico di Torino
87 PUBLICATIONS **541** CITATIONS
SEE PROFILE

Fiaz Gul Khan
COMSATS Institute of Information Technology
18 PUBLICATIONS **58** CITATIONS
SEE PROFILE

Omar Khan
National University of Computer and Emerging …
16 PUBLICATIONS **28** CITATIONS
SEE PROFILE

PARALLEL BUTTERFLY SORTING ALGORITHM ON GPU

Bilal Jan[1,3], Bartolomeo Montrucchio[1], Carlo Ragusa[2], Fiaz Gul Khan[1], Omar Khan[1]
[1] Dipartimento di Automatica e Informatica (DAUIN), Politecnico di Torino, Torino, Italy
[2] Dipartimento di Energia (DENERG), Politecnico di Torino, Torino, Italy
[3]bilal.jan@polito.it

***Abstract*—Efficient sorting is vital for overall performance of the underlying application. This paper presents Butterfly Network Sort (BNS) for sorting large data sets. A minimal version of the algorithm Min-Max Butterfly is also shown for searching minimum and maximum values in data. Both algorithms are implemented on GPUs using OpenCL exploiting data parallelism model. Results obtained on different GPU architectures show better performance of butterfly sorting in terms of sorting time and rate. The comparison of butterfly sorting with other algorithms:bitonic, odd-even and rank sort show significant speedup improvements against all on Nvidia Quadro-6000 GPU with relatively better sorting time and rate.**

***Index Terms*—Parallel Computing, GPU, Sorting Algorithms, OpenCL**

	NVIDIA			Intel
Architecture Details	GT320M	GTX 260	Quadro 6000	Core2Quad Q8400
Micro-processors	3	27	14	2
Cores per Processor	8	8	32	2
Total-Cores	24	216	448	4
Clock-Rate in MHz	1100	1242	1147	2660
FLOPS	158	874.8	1030.4	42.56
Memory-Bandwidth (GB/s)	9.95	91.36	144	-

TABLE I: Specifications of CPU and GPUs

1. INTRODUCTION

Parallelism, concurrency enabled at hardware, has played vital role in advancements of micro-processor architectures in the area of high performance computing. Such high performance, in one case, is accomplished by more than one processors working together on a single compute unit forming multicore CPUs[1]. However for high performance data-computation intensive applications, requiring massive parallelism, such core-processors are not sufficient. Recently GPUs (Graphic Processing Units), introduced primarily for higher resolution games, are now widely used for parallel computation where concurrent threads run simultaneously over multiple processing elements. High resolution games and scientific applications requiring high computations, are amongst the greatest beneficiaries. Earlier credits to NVIDIA (GeForce3) by adding programmable graphics pipeline to GPUs and AMD/ATI (Radeon9700) introducing floating point math capability, has led GPUs for general purpose computation as the name *"GPGPU"* i.e. General Purpose Graphic Processing Units coined by M.J Harris[2]. Recent developments in dedicated and heterogeneous parallel programming model and APIs like NVIDIA-CUDA and OpenCL specification by Khronos Group, enabled GPUs offload CPU burden working as co-processor for fast numerical crunching [3], [4]. The GPU itself is a multi-core processor where dozens of streaming processors with hundreds of cores support thousands of threads [5] running concurrently. CPU bound programs with high data in-dependency perform relatively better on GPUs. Generally large and complex applications have both data-dependent and independent modules which require both CPU and GPU working together as the name *"GPU Computing"* [6]. GPUs are adopted widely for computation intensive numerical simulations as they provide best price/performance ratio[7]. Table-1 highlights specification details of the GPUs and CPU focused in this paper for implementation of sorting algorithms. Performance comparison in GFLOPS shows that GPUs surpass CPUs by a wide range.

Several tools and APIs are available for programming GPU with a trade off for scalability, compatibility and user friendliness. Two of such tools and APIs are CUDA (Compute Unified Device Architecture) by NVIDIA[1] and OpenCL (Open Computing Language) by the Khronos[2]. Although CUDA is more scalable but designed specifically for NVIDIA platforms where as OpenCL is an open standard for heterogeneous computing targeting different platforms meeting its specification [8], [9]. OpenCL encompasses C99 based language extension for writing kernel code (device-program) and an API for manipulating (host-program) these kernels. Host program, running on CPU, controls access to kernel code run simultaneously by threads for data-task parallelism. The crux of parallel computing is finding sequential and parallel modules in the application to be run on CPU and GPU respectively. Threads are managed at hardware level

[1]NVIDIA Coporation www.nvidia.org
[2]OpenCL Specification. www.khronos.org/opencl

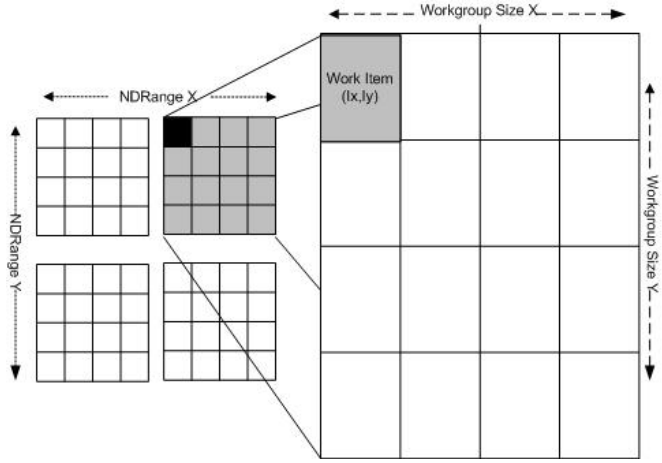


Fig. 1: 2D Addressing scheme for work-items

and developer only organizes the work domain into work items (threads) and to workgroups. Just like CUDA's thread, a workitem in OpenCL nomenclature is the basic execution unit. Workitems are identified uniquely by a global addressing scheme obtained through $get\_global\_id(0)$ built in function. Inside a workgroup workitems can be identified by local addressing with scope only to workgroup they belong to. Workitems belonging to different workgroups can have same local addressing but not global one. The organization of workitems inside NDRange for a 2-dimensional scheme is shown in Fig.1. The entire problem domain, ND-Range, can be in 1D, 2D or 3D. The dimension size varies from device to device with a limit of up to maximum of 3 dimensions 0, 1 and 2. For example in Quadro 6000 total thread size for the 3-dimension is 1024*1024*64. Threads are executed in groups called a thread block/warp which consists of 32 threads per block. Number of threads per block with their respective memory access distribution greatly affect throughput of memory accessing instructions. Objective of the GPU implementation for a sorting algorithm is minimizing time spent in sorting data. This time is vital for overall efficiency of the whole application, for example in discrete event simulators where FES (Frequent Even Set) sorting time is of great concern in overall performance of simulation. Selecting a sorting algorithm depends on both application and underlying hardware architecture. The paper is organised as followsn. A related work on parallel sorting is presented in section II. Section III briefly discusses design, algorithm and implementation of butterfly sorting on GPU architecture. Performance analysis and concluding remarks are reported in section IV and V respectively.

2. Related Work

Sorting is the most common operation performed in numerical computation and thus is one of the widely studied area in computer science. Sorting algorithms are very rich in literature. The focus here is on parallel sorting algorithms relating to GPUs only. A quick overview of parallel algorithms is presented in [10]. A quick-sort implementation on GPU using CUDA is considered in[11]. The quick-sort algorithm discussed in [11] works in two steps, creation of sub-sequences and assigning threads to the sub-sequences generated in first step. Their algorithm works in divide and conquer fashion on left-right sequences formation in accordance to the current value, greater or smaller than the value of pivot. The results in [11] show better performance of quick sort over bitonic and radix sort with complexity of $O(n\,log(n))$. A GPU implementation of merge sort and radix sort is presented in [12]. In this case, the radix sort divides the sequence of $N-items$ into N/P blocks. In next phase, in order to maximize coherence of scatters and minimize it to global memory, every sequence then is sorted by radix sort exploiting shared memory on the chip. The merge sort algorithm discussed in [12] adopts same divide-conquer approach by dividing sequence into p number of equal size blocks/tiles. An adaptive bitonic sorting algorithm is shown in [13]. Their implementation achieves optimal complexity of $O\left(\frac{nlogn}{p}\right)$ for sorting n numbers on p streaming processors. A GPU implementation of bitonic sort is discussed in [14] and CUDA based in-place bitonic sort is implemented in [15]. An overview of sorting on queues is covered in[16] focusing mainly on traffic simulations for studying the behavior of transport agents in large groups. A parallel implementation of odd-even sort suggested in [17] shows that parallelism can be introduced at each stage only internally i.e. at compare-exchange process but not stage by stage meaning that no two stages can be executed in parallel as output at any stage s_i is input for subsequent stage s_{i+1}. Same holds true for both min-max butterfly and full butterfly sorting where consecutive two stages can not be executed in parallel.

3. Butterfly Network Sorting

A. Overview

Network Routing and sorting on hypercube is of high concern for efficiency and throughput as mentioned in [18], [19]. Generally sorting N size data on a k-dimensional hypercube with $p-processors$ require running time of $\Theta((nlogn)/p)$ with $p=2^k$. We have considered 2×2 butterfly acting as compare-exchange circuit for both min-max butterfly and full butterfly sorting. The output values are placed at upper and lower wing of the butterfly. Searching minimum and maximum in this way increases over all throughput of several applications and dynamic systems involving operations on such smallest and largest values. The butterfly-network structure can be found in many diverse set of areas like FFT calculation in DSP, Benes network for switching fabrics, Hamiltonian cycle construction in network flows, dealing with min-max fairness problems and several other semi/complete sorting applications.

B. Working and Algorithms

1) Min-Max Butterfly: The min-max butterfly finds minimum and maximum in large volume of data in relatively small time. Min-max butterfly for searching minimum and maximum in N size data has total of $log_2 N$ stages. Complexity in terms of butterflies (comparators) is $(N/2)log_2 N$ butterflies where $N/2$ are number of butterflies in each stage. An example diagram of length 8 min-max butterfly is shown in Fig.2. Here $x(0), x(1)...x(7)$ can be any random values. At each stage $N/2$ butterflies are carried out in parallel where each butterfly fetches two values, Pos_{start} and Pos_{end}, from queue and then compares these values to be placed either at its upper or lower wing accordingly as shown in the algorithm below. After successful complete run of the algorithm in this case minimum and maximum values, 0 and 7, are output at $x(0)$ and $x(7)$ respectively. The min-max algorithm is carried out stage by stage with parallelism introduced in butterflies in execution in parallel inside a single stage. In addition to finding minimum and maximum in data, the min-max butterfly does complete sorting in special cases where input data is completely in descending order and vice versa.

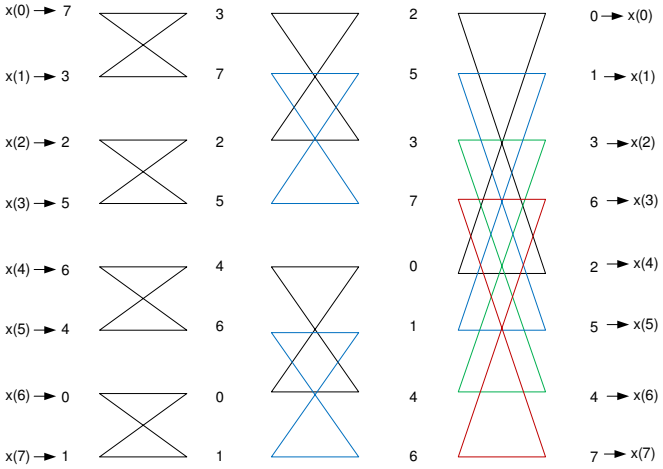


Fig. 2: 8x8 Min-Max Butterfly

The min-max butterfly algorithm works as follows;-

2) Full Butterfly Sort: The butterfly sort orders input data following any distribution type:uniform, random, exponential etc. Like min-max butterfly, in full butterfly sort the number of butterflies in any stage are constant i.e. $N/2$. For complexity in terms of total butterflies, we first find total number of stages which is given by the following formula.

$$T.Stages = log_2 N + \sum_{i=1}^{log_2 N - 1} i \quad (1)$$

$$T.Butterflies = N/2 \times T.Stages \quad (2)$$

In equation 1 $log_2 N$ are total number of out-kernals represented by the first $do-parallel$ block of the algorithm where

input : data$_{random}$,size=N
output: data$_{sorted}$

begin
  for $x_{out} \leftarrow 1$ *to* $\log_2(size)$ **do**
    PowerX = $radix^{x_{out}}$;
    do parallel T
      yIndex = $t/$ (PowerX/$radix$);
      kIndex = $t\%$ (PowerX/$radix$);
      Pos$_{start}$ = kIndex + yIndex $\times$ PowerX;
      Pos$_{end}$ = kIndex + yIndex $\times$ PowerX + PowerX/radix;
      if Pos$_{start}$ > Pos$_{end}$ **then**
        swap (Pos$_{start}$,Pos$_{end}$)
      end
    end
  end
end

Algorithm 1: Min/Max Butterfly Sorting Network

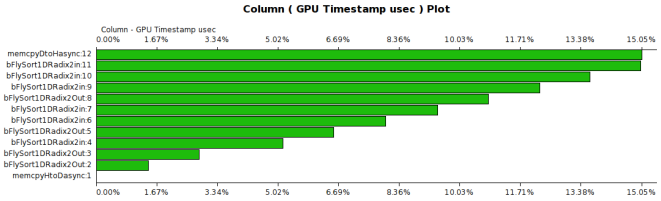


Fig. 4: Time breakdown for memory transfer and GPU execution of each stage/sub-stage for input data of size 16

as $\sum_{i=1}^{log_2 N-1} i$ are total number of in-kernels represented by second $do-parallel$ block in the algorithm. The visual profiler output for input data of size 16 clarifies this fact as shown in Fig. 4. Figure3 shows an example of 16x16 full butterfly network.

C. Visual Profiler Analysis

NVIDIA profiler presents some visual information of the code with the one shown in Fig. 4. Number of workgroups and their size plays significant role in over performance of the application. Throughput is also affected by global memory access that is typically accessed in 32, 64 or 128-byte chunks [20]. Fig.5 shows number of diverging threads in case of one stage of the full butterfly sort for input sizes 128 and 512 using float data type that is 4-byte long. For each butterfly two memory accesses are carried out; Pos_{Start} and Pos_{End}, for each memory load and store operation in 32, 64 or 128 byte chunks. After every access, the chunk size may be changed depending on device compute capability rules. Host program sets $globalSize$ and $localSize$ for workgroup size and for number of concurrent threads executing kernel respectively, which are given as follows:-

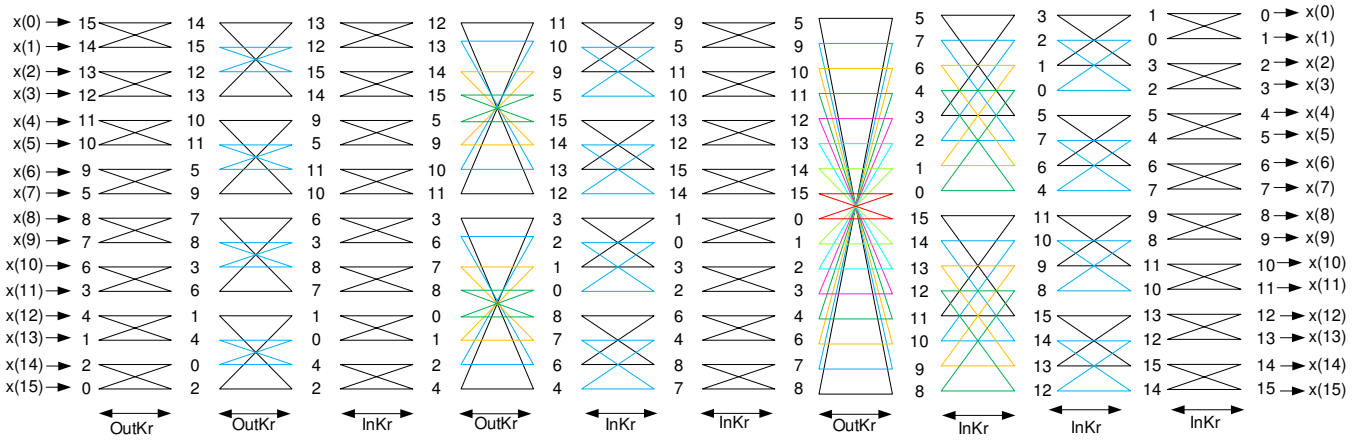


Fig. 3: 16x16 Full Butterfly Sorting

input : data$_{random}$,size=N
output: data$_{sorted}$

begin
  for $x_{out} \leftarrow 1$ *to* $\log_2(size)$ **do**
    PowerX = $radix^{x_{out}}$;
    do parallel T
      yIndex = $t/$ (PowerX/$radix$);
      kIndex = $t\%$ (PowerX/$radix$);
      Pos$_{start}$ = kIndex + yIndex $\times$ PowerX;
      Pos$_{end}$ = PowerX $-$ kIndex $-$ 1 + yIndex $\times$ PowerX;
      if Pos$_{start}$ > Pos$_{end}$ **then**
        swap (Pos$_{start}$,Pos$_{end}$)
      end
    end
    if $x > 1$ **then**
      for $x_{in} \leftarrow x$ *to* 1 **do**
        PowerX = $radix^{x_{in}}$;
        do parallel T
          yIndex = $t/$ (PowerX/$radix$);
          kIndex = $t\%$ (PowerX/$radix$);
          Pos$_{start}$ = kIndex + yIndex $\times$ PowerX;
          Pos$_{end}$ = kIndex + yIndex $\times$ PowerX + PowerX/radix;
          if Pos$_{start}$ > Pos$_{end}$ **then**
            swap (Pos$_{start}$,Pos$_{end}$)
          end
        end
      end
    end
  end
end

Algorithm 2: Full Butterfly Sorting

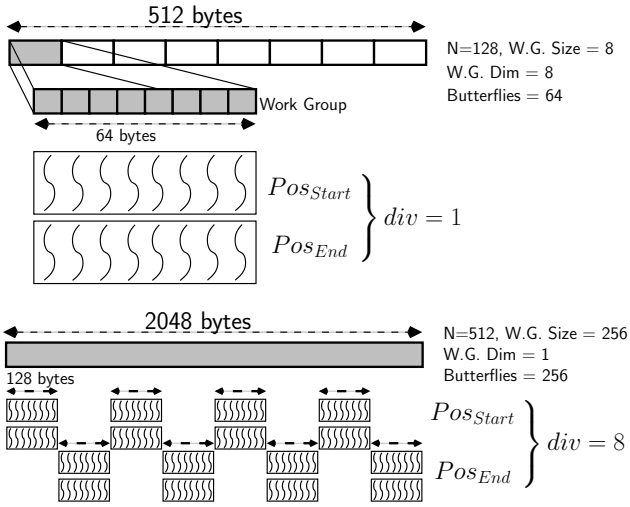


Fig. 5: Thread Divergence

$$globalSize = size/2$$
$$localSize = \begin{cases} globalSize/8 & \forall globalSize < 256 \\ 256 & otherwise \end{cases}$$

Throughput is affected mainly by *global-local* workgroup size ratio and distribution of memory accesses inside threads as shown in Fig. 6. Memory load (ρ_l) and store (ρ_s) throughput for all stages (*st*) from visual profiler are analysed as percentage average values for data of various sizes (n) by using following equations:

$$\rho_l = \sum_{st} \left[\frac{gld_{32} \times 32 + gld_{64} \times 64 + gld_{128} \times 128}{n \times sizeof(float)} \right]$$
$$\rho_{l_{avg}} = \frac{\rho_l}{st} \times 100$$

$$\rho_s = \sum_{st} \left[\frac{gst_{32} \times 32 + gst_{64} \times 64 + gst_{128} \times 128}{n \times sizeof(float)} \right]$$

$$\rho_{s_{avg}} = \frac{\rho_s}{st} \times 100$$

4. Performance Analysis

Performance of the sorting algorithms discussed here is evaluated both on CPU and GPUs considering their sequential and parallel implementations in terms of sorting time, sorting rate and speedup.

A. Experimental Setup

All simulations are carried out in OpenCL 1.2 and standard C compiler for different queue sizes in the power of 2. Input data of type float, is taken from a random number generator with size in the range of 2^{10} to 2^{25}. Variable declaration/initializations, random variate generators and other memory reads/writes to/from queues are mainly limited to CPU in host program. Actual sorting, butterfly computation, is carried out on GPU in kernel code. Hardware architectures used for simulations are Nvidia-Quadro6000, GeForce GTX260 and GeForce GT320M for parallel implementation and Intel Core2Quad CPU Q8400 for serial implementation. Visual profiler analysis are carried out on NVIDIA-GeForce GTX 260 of compute capability 1.3[3].

B. Results

1) Sorting Time: Sorting time of the algorithm is recorded as real time in seconds and is the time spent by the algorithm only for sorting data and excludes any other time spent in variable initialization, memory read/write and contention times etc. Sorting times for min-max butterfly and full-butterfly sorting on different GPU and CPU architectures are depicted in fig 7a and fig 8a respectively. Performance is improved by exploiting high parallelism inside any stage of the algorithm. Sorting time and rate values for full butterfly sorting are relatively better than bitonic sort, odd even sort and rank sort as shown in [17].

2) Sorting Rate: Sorting rate is the ratio of queue size to sorting time. Sorting rates for bitonic, odd/even and rank reported in[17] and are used only for comparisons with sorting rates of min-max butterfly and full butterfly. Our results for sorting rates, Fig 7b and Fig.8b, of min-max and full butterfly sort show better performance than all the three algorithms.

[3]Nvidia Compute Capability. www.nvidia.co.uk/object/cuda_gpus_uk.html

3) Speedup: Figure [9a,9b,9c] report improvement in speedups of our butterfly sort against others. It achieves 2x speedup over bitonic sort, a speedup of nearly 10^4x on Quadro6000 over rank and odd even sort for parallel implementation and speedup of nearly 10^3x against odd-even and rank sort for serial implementation. Speedup factor increases for large queue sizes on GPUs with larger number of cores.

5. Conclusion

We tested parallel and serial implementation of novel sorting algorithms: min-max butterfly and full butterfly sorting on different GPU and CPU architectures and evaluated better performance of our algorithms in comparison to bitonic, odd-even and rank-sort in terms of sorting time, sorting rate and speedup. In future the work will be transported to multiple GPUs with optimization techniques like memory coalescing etc and uses of these algorithms for $hold - operation$.

References

[1] M. Creeger, "Multicore cpus for the masses," *Queue*, vol. 3, pp. 64–ff, Sept. 2005. 1

[2] M. J. Harris, *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina at Chapel Hill, 2003. 1

[3] H. Hacker, C. Trinitis, J. Weidendorfer, and M. Brehm, "Considering GPGPU for HPC centers: Is it worth the effort?," in *Facing the Multicore-Challenge* (R. Keller, D. Kramer, and J.-P. Weiss, eds.), vol. 6310 of *Lecture Notes in Computer Science*, pp. 118–130, Springer, 2010. 1

[4] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010. 1

[5] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *HPCA*, pp. 382–393, IEEE Computer Society, 2011. 1

[6] Nvidia, Corporation, "What is gpu computing?." http://www.nvidia.com/object/what-is-gpu-computing.html, 2012. 1

[7] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, pp. 879 –899, may 2008. 1

[8] M. Garland, "Parallel computing with CUDA," in *IPDPS*, p. 1, IEEE, 2010. 1

[9] V. V. Kindratenko, J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. mei W. Hwu, "GPU clusters for high-performance computing," in *CLUSTER*, pp. 1–8, IEEE, 2009. 1

[10] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press, 1985. 2

[11] D. Cederman and P. Tsigas, "GPU-quicksort: A practical quicksort algorithm for graphics processors," *ACM Journal of Experimental Algorithmics*, vol. 14, 2009. 2

[12] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for many core GPUs," in *Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (23rd IPDPS'09)*, (Rome, Italy), pp. 1–10, IEEE Computer Society, May 2009. 2

[13] A. Gres and G. Zachmann, "GPU-bisort: Optimal parallel sorting on stream architectures," in *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)*, (Rhodes Island, Greece), IEEE Computer Society, Apr. 2006. 2

[14] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of Graphics Hardware 2003*, pp. 41–50, 2003. 2

[15] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, (Berlin, Heidelberg), pp. 403–410, Springer-Verlag, 2010. 2

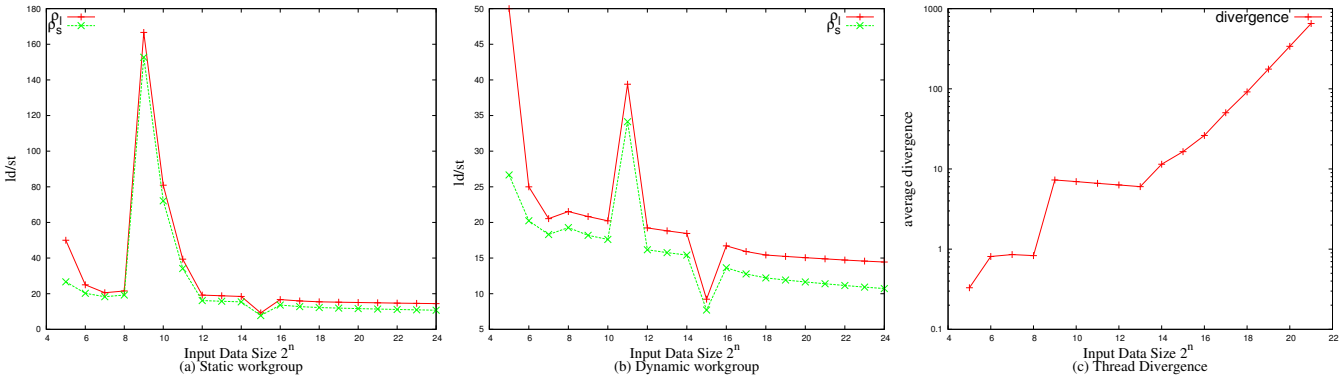
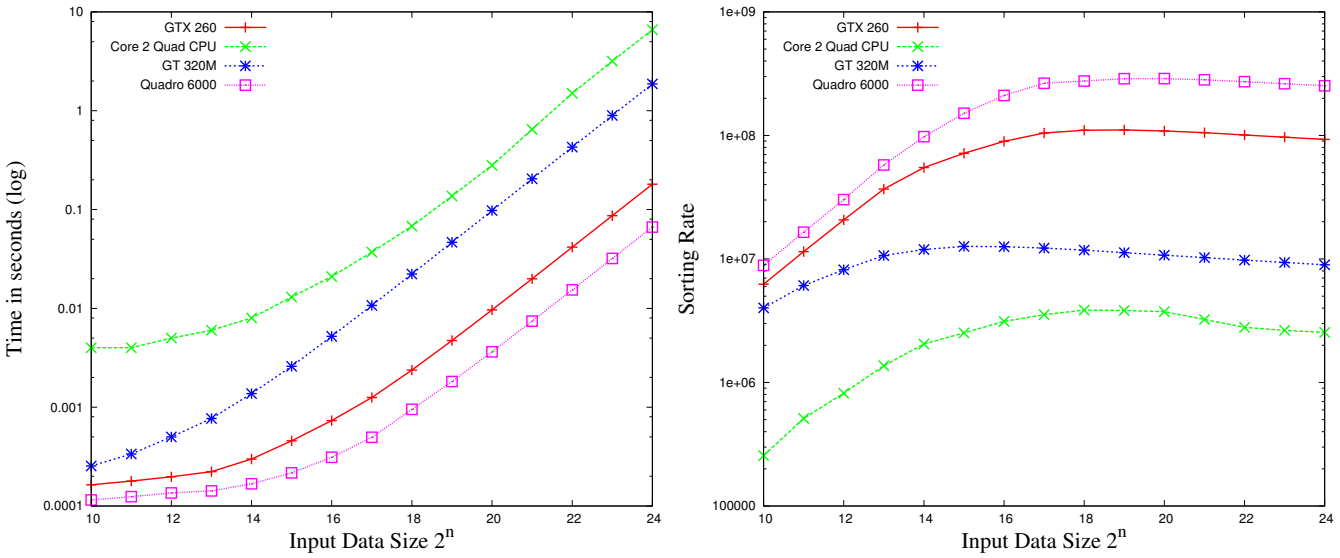


Fig. 6: Memory load (ρ_l)-store (ρ_s) counts and Thread Divergencecounts for a single stage of full butterfly sort. The position of peak's observable in (a) and (b) conform to the length of the work-group sizes and are un-avoidable because the work-group sizes cannot be exceeded beyond 512. The thread divergence grows exponentially as the size increases. Figure (c) showing the log scales reports a step again conforming to the length of the work-group size. This step is explained in Figure-5



(a) Sorting Time (b) Sorting Rate

Fig. 7: Min-max Butterfly

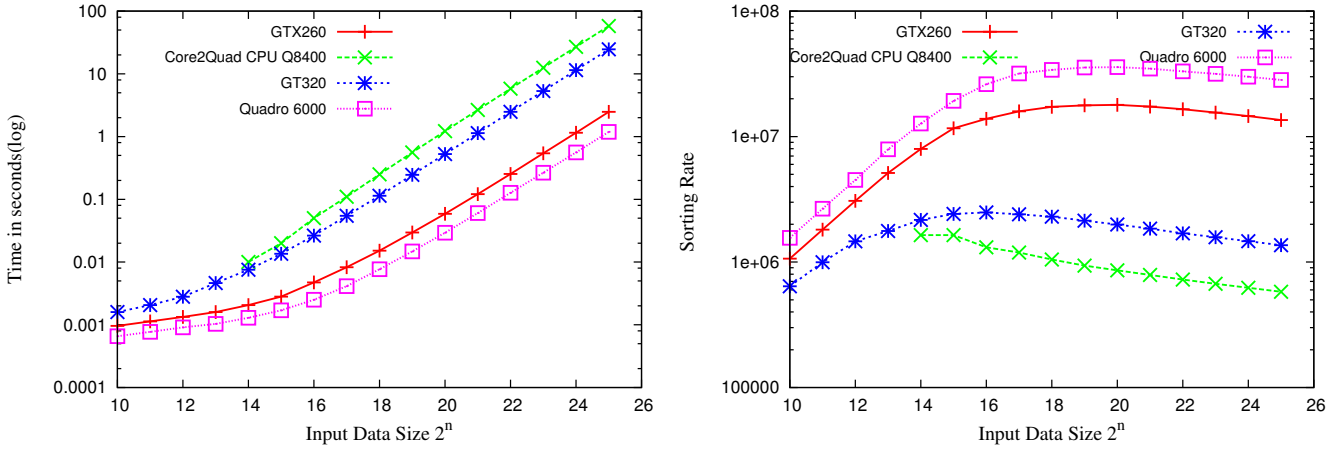
[16] D. Strippgen and K. Nagel, "Using common graphics hardware for multi-agent traffic simulation with CUDA," in *SimuTools* (O. Dalle, G. A. Wainer, L. F. Perrone, and G. Stea, eds.), p. 62, ICST, 2009. 2

[17] F. G. Khan, O. U. Khan, B. Montrucchio, and P. Giaccone, "Analysis of fast parallel sorting algorithms for gpu architectures'," in *Proceedings of the 2011 Frontiers of Information Technology*, pp. 173–178, IEEE Computer Society, 2011. 2, 4.21, 4.22

[18] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992. 3.1

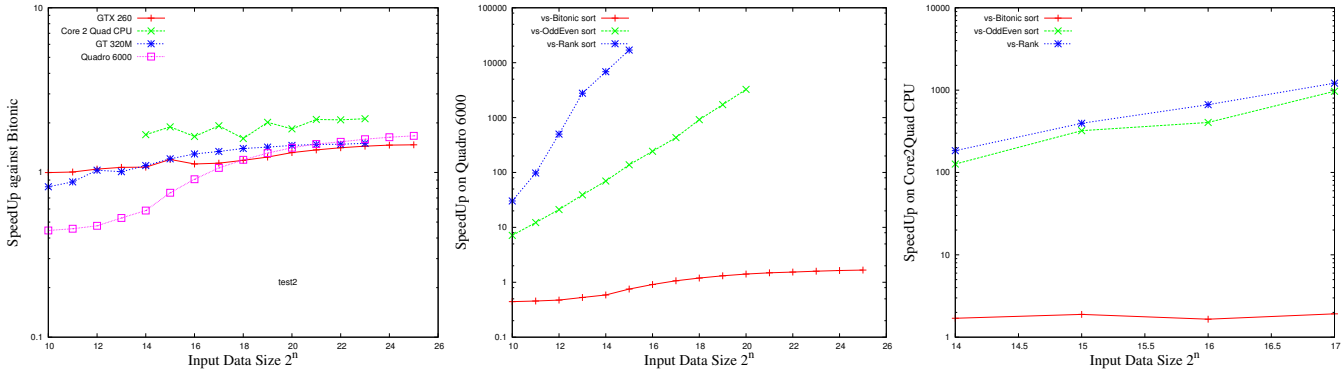
[19] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures (Series in Computer Science)*. Springer, 1 ed., 1 1999. 3.1

[20] *NVIDIA CUDA C Programming Guide Version 4.2*. NVIDIA, 4 2012. 3.3



(a) Sorting Time (b) Sorting Rate

Fig. 8: Full Butterfly Sorting Time and Rate



(a) Speedup Full Butterfly against Bitonic (b) Speedup Full Butterfly against Others on Quadro 6000 (c) SpeedUp Serial Full Butterfly against Others on CPU

Fig. 9: SpeedUp Improvement of Full Butterfly Sort