FAST PARALLEL SORTING ALGORITHMS ON GPUS

Bilal Jan[1,3], Bartolomeo Montrucchio[1], Carlo Ragusa[2], Fiaz Gul Khan[1] and Omar Khan[1]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, I-10129 taly
[2] Dipartimento di Ingegneria Elettrica, Politecnico di Torino, Torino, I-10129 Italy
[3]Bilal.Jan@polito.it

ABSTRACT

This paper presents a comparative analysis of the three widely used parallel sorting algorithms: Odd-Even sort, Rank sort and Bitonic sort in terms of sorting rate, sorting time and speed-up on CPU and different GPU architectures. Alongside we have implemented novel parallel algorithm: min-max butterfly network, for finding minimum and maximum in large data sets. All algorithms have been implemented exploiting data parallelism model, for achieving high performance, as available on multi-core GPUs using the OpenCL specification. Our results depicts minimum speed-up19x of bitonic sort against odd-even sorting technique for small queue sizes on CPU and maximum of 2300x speed-up for very large queue sizes on Nvidia Quadro 6000 GPU architecture. Our implementation of full-butterfly network sorting results in relatively better performance than all of the three sorting techniques: bitonic, odd-even and rank sort. For min-max butterfly network, our findings report high speed-up of Nvidia quadro 6000 GPU for high data set size reaching 2^{24} with much lower sorting time.

KEYWORDS

Parallel Computing, Parallel Sorting Algorithms, GPUs, Butterfly Network, OpenCL

1. INTRODUCTION

Parallelism on chip level is the hub for advancements in micro processor architectures for high performance computing. As a result of which multi-core CPUs [1] are commonly available in the market. These core-processors, in personal computers, were not sufficient for high data-computation intensive tasks. As a result of collective efforts by industry and academia, modular and specialized hardware in the form of sound cards or graphic accelerators are increasingly present in most personal computers. These cards provide much high performance as compared to legacy on-board units. Recently, graphics cards or graphics processing units (GPU), introduced primarily for high-end gaming requiring high resolution, are now intensively being used, as a co-processor to the CPU, for general purpose computing[2, 3]. The GPU itself is a multi-core processor having support for thousands of threads [4] running concurrently. GPUs are result of dozens of streaming processors with hundreds of core aligned in a particular way forming a single hardware unit. Thread management at such hardware level requires context-switching time close to null otherwise penalizing performance. Apart from high-end games, general purpose CPU-bound applications which have significant data in-dependency are well suited for such devices. Hence data parallel codes are efficiently performed since the hardware can be classified as SIMT (single-instruction, multiple threads). Performance evaluation in GFLOPS (Giga Floating Point Operations per Second) shows that GPUs outperforms their CPU counterparts. For example a high-end Core I7 processor (3.46 GHz) delivers up to a peak of 55.36 GFLOPs[1]. Table-1 reports some architecture details of GPUs versus Intel Core2 system that we have used for our implementation of sorting algorithms. The devices include both high-end graphics card like Quadro 6000 comprising of 14 stream processors with 32 cores each, and

[1] Intel Core I7 Specification, www.intel.com

DOI : 10.5121/ijdps.2012.3609

also low-end graphic cards that is GeForce GT 320M with 3 processors of 8 cores each. One such architecture of GTX 260 with 27 processors having 8 cores each, is depicted in Fig. 1. To low-end we have used GT 320 M as is more ideally suitable for laptops and hence the fewer cores provide good balance for battery power. The high powerful Quadro-6000 and GTX-260 is well suited for desktops with power requirement of 204W and 182W respectively.

Architecture Details	NVIDIA			Intel
	Quadro 6000	GTX 260	GT 320M	Core2 Quad Q8400
Total Cores	448	216	24	4
Micro Processors	14	27	3	1
Clock Rate (MHz)	1147	1242	1100	2660
FLOPs	1030.4	874.8	158	42.56
Mem. Bandwidth (GB/s)	144	91.36	9.95	-

Table 1Architecture details of GPUs and CPU

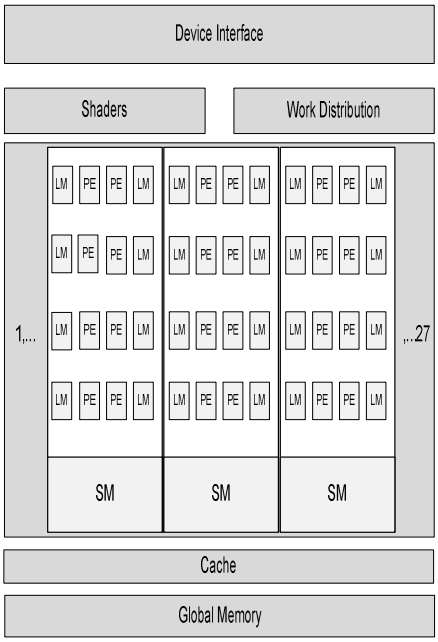


Figure 1GTX-260 Device Archite

To program a GPU, one needs vendor provided API's which is NVIDIAs CUDA[2], ATIs FireStream[3] or the OpenCL specification by Khronos group[4] . One difference between CUDA and OpenCL is that CUDA is specific for GPU devices whereas OpenCL is heterogeneous and targets all devices conforming its specification [5], [6]. This may include GPUs and/or CPUs but for to achieve high performance, it primarily focuses on the GPUs. OpenCL adopts C-style and is an extension of C99 with some extra keywords and a slightly modified syntax for threads driving kernels. OpenCL runs two pieces of codes. One is kernel, also called device program which is a specific piece of code running on device, is executed concurrently by several threads and this is where task parallelism takes place consisting of thousands of threads on the target device. The other, called a host program, runs entirely on CPU side that launches kernels i.e. SIMT based programs. Thread management is hardware based and programmer only organizes the work-domain into several work-items divided into one or more work-groups. The overall problem domain, called the ND-Range, can support up to three dimensions. A work-item or thread which is the basic execution unit in NDRange, is identified by a global and local addressing scheme in NDRange for each dimension of NDRange and work-groups. Global addressing obtained by $get\_global\_id(0)$ is unique for all threads, whereas any two threads of

[2] Nvidia Cuda GPGPU framework.www.nvidia.com
[3] AtiFirestream DAAMIT GPGPU Framework. www.amd.com
[4] OpenCL Specification. www.khronos.org/opencl

different groups can have same local address. This scheme is outlined in Fig. 2 for a 2 dimensional problem.

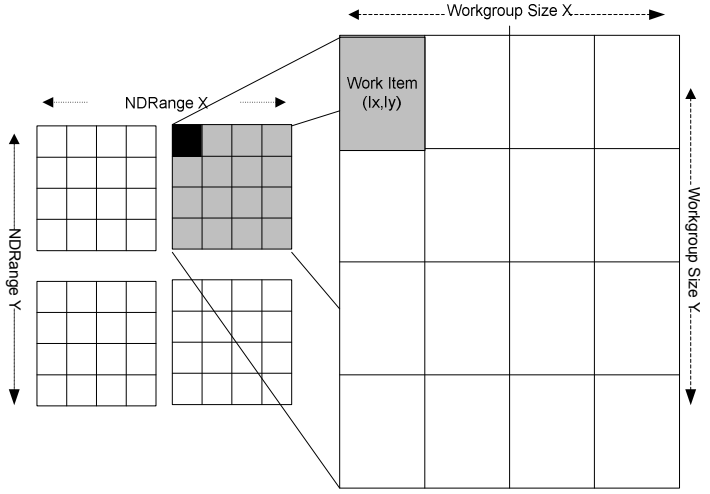


Figure 2 ND Range Adressing Scheme

A single dimensional address can be computed as: $global_{idd} = (group_{id} * group_{id})$

Provided that it fulfils the following expressions:

$$global_{idd} = (group_{id} * group_{id})$$
$$0 \le local_{id} \le global_{size}$$
$$NDRange = \max(group_{id}) * group_{size}$$
$$\left[\frac{NDRange}{\max(group_{id})} \% 2 = 0 \right]$$

Here, $group_{size}$ and $NDRange$ is set in host program by the programmer. The dimensional limits differs from device to device with a limit of up to maximum of 3 dimensions 0,1 and 2. For Quadro 6000 maximum size for the 3-dimensions are 1024*1024*64 and 512*512*64 for both GTX260 and GT320M. This corresponds to approximately 67 and 16 million threads for Quadro and GTX,respectively. All threads are executed in form of thread blocks containing 32 threads, referred to as warps. However, some devices support execution of half warps. Our focus in this paper is to report performance of sorting algorithms using graphics cards which is of significant importance to various computer science applications. The choice of sorting technique is vital in performance for some applications, for instance discrete event simulations, where sorting frequent events can directly affect the performance of simulation. The

algorithms discussed int the following are bitonic, odd-even, and rank sorting algorithms.

2. PARALLEL SORTING ALGORITHMS

Sorting on GPU require transferring data from main memory to on-board GPU global memory. Although on-device bandwidth is in the range of 144Gb/s, thus only those sorting techniques are efficient which require minimum amount of synchronization because the PCI bandwidth is to the range of 2.5Gb/s. i.e., synchronization and memory transfers between CPU and GPU will affect system performance adversely. Compared to serial sorting algorithms, parallel algorithms are designed requiring high data independence between various elements for achieving better performance. Those techniques which involve large data dependency are categorized as sequential sorting algorithms.

2.1. Odd-Even Sort

The odd-even sort is a parallel sorting algorithm and is based on bubble-sort technique. Adjacent pairs of items in an array are exchanged if they are found to be out of order. What makes the technique distinct from bubble-sort is the technique of working on disjointed pairs, i.e., by using alternating pairs of odd-even and even-odd elements of the array. The technique works in multiple passes on a queue Q of size N. In each pass, elements at odd-numbered positions perform a comparison check based on bubble-sort, after which elements at even-numbered positions do the same. The maximum number of iterations or passes for odd-even sort is $N/2$. Total running time for this technique is $O(\log 2N)$. The algorithm works as:-

Algorithm 1 Odd Even Sort

```
for k = 1 → N/2 do
    do parallel
    if i > i + 1 ∀ i%2 != 0 then
        swap i, i + 1
    end if
    end parallel
    do parallel
    if i > i + 1 ∀ i%2 == 0 then
        swap i, i + 1
    end if
    end parallel
end for
```

2.2. Rank Sort

There are two phases of the rank-sort algorithm. In the first phase, for each element in queue Q of size N, the total number of elements less than itself is maintained in another data structure of same size N. This is called the ranking phase and is depicted in Algorithm-2. Since each element n is compared against $n-1$ other elements, therefore there are a total of $n(n-1)$ total computational steps. But since the comparison requires sharing of data and not changing of data, the comparison can be made in $O(N)$ total steps for N processors. This also means that the technique is feasible for shared memory architectures. The second phase involves sorting of elements in queue Q based on its rank. The phase is shown in Algorithm-2. The second phase sorting can be performed in $O(\log 2n)$ steps. For optimization, the number of elements is divided based on number of processors using $m = n/p$.

Algorithm 2 Rank Sort

```
for k = 1 → N do
    do parallel
    Q[n] = 0
    for i = 1 → N do
        for j = 1 → N do
            if Q[j] < Q[i] then
                r[i] + +
            else
                r[j] + +
            end if
        end for
    end for
    end parallel
end for
do paralle
for k = 1 → N do
    U[r[k]] = a[k]
end for
end parallel
do paralle
for k = 1 → N do
    a[k] = u[k]
end for
end parallel
```

2.3. Bitonic Sort

Bitonic sort with the property that sequence of comparisons is data-independent makes it one of the fastest and suitable parallel sorting algorithms. To sort an arbitrary sequence bitonic sort have two steps. In the first step it makes the arbitrary sequence in to bitonic sequence. A bitonic sequence is a sequence which either monotonically increases or decreases, reaches a single maximum or minimum, and then after that maximum or minimum value it again monotonically increases or decreases. For example, the two sequences 3 5 8 9 7 4 2 1 and 5 8 9 7 4 2 1 3 are bitonic. The first one increases from 3 to 9, then decreases. The second one can be converted to the first one by cyclically shifting. In the second step the bitonic sequence is sorted in such a way that, lets we have a bitonic sequence N with length $n = 2^k$, which would require k steps to sort an entire length of n elements. In the first step $N(0)$ would be compared to $N(n/2)$, $N(1)$ with $N(n/2+1)$ up to $N(n/2-1)$ with $N(n-1)$ and elements are exchanged according either subsequence from $N(0)$ to $N(n/2)$ and from $N(n/2+1)$ up to $N(n-1)$. Then in the second step same procedure would be applied to each subsequence and each subsequence would yield another subsequence and after the k^{th} step it yields the 2^k sub-sequences of length 1 so all the elements in the bitonic sequence are being sorted. Bitonic sort consists of $O(n \log n^2)$ comparators as in every step $n/2$ compare/exchange operations are performed and total number of steps are $k = \log n$, so in parallel processing it would take n processor to sort it with $O(\log n^2)$ complexity. The total number of steps required in bitonic sort in both steps that are creating a bitonic sequence and the sorting $\frac{k(k+1)}{2}$. For example arbitrary sequence of 16 elements (2^4) would take 10 steps.

2.4. Min-max Butterfly Network

Butterfly network is a special form of hypercube. A k-dimensional butterfly has $k(k+1)2^k$ vertices and $k2^{k+1}$ edges [7]. In this case vertices represent input data whereas edges represent possible data movements. We are considering 2X2 butterfly-network, acting as a comparator, placing minimum and maximum number at their respective upper and lower leaves. The min-max butterfly of N numbers has $\log_2 N$ stages resulting into a total complexity of $N/2 \log_2 N$ butterflies in terms of comparators or cross points with $N/2$ as the number of 2x2 butterflies in each stage. The min-max butterfly network is of significance importance to several network applications and dynamic systems involving minimum and maximum values/quantities. The butterfly-network, in general, has its roots in many diverse areas: DSP-FFT calculation, Switching Fabric-Benes Networks, Network Flows-Hamiltonian, cycle construction, min-max fairness problems etc. This paper exploits the butterfly network in a parallel way for finding minimum and maximum values of the input data. The Fig. 3 shows a schematic of a min-max butterfly of 8 random numbers *x(0),x(1)...x(7)* in increasing order. The input to stage 1 is obtained from a random variate generator. At each stage a single butterfly compares two numbers and places it at upper and lower level accordingly. At last stage i.e. $\log_2 N^{th}$ stage minimum and maximum values are output at upper and lower leaf of first and last butterfly respectively. In this case *x(0)* and *x(7)* are the resulting min-max values. The stages of min-max butterfly structure could not be parallelized as output of any stage s_i is input to subsequent stage s_{i+1} and hence algorithm works stage-by-stage sequentially. For efficient resource utilization and high performance, we have introduced parallelism inside stage.

Referring to Fig. 3 it can be judged that at any stage $s_i < \log_2 N$ parallelism can be imposed by executing similar operations concurrently. For example at stage s_1 each 2x2 butterfly can be executed in parallel constrained by maximum number of threads running in parallel imposed by underlying hardware architecture. Degree of parallelism remains constant throughout all stages $s_1, s_2 ... \log_2 N$.

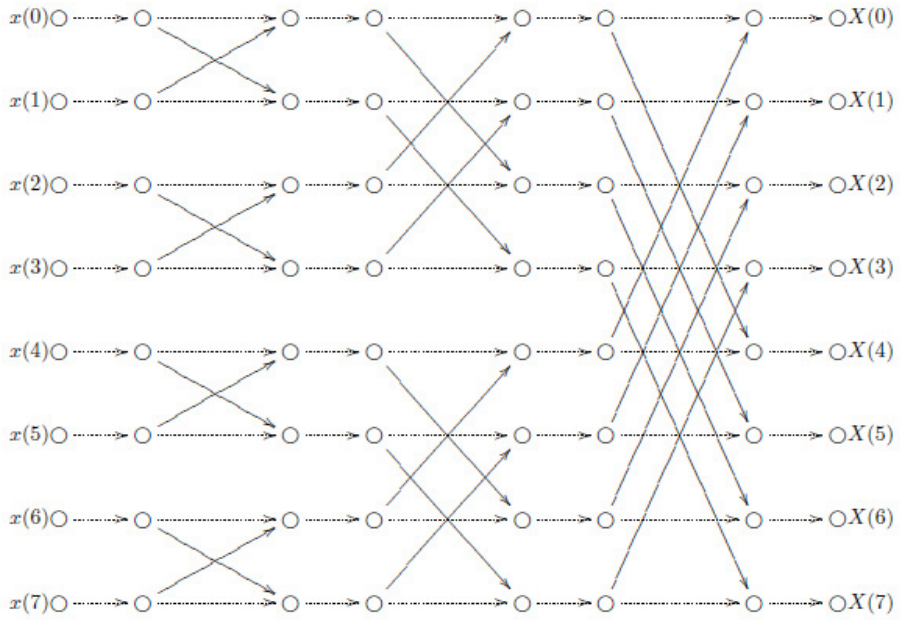


Figure 3 Min-Max 8x8 Butterfly

The min-max butterfly algorithm works as follows;-

Algorithm 3 min-max butterfly

```
t = 1
for x = log2N → 1 do
    do parallel t , t < N/r
    p = r^x
    pm1 = p/r
    y = t/pm1
    s = t * p
    e = s + pm1
    if (d[s] > d[e]) then
        swap ( d[s] , d[e] )
    end if
    t = t * 2
    end parallel
end for
```

3. RELATED WORK

Sorting is one of the widely studied algorithmic topics for the last three decades. Due to space limitation for mentioning different kinds of sorting algorithms, we discuss only relevant parallel sorting techniques on GPUs in this section. An overview of sorting algorithms in parallel is given in [8]. A quick-sort implementation on GPU using CUDA is considered in [9] which results quick-sort as an efficient alternative to both bitonic and radix sort over GPU's for larger data sequences. Moreover bitonic sort is suggested for smaller sequences. The quick-sort algorithm discussed in [9] uses a divide-and-conquer approach for sorting, forming left and right sequences depending on whether current value is greater or smaller than pivot value. For each recursive call, a new pivot value has to be selected. On the GPU, [9] have proposed two steps (1) sub-sequences creation, and (2) assigning each sub-sequence to thread for sorting. The overall complexity of above GPU-quick sort technique is $O(n \log n)$, with a worst case of $O(n^2)$.

A merge and radix-sort implementation for GPU's is provided in [10]. Here, radix sort first divides full sequence n into n/p thread blocks with p as total available threads. Each sequence then is locally sorted by radix sort on-chip shared memory reducing number of scatters to global memory and maximizing their coherence. Scattered I/O is efficient by placing single procedure call to write data to a single data stream coming from multiple buffers. But it has no support in all GPU devices and thus all writes are sequential [11]. In recent cards, including the NVIDIA G80 series and AMD R600 series this is however no longer a problem. Their technique achieves complexity of $O(\sqrt{t})$ with t threads handling $2b$ buckets. The merge sort [10] follows the same divide-and-conquer technique where complete sequence is divided into p same size tiles. Afterwards all tiles are sorted in parallel using odd-even sort with p thread blocks, and then merged together using merge-sort conventions on a tree of $\log p$ depth. This technique is well suitable for external sorting, where a processor has access only to a small memory address space. Moreover, degree of parallelism is reduced as higher levels are sorted and thus not fully utilizing parallel GPU architecture. An adaptive bitonic-scheme is proposed in [12]. Their technique sorts n values using p stream processors achieving optimum complexity of $O\left(\frac{n \log n}{p}\right)$. Bitonic sort has also been implemented in [13] using Imagine stream processor. An overview of sorting queues for traffic simulations is covered in [14]. Their approach is to study the behavior of relatively large groups of transport agents.

4. PERFORMANCE ANALYSIS

Several different C data structures and built-in routines are usually used for sorting algorithm implementation. In OpenCL framework this is not the case because only a few supported math functions, most of these are absent. Hence they have to be implemented explicitly by developers. Moreover, as memory cannot be allocated dynamically in kernels, all memory has to be allocated before.

4.1. Experimental Setup

This section is dedicated to examine performance of our sorting algorithms. The performance tests are carried out on varying queue sizes where each queue size is a value of power 2. The input data type is float for all algorithms. Random numbers are generated following uniform and/or exponential distributions to populate the input queue size. For uniform distribution, value ranges from 1 and 2^n. All necessary variable initializations for input/output, random variate generators, output from the queues are performed locally on the CPU, whereas actual sorting implementation is carried out entirety on GPU side. The GPU devices for running our

simulations are the NVIDIA Quadro 6000, NVIDIA GeForce GTX 260 and NVIDIA GeForce GT 320M. The GT320M, designed for notebooks, consumes less power and has less cores with 1GB global memory for the device. The GTX260, on the other hand, is a high-end graphics card with large number of cores, 216 in number and 895MB of global memory. The NVIDIA Quadro 6000, built on innovative NVIDIA fermi architecture, supports 14 micro-processors having 32 cores each, thus resulting into 448 cores in total, arranged as array of streaming multi-processors. For comparison with CPU we have implemented the same algorithms specific to be run sequentially on CPUs. We have used Intel Core2Quad CPU Q8400 with 2.66 GHz processor and 4GB of random memory.

4.2. Results and Discussion

4.2.1. Sorting Time

Sorting time is recorded as the actual sorting duration of the queue in seconds and does not take into account any memory copy and other contention times. Fig. 4 reports sorting times of bitonic, odd-even and rank sort on different GPU devices and CPU. Data in-dependency in case of bitonic and odd-even sorts makes them suitable for parallel systems. Fig. 4.a and 4.b illustrate how faster bitonic and odd-even sort run on GPU devices and get considerable speedup over their respective serial implementation on CPU. While on the other hand as shown in Fig. 4.c rank sort performs considerably well on CPU rather than on GPU devices because of the data dependency during sorting. On quadro 6000, bitonic sort has recorded minimum sorting time for very large queue size i.e. 2^{25} as 1.97 seconds and 0.0012 seconds for queue size 2^{15}. For odd-even sort it is 0.23 seconds for queue size 2^{15} and for rank sort it is 28.8 seconds for queue size 2^{15}. On the GTX 260, rank-sort has recorded maximum sorting time for queues size 2^{15} as 47.2s. Whereas equivalent sized queue using bitonic sort has recorded time of 0.003 seconds and odd-even sort 0.7 seconds. On the GT 320M, for queue size 2^{15}, sorting time for rank-sort recorded is 283 seconds or 4m : 43s, which is huge as expected. The time for complete odd-even sort on the GT320M recorded as 2.17 seconds and 0.016 seconds for bitonic sort. From our results, we see average speed-up of 2.73 for odd-even sort on GTX260 vs GT 320M and speed-up of 12.11 on Quadro 6000 vs GT 320M. In case of bitonic sort the average speedup is 18.93 when Quadro 6000 is used and 10.11 if GTX 260 is used, over GT 320M respectively. Whereas for rank sort an average speed-up of 9.25 and 5.79 is achieved respectively on Quadro 6000 and GTX260. This show that both odd-even and rank-sort will achieve considerable speed-up if number of on-device cores increases. Sorting time for min-max butterfly and full-butterfly network sorting, in both cases, in relatively lower than sorting times of all three: bitonic, odd-even and rank sort. Performance is improved because of the parallel nature of the algorithm and better code optimization. Interestingly, for complete descending ordered data, min-max butterfly,besides its sole purpose of finding minimum and maximum in data, gives complete sorted data in less sorting time than others. Fig. 7.a and Fig. 8.b shows our results for min-max butterfly for large queue sizes.

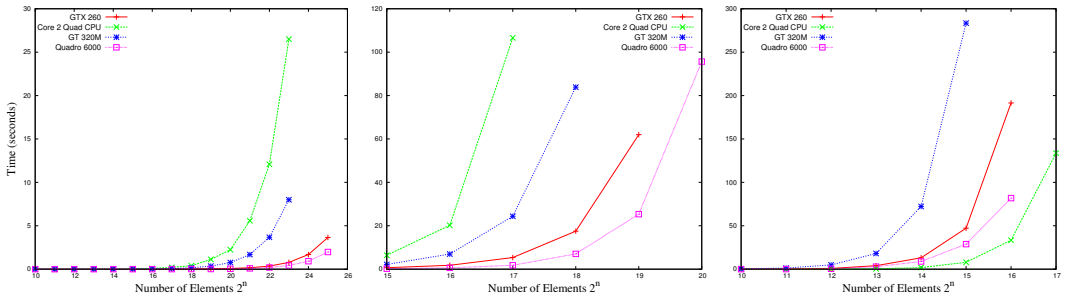
4.2.2. Sorting Rate

Fig. 5 shows sorting rate of bitonic, odd-even and rank sort, which is determined as the ratio of queue length and sorting time. For smaller queue sizes $\approx$ 212, rank-sort has a rounded rate of 800 elements on the GT320M, 4600 elements on the GTX260 and 9000 elements on Quadro 6000. In contrast, odd-even sort shows a rounded rate of 31,000 elements on GT320M, rate of 69,000 on the GTX260 and 0.2 million elements on Quadro 6000. The bitonic sort shows a rounded rate of 1.4 million elements on the GT320M, 2.9 million on the GTX260 and 9.5 million elements on Quadro 6000. In case of serial implementation on Intel Q8400 CPU the sorting rate is 0.9 million, 49,000 and 36,000 elements for bitonic, odd-even and rank sort respectively for same queue size. However, in case of odd-even and rank sort we can observe that the sorting rate approaches to zero as the size of queue increases. Of these, rank-sort

converges more quickly than odd-even sort. This suggests that both odd-even and rank-sort do not scale well for large queue sizes. But on the other hand bitonic sort performs well for all cases on Quadro 6000 and GTX 260. Bitonic sort gives rounded sorting rate of 19 million 10 million elements on Quadro 6000 and GTX 260 respectively even for a very large queue size of 2^{23} . Sorting rates for min-max butterfly are shown in Fig. 7.b for different GPU architectures and Intel system.

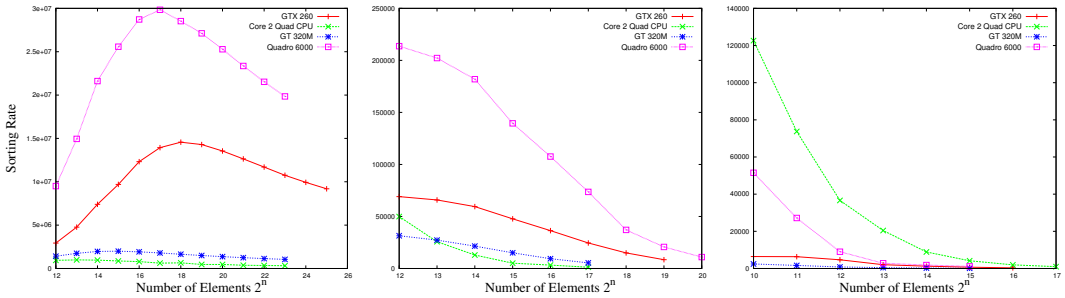
4.2.3. Speedup

Fig. 6 shows different speedups of different algorithms and architectures. Speedup of bitonic sort over odd-even on the GT 320M is recorded 45.04x for queue size 2^{12} and speed-up of 332.7x for queue size 2^{17} . Whereas on the GTX 260 the speed-up of bitonic sort is 42.4x for queue size 2^{12} and 567.91x for queue size 2^{17} and on Quadro 6000 the speed-up bitonic sort is 44.47x for queue size 2^{12} and 405.3x for queue size 2^{17} over odd-even sort. The reduced speed-up on the Quadro 6000 even though it has 18x more cores than the GT320 suggests that bitonic sort may have reduced performance edge over odd-even sort as the degree of parallelism increases. Fig. 6.a and 6.b show the speed up achieved on Quadro 6000 against other architectures for bitonic and odd-even sort respectively. As can be seen from figures, speedup increases by increasing queue size. A speedup comparison of different GPUs against Intel CPU for rank-sort is highlighted in Fig. 6.c. As shown here, the rank sort performs considerably well on CPU rather than on GPU devices because of the data dependency during sorting as it is not designed specifically for parallel systems. One thing is clear until now that increasing of number of cores on GPU, speed up of sorting algorithms also increases. A speedup improvement, on different GPU and CPU architectures, is drawn in Fig. 7.c and Fig. 8 for both min-max butterfly and full-butterfly sorting respectively. Full-butterfly gives complete sorting of large random data and has good performance relatively to other sorting algorithms, discussed here. Due to content and space limitation, we let algorithm and implementation details of full-butterfly network sorting techniques to next paper.



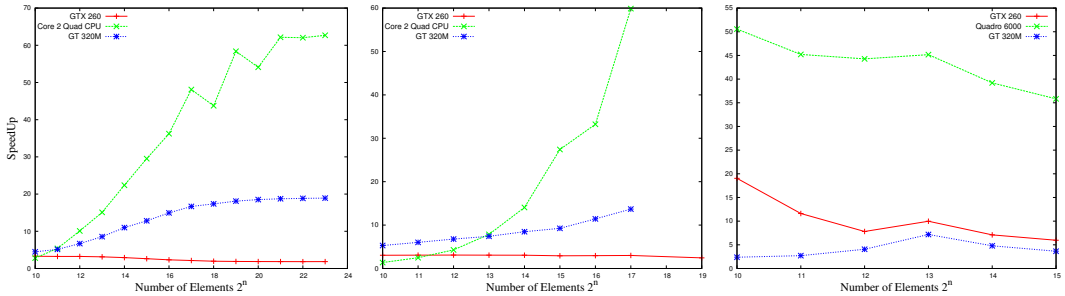
(a) Bitonic sort **(b) Odd-Even sort** **(c) Rank sort**

Figure 4 Sorting Time of different algorithms on different architectures



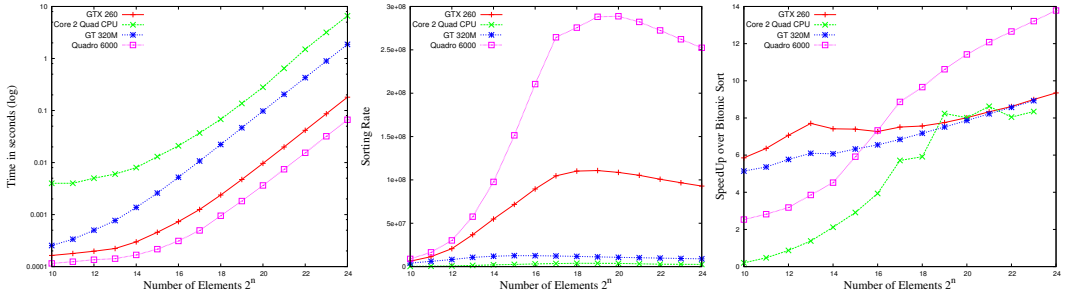
(a) Bitonic sort (b) Odd-Even sort (c) Rank sort

Figure 5 Sorting rate of different algorithms on different architectures



(a) Quadro6000 for Bitonic sort b) Quadro for Odd-Even sort (c) GPUs vs CPUS Rank sort

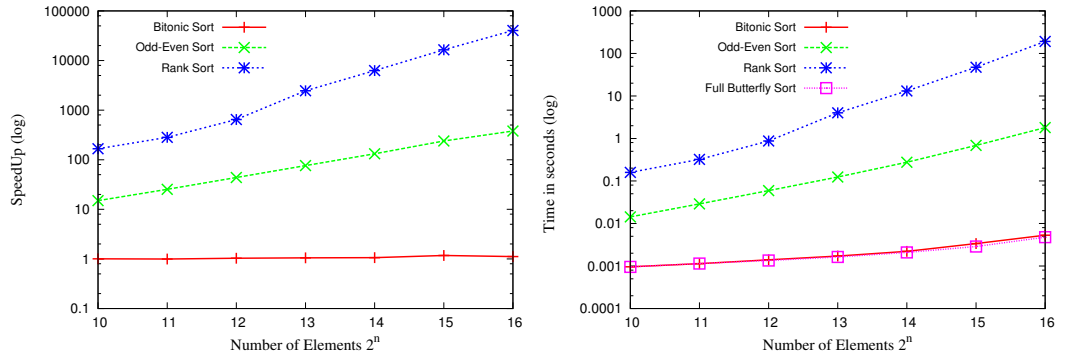
Figure 6 SpeedUp among different architectures for different algorithms



(a) Sorting time b) Sorting rate (c) speed up

Figure 7 Min-max Butterfly

test2



(a) Speedup against all b) Sorting time

Figure 8 Performance Comparison of Full-Butterfly sort and others

5. CONCLUSION

We tested performance of parallel bitonic, odd-even and rank-sort algorithms for GPUs and comparison with their serial implementation on CPU. It is shown that performance is affected mainly by two things: nature of algorithm and hardware architecture. It is shown that bitonic sort, easily parallizable, has maximum of 2300x speed-up against odd-even sorting technique on Quadro 6000 GPU, whereas rank sort performs well on CPU as data dependency of that algorithm. The performance of our algorithms: min-max butterfly and full-butterfly sort is relatively higher than the rest. Future work will be dedicated to design and implementation details of our full-butterfly sort and a feasibility report of parallel sorting algorithms for $hold\_operation$.

REFERENCES

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

[2] 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.

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

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

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

[6] 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.

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

[8] S. G. Akl, Parallel Sorting Algorithms. Academic Press, 1985.

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

[10] 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.

[11] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in Graphics Hardware (M. Segal and T. Aila, eds.), (San Diego, California, USA), pp. 97–106, Eurographics Association, 2007.

[12] 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.

[13] 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.

[14] 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.