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.

KEY WORDS
Parallel Computing, GPU, Sorting Algorithms, OpenCL

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[3]. 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[7]. 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 [6, 11].

The GPU itself is a multi-core processor where dozens of streaming processors with hundreds of cores support thousands of threads [20] running concurrently. CPU bound programs with high data independency 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 [13]. GPUs are adopted widely for computation intensive numerical simulations as they provide best price performance ratio[14]. 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 NVIDIA1 and OpenCL (Open Computing Language) by the Khronos2. 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 [4, 9]. OpenCL en-

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

Table 1: Specifications of CPU and GPUs
compares 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 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 NRange 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 organized in this way. A related work on parallel sorting is presented in section 2. Section 3 briefly discusses design, algorithm and implementation of butterfly sorting on GPU architecture. Performance analysis and concluding remarks are reported in section 4 and 5 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 [1]. A quick-sort implementation on GPU using CUDA is considered in [2]. The quick-sort algorithm discussed in [2] 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 [2] 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 [18]. In this case, the radix sort divides the sequence of $N - items$ into $N/P$ blocks. In the next phase, in order to maximize coherence of scatterers 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 [18] adopts same divide-conquer approach by dividing sequence into $p$ number of equal size blocks/tiles. An adaptive bitonic sorting algorithm is shown in [5]. Their implementation achieves optimal complexity of $O\left(\frac{n \log(n)}{p}\right)$ for sorting $n$ numbers on $p$ streaming processors. A GPU implementation of bitonic sort is discussed in [17] and CUDA based in-place bitonic sort is implemented in [16]. An overview of sorting on queues is covered in [19] focusing mainly on traffic simulations for studying the behavior of transport agents in large groups. A parallel implementation of odd-even sort suggested in [8] 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

3.1 Overview

Network Routing and sorting on hypercube is of high concern for efficiency and throughput as mentioned in [10, 15]. Generally sorting $N$ size data on a $k$-dimensional hypercube with $p - processors$ require running time of $\Theta((n \log n)/p)$ with $p = 2^k$. We have considered $2 \times 2$ 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.

3.2 Working and Algorithms

3.2.1 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 \( \frac{N}{2} \log_2 N \) butterflies where \( \frac{N}{2} \) are number of butterflies in each stage. An example diagram of length 8 min-max butterfly is shown in Fig. 2. Here \( x(0), x(1)\ldots x(7) \) can be any random values. At each stage \( \frac{N}{2} \) 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 by executing butterflies in parallel inside any 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.

![Figure 2: 8x8 Min-Max Butterfly](image)

The min-max butterfly algorithm works as follows:

```plaintext
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
```

3.2.2 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. \( \frac{N}{2} \). For complexity in terms of total butterflies, we first find total number of stages which is given by the following formula.

\[
T.\text{Stages} = \log_2 N + \sum_{i=1}^{\log_2 N-1} i \quad (1)
\]
\[
T.\text{Butterflies} = \frac{N}{2} \times T.\text{Stages} \quad (2)
\]

In equation 1, \( \log_2 N \) are total number of out-kernels represented by the first \( do-parallel \) block of the algorithm where 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 length 16 full butterfly network.
Algorithm 2: Full Butterfly Sorting

begin
    for $x_{out} \leftarrow 1$ to $\log_2(size)$ do
        $PowerX = \text{radix}^{x_{out}}$;
        do parallel $T$
            $yl\text{ndex} = t / (PowerX / \text{radix})$;
            $kl\text{ndex} = t \% (PowerX / \text{radix})$;
            $Pos_{start} = kl\text{ndex} + yl\text{ndex} \times PowerX$;
            $Pos_{end} = PowerX - kl\text{ndex} - 1 + yl\text{ndex} \times PowerX$;
            if $Pos_{start} > Pos_{end}$ then
                swap ($Pos_{start}$, $Pos_{end}$)
            end
        end
    end
    if $x > 1$ then
        for $x_{in} \leftarrow x$ to 1 do
            $PowerX = \text{radix}^{x_{in}}$;
            do parallel $T$
                $yl\text{ndex} = t / (PowerX / \text{radix})$;
                $kl\text{ndex} = t \% (PowerX / \text{radix})$;
                $Pos_{start} = kl\text{ndex} + yl\text{ndex} \times PowerX$;
                $Pos_{end} = kl\text{ndex} + yl\text{ndex} \times PowerX + PowerX / \text{radix}$;
                if $Pos_{start} > Pos_{end}$ then
                    swap ($Pos_{start}$, $Pos_{end}$)
                end
            end
        end
    end
end

3.3 Visual Profiler Analysis

NVIDIA Compute Visual Profiler presents useful information in a graphical way, generated after compiling the code. Such information include GPU-CPU times, memory load-store analysis, occupancy etc. One such profiler output of the GPU Time Summary plot is shown in Fig. 4. The corresponding bars in the figure, are sorted in decreasing order of the GPU time spent by each method. All profiler analysis are taken on NVIDIA GeForce GTX 260 with compute capability 1.3. Number of workgroups and their sizes plays significant role in over performance of the application. Throughput is also affected by the global memory access type that is typically accessed in 32, 64 or 128-byte chunks [12]. Figure 5 shows number of diverging threads in the 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; $Pos_{start}$ and $Pos_{end}$, are carried out 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:-

$$globalSize = \frac{size}{2}$$

$$localSize = \begin{cases} \frac{globalSize}{8} & \text{if } globalSize < 256 \\ 256 & \text{otherwise} \end{cases}$$

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

$$\rho_l = \frac{\sum_{st} \left[ gld\_{32} \times 32 + gld\_{64} \times 64 + gld\_{128} \times 128 \right]}{n \times sizeof(\text{float})}$$

$$\rho_{l_{avg}} = \frac{\rho_l}{total_{st}} \times 100$$

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

$$\rho_{s_{avg}} = \frac{\rho_s}{total_{st}} \times 100$$
Figure 3: 16x16 Full Butterfly Sorting

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

Figure 6: Memory load ($\rho_l$)-store ($\rho_s$) counts and thread Divergence counts for a single stage of full butterfly sort. The position of peaks observable in (a) where localSize comes from equation 3 and (b) localSize = 256 for dataSize $\geq$ 512 conform to the length of the work-group sizes and are inevitable 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.
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.

4.1 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.

4.2 Results

4.2.1 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 [8].

4.2.2 Sorting Rate

Sorting rate is the ratio of queue size to sorting time. Sorting rates for bitonic, odd/even and rank reported in[8] 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.

4.2.3 Speedup

Figures 9a,9b and 9c report improvement in speedups of our butterfly sort against others. It achieves 2x speedup over bitonic sort, a speedup of nearly $10^4 x$ on Quadro6000 over rank and odd even sort for parallel implementation and speedup of nearly $10^3 x$ 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


Figure 7: Min-max Butterfly

Figure 8: Full Butterfly Sorting Time and Rate

Figure 9: SpeedUp Improvement of Full Butterfly Sort