An OpenCL Method of Parallel Sorting Algorithms for GPU Architecture

Krishnahari Thouti *kthouti@gmail.com*
Department of Computer Science Engg.
Visvesvaraya National Institute of Technology
Nagpur, 440010, Maharashtra, India

S. R. Sathe *srsathe@cse.vnit.ac.in*
Department of Computer Science Engg.
Visvesvaraya National Institute of Technology
Nagpur, 440010, Maharashtra, India

Abstract

In this paper, we present a comparative performance analysis of different parallel sorting algorithms: Bitonic sort and Parallel Radix Sort. In order to study the interaction between the algorithms and architecture, we implemented both the algorithms in OpenCL and compared its performance with Quick Sort algorithm, the fastest algorithm. In our simulation, we have used Intel Core2Duo CPU 2.67GHz and NVidia Quadro FX 3800 as graphical processing unit.

Keywords: GPU, GPGPU, Parallel Computing, Parallel Sorting Algorithms, OpenCL.

1. INTRODUCTION

The GPU (Graphics Processing Unit) [1] is a highly tuned, specialized machine, designed specifically for parallel processing at high speed. In recent years, Graphic Processing Unit (GPU) has been evolved as massive parallel processor for achieving high computing performance. The architecture of GPU is suitable not only for graphics rendering algorithms but for also general parallel algorithms in a wide variety of application domains.

Sorting is one of the fundamental problems of computer science, and parallel algorithms for sorting have been studied since the beginning of parallel computing. Batcher's $\Theta(\log^2 n)$ - depth bitonic sorting network [2] was one of the first methods proposed. Since then many different parallel sorting algorithms have been proposed [7, 9, 10]. The $\Theta(\log n)$ - depth sorting circuit was proposed in [4, 6].

Given, a diversity of parallel architectures and a number of parallel sorting algorithms, there is a question of which is the best fit for a given problem instance. An extent to which an application will benefit from these parallel systems, depend on the number of cores available and other parameters. Thus, many researchers have become interested in harnessing the power of GPUs for sorting algorithms. Recently, there has been increased interest in such research efforts [8, 11, 16]. However, more studies are needed to claim whether a certain algorithm can be recommended for a particular parallel architecture.

In this paper, we present an experimental study of two different parallel sorting algorithms: Bitonic sort and Parallel Radix sort.

This paper is organized as follows. Section - 2 provides previous work done. In Section - 3, we present GPU architecture and OpenCL Programming model. Parallel Sorting algorithms are explained in Section - 4. Test results and analysis are provided in Section - 5. Section - 6 concludes our work and makes future research plans.

2. RELATED WORK

In this section, we review previous work on parallel sorting algorithms. Study of parallel algorithms using OpenCL is still in progress and there is not much work done in this topic. However, an overview of parallel sorting algorithms is given in [5]. Here we review parallel algorithms with respect to GPU architecture.

A parallel sorting algorithm is presented in [12] for general purpose internal sorting on MIMD machines where performance of the algorithm on the Fujitsu AP1000 MIMD supercomputer is discussed. A comparative performance evaluation of parallel sorting algorithms presented in [13]. They implement parallel algorithms with respect to the architecture of the machine. An on-chip local memory version of radix sort for GPU's has been implemented [21]. As expected, OpenCL local memory is much faster than global memory. Bitonic sorting algorithm has been implemented using stream processing units and Image Stream processors in [17, 15].

An *O(n)* radix sort is implemented in [21]. As reported in [21] radix sort is roughly twice as fast as the CUDAPP[19] radix sort. Quick-sort algorithm for GPU's using CUDA has been implemented in [20] where their results suggest that given a large data set of elements, quick-sort still gives better performance as compared to radix and Bitonic sort. A portable OpenCL implementation of the radix sort algorithm is presented in [24] where authors test radix sort on several GPUs and CPUs. An analysis of parallel and sequential bitonic, odd-even and rank-sort algorithms for different CPU and GPU architectures are presented in [23] where they exploit task parallelism using OpenCL.

3. GPU ARCHITECTURE and OPENCL FRAMEWORK

NVidia GPUs comprises of array of multi-processor units called Streaming Multiprocessors (SMs), also called as Compute Units (CU) and each one consists of multiple Scalar Processor (SP) cores, also known as Processing Elements (PE). The NVidia Quadro FX 3800 has 24 SMs with 8 PEs in each SM as shown in Figure 1. There is on-chip local store called shared memory, through which the PEs communicate with SM and different SMs communicate through off-chip memory called global memory.

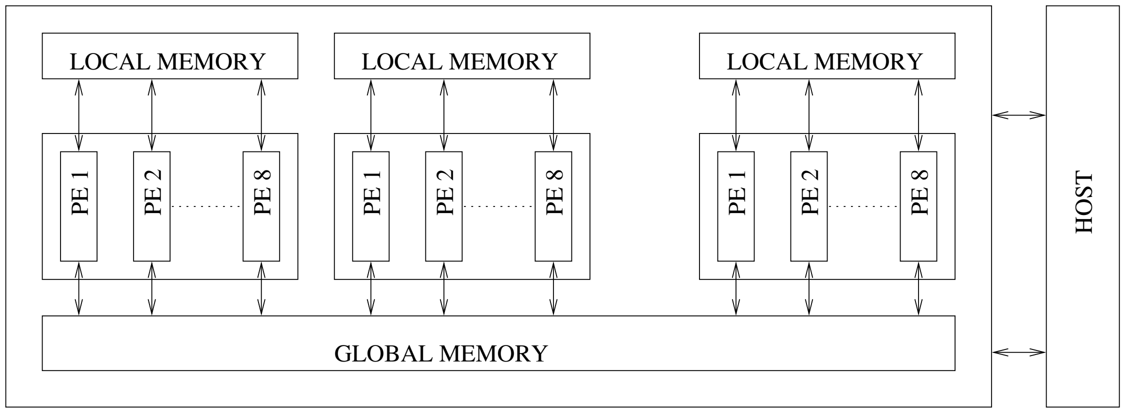

FIGURE 1: GPU Architecture

The GPU is programmable using vendor provided API's such as NVIDIA's CUDA [18], OpenCL specification by Khronos group [22]. While CUDA targets GPU specifically, OpenCL targets heterogeneous system which includes GPUs and/or CPUs. OpenCL programming model involves a host program on the host (CPU) side that launches Single Instruction Multiple Threads (SIMT) based programs called kernels consisting of groups of threads called as warps on the target device. Although management of warps is hardware dependent, programmer can organize problem domain into several work-items, consisting of one or more work-groups. This is explained as ND-Range in GPU architecture. For more information on managing and optimizing ND-Range refer to OpenCL Specifications [22]. In summary, we say, following steps are needed to initialize an OpenCL Application.

- Setting Up OpenCL Environment – Declare OpenCL context, choose device type and create the context and a command queue.
- Declare Buffers & Move Data across CPU & GPU – Declare buffers on the device and enqueue input data to the device.
- Runtime Kernel Compilation – Compile the program from the kernel array, build the program, and define the kernel.
- Run the Program – Set kernel arguments and the work-group size and then enqueue kernel onto the command queue to execute on the device.
- Get Results to Host – After the program has run, read back result array from device buffer to host memory.

See [25, 26, 27, 22] for more details on this topic.

4. PARALLEL SORTING ALGORITHMS

In this section we give brief descriptions of two parallel sorting algorithms selected for implementation.

4.1 Bitonic Sort

Batcher's Bitonic sort [2] is a parallel sorting algorithm which merges two bitonic sequences. Bitonic sorting was originally defined in terms of sorting networks. Sorting networks are comparison networks that always sort their inputs. A sorting network [14, 3] is a special kind of sorting algorithm, where the sequence of comparisons is data independent. This makes sorting networks suitable for implementation in hardware or in parallel processor arrays.

A bitonic sequence is a sequence of values $a = \{a_0, a_1 \ldots, a_{p-1}\}$ with the property that either (1) there exist an index k, where $0<k<p-1$ such that $a_0 \le a_1 \le \ldots \le a_k \ge \ldots \ge a_{p-1}$ or $a_0 \ge a_1 \ge \ldots \ge a_k \le \ldots \le a_{p-1}$ or (2) there exist a cyclic shift of indices so that (1) is satisfied. For example, (4, 8, 12, 15, 11, 6, 3, 2) is a bitonic sequence.

Let $s = \{a_1, a_2 \ldots a_p\}$ be bitonic sequence such that $a_0 \le a_1 \le \ldots \le a_{p/2-1}$ and $a_{p/2} \le a_{p/2+1} \le \ldots \le a_{p-1}$. The bitonic sequence s can be sorted with bitonic split operation which halves the sequence into two bitonic sequences s_1 and s_2 such that all values of s_1 are smaller than or equal to all the values of s_2. That is, bitonic split operation performs:

$$S_1 = \{\min(a_0, a_{p/2}), \ldots, \min(a_{p/2-1}, a_{p-1})\}$$
$$S_2 = \{\max(a_0, a_{p/2}), \ldots, \max(a_{p/2-1}, a_{p-1})\}$$

For example, the bitonic sequence mentioned above s = (4, 8, 12, 15, 11, 6, 3, 2) will be divided to two bitonic sequences s_1 = (4, 6, 3, 2) and s_2 = (11, 8, 12, 15). Thus, given a bitonic sequence, we can use bitonic splits recursively to obtain short bitonic sequences until we obtain sequences of size one, at which point the input bitonic sequence is sorted. This procedure of sorting a bitonic sequence using bitonic splits is called bitonic merge (BM).

The bitonic sorting network for sorting N numbers consists of $\log(N)$ bitonic sorting stages, where i^{th} stage is composed of $N/2^i$ alternating increasing and decreasing bitonic merges of size 2^i. In OpenCL implementation, we set kernel arguments for each of the stages and call the kernel sub-routine bitonic sort. Algorithm 1, 2, and 3 shows bitonic sorting algorithm on GPU device using OpenCL. The algorithm executes on every core in GPU kernel in parallel.

```
__kernel void bitonic_sort(__global *data, int dir)
{
        divide data into in_1 and in_2
        sort(in_1, ASC)
        sort(in_2, DES)
        swap(in_1, in_2, dir)
        sort(in_1, dir)
        sort(in_2, dir)
    result = (in_1, in_2)
}
```

Algorithm 1: Bitonic Sort Kernel for SIMD Architecture

```
for each level i = 1, …, log(n)
{
      for each pass of level j = 1 to i +1
            run_kernel ();
}
```

Algorithm 2: Generalized Bitonic Sort

Algorithm 1 is bitonic sort kernel for SIMD architecture where input data is multiple of 8 data sequence. Algorithm 2 is generalized bitonic sort and its corresponding kernel is shown in algorithm 3.

```
__kernel sort(__global *data, int stage i, int pass_of_stage j,
int dir)
{
      /* using values of i, j, dir – get left_Id & right_Id */
      left_child = data [left_Id]
      right_child = data [right_Id]
      compare(left_child, right_child)

      /* copy left & right child values to data with respect to dir
*/
      data [left_child] = max(left_child, right_child)
      data [right_child] = min(left-child, right_child)
}
```

Algorithm 3: Generalized Bitonic Sort Kernel Using OpenCL

Initially, the host (CPU) device distributes unsorted vector in form of work_groups to GPU cores using the global_size and local_size OpenCL Parameters. Alternate work_items in work_group perform sorting in ascending and descending order. Next, merging stage is performed and result is obtained. For more information, on this parameters please refer OpenCL Specifications [22].

4.2 Parallel Radix Sort

Like the bitonic sort, the radix sort [14] uses a divide-and-conquer strategy; it splits the dataset into subsets and sorts the elements in the subsets. But instead of sorting bitonic sequences, the radix sort is a multiple pass distribution sort algorithm that distributes each item to a bucket according to least significant digit of the elements. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant digit.

Suppose, the input elements are 34, 12, 42, 32, 44, 41, 34, 11, 32, 63.

After First Pass: {[41, 11],   [12, 42, 32, 32],   [63],   [34, 44, 34]}

After Second Pass: {[11, 12], [32, 32, 34, 34], [41, 42, 44], [63]}

When we collect them they are in order: {11, 12, 32, 32, 34, 34, 41, 42, 44, 63}

In OpenCL, the first step of each pass is to compute histogram to identify the least significant digit. Let '*p*' be the number of number of processing elements available on GPU device. Each processing element is responsible for $\lceil n/p \rceil$ input elements. In next step, each processing element counts the number of its elements and then computes the prefix sums of these counts. Next, the prefix sums of all processing elements are combined by computing the prefix sums of the processing element-wise prefix sums. Finally, each processing element places its elements in the output array. More details are given in the pseudo-code below.

```
b ← no. of bits
A← Input Data
cmp ← 1
cnt_0 ← contains zero's count
cnt_1 ← contains one's count
One, Zero ← Bucket Arrays
Mask ← Temporary Array

for ( i = 0 to 2^b – 1)
{
    for ( j = 0 to A.size)
    {
        if (A [j] && cmp)
            cnt_1 ++
            One [cnt_1]  ← a[j]
        else
            cnt_0 ++
            Mask [cnt_0] ← j
    }
    for( j = cnt_0 to A.size)
    Mask [j] ← A.size – cnt_0 + j

  A ← shuffle(A, one, Mask)
  cmp ← left_shift(cmp)
}
result ← A
```

Pseudo-code: Parallel Radix Sort Kernel

The code performs bitwise AND with *cmp*. If AND result is non-zero, code places the element in *One* array and increments one's counter. If the result is zero, the code set appropriate value in *Mask* array and increment zero's counter. Once every element is analyzed, the *Mask* array is further updated to identify each element in *One;s* array. The *shuffle* function re-arranges the *Mask* array data and then process continues.

The computation of histogram is shown in algorithm 4. After this step, histogram is scanned and prefix sum is calculated using the algorithm 5. After this step, re-ordering of histogram takes place and finally result is obtained by transposing the re-ordered histogram. Other implementation details are not mentioned here; only the method is presented in this paper. For more information refer [27].

5. EXPERIMENTAL RESULTS

In this section, we discus machine specifications on which experiments were carried out and present our experimental results. In all cases, the elements to be sorted were randomly generated 10 bit integers. All experiments were repeated 30 times and the results were reported are averaged over 30 runs.

```
Let n = no. of elements
$w_i$ = no. of work_items
$w_g$ = no. of work_groups

/* $w_i$ & $w_g$ can be computed using clDeviceInfo()
    : see [22] */
for ( i = $w_i$ to $w_i$ + $w_g$)
{
      Extract the group of bits of pass i,and
      Store the result in hist []
}
```

Algorithm 4: Compute Histogram

```
for each processing element, $PE_i$
{
      sum[i] = list [ (n/p) * i]
      for ( j = 1 to n/p)
      sum[i] = sum[i] + list[(n/p) * i + j ]

   result = ∑(sum)
}
```

Algorithm 5: Parallel Prefix Sum

5.1 Machine Descriptions

The GPU device used for testing simulation is NVidia Quadro FX 3800 which has 192 processing cores and 1 GB device global memory. For comparison purpose, we have implemented and tested the results of quick-sort algorithm on 2.66GHz Intel Core2DUO CPU E7300 with 1GB RAM. The cache specifications are 32KB data cache, 32KBinstruction cache and 3MB shared L2 cache.

5.2 Comparison of the Algorithms

Figure 2 shows the comparison of above mentioned algorithms for different size of input sequence. For comparison purpose, we have taken the sequential version of Quick sort and have compared with OpenCL version of Parallel Bitonic Sort and Parallel Radix Sort. As expected, in all cases, radix sort is fastest, followed by Bitonic sort, and then quick sort. GPU is a large computation unit and thus we measured the GPU runtime called as GPU PROFILE time only, excluding the time for GPU memory allocation, data and memory transfer between CPU and GPU. However, if we take into account, all the parameters concerning GPU application, as explained in Section – 3, we find that quick sort is still the fastest.

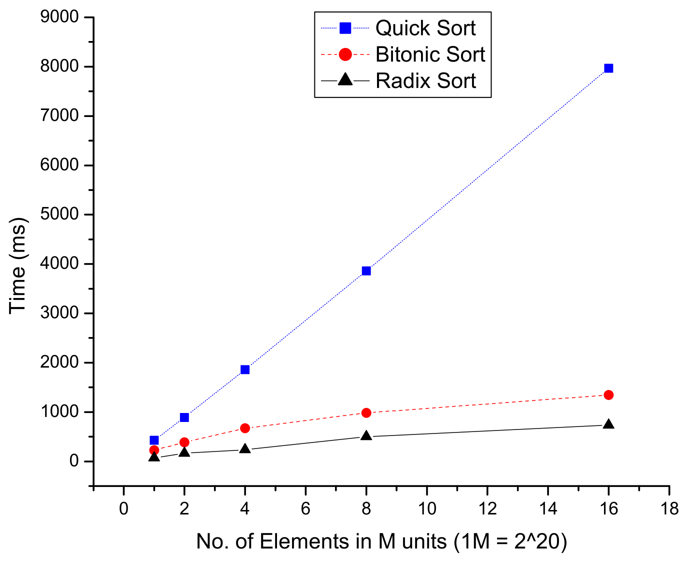

FIGURE 2: Comparison of Sorting Algorithms

6. CONCLUSION AND FUTURE SCOPE

We have presented an analysis of parallel bitonic and radix sort algorithms for GPUs using OpenCL and their comparison with the serial implementation of quicksort on CPU Dual-core machine. We have shown their GPU performance and compared with CPU implementation of quick sort. Our finding reports that radix sort is still the fastest, followed by Bitonic sort, and then quick sort. In future work, along with these sorting algorithms, we are planning to investigate some other parallel sorting algorithms including quick sort and use different GPU architecture from different vendors for our analysis.

REFERENCES

[1] General Purpose Computations Using Graphics Hardware, http://www.gpgpu.org/

[2] K. E. Batcher. “Sorting networks and their applications”. in AFIPS Spring Joint Computer Conference, Arlington, VA, Apr. 1968, pages 307–314.

[3] D.E. Knuth. The Art of Computer Programming. Vol. 3: Sorting and Searching (second edition). Menlo Park: Addison-Wesley, 1981.

[4] M. Ajtai, J. Komlos, Szemeredi. “Sorting in parallel steps”. Combinatorica 3. 983, pp. 1 -19.

[5] S. G. Akl. “Parallel Sorting Algorithms”, Academic Press, 1985.

[6] J. H. Reif, L. G. Valiant. “A Logarithmic Time Sort for Linear Size Networks”. Journals of the ACM, 34(1): 60 – 76, 1987.

[7] G.E. Blelloch,” Vector Models for Data-Parallel Computing”. The MIT Press, 1990.

[8] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha. “A Comparison of Sorting Algorithms for the Connection Machine CM-2”. in Annual ACM Symp. Paral. Algo: Arc. 1991, Pages 3 -16.

[9] F. T. Leighton, “Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes”. Morgan Kaufmann, 1992.

[10] J.H. Reif. ”Synthesis of Parallel Algorithms”. Morgan Kaufmann, San Mateo, CA, 1993.

[11] H. Li, K.C. Sevcik. “Parallel Sorting by Over-partitioning”. in Annual ACM Symp. Paral. Algor.Arch. 1994, pages 46 – 56.

[12] A. Tridgell, R. P. Brent. “A general-purpose parallel sorting algorithm” in International J. of High Speed Computing 7 (1995), pp. 285-301.

[13] N. Amato, R. Iyer, S. Sundaresan, Y. Wu. “A Comparison of Parallel Sorting Algorithms on Different Architectures” Texas A & M University, College Station, TX, 1998.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. 2nd edition, The MIT Press. 2001.

[15] T. J. Purcell, C. Donner, M. Cammarano, H. Jensen, P. Hanrahan “Photon mapping on programmable graphics hardware”, in Annual ACM SIGGRAPH / Eurographics conference on Graphics Hardware, 2003, pp. 41 – 50.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware.” in Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.

[17] A. Greb, G. Zachmann. “GPU-AbiSort: Optimal Parallel Sorting on Stream Architectures” in IPDPS'06 Proceedings of the 20th international conference on Parallel and distributed processing. 2006.

[18] NVidia CUDA GPGPU Framework. http://www.nvidia.com/

[19] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens. “Scan primitives for GPU computing,” in Graphics Hardware 2007, Aug. 2007, pp. 97–106.

[20] D. Cedermann, P. Tsigas. “A practical quicksort algorithm for graphic processors”, Tech. Rep, Chalmers University of Technology and Goteberg University, 2008.

[21] N. Satish, M. Harris, M. Garland. “Designing efficient sorting algorithms for manycore GPUs”. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing. May 23-29, 2009, pp.1-10.

[22] OpenCL Specification, http://www.khronos.org/opencl/

[23] F. Gul, O. Usman Khan, B. Montrucchio, P. Giaccone. “Analysis of Fast Parallel Sorting Algorithms for GPU Architectures”. in Proceeding FIT '11 Proceedings of the 2011 Frontiers of Information Technology Pages 173-178.

[24] P. Helluy. “A portable implementation of the radix sort algorithm in OpenCL”. http://code.google.com/p/ocl-radix-sort/ May 2011

[25] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, D. Schaa. Heterogeneous Computing with OpenCL. Morgan Kaufmann. 2011.

[26] AMD Accelerated Parallel Processing OpenCL Programming Guide, Advanced Micro Devices, Inc. 2012. http://developer.amd.com/appsdk

[27] M. Scarpino. OpenCL in Action. Manning Publications, 2011.