VOLUME-6, ISSUE-6, JUNE-2017 • ISSN No 2277 - 8160

Original Research Paper

Surgery

ANALYSIS OF VARIOUS SORTING ALGORITHMS IN OPENCL

Tanvi Principal Author, Department of Computer Science, PGDAV(M) college University of Delhi, New Delhi, India-110065

Shreshtha Sharma Corresponding Author, Analyst-Technology, Estee Advisors, Gurgaon, India

ABSTRACT With the availability of multi-core processors and graphics processing units in the market, heterogeneous computing environment with the immense performance capability can be easily constructed. Heterogeneous computing accelerates the performance by transferring the computation intensive code to the GPU and the remaining code runs in the CPU. OpenCL is a standardized framework for the heterogeneous computing. Sorting algorithms are considered as fundamental building blocks for algorithms and has many applications in computer science and technology. In this paper, we implement Selection Sort, Btonic Sort and Radix Sort in OpenCL. The algorithms are designed to exploit the parallelism model available on multi-core GPUs with the help of OpenCL specification. In addition, the comparison between the traditional sequential sorting algorithms and parallel sorting algorithms are made on the Intel ® Core™ i5-3317U CPU @ 1.70 GHz architecture and AMD Radeon HD 7600 M series GPU.

KEYWORDS : GPU, Selection Sort, Bitonic Sort, Radix Sort & OpenCL

1. INTRODUCTION

General purpose GPU computing is defined as using graphic processing units for non-graphical purpose. To program hybrid devices, OpenCL is used. It is a non-proprietary standard and it can be compiled and executed on any processor whereas Nvidia's CUDA is used to program on Nvidia's GPUs only.

Before OpenCL, there was no famous architecture in the field of high computing and graphics. CPU's and GPU's have different type of programming models. To develop a unified programming model for heterogeneous platforms is a difficult task to achieve. The lack of a standardized framework is a major obstacle in this field of heterogeneous computing.

In heterogeneous programming, OpenCL emerged as a promising framework. It is supported by all GPU vendors and provides a unified programming model for heterogeneous computing. Due to active support for OpenCL from the vendors of CPU and GPU, OpenCL has become the standard framework for CPU and GPU platforms. The motivation of this work is to prove the achievability of OpenCL as a standard framework for heterogeneous platforms. Moreover, the performance of parallel sorting algorithms on heterogeneous platforms is compared with the sequential sorting algorithms on CPU.

The objective of this paper is to implement various sorting algorithms in OpenCL. Sorting is a fundamental algorithmic building block. The time taken by various parallel sorting algorithms is compared with the time taken by various sequential sorting algorithms running on the CPU. Here, we choose three different algorithms with totally different nature of parallelism. The sorting algorithm includes implementation of parallel selection sort, parallel bitonic sort and parallel radix sort. The performance of these sorting algorithms is analyzed and then it is compared with the sequential sorting algorithms.

2. RELATED WORK

Since the late 1960s, parallel sorting has been a topic of study. In 1968, Batcher [1] has explained his work on network sorting and describes comparison sorts which carry out individual comparisons in parallel. After this, a large amount of work is done in the field of parallel sorting algorithms. Batcher's [1] work has been adapted by Nassimi and Sahni [2]. This work describes parallel computers that have mesh interconnect. Parallel merging techniques which are used to sort datasets are developed by Francis and Mathieson [3]. Blelloch et al. [4] has presented work on several parallel sorting algorithms including bitonic sort, radix sort and sample sort. This work is similar to our work as it develops sorting algorithm to efficiently utilize modern hardware. However, the difference is that it does not target heterogeneous systems. The comparison of performance of sorting algorithms is described by Amato [5].

Prior to the release of CUDA in 2007 [7] and OpenCL in 2008 [6], GPU's have been used to carry out parallel sorting operations. In 2005, Kipfer and Westermann [8] described sorting algorithms implemented on GPU's using a framework known as pug. This work describes odd-even merge sorting and bitonic networks. In 2006, the implementation of bitonic sort on the GPU devices has been presented by Greb and Zachmann [9]. This work proves the time complexity of O(nlogn).

After the release of CUDA framework, Harris et al. [10] explained an algorithm for parallel radix sort which is based on the efficient parallel prefix sum algorithm. After this Harris et al. [11] developed parallel radix sort and merge sort for NVIDIA GPU device. Leischner et al. [12] has presented work on parallel sample sort for GPU devices.

In 2008, OpenCL was released. Since, most of the functionality of OpenCL is similar to CUDA; OpenCL versus CUDA is a frequently mentioned topic in literature. Helluy [13] presents a portable OpenCL implementation of radix sort algorithm where comparison of radix sort on several platforms is done. In 2011, an analysis of parallel and sequential sorting algorithms like bitonic sort, odd-even sort and rank sort algorithms on different architectures are presented by Gul and Khan [14] where task Parallelism is used.

3. OPENCL ARCHITECTURE

OpenCL is an open standard framework for programming on heterogeneous platforms. It is a framework for parallel progra mming. The main aim of OpenCL is to write a portable yet effective code. The following hierarchy of models describe the OpenCL in detail:

3.1 PLATFORM MODEL [6]

This model consists of a host device which is connected with the OpenCL compliant devices. OpenCL device consists of many compute units which are divided into processing elements. Processing elements are responsible for performing computations on a device. Host device executes the host application and it sends commands to the processing elements which present in the GPU devices for execution of the parallel code. Figure 1 explains the platform model for OpenCL.

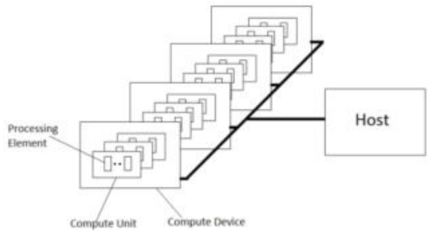


Figure 1. Platform Model for OpenCL

3.2 EXECUTION MODEL [6]

Execution of OpenCL program consists of two parts: Kernels and host program. Kernels execute on OpenCL compliant devices and host program executes on host. Host program is responsible for defining the context for kernels and its management. The main task of execution model is execution of kernels.

In OpenCL, tasks are known as kernels. Kernels are functions that are sent to OpenCL complaint devices by host application and host application is a regular C/C++ application program. Host application manages devices with the help of context and context act as a device container. Program is a kernel container from which host selects a function to create a kernel. Kernel is dispatched to a command queue. Through command queue, host tells devices what to do. Figure 2 explains the Kernel distribution among devices.

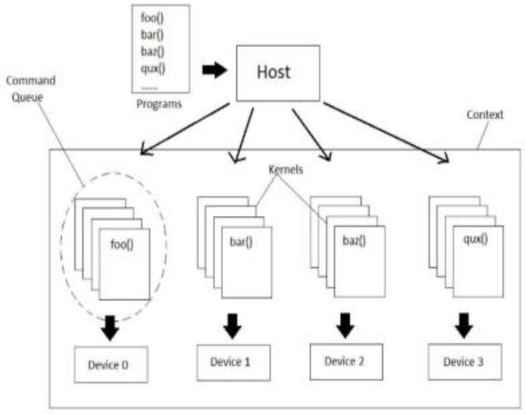


Figure 2. Kernel distribution among OpenCL compliant devices

3.3 MEMORY MODEL [6]

In OpenCL, every kernel argument that references memory has an address space modifier. There are four address spaces. Global memory stores data for entire GPU device. It is a read and write memory. Constant is similar to global memory but it is a read only memory. In local memory, data for all the work items in a work group is stored in this memory. Private memory stores data for a particular work item.

3.4 PROGRAMMING MODEL [6]

OpenCL supports data parallel and task parallel parallel program ming models. Data parallel programming model is the primary model for the design of the OpenCL. In a data parallel system, each device receives the same instructions. But it operates on different sets of data. Task parallel programming model allows different devices to perform different tasks. Each task operates on different data.

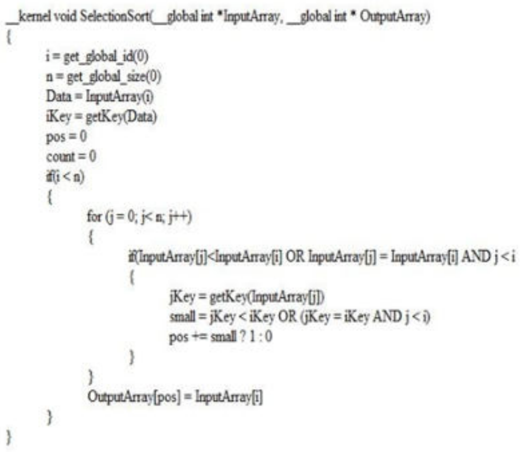
4. IMPLEMENTATION

In this section, various sorting algorithms are discussed and also their implementation on the OpenCL framework has been explained. The performance of these sorting algorithms is tested on AMD Radeon HD 7600 M series GPU and Intel® Core™ i5-3317U CPU @ 1.70 GHz architecture. After that performance of parallel sorting algorithms implemented in OpenCL is compared with the traditional sorting algorithms.

4.1 SELECTION SORT

Selection sort is a sequential algorithm. The implementation of traditional selection sort is very simple. Firstly, it finds out the smallest element and then put it into its right position. This process is to be repeated till all the elements are sorted out.

While implementing parallel selection sort, host device, sends the unsorted array to the GPU devices and devices will sort the elements in a parallel manner. Let N be the size of elements to be sorted. In algorithm, we set the N work items and each work item will process on the entire set of N elements in order to find out the exact position of a particular element in the array. After completion of sorting, the sorted array is sent back to the host device. The algorithm for the parallel selection sort is defined in Figure 3.



```
__kernel void SelectionSort(__global int *InputArray, __global int * OutputArray)
{
    i = get_global_id(0)
    n = get_global_size(0)
    Data = InputArray(i)
    iKey = getKey(Data)
    pos = 0
    count = 0
    if(i < n)
    {
        for (j = 0; j< n; j++)
        {
            if(InputArray[j]<InputArray[i] OR InputArray[j] = InputArray[i] AND j < i
            {
                jKey = getKey(InputArray[j])
                small = jKey < iKey OR (jKey = iKey AND j < i)
                pos += small ? 1 : 0
            }
        }
        OutputArray[pos] = InputArray[i]
    }
}
```

4.2 BITONIC SORT

A monotonic sequence is a sequence in which all elements are sorted in one direction i.e. the value increases (or decreases) from left to right. If ak < ak+1 for all k < m then the sequence a1, a2, a3.....am is considered as monotonically increasing. The sequence which increases monotonically reaches a single maximum point and after that monotonically decreases is known as a bitonic sequence. Thus, the sequence becomes bitonic by cyclically shifting the sequence.

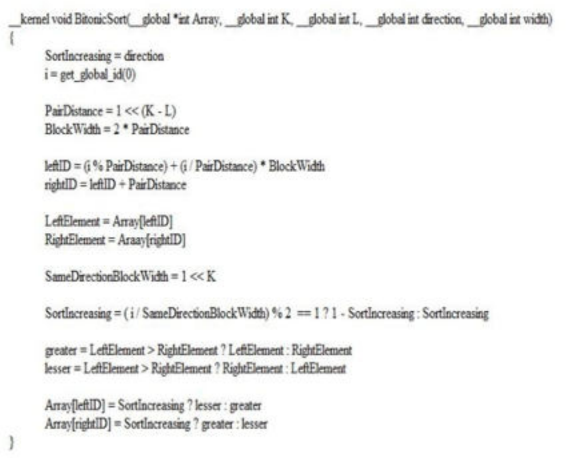
In order to sort the elements using Bitonic Sorting, bitonic split property is used. In bitonic split, if ak > ak+m/2 then the two elements are exchanged, where 1 < k < m. After this step, two bitonic sequences X and Y are produced such that all the elements present in Y is greater than all the elements present in X. Bitonic sequence can be converted into a monotonic sequence by performing bitonic split repeatedly. In bitonic sort, total k steps are required to sort n elements where n = 2k.

Bitonic Sort is a parallel sorting algorithm. While implementing Bitonic Sort in OpenCL, host device sends the unsorted elements to the GPU cores in the form of work groups which uses global size and local size parameters. In work group, alternate work items perform sorting in descending and ascending order respectively.

Figure 4 defines the algorithm for the Bitonic Sort in OpenCL. Let N denotes the total number of elements present in an array. Then total wok items will be set to N/2. Bitonic Sort kernel has five arguments and all of them are stored in global memory. The K argument defines the total number of steps required to sort an unsorted array and L argument defines the sub-steps of each K. The direction argument defines the direction in which sorting is performed i.e. ascending order or descending order and the total number of elements to be sorted is defined by width argument. Host program will call the L*K

VOLUME-6, ISSUE-6, JUNE-2017 • ISSN No 2277 - 8160

kernels. After completion of sorting, the sorted array is moved back to the host device and control goes to the host program.



```
__kernel void BitonicSort(__global *int Array, __global int K, __global int L, __global int direction, __global int width)
{
        SortIncreasing = direction
        i = get_global_id(0)

        PairDistance = 1 << (K - L)
        BlockWidth = 2 * PairDistance

        leftID = (i % PairDistance) + (i / PairDistance) * BlockWidth
        rightID = leftID + PairDistance

        LeftElement = Array[leftID]
        RightElement = Araay[rightID]

        SameDirectionBlockWidth = 1 << K

        SortIncreasing = ( i / SameDirectionBlockWidth) % 2  == 1 ? 1 - SortIncreasing : SortIncreasing

        greater = LeftElement > RightElement ? LeftElement : RightElement
        lesser = LeftElement > RightElement ? RightElement : LeftElement

        Array[leftID] = SortIncreasing ? lesser : greater
        Array[rightID] = SortIncreasing ? greater : lesser
}
```

Figure 4. Parallel Bitonic Sort Kernel

4.3 RADIX SORT

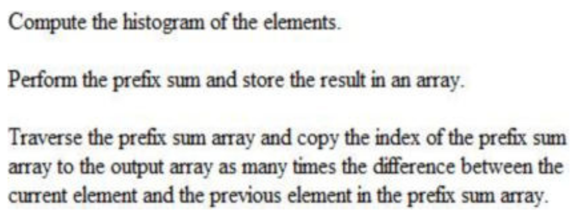
Radix sort is an efficient stable sort algorithm for sorting elements in a list. It sorts the data by distributing each element in a bucket which has the same value. After completion of each pass, the items are collected and stored in order in an array. Then again the previous step is repeated according to the next significant digit. Radix sort can be easily parallelized.

In Radix Sort, each element is between 0 and 2^B-1 where B is the total number of bits required to represent the keys. Total number of passes p in the algorithm is represented by B/R, where R is the number of bits which are required to represent the radix.

In our implementation, first step of each pass is to compute a histogram. Here, we assume G groups, each group has I items. Therefore, the total numbers of processors are GI. This part of an algorithm is completely parallel. In this list of elements are considered as a matrix with IG rows and N/IG columns which are stored in a row-major order. Each row is processed by a single work item. Firstly, transposition of matrix is calculated. It makes the matrix in the column-major order. In the end, same step is repeated in order to recover the sorted list of elements. Then the histogram is calculated by computing the xth digit in the list. It basically identifies the least significant digit.

In the second part of the pass, parallel prefix sum is calculated for the resultant array calculated from the previous step. Here, the array is firstly split into m parts and then they are separately scanned and sum is stored in an auxiliary array. Finally, all the sums are combined to obtain the global sum of the histogram array.

After the computation of prefix sum, each item finds its part on the list and using the resultant array obtained from the previous steps, it puts the keys at their right position.



Compute the histogram of the elements.

Perform the prefix sum and store the result in an array.

Traverse the prefix sum array and copy the index of the prefix sum array to the output array as many times the difference between the current element and the previous element in the prefix sum array.

Figure 5. Parallel Radix Sort Kernel

5. RESULTS

In this section, the results of sequential sorting algorithms and parallel sorting algorithms on Intel® Core™ i5-3317U CPU @ 1.70 GHz architecture and AMD Radeon HD 7600 M series GPU are compared. Sorting time i.e. the time taken by an algorithm to sort the elements is the metric of performance. The time taken by various sorting algorithms in milliseconds is shown in table 1. On the basis of this figure, graphs are drawn between the time taken in milliseconds and the input size which compares the traditional sequential algorithms with the parallel sorting algorithms.

Table 1. Time Taken by sorting algorithms for different input size

Sorting	Time	Time	Time	Time	Time	Time
Algorithm	(in ms)	(in ms)	(in ms)	(in ms)	(in ms)	(in ms)
	for n	for n	for n	for n	for n	for n
	= 8192	= 16384	= 32768	= 65536	= 131072	= 262144
Sequential	297	1109	4407	16781	58281	274375
Selection Sort						
Parallel	1000	1031	1297	2407	6422	22844
Selection Sort						
Sequential	16	46	93	187	422	922
Bitonic Sort						
Parallel	47	47	47	62	78	109
Bitonic Sort						
Sequential	5062	5094	5266	5250	5859	5781
Radix Sort						
Parallel	1172	1203	1219	1219	1265	1375
Radix Sort						

Figure 6 shows the comparison between the sequential selection sort and the parallel selection sort. For small number of elements, the performance of both the traditional algorithm and parallel algorithms does not show noticeable difference. But for large number of elements in an array, traditional selection sort is very slow and ineffective whereas parallel selection sort is very effective and provides high performance.

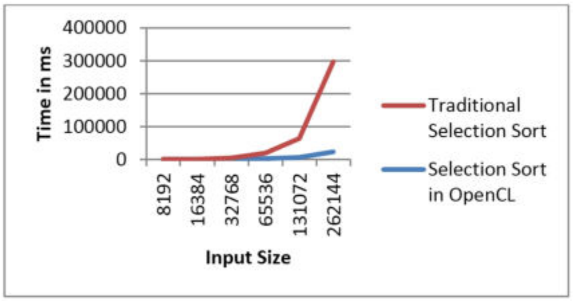


Figure 6. Comparison between Traditional Selection Sort and Parallel Selection Sort

Figure 7 compares the traditional bitonic sort with the parallel bitonic sort. The graph clearly shows the improvement in the performance of the parallel bitonic sort especially for large numbers.

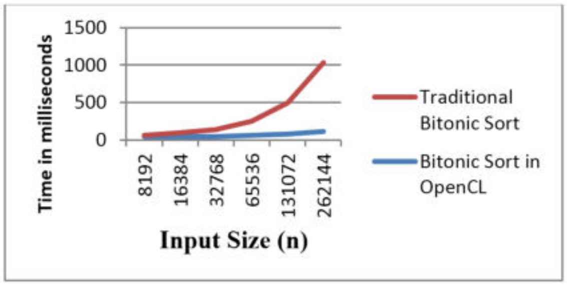


Figure 7. Comparison between Traditional Bitonic Sort and Parallel

Bitonic Sort

Figure 8 shows the comparison between the sequential radix sort and the parallel radix sort. It can be clearly concluded from the graph that the implementation of radix sort in OpenCL is very effective and takes noticeable less time than the sequential radix sort.

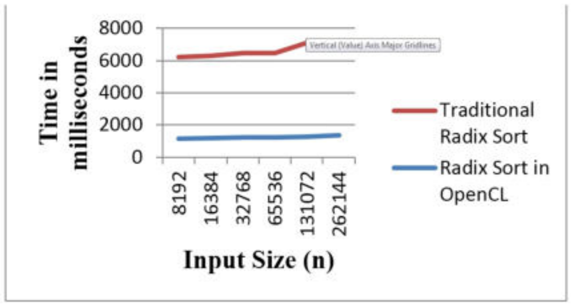


Figure 8. Comparison between Traditional Radix Sort and Parallel Radix Sort

Now, from the above comparisons we can clearly say that since selection sort is a sequential sorting algorithm and it does not exploit the full processing capabilities of GPU takes maximum time. Bitonic Sort is a parallel sorting algorithm. It takes minimum time in both sequential and parallel implementation. Radix sort fully utilizes the GPU and can easily be parallelized. Although it takes more time than the bitonic sort but it shows noticeable improv ement in the performance when implemented on GPU.

Also, it may be noted that for small n, there is not much difference between the performance of the parallel sorting algorithms and traditional sorting algorithms and even sometime traditional sorting algorithms are better. This is because the cost of reading data from the CPU and copying it to the GPU and after computing the result, writing back data to the CPU from GPU is an inefficient task. Therefore, for small n, sequential algorithms are more efficient and for large n, optimized algorithms implemented in OpenCL are more suitable and efficient.

6. CONCLUSIONS

In our paper, parallel sorting algorithms implemented on GPUs are compared with their serial implementation on CPU. It can be concluded that OpenCL provides a noticeable improvement in the performance of various parallel algorithms as compared with the traditional sorting algorithms. The results of our implementation on AMD GPU are that the bitonic sort is fastest followed by radix sort and selection sort. Also, it can be concluded that for small n, sequential algorithms are more efficient and for large n, optimized algorithms implemented in OpenCL are more suitable and efficient.

7. REFERENCES

[1] K. Batcher. Sorting networks and their applications. In AFIPS Spring Joint Computer Conference, vol. 32, pages 307{314, 1968.
[2] D. Nassimi and S. Sahni. Bitonic Sort on a mesh-connected parallel computer. Computers, IEEE Transactions on, C-28(1):2 {7, jan. 1979.
[3] R. Francis and I. Mathieson. A benchmark parallel sort for shared memory multiprocessors. Computers, IEEE Transactions on, 37(12):1619 { 1626, dec 1988.
[4] G. Belloch, C. Leiserson, B. Maggs, C. Plaxton, S. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. pages 3{16, 1991.
[5] Nancy Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A comparison of parallel sorting algorithms on different architectures. Technical report, College Station, TX, USA, 1998.
[6] The OpenCL specification, version: 1.0. http:// www.khronos.orf/ registry/ cl/specs/opencl-1.0.pdf, 2008.
[7] CUDA toolkit archive | NVIDIA developer zone. https:// developer. nvidia.com/ cuda-toolkit-archive, 2012.
[8] P. Kipfer and R. Westermann. Improved GPU Sorting. In Matt Pharr, editor, GPU Gems 2, chapter46. Addison Wesley, March 2005.
[9] A. Greb and G. Zachmann. GPU-ABiSort: optimal parallel sorting on stream architectures. In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, page 10 pp., April 2006.
[10] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851 {876. Addison Wesley, August 2007.
[11] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium, pages 1 { 10, May 2009.
[12] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1 {10, April 2010.
[13] Philippe Helluy. A portable implementation of the radix sort algorithm in opencl. 2011.
[14] B. Montrucchio P. Giaccone F. Gul, O. Usman Khan. Analysis of fast parallel sorting algorithms for gpu architectures. In Proceeding FIT '11 Proceedings of the 2011 Frontiers of Information Technology, 2011.