GPU Matrix Sort (An efficient implementation of Merge Sort)

Mukul Panwar
DIT University, Dehradun

Monu Kumar
DIT University, Dehradun

Sanjay Bhargava, Ph. D
DIT University, Dehradun

ABSTRACT
Sorting is one of the frequent used operations in computer science. Due to highly parallel computing nature of GPU architecture; it can be utilized for sorting purpose. We have considered the input array that is to be sorted in a 2D matrix form and applied a modified version of merge sort on that matrix. This modification leads to a much efficient sorting algorithm with reduced complexity. Therefore a lot of work has already been done to improve the efficiency of sorting algorithms. In this paper We have used the GPU architecture for solving the sorting problem.

Keywords
Sorting, Multi-Core, CUDA, Quicksort.

1. INTRODUCTION
In this paper We have described an efficient parallel algorithmic implementation of merge sort, GPU matrix sort, designed to take advantage of the highly parallel nature of the graphics card. Merge sort is an efficient sorting algorithm in practice for single processor system. It is well known that merge sort is a comparisons –based sorting requires 0(nlgn) comparisons to sort n elements[1][2].Merge sort best case, average case and worst case time complexity is 0(nlgn). Today's graphics cards contain very powerful multicore processors. These multicore processors are mainly used in gaming. But since the processors are specialized for compute intensive highly parallel computations, they could be used to solve problems. Standard graphics API such as open GL [3] and DirectX [4] do not expose much of the underling computational capabilities that graphic hardware can provide. The lack of convenient data types, basic computational functionality, and a generic model renders this environment far from attractive from developers. The Compute Unified Device Architecture (CUDA) introduced by NVIDIA [5] is a significant advance, exposing several hardware features that are not available via the graphics API. CUDA consist of minimal set of extensions to the C language and a runtime library that provide function to control the GPU from the host, as well as device specific functions and data types. At the top level, an application written for CUDA consist of serial program running on the CPU, and a parallel part, called a kernel that runs on the GPU. A kernel however, can only be invoked by a parent process running on the CPU. As a consequence, a kernel cannot be initiated as a standalone application and it strongly depends on the CPU process that invokes it. Each kernel is executed on the device as many different thread organized in thread block. Thread blocks are executed by multiprocessors of the GPU in parallel.

2. RELATED WORK
Quick sort is a very famous sorting algorithm and it can be parallelized. The obvious way to take advantage of its inherent parallelism by just assigning a new processor to each new subsequence. This means, however, that there will be a very little parallelization in the beginning, when the sequence is few [6]. To deal with this issue, Caderman [7] introduces a parallel version of quick sort combining CPU process and GPU process. The algorithm is theoretically optimal, but the data transferring between the CPU and GPU slows down the running time in practice.

If we use the concept of merge sort than it will proceed from bottom to top while quick sort is a top down approach. It implies from the very beginning of algorithm we will have sufficient sub arrays to work with different processors. A more general usage of merge sort is to sort many sorted sub arrays into whole sorted array, due to its GPU friendly memory access pattern[8][9]. Currently there is a big interest in sorting algorithms on GPU. Prucell et al [10] have presented an implementation of bitonic merge sort on GPU based on an implementation by kapasi et al [11]. Sintorn et al [12] presented a hybrid sorting algorithm which splits the data with a bucket sort and then uses merge sort on the resulting blocks.

3. IMPLEMENTATION
Let A be an array having n items such that $A= \{a_1, a_2, a_3, a_4 \_ \_ \_ \_ \_ \_ a_n\}$ and each a_i has a key value k_i .Array A has to be sorted in increasing order of the keys. The input array A is organized into a 2D matrix which has w rows and h columns and the items are filled in row major i.e. row by row. In the above mentioned scheme row of any item a_i is given by floor (i/h). The column of any element a_i is given by (i mod h). Any item a_i can be referred as A(x, y) where x is row number and y is column number. Any item a_i is smaller than item a_j if $k_i<k_j$. The basic scheme of matrix sort has 2 steps.

i) Sorting of individual rows of matrix.

ii) Merge the two sorted rows recursively until we get one sorted array.

In the GPU architecture we can assign a single processor to each row for the sorting of that row. A single row is having h elements (due to h columns in the matrix) therefore the sorting of h elements will take hlgh time using quick sort. Therefore initially to work with w rows at least w processors are required. We can have more than w processors. In this case more than 1 processor will work on a single row. As in GPU architecture all the processor will work simultaneously therefore all the w rows can be sorted simultaneously. The sorting of a single row by a single processor require hlgh time where h is the size of a row. Total time required for sorting of w rows will also be o (hlgh) because all the processors are working simultaneously.

Table 1

S.No.	Row	Time Required	Sorted By
Row 1	[h elements]	hlgh	Processor P1
Row 2	[h elements]	hlgh	Processor P2
Row 3	[h elements]	hlgh	Processor P3
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
Row w	[h elements]	hlgh	Processor w

This will complete the first step of the GPU matrix sort i.e. sorting of individual rows of matrix. The next step of the GPU matrix sort will be merging of the two sorted rows recursively until we get one sorted array. The two sorted rows can be merged by a single processor using the merge procedure given below.

Merge Procedure: [1]-

Input:-Merge procedure will take two sorted rows say w_i and w_k as input both having h elements

Output: - Merge procedure will return a sorted array sw having 2h elements.

Algorithm:-

```
Merge (wi,wk)
    m=1, c=1, t=1;
    While(c<=h and t<=h)
            If (wi[c] <= wk[t])
                    Sw[m] = wi[c];   m++;   c++;
        else
                    Sw[m] = wk[t];  m++;   t++;
    If(c>h)
            While(t<=h)
                    Sw[m]=wk[t];
                    t++;  m++;
    else
                    While(c<=h)
                    Sw[m] =wi[c];
                    c++;  m++;
return sw;
```

The time complexity of the merge procedure will be 0(2h). After the first pass of merge w rows will become w/2 such that each row will have 2h elements. The first pass of merging will take 0(2h) time because the merging of all the rows can be done simultaneously by w/2 or more processors.

Sorted row 1 Sorted row (1 & 2)

Sorted row 2 Sorted row (3 & 4)

Sorted row 3 After merging of Sorted row (4 & 5)

Sorted row 4 pass 1

Sorted row (5 & 6)

..

.

Sorted row $_{w-1}$

Sorted row (w-2& w-3)

Sorted row $_{w}$ Sorted row (w-1 & w)... (Total w/2 rows)`

The above procedure of merging the rows will take place repeatedly until we get one array having n elements in sorted order.

After pass 1 each row will have 2h elements = 2^1h elements.

After pass 2 each row will have 4h elements= 2^2h elements.

After pass 3 each row will have 8h elements= 2^3h elements.

After pass 4 each row will have 16h elements= 2^4h elements.

Similarly

After pass k each row will have 2^kh elements.

For k to be final merge pass 2^kh =n

2^k=n/h

Taking lg_2 on both sides we get

k $lg_2$2= lg_2 (n/h)

k=lg_2 (n/h)---------- eqn (1)

Since n is the total number of array elements therefore n can be written as n= w×h

Where w= number of rows in matrix, h=number of columns in matrix.

Put the value of n in equation 1

Therefore k= lg_2 ((w×h)/h)

K= lg_2w. So from the above discussion it is clear that lg_2w merge passes will be required for getting a sorted array of n elements. Now the total time complexity of merging will be $\sum_{i=1}^{k=lgw}$ time complexity of merging in Pass i.

That is equal to:-

Time complexity merge in pass 1 +Time complexity of merge in pass 2 +……Time complexity of merge in pass k .

=2^1h +2^2h +2^3h +2^4h +……2^kh.

=2^1h +2^2h +2^3h+ 2^4h+……2^{lgw}h.

=h (2^1 +2^2 +2^3+ 2^4+……2^{lgw}) ----eqn (2)

Above series is a G.P. having a=2, r=2 and n=lg_2w.

Sum of G.P. = a (r^n-1)/(r-1)

Therefore sum will be = 2(2^{lgw}-1)/ (2-1)

= 2×2^{lgw}- 2

$= 2 \times w-2$

On putting value of sum in eqn 2 we get = h (2w-2)

$= 0(wh)$

The total time complexity of the proposed algorithm= Time complexity of sorting rows + Total time complexity of merging.

Therefore total time complexity of algorithm will be=0(hlgh+ wh).

4. EXPERIMENTS

As we have proved in implementation section the total running time complexity of the G.P.U matrix sort is 0(hlgh+w)----eqn (1)

Where w- No of rows in the input matrix.

h- No of columns in the input matrix.

If we arrange the Input in a square matrix then number of rows will be equal to the number of columns i.e. w=h.

On putting w=h in equation (1) we get

Total time complexity = 0(hlgh +h.h)

$= 0(hlgh + h^2)$

$= 0(h^2)$

If we compare the above calculated time complexity with the other standard sorting algorithm on single CPU say quick sort. We can see for sorting n elements on a single CPU by quick sort time complexity will be 0(nlgn). If we want to calculate the speed up factor with respect to quick sort speed up will be given by

Speed up=Time complexity of Quick sort / Time complexity of GPU matrix sort

$= nlgn / h^2$_____eqn (2)

Here n is the total number of items in input array. In case of matrix arrangement n can be written as n= Number of rows × Number of columns. For similar matrix, Number of rows = Number of columns

Therefore $n= h\times h= h^2$

On putting the value of n in equation (2) we get

Speed up = $h^2 \lg_2 h / h^2$

$= \lg_2 h^2$

Therefore Speed up= $2\times \lg_2 h$ ---------- eqn (3).

If we take n= 1024

Then h=32 (n=h×h)

Put the value of h in equation 3

Speed up= $2 \times \lg_2 32$

Speed up=10.

Therefore the GPU matrix sort will be 10 times faster than quick sort on single CPU if input size is 1k and it will increase reasonably as we will increase the input size. We have done some experiments on an NVIDIA GIX 280 graphics card with 240 streaming processors.

Table 2

Input Size	w	h	Best running time
1M	32	32	18.768 ms
2M	64	32	41.861 ms
4M	64	64	93.121 ms
8 M	128	64	198.207 ms

5. CONCLUSION

This paper presents a sorting algorithm which is an efficient implementation of merge sort using GPU architecture. The implementation is simple and straightforward, it arranges the input array to be sorted into a 2D matrix. Each row of input matrix can be sorted by quick sort simultaneously by assigning each row a different thread. By this way We have sorted every row of input matrix in same time span after We have applied merging of individual rows in sorted manner. By this way the time complexity of the sorting procedure comes as O (h2) where h is the number of rows in input matrix.

6. REFERENCES

[1] T.H. Coremen, C.E. Leiserson, R.L Rivest and C Stein, Introduction to algorithms, second Edition. The MIT Press and McGrawHill Book Company, 2001.

[2] D.E. Knuth, Art of Computer Programming, Volume 3: Sorting and Searching. Addison-wesely Professional, second ed., April 1998.

[3] OpenGL- The Industry Standard for High Performance Graphics. http://www.opengl.org.

[4] Microsoft's DirectXdeveloper site: http://msdnmicrosoft.com/directx.

[5] NVIDIA Compute Unified Device Architecture (CUDA) Toolkit, version3.2. http://www.developer.nvidia.com/object/cuda-3-2-downloads.html.

[6] D.J. Evans and R.C. Dumbar. The parallel Quick sort algorithm Part 1. Run time analysis. Int J. Compute Math, 12:19-55, 1982.

[7] D Cederman and P Tsigas "GPU- Quicksort: A practical quick sort algorithm for graphics processor". J. Exp Algorithms, vol. 14 PP.1.4.-1.24-2009.

[8] N Satish, M Harris and M. Garland "Designing efficient sorting algorithms for manycore GPU's" in 23rd IEEE International Symposium on Parallel and Distributed Processing 1P.1-10-2009.

[9] M Harris, S Sengupta and JD Owens, "Parallel Prefix sum (scan) with CUDA" in GPU Gems 3(H. Naguyen, ed), Addison Wesley, August 2007.

[10] Timothly J. Prucell Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pal Hanrahan. Photon Mapping on Programmable Graphics Hardware Pages 41-50. Eurographics Association 2003.

[11] Ujval J Kapasi, William J Dally, Scott Rixner, Peter R Mattson, John D Owens, Operations for data –parallel architecture.

[12] Erik Sintorn and Ulf Assarsson. Fast parallel GPU sorting using a Hybrid algorithm. In workshop on General purpose processing on Graphics processing units 2007.