International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 07 | July 2021 www.irjet.net p-ISSN: 2395-0072

A GPU Parallel Implementation of Bitonic Sort using CUDA

Saurabh Rudrawar[1], Dinesh Hebare[2], Atharva Pophale[3] and Mohit Lokhande[4]

Abstract - *Modern GPUs (Graphics Processing Unit) have evolved as massively parallel processors and can also be used for general purpose computing. GPUs provide high computational power with less cost and less power consumption and thus is a popular computing alternative. In this paper, we are presenting a GPU [5] parallel method of sorting elements of a large array using Bi-tonic [12] sort technique and comparing the serial and parallel execution method performance. The parallel implementation yields maximum speed-up of 192 and 99% gain in time on NVIDIA's single Tesla K20 GPU, for an array of four million integer elements. We conclude from the work performed for sorting using GPU's that for larger datasets, sorting can be done in an efficient manner by launching millions of parallel threads on modern GPUs.*

Key Words: **Parallel processing, parallel programming, sorting, graphics processing unit, CUDA**

1.INTRODUCTION

Sorting is none other than arranging data in ascending or descending order. Sorting is important for retrieving the required data efficiently. There are many applications in our real life that require searching, especially scenarios like searching a particular record in the database, roll numbers in the merit list, a particular contact in the mobile phone or in a telephone directory, a particular page in the book etc. All of these applications demand performance efficiency which cannot be achieved if the data to be processed is unordered and unsorted, but fortunately, the concept of sorting data makes it easier for everyone to arrange data in an order and making it easier to search. The importance of sorting lies in the fact that data searching can be optimized to significant level if data is stored in a sorted manner. Sorting is also used to change the format of data and represent it in more readable formats.

1.1 Classification of sorting techniques

There are a variety of different techniques available for sorting which are differentiated by their efficiency and space requirements. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. Sorting algorithms can be classified based on multiple factors such as following-

1) Computational complexity in terms of the number of swaps. Sorting methods perform various numbers of swaps in order to sort a data.
2) System complexity of computation. In this case, each method of sorting algorithm has different cases of performance. They are the worst case, when the integers are not in order and they have to be swapped at least once. The term Best Case is used to describe the way an algorithm behaves under optimal conditions.
3) Memory usage and other computer resources is also a factor in classifying the sorting algorithms.
4) Recursion is applied in some algorithms while others may be non-recursive.

1.2 Categories of sorting algorithms

Some sorting algorithms require some amount of extra-space or temporary storage for comparison of data elements and use this storage extensively. This is termed as out-of-place sorting. Some algorithms do not require any extra space and perform sorting within the array itself. This is termed as in-place sorting. Bubble sort is an example of in-place sorting. On the other hand, in the merge sort algorithm it is necessary to allocate extra space which is more than or equal to the number of elements being sorted for comparison.

In the stable sorting, after sorting the contents, the relative order of equal elements is preserved. In the unstable sorting, after sorting the contents the relative order is changed. In an adaptive sorting algorithm, while performing sorting, if the source list already contains some elements in the desired sequence, it will take this into account and will try not to re-order them. It takes advantage of already 'sorted' elements in the list that is to be sorted.

A non-adaptive algorithm is the one which does not perform any operations on the elements which are already sorted. They try to force every single element to be re-ordered to determine whether they are sorted or not.

1.3. Sequential Execution complexities

Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm(N)[1]. Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input(N) to the algorithm.

Table -1: TIME AND SPACE COMPLEXITIES OF SORTING ALGORITHMS

Algorithm	Time Complexity			Space Complexity Worst
	Best	Average	Worst	
Quicksort	Ω(n log(n))	Θ(n log(n))	$O(n^2)$	O(log(n))
Mergesort	Ω(n log(n))	Θ (n log(n))	O(n log(n))	O(n)

Heapsort	Ω(n log(n))	Θ (n log(n))	O (n log(n))	O(1)
Bubble Sort	Ω(n)	$\Theta(n^2)$	$O(n^2)$	O(1)
Insertion Sort	Ω(n)	$\Theta(n^2)$	$O(n^2)$	O(1)
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	$O(n^2)$	O(n)
Shell Sort	Ω(n log(n))	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	$O(n^2)$	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)

As per table 1, the sequential time complexities for sorting algorithms can be quadratic in worst case scenarios. For sorting algorithms where the number of elements is more than a million, computational power is limited and this results in prolonged execution time.

2. GPU COMPUTING

Many applications (data parallel) executed on the CPU can be accelerated by computing the parallelism supportive and time-consuming portions of the code on GPU. The rest of the application code still executes on the CPU. From the user's perspective, the application executes faster because it's using the massively parallel processing power of the GPU to boost performance which is known as heterogeneous or hybrid computing [5]. Nowadays NVIDIA GPUs has many core chips developed around an array of parallel processor [7]. A kernel performs the scalar sequential code across a set of parallel threads. These threads are organized as thread blocks, the kernel consists of a grid of one or more thread blocks. Execution of these thread blocks takes place on an array of SMs. The Hardware performs all thread management operations like creation and scheduling.

To efficiently manage a large population of threads Tesla SM has SIMT (Single Instruction and Multiple Threads) architecture [7,8]. Threads are executed in groups of 32 called warps. The threads of warp are executed on separate SP which consists on single multi-threaded instruction unit. The warp threads can load and store any valid address, supporting gather and scatter access to memory. When threads access consecutive words in memory, coalesce accesses result in high memory throughput.

2.1. CUDA Program

A CUDA program dissolves itself into one or more phases which are executed on either CPU i.e serial or GPU i.e parallel [4] systems. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit the very high amount of data parallelism are implemented in the device code. A CUDA program is a unified source code encompassing both host and device code. The compiler (nvcc) separates the two during the compilation process. The CPU (serial) code is straight ANSI C code; it is further compiled with the CPU's standard C compilers and runs as an ordinary CPU process[8]. The device code comprises of ANSI C code with some extended key-words for kernels, data structures associated with it. The device code is consistently further compiled by the nvcc compiler and executed on a CUDA supported GPU device.

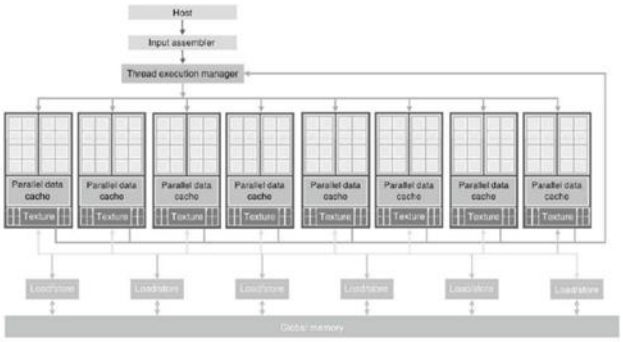

Fig -1: Architecture of CUDA Capable GPU [4]

From [6], CUDA includes such a programming model along with hardware support that facilitates parallel implementation. The number of times the speedup is obtained for an application by using parallelism de-pends on the portion of the application that can be Parallelized. CUDA works in terms of threads (Single Instruction Multiple Thread). Threads are the smallest unit of processing within a program. Threads execute independently of each other unless explicitly synchronized (or part of some warp). Each thread has access to global memory and its shared memory (within a block).

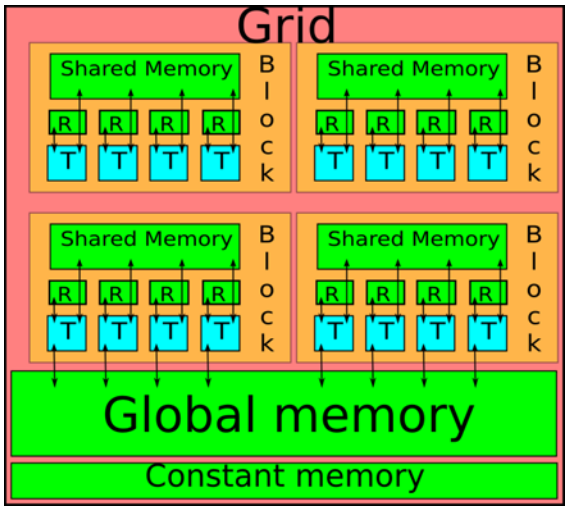

Fig -2: Conceptual Memory Diagram [6]

3. BI-TONIC SORT

Bitonic sort falls into group of sorting networks which means that the sequence and direction of comparisons are known in advanced irrespective of the input se-quence. It is based on Bitonic sequence [9,10,11]. Bitonic sort does O(n(〖log〗^2n)) comparisons given in "Ref. [12]". The no. of comparisons done by bitonic sort are more than other sorting algorithms such as merge sort but its effective for parallel implementation. A sequence is called Bi-tonic sequence if it is increasing up to an element and then decreasing i.e. for an array a[0-(n-1)] if there exists an index i such that –

$X_0<=X_1<=X_2 \ldots\ldots X_i$ and $X_i>=X_{(i+1)}>=X_{(i+2)} \ldots\ldots X_{(n-1)}$

An increasing sequence with decreasing part empty as well as a decreasing sequence with increasing part empty is considered as bi-tonic sequence. Forming a bi-tonic sequence for random input sizes. We consider consecutive two element sequences and form a 4-element sequence from it i.e. consider 4 element sequences with X0,X1,X2,X3 as its elements then we sort X0,X1 in increasing order and X2,X3 in decreasing order. We concatenate them to form a 4-element bi-tonic sequence as shown in "Ref. [12]" then we take two 4 elements bi-tonic sort and sort one in increasing order while the other in decreasing order.

Following is the bitonic sorting network for 16 elements of a random sequence-

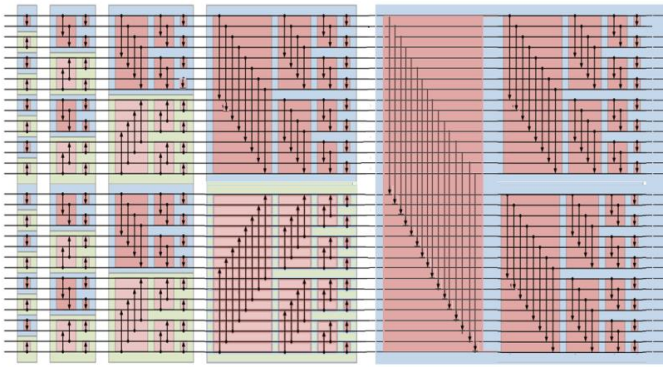

Fig -3: Bi-tonic sorting network for 16 elements

Here the blue area depicts the elements comparisons in increasing order and the green area depicts the elements comparisons in decreasing order.

In the diagram above downward moving, arrows per-form a minimum comparison operation between the element representing at the start of arrow and element pointed towards the end of the arrow. For minimum comparison operation if the element at the start of the arrow is greater than element at end of arrow then we swap both elements and if the element at the start of the arrow is lesser than element at end of arrow then we keep both elements as it is. Similarly, in the diagram above upward moving arrows performs a maximum comparison operation between element representing at the start of arrow and element pointed towards the end of the arrow. For maximum comparison operation if the element at the start of the arrow is greater than element at end of arrow then we swap both elements and if the element at the start of the arrow is lesser than element at end of arrow then we keep both elements in same order.

3.1 Bi-tonic sort implementation

For every block, input is in 2^kformat. In our Multi-Block sorting algorithm using Bi-tonic sort, there is one thing common for every stage that is we always get same number of Bi-tonic sequences after every stage. Before the first stage we have 'n' Bi-tonic sequences for every block where 'n' is total number of elements in one block. For every stage n/(2) threads are qualified to perform the specified operation. After first stage we get n/2 Bi-tonic sequences for every block. In the similar way we get n/4,n/8,….(n)/n Bi-tonic sequences after performing every stage and at the last stage it will give us complete sorted sequence. For having every single block sorted by the above discussed method, we have used two decision parameters.

1) First parameter is used to determine which threads are qualified to be used for performing specified operations.
2) Second decision parameter is used for determining whether to perform minimum comparison or maximum comparison between numbers representing qualified thread and numbers at position (i+x)/2 where 'i' is Thread ID of qualified thread and 'x' is size parameter which is passed to Kernel as function as an argument. 'X 'starts from 2 and increases exponentially in powers of 2 that is 2,4,8,16, ... ,n.

3.2 Algorithm

In the proposed implementation of algorithm, we make a kernel call for the respective size and we loop it accordingly. Here 'y' and 'z' are the decision parameters i.e thread qualifier and min/max differentiator. At any stage no. of threads qualifying will be n/2. Additionally, we make use of shared memory for faster memory accesses.

Below is the implementation for kernel function:

Kernel Call:

```
for(l=2; l<=2k; l*=2)
{
  for(x=l; x>=2; x/=2)
  {       //calling bitonic kernel
          bitonic_k<<<blocks,blocksize>>>(d_input,l,x);
  }
cudaDeviceSynchronize();
}
```

Kernel –

```
y = k%p;      //for thread qualification
z = k%(2*size);    //for min or max condition
```

```
if(y<p/2)
{
    if(z<size)//min condition(up->down)
    {
     if(d_input[k]>d_input[k+p/2])
     {
      temp = d_input[k];
      d_input[k] = d_input[k+p/2];
      d_input[k+p/2] = temp;
     }
    }
    else     //max condition(down->up)
    {
    if(d_input[k] <d_input[k+p/2])
    {
      temp = d_input[k+p/2];
      d_input[k+p/2]=d_input[k];
      d_input[k] = temp;
    }

  }//end else
}// end if
```

For complete Bi-tonic sort, we removed for loop in kernel function because of irregular data syncing and nested the loops for calling kernel function. Here outer loop will iterate till total size i.e. 2k starting from size 2 and after every iteration it exponentially increases in powers of 2. Unlike block-wise sorting method here the loop will take values till it encompasses total number of inputs. Such that we will receive whole array or input vector in desired sorted state. Swapping of two numbers and obtaining Bi-tonic sequence operations are similar to those that are explained in block-wise sorting earlier in this paper.

3. TESTING ENVIRONMENT

All the operations are implemented on Intel Haswell E5-2620V3 Six core/2.4 GHz/15MB Cache 2*16 GB DDR4 RAM, NVIDIA'S Tesla K20,2496 CORES,5GB Memory with performance of 3.52 TFLOPS(Single preci-sion) and 1.17 TFLOPS(Double precision).

4. RESULTS AND ANALYSIS

Serial and parallel implementations of-tonic sort were executed for different vector sizes and block sizes. The comparisons are as shown in following tables. We compute speedup as difference between serial execution and parallel execution divided by serial execution.

Input size	Speed up	%gain
2^{17}	139.53	99.28
2^{18}	168.11	99.40
2^{19}	153.28	99.34
2^{20}	174.29	99.42
2^{21}	183.39	99.45
2^{22}	192.72	99.48

Table 2. Speed up & %gain for 128 block size.

Input size	Speed up	%gain
2^{17}	138.56	99.28
2^{18}	164.38	99.39
2^{19}	152.24	99.34
2^{20}	171.80	99.99
2^{21}	180.90	99.99
2^{22}	189.96	99.47

Table 3. Speed up & %gain for 256 block size.

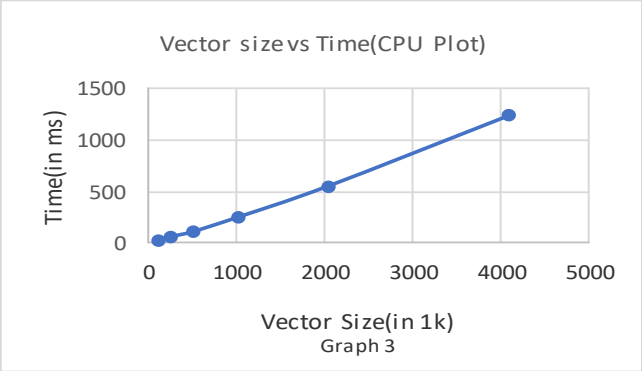

Chart -1: CPU plot for input size vs time required

Based on empirical analysis, we have executed the parallel implementations for block size 64, 128, 256 and 512. The best results were obtained for block size 128. The performance impact of block size on code to be executed depends on code and the hardware platform that is used. As the block sizes are finite, the best configuration for the code is relatively easy to find. We experimented the parallel implementations for input data up to 4 million and achieved greater speedup for large input vectors.

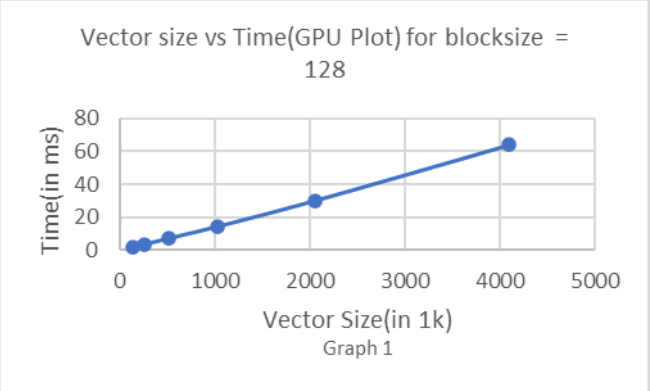

Chart -2: GPU plot for input size vs time required

5. CONCLUSION

After performing parallel computations on varied sizes of input, we conclude that using GPUs for sorting is a viable and

efficient alternative to conventional computation methods. Based on the performance of serial and parallel implementations of the Bi-tonic sort we conclude that significant increase in speedup is obtained which increases proportionally to N (vector size). Furthermore, we suggest an analogous parallel implementation of merge sort algorithm due to the parallel nature of divide and conquer algorithm.

REFERENCES

[1] Data structures and algorithms, introduction to sorting: https://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson8_1.htm.

[2] Data structure and algorithms and sorting techniques: https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorihms.htm.

[3] Array sorting algorithms: http://www.bigocheatsheet.com/.

[4] Programming massively parallel processors by David B kirk and Wen-meiW.hwu.

[5] CUDA by Example, Jason Sanders and Edward Kandrot (2010).

[6] Introduction to GPU Hardware and CUDA, University of Cambridge, PhilipBlakely: https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&ved=0ahUKEwjGkLCbs63aAhUFR48KHVw9AdsQFghxMAc&url=http%3A%2F%2Fpeople.ds.cam.ac.uk%2Fpmb39%2FGPULectures%2FLecture_1.pdf&usg=AOvVaw1bzMxlXsOoST_bHkjBBlpY.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym.NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28(2):39–55, Mar/Apr 2008.

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. Queue, 6(2):40–53,Mar/Apr 2008.

[9] Batcher KE. Sorting networks and their applications. Proceedings of the April 30–May 2,1968, Spring Joint Computer Conference, AFIPS '68

[10] (Spring),ACM:NewYork,NY,USA,1968;307314,doi:10.1145/1468075.1468121.URL http://doi.acm.org/10.1145/1468075.1468121.

[11] Peters H, Schulz-Hildebrandt O, Luttenberger N. Fast in-place, comparison-based sorting with CUDA: A study with bitonic sort. Concurr. Comput. :Pract. Exper. May 2011;23(7):681–693, doi:10.1002/cpe.1686. URL http://dx.doi.org/10.1002/cpe.1686.

[12] Peters H, Schulz-Hildebrandt O, Luttenberger N. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012, 2012;227–237, doi:10.1109/IPDPS.2012.30. URL http://dx.doi.org/10.1109/IPDPS.2012.30

[13] Bitonic sort and sequence: https://www.geeksforgeeks.org/bitonic-sort.