An Efficient Parallel Implementation of CPU Scheduling Algorithms Using Data Parallel Algorithms

Suvigya Agrawal, Aishwarya Yadav, Disha Parwani and Veena Mayya

Abstract Modern graphics processors provide high processing power, and furthermore, frameworks like CUDA increase their usability as high-performance co-processors for general-purpose computing. The Graphical Processing Units (GPUs) can be easily programmed using CUDA. This paper presents an efficient parallel implementation of CPU scheduling algorithms on modern The Graphical Processing Units (GPUs). The proposed method achieves high speed by efficiently exploiting the data parallelism computing of the The Graphical Processing Units (GPUs).

1 Introduction

The modern GPU is a many-core processor which supports execution of thousands of threads concurrently. GPU comprises of a series of streaming processors with hundreds of core aligned in a particular way which facilitate single instruction multiple threads (SIMTs) programming model.

General-purpose computing on GPU is a graphical processing unit which is very efficient at computer graphics manipulation and image processing. The highly parallel structure of GPU makes it easy to use to perform the general-purpose computation and accelerate traditional CPU-based computational tasks. Recently, general-purpose

S. Agrawal (✉) · A. Yadav · D. Parwani · V. Mayya
Department of Information and Communication Technology, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal 576104, India
e-mail: suvigya1st@gmail.com

A. Yadav
e-mail: aishwaryayadav91@yahoo.com

D. Parwani
e-mail: disha.parwani9927@gmail.com

V. Mayya
e-mail: veena.mayya@manipal.edu

R. Kamal et al. (eds.), *International Conference on Advanced Computing Networking and Informatics*, Advances in Intelligent Systems and Computing 870,
https://doi.org/10.1007/978-981-13-2673-8_45

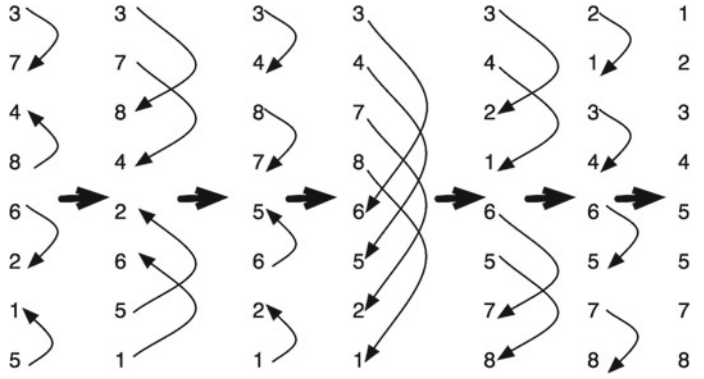

Fig. 1 Example of bitonic parallel sort algorithm

computation on graphics processing units (GPGPU) has been adopted in accelerating many algorithms such as sorting [11], graph algorithms [12], data encryption algorithms [5], and other generic algorithms [13]. Several libraries and programming environment are available that allow programmers to perform GPGPU and exploit computational power of GPU. Compute Unified Device Architecture (CUDA) [8] framework is one of the programming environments provided by NVIDIA that allows to exploit data parallelism of GPU using C-style programming language.

Scheduling is a way by which work specified by some means is assigned to resources that complete the work. Job/process scheduling is the process of arranging, controlling, and optimizing the allocation of system resources to threads, processes, and data flows for maximum utilization. The operating system schedules the processes for execution using several scheduling algorithms based on various scheduling criteria such as CPU utilization, throughput, turnaround time, waiting time, and priority. System resources can be equally and effectively utilized by properly scheduling the processes and hence achieve a target quality of service. The major scheduling algorithms types of CPU/job scheduling algorithms include as follows: First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority-Based Scheduling (PBS). Scheduling is required to perform multitasking and multiplexing. Scheduling is a complex job requiring extensive processing which is better performed on a parallel platform.

In this paper, a novel parallel approach to perform scheduling using CUDA technology to enhance the performance of process scheduling algorithms is proposed. A combination of well-established data parallel algorithms and parallel sorting techniques has been adopted to achieve drastic performance increase in execution time of scheduling algorithms (Fig. 1).

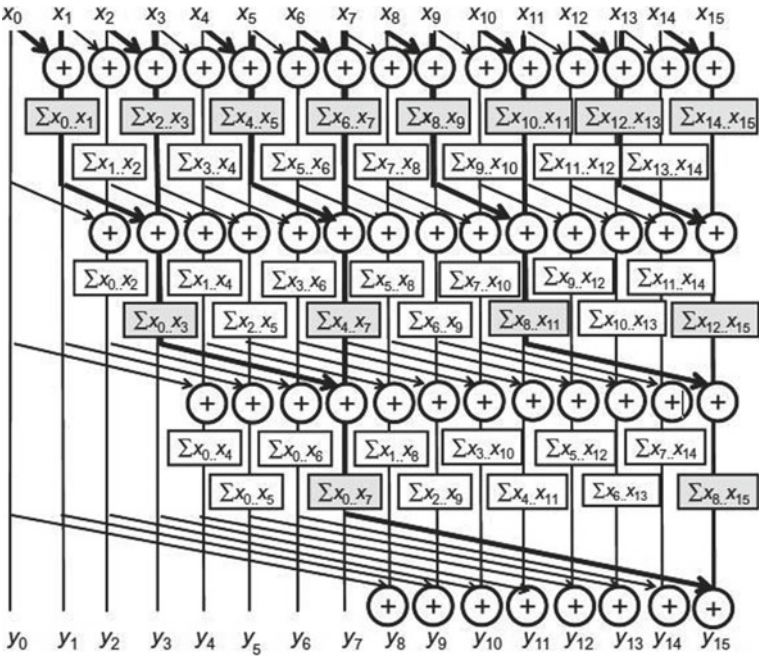

Fig. 2 Prefix sum data parallel algorithm

1.1 Data Parallel Algorithms

In data parallel algorithms, parallelism is involved in simultaneous operations across large sets of data, rather than from multiple threads of control [4]. Prefix sum or scan is one among the most common data parallelism algorithms. The prefix sum, cumulative sum, inclusive scan, or simply scan of a sequence of numbers $x_0, x_1, x_2, \ldots x_{n-1}$ are the second sequence of numbers $y_0, y_1, y_2, \ldots y_{n-1}$, the sums of prefixes (running totals) of the input sequence where $y_0 = x_0; y_1 = x_0 \oplus x_1; y_1 = x_0 \oplus x_1 \oplus x_2 \ldots$, as shown in Fig. 2. Mathematically $y_i = x_0 \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_i \; \forall_i \in 0 \ldots n-1$. Figure 2 depicts the pictorial representation for the scan data parallel algorithm [6].

1.2 Bitonic Sort

There are multiple sorting algorithms available such as Merge sort, Quick sort, Radix sort, and Heap sort. Bitonic sort is one among the sorting algorithms that can make use of GPU computing power and thus is efficient in terms of both space and time complexities [7].

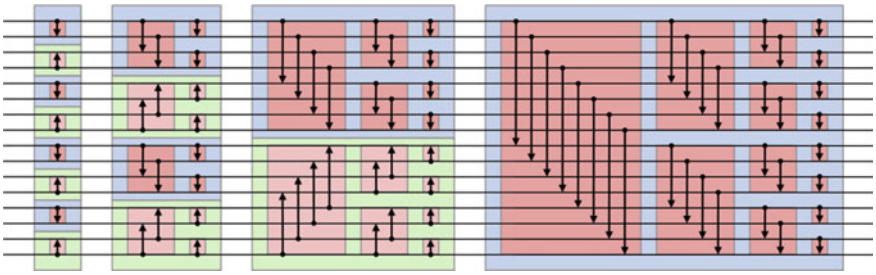

Fig. 3 Representation of bitonic sort

Bitonic Sequence: A sequence is called Bitonic if it is first increasing, then decreasing. In other words, an array arr[0..n − 1] is Bitonic if there exists an index i where $0 \leq i \leq n-1$ such that,

$$a_0 \leq a_1 \leq \cdots \leq a_{n/2-1} \, and \, a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$$

The Bitonic Sort Algorithm (as illustrated in Fig. 1):

(1) Let s = $\langle a_0, a_1, a_0, \ldots, a_{n-1} \rangle$ be a bitonic sequence such that

   (a) $a_0 \leq a_1 \leq \ldots \leq a_{n/2-1}$, and
   (b) $a_{n/2} \geq a_{n/2+1} \geq \ldots \geq a_{n-1}$

(2) Consider the following subsequences of s

   (a) $s_1 = \langle min(a_0, a_{n/2}), min(a_1, a_{n/2+1}), \ldots, min(a_{n/2-1}, a_{n-1}) \rangle$
   (b) $s_2 = \langle max(a_0, a_{n/2}), max(a_1, a_{n/2+1}), \ldots, min(a_{n/2-1}, a_{n-1}) \rangle$

(3) Sequence properties

   (a) $s_1 \, and \, s_2 \, are \, both \, bitonic$
   (b) $\forall x \forall y \, x \in s_1, y \in s_2, x < y$

(4) Apply recursively on $s_1 \, and \, s_2$ to produce a sorted sequence
(5) Works for any bitonic sequence, even if $|s_1| \neq |s_2|$.

Given an unordered sequence of size $2n$, exactly $log_2 2n$ stages of merging are required to produce a completely ordered list (Fig. 3).

1.3 Prefix Sum (Scan)

Prefix sum of a sequence of numbers $x_0, x_1, x_2, \ldots, x_n$ is another sequence of numbers $y_0, y_1, y_2, \ldots, y_n$ given by:

$$y_0 = x_0; y_1 = x_0 \oplus x_1; y_1 = x_0 \oplus x_1 \oplus x_2$$

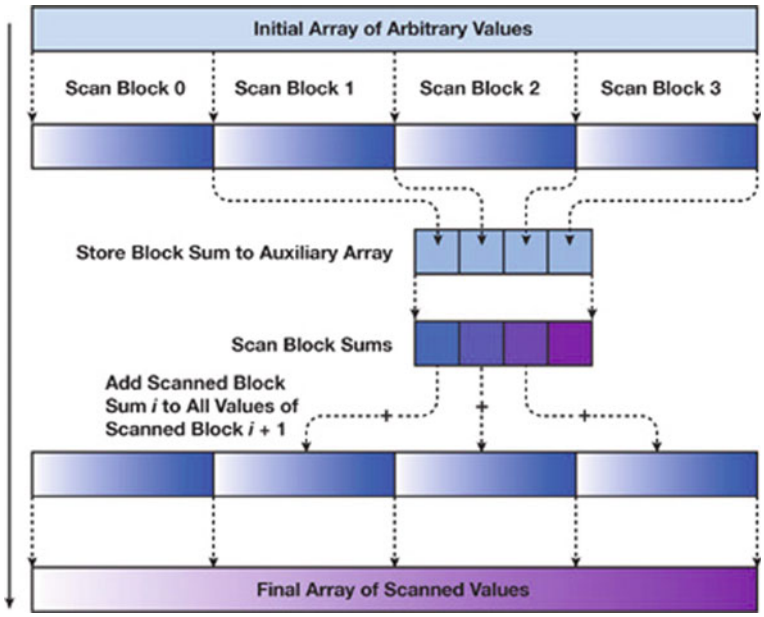

Fig. 4 Block-wise data parallel scan operation

The general mathematical representation of inclusive prefix sum for computing the output value in a sequential order is:

$$y_i = x_i; \quad i = 0$$
$$y_i = y_{i-1} \oplus x_i; \quad i > 0$$

The given data set is divided into blocks of fixed size. The scan is performed individually on each block. The result obtained is not the final result. It is a temporary array, as second block does not counter for first block elements and similarly henceforth. Each block sum is the last element of that block. The concept used first involves extracting the last element of the block that will give the individual block sum and storing these in a separate array let's say Y in the corresponding positions, i.e., block 0 sum stored at zeroth index and block 1 sum stored in first index. The final result is obtained by adding the elements in array Y to the corresponding temporary array elements by using indices of block and the thread. For instance, the block 0 contains final result so we do not have to perform any computation. But for block 1, Y[0] (which is nothing but the block sum of block 0) is to be added to each and every element of block 1 present in the temporary array. For block 2, Y[0] + Y[1] is added to every element, which is nothing but the block sum of zeroth and first block, respectively, as shown in Fig. 4, similarly for further blocks. Finally, the resultant array is obtained.

2 Related Work

Job/process scheduling is one of the important tasks performed by the operating system (OS). The performance of the OS depends on the CPU scheduling algorithms. Recently, several improved CPU scheduling algorithms such as [2, 9, 10] have been introduced for improving the system performance. Scheduling needs to be carried out very frequently by the operating system and is a complex job which may require repetitive computation. State-of-the-art performance is achieved by implementing many generic algorithms [1, 3] on GPU. This motivates to implement scheduling algorithms on GPU and analyze the performance.

3 Implementation

Algorithm 1 provides the steps carried out to implement the parallel non-preemptive scheduling algorithm.

Algorithm 1 Algorithm for parallel scheduling using data parallel algorithm

```
1: procedure MainModule
2:     Allocate memory for the input using cudaMallocManaged
3:     Read and store the input priority, burst time, arrival
       time in the above allocated variables
4:     Launch the SORT kernel as depicted in Algorithm 2.
5:     Perform block  SCAN kernel  as depicted in Algorithm 4
       with required parameters.
6:     Launch the Reduction kernel with required parameters
       to find the average waiting time and turnaround time
```

Algorithm 2 Algorithm for Bitonic Sort kernel launch

```
1: procedure SortingFirst (int *pr, int *bt)
2:     dim3 blocks(BLOCKS,1);
3:     dim3 threads(THREADS,1);
4:     for (int k = 2; k <= NUM; k <= 1) do
5:         for (int j = k >>1; j >0; j = j >>1) do
6:             BitonicSortStep <<<blocks,threads>>>(pr,j,k);
7:     //The kernel is shown in Algorithm 3.
```

Algorithm 3 Main Algorithm of Bitonic Sort

```
1: procedure BitonicSortStep(float *values, int j, int k)
2:     unsigned int i, ixj;
3:     i = threadIdx.x + blockDim.x * blockIdx.x;
4:     ixj = i ^ j;
5:     if ((ixj)>i) then
6:        if ((i & k )==0) then
7:           if (values[i] > values[ixj]) then
8:              float temp = values[i];
9:              values[i] = values[ixj];
10:             values[ixj] = temp;
11:       if ((i & k)!=0) then
12:          if (values[i] < values[ixj]) then
13:             float temp = values[i];
14:             values[i] = values[ixj];
15:             values[ixj] = temp;
```

To form a bitonic sequence from a random input, we start by forming four-element bitonic sequences from consecutive two-element sequence. Consider four-element in sequence x_0, x_1, x_2, x_3. We sort x_0 and x_1 in ascending order and x_2 and x_3 in descending order. We then concatenate the two pairs to form a four-element bitonic sequence. Next, we take two four-element bitonic sequences, sorting one in ascending order, the other in descending order (using the bitonic sort which we will discuss below), and so on, until we obtain the bitonic sequence as shown in Fig. 3.

Algorithm 4 Algorithm for Scan kernel

```
1: procedure BlockSum(x)
2:     n = length(x)
3:     y = fill(x[1],n)
4:     for i = 2 : n do
5:        xy[i] = xy[i-1] + x[i]
6:     Wait for all threads in a block to finish
7:     Copy xy into y
8:     Wait for all threads of all block to finish
9:     if (threadIdx.x <blockIdx.x) then
10:       Extract last element of every block and put in xy.
11:    Wait for all the threads to finish.
12:    for i = 0 : blockIdx.x do
13:       Add the block sums to all the elements of that
          block from xy.
```

The experiment done was on analyzing the performance of the Priority-Based Scheduling (PBS) algorithm. A large number of processes are taken which include parameters such as burst time, arrival time, and priority. The processes are sorted based on the priority first, and then, the tie is broken between the processes having same priority depending upon their burst time. The scheduling is performed placing the processes with the highest priority and lowest burst time first (the highest priority corresponds to lowest number). The input elements are sorted using bitonic sort. Further, the block-wise work-efficient scan operation is performed to obtain the waiting time and turnaround time as shown in Fig. 4. Data parallel reduction algorithm is applied to sum waiting time and turnaround time and thus compute average waiting time and turnaround time. The scan algorithm iterates $log(n)$ time.

4 Results

The proposed method uses parallel bitonic sort, and the computation of this sorting has a complexity of $O((\log N)^2)$ that makes it n times faster than its serial complexity $O(N\ (\log N)^2)$. Param Shavak with Kepler GTX GPU card is used to analyze the proposed method. Screenshots of execution are shown in Figs. 5 and 6 that depict the time taken to execute serial and parallel scheduling code, respectively. Figure 7 shows the graphical representation for the same. It can be seen that execution speed of parallel code is almost 10–15 times more than that of serial code.

```
Serial Computation Elapsed time: 0.000263s for 256 process
Serial Computation Elapsed time: 0.000921s for 512 process
Serial Computation Elapsed time: 0.003582s for 1024 process
Serial Computation Elapsed time: 0.013594s for 2048 process
Serial Computation Elapsed time: 0.044259s for 4096 process
Serial Computation Elapsed time: 0.121259s for 8192 process
```

Fig. 5 Screenshot of serial execution of the PBS

```
Parallel Computation Elapsed time: 0.000489s for 256 process
Parallel Computation Elapsed time: 0.000557s for 512 process
Parallel Computation Elapsed time: 0.000615s for 1024 process
Parallel Computation Elapsed time: 0.000730s for 2048 process
Parallel Computation Elapsed time: 0.000890s for 4096 process
Parallel Computation Elapsed time: 0.001073s for 8192 process
```

Fig. 6 Screenshot of parallel execution of the proposed method

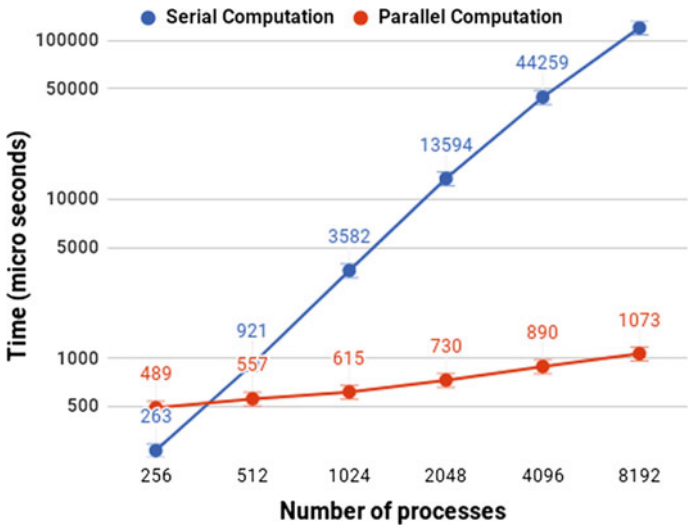

Fig. 7 Graphical representation of the comparison of serial and parallel scheduling algorithms

5 Conclusion

The paper focuses on solving a priority-based algorithm using bitonic sort and the prefix scan, where all the codes are implemented in parallel and executed on GPU, to enable faster execution of the problem.

The entire task is broadly classified into two sub-tasks that are as follows:

1. Sorting the processes based on priority first and then on burst time (execution time) using the parallel code for the bitonic sort by launching multiple kernels.
2. Priority-Based Scheduling is implemented by using the parallel code for prefix scan which again has two stages:

    a. Data that are divided into blocks are first put through initial prefix scan which results in scanned results, but block-wise.
    b. The second stage involves the different blocks to encounter for all the elements that are present in the block previous to it.

The prefix scan is again done by launching multiple kernels.

It is observed that when the number of processes to be scheduled increases, the amount of the time taken for the CPU to schedule these processes also increases drastically. But, however, in the case, when there is an increase in the number of processes that are to be scheduled on the GPU, the amount of the time taken by it to schedule these processes increases but by a relatively very less value.

In other words, the proposed parallel implementation is 10–15 times faster than the serial implementation. Future work involves analyzing the proposed method for non-preemptive scheduling algorithms.

References

1. J.P. Arun, M. Mishra, S.V. Subramaniam, Parallel implementation of MOPSO on GPU using OpenCL and CUDA, in *2011 18th International Conference on High Performance Computing (HiPC)*, pp. 1–10
2. N. Goel, R.B. Garg, Simulation of an optimum multilevel dynamic round robin scheduling algorithm (2013), http://arxiv.org/abs/1309.3096
3. P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in *Proceedings of the 14th International Conference on High Performance Computing*, p. 197, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1782174.1782200
4. W.D. Hillis, G.L. Steele Jr., Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986), http://doi.acm.org/10.1145/7902.7903
5. H. Jo, S.T. Hong, J.W. Chang, D.H. Choi, Data encryption on GPU for high-performance database systems. Procedia Comput. Sci. **19**(Supplement C), 147–154 (2013), http://www.sciencedirect.com/science/article/pii/S1877050913006327, The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), The 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013)
6. D.B. Kirk, W.M.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd edn. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013)
7. Q. Mu, L. Cui, Y. Song, The implementation and optimization of Bitonic sort algorithm based on CUDA, (2015), http://arxiv.org/abs/1506.01446
8. J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda. Queue **6**(2), 40–53 (2008), http://doi.acm.org/10.1145/1365490.1365500
9. A. Pandey, P. Singh, N.H. Gebreegziabher, A. Kemal, Chronically evaluated highest instantaneous priority next: a novel algorithm for processor scheduling. J. Comput. Commun. **4**, 146–159 (2016), http://www.scirp.org/JOURNAL/PaperInformation.aspx?PaperID=65949
10. H.B. Parekh, S. Chaudhari, Improved round robin CPU scheduling algorithm: round robin, shortest job first and priority algorithm coupled to increase throughput and decrease waiting time and turnaround time, in *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, pp. 184–187, Dec 2016
11. 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* (*IPDPS'09*) (IEEE Computer Society, Washington, DC, USA 2009), pp. 1–10, https://doi.org/10.1109/IPDPS.2009.5161005
12. P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, A. Lumsdaine, Dynamic parallelism for simple and efficient GPU graph algorithms, in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (*IA3'15*) (ACM, New York, NY, USA, 2015), pp. 11:1–11:4, http://doi.acm.org/10.1145/2833179.2833189
13. Y. Zhang, J.D. Owens, A quantitative performance analysis model for GPU architectures, in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture* (HPCA'11) (IEEE Computer Society, Washington, DC, USA, 2011), p. 382, http://dl.acm.org/citation.cfm?id=2014698.2014875