



## Analysis of Parallel and Sequential Radix Sort for Graph Exploration using OpenMP and CUDA: A Review

<sup>1</sup>Prince Sharma\*, <sup>2</sup>Shailendra Shukla

<sup>1</sup>Jaypee University of Information Technology, Solan, HP, India

<sup>2</sup>Department of Computer Science and Engineering, Jaypee University of Information Technology, Solan, HP, India

---

**Abstract:** *In this paper we have analyzed the comparison of radix sort algorithm on sequential and parallel procedures across three programming language platforms namely C, OpenMP based C++ and CUDA programming fixtures. The importance of radix sort to be a non comparison based sorting algorithm has been truncated. The algorithmic flow of data from the unsorted input to the desired output has also been shown using flowchart. Next to it, the theoretical complexity for radix sort is being illustrated and experimented for varied datasets of non-fixed sizes with varied bucket size ranges. Towards the end, the algorithmic complexities for radix sort have also been evaluated for the three algorithmic procedures used. The comparative efficiencies of radix sort for sorting a list of integers of different sizes and ranges have been evaluated. On the basis of analysis we have concluded the CUDA program to be approximately 100 times faster than the sequential C program and the multi-threaded parallel program enabled by OpenMP in C++ to be approximately 10 times faster than the sequential C program to sort a varied sized text file according to radix sort in an increasing order. This paper is a review of the serial and parallel algorithm implementation for sorting anonymously large dataset which can be used for graph explorations implemented over network analysis, which is also the future proposed work for our analytic study.*

### Categories and Subject Descriptors

F.2.2 [ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY]: Nonnumerical Algorithms and Problems –

- Complexity of proof procedures, Sorting and searching.

**General Terms:** Algorithms, Performance, Design, Reliability, Experimentation, Languages, Verification.

**Keywords:** Radix sort, thrust library, program complexity, threading, NVIDIA CUDA.

---

## I. INTRODUCTION

Sorting can simply be defined as a procedure of arranging a sequence of elements in some order which may be ascending or descending or in some chronological order. In respect of the field of mathematics, sorting may be described in terms of some permutation function operated on a certain set of elements. Thus there are various sorting algorithms which are taught in the sciences of mathematics and computer science. Several sorting algorithms are being discussed in the field of Computer Science which include Bubble Sort, Merge Sort, Selection Sort, Bitonic Sort and many others. The efficiency of these sorting procedures is measured by their sorting complexities, measured in terms of either the space or the time requirements of these sorting algorithms. Nowadays due to the development of heterogeneous architecture based computing processors, which include the CPUs as well as the GPUs, the need of hour is to use and implement the algorithms in an efficient manner so that their better utilization can be done. The utilization of these architectures can be done by using efficient subroutines at block level or thread level for the processes which include either direct or indirect dependence on the sorting procedures. The algorithmic improvisations in sorting thus play an important role, which render numerous set of applications based on sorting. It is because many parallel versions of previously sequential sorting algorithms have continuously been evolved. We have considered our focus on Radix Sort for this paper because of being an efficient sorting algorithm since a very past experience of history of this sorting strategy. For larger value of datasets, the radix sort is being preferred more in comparison to any other sorting procedure. The drift of serial world into parallel has led to the parallel evolution in the field of application development also. Several parallel programming languages e.g. CUDA [12], OpenMP and MPI [19] have thus gained importance due to the similar purpose, which render the better utilization of architectures of GPUs and multicore CPUs. The high performance usages of these parallel constructs lay down the usage of OpenMP [20] and CUDA initiatives [15] in the field of physics and other computational sciences [1]. Along with the usage of these techniques, continuous evolution is also under refreshment of the basic architectural models used by these architectures. The computational models for GPUs have also been proposed time and again overwhelming the drawbacks and the limitations of the previous time respective approaches [5]. Continuous work on the improvements in the design of high performance parallel sorting routines like radix sort and merge sort by using the high computing enabled by the GPUs have been in the trend [6]. There have been proposals and methods demonstrating how string sorting algorithms can be parallelized on multicore shared memory machines [9]. The transformation system from normal C to CUDA programs has also been brought into picture by few authors [10].

## **II. OVERVIEW OF APPROACH**

The performance measurement for any machine can be studied with respect to either its latency or throughput. The latency is generally observed in terms of time taken to complete the task whereas the throughput is treated in terms of the tasks completed per unit time. Meaning thereby if either of these is enhanced i.e., latency or throughput we have an efficient solution to the problem. In order to perform the power efficient high performance processor we have just two options i.e., either to minimize latency or to enhance throughput. The purpose of minimizing latency can be achieved by enhancing the clock size of the CPU and the goal of maximizing throughput can be achieved by assigning the job to GPU. The best performance can be achieved by efficiently using the CPU and GPU in an alternatively parallel routine.

### **2.1 Radix sort Complexity**

The radixsort is a classic example of non-comparison based sorting techniques[7] which succeed in sorting special kinds of key arrays faster than  $n \log n$ . Likewise the normal sorting procedures, radixsort is a special sorting with the space sorting complexity of  $\Theta((n+k)d)$ . The constant  $k$  demarcates the bucket size which also represents the size of significant figures in each number and  $n$  represents the total number of integers to be sorted in the routine.  $d$  represents the number of passes which would be utilized to perform this radixsort. Within each pass, i.e., under all  $d$  different passes, we would need to sort  $n$  numbers each time by hashing them or binning them. When we consider elements in  $k$  different bins, constant  $k$  also comes in picture and the resulting complexity changes from  $\Theta(nd)$  into  $\Theta((n+k)d)$ , where for all different bins each time,  $d$  set of numbers will be sorted.

### **2.2 Radix Sort across Different Platforms**

Our implementation of radix sort in C programming language, OpenMP construct in C++ and CUDA programming language on a specified architecture shows the differences in performance of these sorting programs with respect to sequential and parallel routines opted by our programs. The performance of these constructs has been compared in the end. The analysis shows that how a CUDA parallel program performs better than the parallel OpenMP program which is itself better than the sequential C program..

## **III. RELATED WORK**

### **3.1 OpenMP**

OpenMP [16] is basically a standard adopted by scientists which act as a programming interface for shared memory machines, which include the Symmetric Multiprocessors as well as the Non-uniform memory architecture based processors [18]. The thread level dealing of the processors is enabled by this standard. The only way to create threads in OpenMP [17] is the use of the parallel constructs. The symmetric multiprocessors are the processors which share an address space with equal access time for each processor and thus for an operating system each processor acts the same way. The non-uniform address space multiprocessors are the processors with different memory regions having different access costs. The present day machines have a basic NUMA architecture. The importance of sorting based algorithms can be well understood with the use of improvements in normal sorting routines like Bubble Sort[3], Merge Sort, Bitonic Sort and many others.

### **3.2 CUDA**

**CUDA** is a programming language developed by NVIDIA which takes the advantage of using GPU based performance improvement enabled by the NVIDIA GPUs. CUDA [12] basically is a set of associations of CPUs and GPUs which may have homogenous or heterogenous architecture. The drivers from NVIDIA, the corresponding toolkit and the SDK all together enable the program to control the flow of data to and fro in between a CPU and a GPU [13]. The CPU uses the normal code supporting the standard C compilers which act a CPU host code for the block of CUDA code framing an integrated CPU and GPU source code in C. Later the intermediary NVIDIA C compiler transfers the control to the GPU using the CUDA optimized libraries [11] or the subroutines launched by the kernel transforming the code into NVIDIA assembly language for computing purposes. The CUDA driver and the debugger profiler are thus handled by the GPU. The host-to device interaction in between the CPU and GPU is thus enabled by the CUDA programs. If we write a simple C program, doesn't matter we write a sequential code or a massively parallel code, it runs on the CPU only. So we need to write the code that has to be over run on the GPU also. CUDA programming model [14] allows us to program both the GPU and the CPU, with one program so that we may use the power of GPU in that program itself. The CUDA compiler compile our program splitted into pieces that run on CPU and GPU, and generate a code for each. The relationship in between the CPU and GPU is handled by the CPU as the incharge which sends directions to GPU to tell what to do. This enables thee normal sequential subroutines to be run on these heterogenous architectures and take full advantage of heavy memory machines with unique individual levels of memory that serve a dedicated purpose. Several sorting algorithmic improvisations to the previous architectural techniques have been proposed by different authors, the basis of whose work is basically one or the other Sorting algorithm. Quicksort improvements in CUDA has been proposed by [2] Cederman. Several GPU based sorting new algorithmic clusters have been found gaining importance like GPUMemSort[4]. The algorithm for modern GPU architecture with optimized and high performance has been proposed by Shifu Chen et al. in Fast and Flexible Sorting algorithm with CUDA[8].

## **IV. PERFORMANCE ANALYSIS**

For our analysis we have implemented the sorting algorithm on the normal gcc compiler, g++ compiler and the nvcc compiler for sequential C program, parallel threaded OpenMP based C++ program and CUDA enabled C program

respectively. The results have been evaluated on for Linux kernel 3.16 distributed by Ubuntu 14.04. The hardware used for our analysis is 3.8 GiB for memory utilized by Quad Core Intel Core i3 CPU cycling at 2.93 GHz. For the CUDA enabled program evaluation the same machine was powered up by the NVIDIA powered GPU, namely GeForce GTX 460. The gcc version 4.9.2 have been used for C program. Similarly for the Threaded OpenMP program, the results have been analysed on the parallel g++ compiler version. In the end the CUDA program results have been tested on nvcc version 5.5 to check the GPU based performance evaluation results.

Figure 1 shows the evaluation results for C program. The results of OpenMP enabled C++ program have been shown in the Figure 2. The implementation results of sorting using CUDA library named as THRUST have been demonstrated in the Figure 3. A comparative analysis for all these three implementation of radixsort procedures has been illustrated in the Figure 4. The last figure shows how the CUDA implementation yields the faster throughput for the elements of different sizes. When the results were compared to the discrete levels of distinction, the threaded parallelism was found to be 10 times faster than the sequential routine and the GPU powered outcome was further evaluated to be even 10 times faster than the parallel threaded architecture.

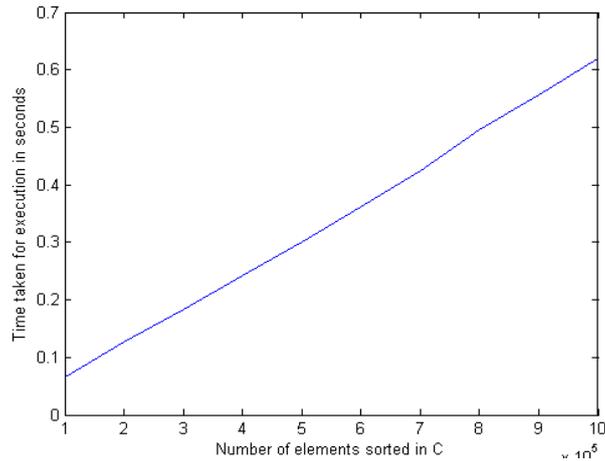


Figure 1. C program sorting time usage

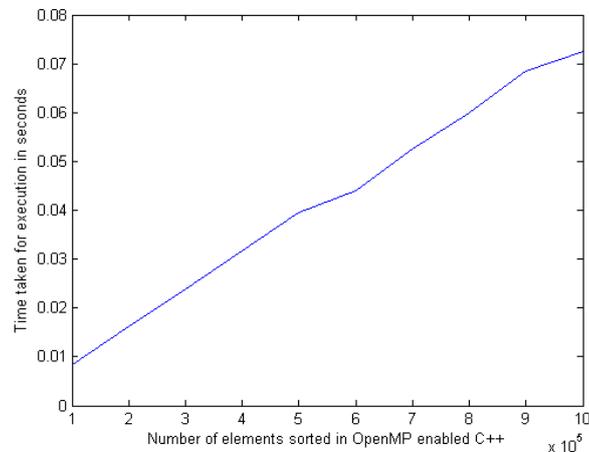


Figure 2. OpenMP program sorting time usage

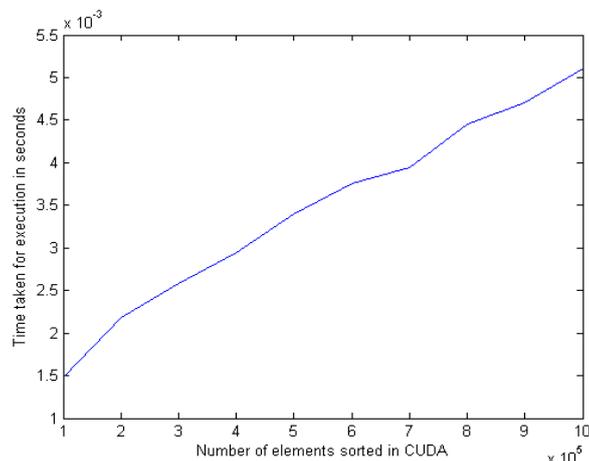


Figure 3. CUDA program sorting time usage

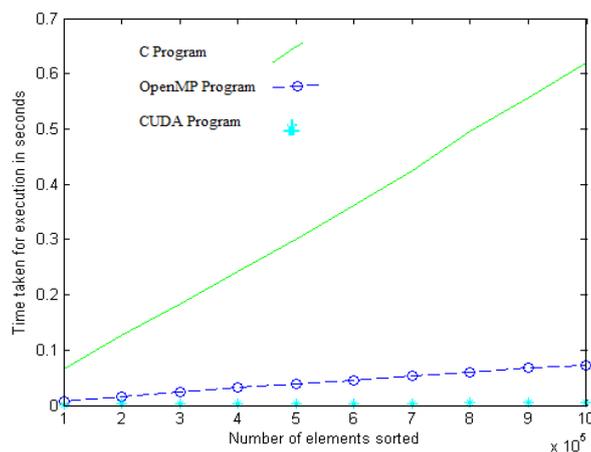


Figure 4. Comparative analysis

## 4.1 Programmatic Analysis

### 4.1.1 Our C Program:

We have implemented a sequential algorithm to perform radix sort on a file which is of different size each time for which its time is evaluated for performing the radix sort. In order to have a different size of file to be sorted we simply have a list of integers generated into a separate text file which acts as an input for our C program.

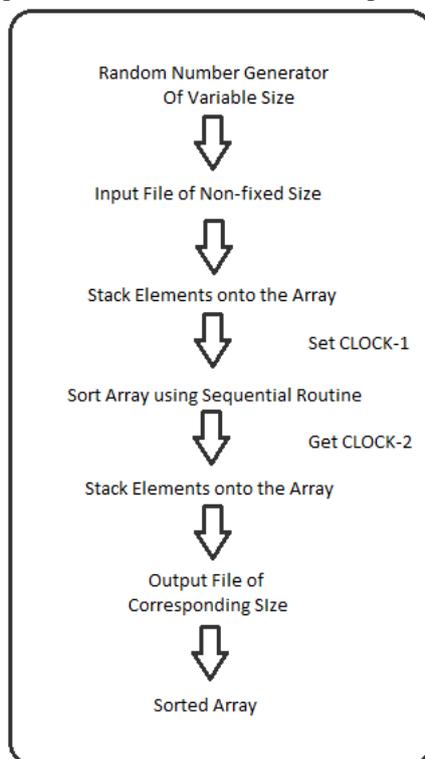


Figure 4. Flowchart for Sequential C Program

Later all the values from the text file containing the generated random integers are printed on to the terminal in order to assure, all the entries are being read into our program. These values are stacked into an array. Then a separate function to sort the array elements is called, which includes the procedure to perform radix sort on them. The final sorted array is then returned back on to the stack in order to check the sorted results. Time evaluation checks have been done before and after the function call for the radix sort function. Thus the difference in between these calls ensures the interval of time taken by the program to perform radix sort in the selected file. In order to ensure, what is the input to the program and what the corresponding output of the program is, we have made separate calls that check what the contents of the file are, before and after the radix sort has been done. The flowchart of the program that we have implemented has been displayed as above.

### 4.1.2 Our OpenMP Program:

Analogous to the input fed to the C program, we have the body structure of our OpenMP program, but the only difference lies in the root level functioning of the program. At the basic level of radix sort, where we encounter the radix sort function in our program, we divide the tasks to be handled by the thread in parallel. Each thread is independently

working on the sorting buckets and the number of threads created, have initially been set to sixty-four in our case. Also the thread level parallelism is the only advantage offered by the OpenMP construct of our C/C++ program. On the basis of similar functioning, the time calls before and after the radix sort function calls ensure that what would be the time taken by our algorithmic procedure in order to sort the respective integer stack or array. Furthermore we have also used the threads in order to display the contents of our input and output, so that parallelism could be used there also. A variable number of thread declarations was used and the best results that were obtained corresponding to it have been used for the CPU level, ensures better results threads our results. The thread level parallelism at

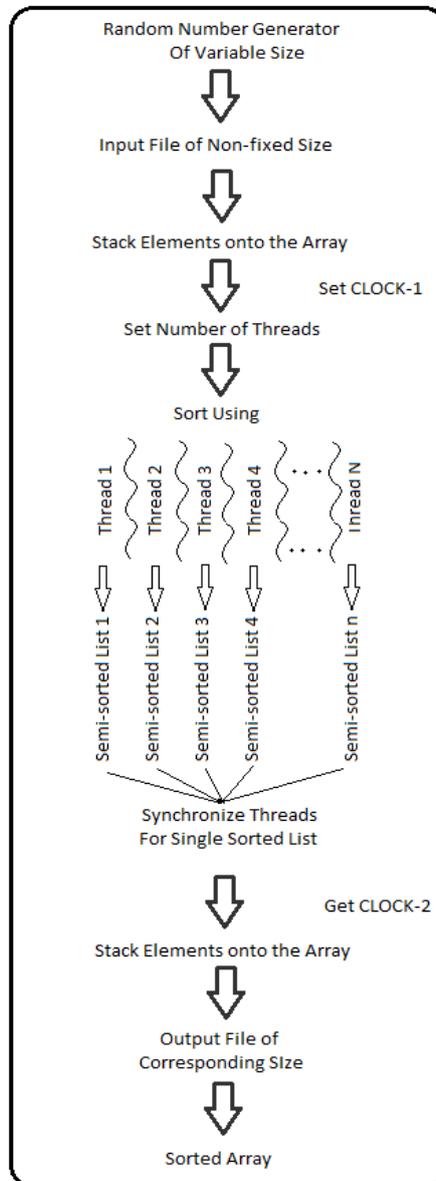


Figure 5. Flowchart for Threaded OpenMP Program

The results when compared with the normal sequential routine with no threads included yielded better performance and a better response time in comparison to the earlier one.

#### 4.1.3 The CUDA Program:

The C and OpenMP programs had the only disadvantage of having all the efficiency to be evaluated to be run at the CPU level constrained to the usage of local architecture of the normal CPU. But coming into the picture of the GPU plus CPU architecture we enabled the provided CUDA program from the THRUST library to run for similar number of elements being fed onto the program. The CUDA program first incorporates a local cache of structure to stack the similar number of elements to be set for sorting. On the analogous platform we further make a call to the sorting subroutine which ensures the flow to be transferred from the CPU on to the GPU through the kernel call, which is enabled by the CudaMemcpy function of the CUDA construct which transfers the contents from the host to the device. The sorting for the entire structure is done by the GPU. Once the sorting is over, the stack is again moved back to CPU using the similar CudaMemcpy function, this time from the device back to host. The clock calls both before and after the sorting procedure renders us to have the time calculated for sorting. The structured block for normal CUDA program has been shown diagrammatically below in Figure 6:

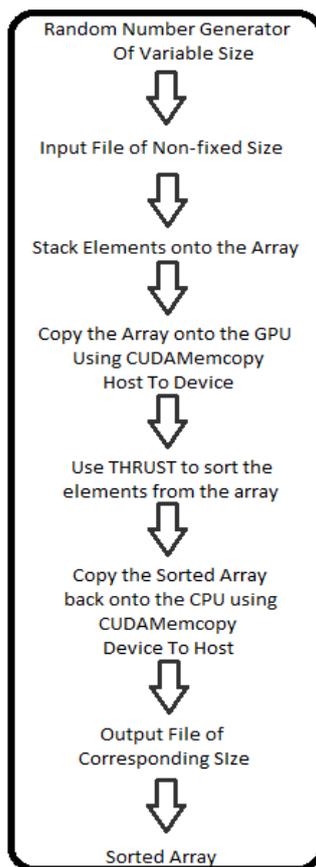


Figure 6. Flowchart for CUDA Program

## V. CONCLUSION

We have tested the results for the performance of radixsort on different algorithmic routines comprising of several parallel executions at different levels and sublevels. Our experimental results show that the GPU usage for performance improvement is about 100 times more efficient than the normal CPU usage. Thus this performance can be well utilized by the parallel applications that need very high levels of minute analysis in simulation and modulation. This mode of sorting efficiently can be used for faster graph explorations and image rendering.

## ACKNOWLEDGMENTS

Our sincere thanks to NVIDIA for providing the hardware at the CUDA Teaching Center, Jaypee University of Information Technology and to Mr. Michael Garland for guidance inspired towards CUDA Thrust library usage.

## REFERENCES

- [1] Stephan C. Kramer and Johannes Hagemann. 2015. SciPAL: Expression Templates and Composition Closure Objects for High Performance Computational Physics with CUDA and OpenMP. *ACM Trans. Parallel Comput.* 1, 2, Article 15 (February 2015), 31 pages. DOI=10.1145/2686886 <http://doi.acm.org/10.1145/2686886>.
- [2] D. Cederman, and P. Tsigas, "A Practical Quicksort Algorithm for Graphics Processors", Technical Report, Gothenburg, Sweden, pp 246–258..
- [3] Zaid Abdi Alkareem Alyasseri, Kadhim Al-Attar, Mazin Nasser, "Parallelize Bubble Sort Algorithm Using OpenMP", in *Distributed, Parallel, and Cluster Computing*, 2014, arXiv preprint arXiv:1407.6603.
- [4] Yin Ye, Zhihui Du, and David A. Bader. "GPUMemSort: A High Performance Graphic Co-processors Sorting Algorithm for Large Scale In-Memory Data," *Annual International Conference on Advances in Distributed and Parallel Computing* (ADPC 2010), Singapore, November 1-2, 2010.
- [5] Koike, A. and Sadakane, K., "A Novel Computational Model for GPUs with Applications to Efficient Algorithms", *Parallel & Distributed Processing Symposium Workshops*, 2014, Phoenix, AZ, pp 614-623.
- [6] N. Satish, M. Harris and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs", in *Proc. IEEE Int'l Symp. Parallel & Distributed Processing*, pp. 1-10, 2009.
- [7] Christos H. Papadimitriou, "Computational complexity". in *Encyclopedia of Computer Science*(4th ed.), 2003, Anthony Ralston, Edwin D. Reilly, and David Hemmendinger (Eds.). John Wiley and Sons Ltd., Chichester, UK 260-265.
- [8] Shifu Chen et al., "A Fast and Flexible Sorting Algorithm with CUDA" in *Algorithms and Architectures for Parallel Processing* 2009, Taipei, Taiwan, pp 281-290.

- [9] Timo Bingmann, Peter Sanders, "Parallel String Sample Sort" in Algorithms, 2013 Sophia Antipolis, France, pp 170-180.
- [10] Muthu Manikandan Baskaran, J. Ramanujam and P. Sadayappan, Automatic C-to-CUDA Code Generation for Affine Programs ", in *Compiler Construction 2010*, pp244-263
- [11] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2), 40-53.
- [12] Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3), 66-73.
- [13] Yang, C. T., Huang, C. L., & Lin, C. F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1), 266-269.
- [14] Kirk, D. B., & Wen-me, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.
- [15] Kirk, D. (2007, October). NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM* (Vol. 7, pp. 103-104).
- [16] Lee, S., Min, S. J., & Eigenmann, R. (2009). OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4), 101-110.
- [17] Jarp, S., Lazzaro, A., Leduc, J., Nowak, A., & Pantaleo, F. (2011, December). Parallelization of maximum likelihood fits with OpenMP and CUDA. In *Journal of Physics: Conference Series* (Vol. 331, No. 3, p. 032021). IOP Publishing.
- [18] Ohshima, S., Hirasawa, S., & Honda, H. (2010). OMPCUDA: OpenMP execution framework for CUDA based on omni OpenMP compiler. In *Beyond loop level parallelism in OpenMP: accelerators, tasking and more* (pp. 161-173). Springer Berlin Heidelberg.
- [19] Lee, S., & Eigenmann, R. (2010, November). OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-11). IEEE Computer Society.
- [20] Jang, H., Park, A., & Jung, K. (2008, December). Neural network implementation using cuda and openmp. In *Digital Image Computing: Techniques and Applications (DICTA), 2008* (pp. 155-161). IEEE.