Enhancing Performance in Heterogeneous Computing: A Comparative Study of CUDA on GPUs and CPUs

Mitul N. Takodara
Member, IEEE
Dept. of Computer Engineering
Marwadi University
Rajkot, India
mitul.takodara@gmail.com

Parth Parmar
Dept. of Computer Engineering
Marwadi University
Rajkot, India
parmarparthvv@gmail.com

Ravikumar R N
Member, IEEE
Dept. of Computer Engineering
Marwadi University
Rajkot, India
rnravikumar.cse@gmail.com

Sivakumar N
School of Computer Science & IT,
Jain (Deemed to be University)
Banglore, India
drsivakumar.nadarajan@gmail.com

Rohit Kumar Tiwari
School of Computer Engineering
KIIT Deemed to be University
Bhubaneswar, India
rohittiwarinitw@gmail.com

2023 IEEE Fifth International Conference on Advances in Electronics, Computers and Communications (ICAECC) | 979-8-3503-0544-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICAECC59324.2023.10560120

***Abstract*— Heterogeneous computing, in which Graphics Processing Units (GPUs) serve as coprocessors to CPUs for arithmetic-intensive data-parallel tasks, has become increasingly popular due to the pressing demand for higher computational efficiency in science and engineering. When it comes to heterogeneous computing on NVIDIA GPUs, the emerging standard is CUDA, or Compute Unified Device Architecture. The primary objective of CUDA is to free developers from the burden of implementing parallel algorithms and instead allow them to concentrate on the work at hand. The main objective of the work is to obtain the performance gain in execution speed for the dynamic algorithms which generally are complex and takes a very long time for execution and compare results on different GPU processors and CPU and have a comparative study of algorithms. This work is mainly concentrated on working with dynamic algorithms such as Knapsack, Binary Search, Longest Common Subsequence, Kruskal's Algorithm on GTX 480 and Tesla C2070 GPUs. Also, the work consists of preliminary sorting algorithms such as Insertion Sort, Selection Sort, and Bubble Sort. Report contains study of the NVIDIA CUDA architecture, CUDA SDK tool kit and different methods to get performance benefit, implementation of intermediate tool to find out functions and their dependencies from the source code and the implementation of the complete algorithm, testing and obtaining statistics for the same. The performance of such algorithms was improved by a factor of 2X to 64X when implemented on GPU as opposed to CPU.**

***Keywords*— CUDA Architecture, Performance, Dynamic algorithms, NVCC compilation, GPU**

I. Introduction

Because of its potent capacity to execute general-purpose sequential programmes, the Central Processing Unit (CPU) processor has garnered the industry's attention throughout the history of microprocessors. Despite being incredibly successful in satisfying the majority of computing requirements, programmable accelerators have a history of enhancing performance for particular application domains. Digital Signal Processors (DSPs) are devices that encode and decode audio in embedded systems. Since the 1970s, there have been numerous examples of coprocessors in High Performance Computing (HPC) that were created specifically for floating point calculations or other specialized activities [1]. Discrete video processors have long been used in consumer computers to address the unique requirements of rendering pictures at the demanding rate of stutter-free video Since then, GPUs have been utilized for massively parallel computations on data other than graphics. Coprocessors and programmable accelerators have long been used in scientific computing to meet the field's seemingly insatiable demand for processing power [2]. In reality, modern high-end supercomputers increasingly combine graphics processing units (GPUs) with traditional CPUs and stream processors. In actuality, current supercomputers are increasingly combining graphics processing units (GPUs) with classical CPUs and stream processors. The door was therefore flung wide for GPUs to be utilized for massively parallel computations on data not normally associated with graphics. For a long time, scientific computing has relied on coprocessors and programmable accelerators to meet its seemingly insatiable demand for computational performance. In fact, graphics processing units (GPUs) are becoming common in today's high-end supercomputers, which combine the best of both classic CPUs and stream processors. The top-tier GPUs of today have the potential to outperform even the most powerful X86 quad-core CPU by a factor of more than four [3].

A. GPU Hardware Architecture

The number of cores in modern multi-core CPUs ranges from 2-8. These cores often operate in a decentralized and asynchronous fashion. This means that various instructions can be run on separate data by each core simultaneously. In this, discussion has been done on MIMD (Multiple Instruction, Multiple Data) class of computer architectures, to use Flynn's terminology [4]. In contrast, graphics processing units (GPUs) are optimized for arithmetic operations due to their primary function of creating a graphic scene for display. Modern graphics accelerators are multi-processor devices with as many as 30 processors. Each multiprocessor has a number of ALUs (8, 12, or 16 are common) that it uses to perform calculations. The most powerful modern GPUs pack as many as 480 processing cores. CPU and GPU high-level architecture is depicted in Figure-1.

Fig. 1 GPU Devoted More Transistor to Data Processing [1]

979-8-3503-0544-9/23/$31.00 ©2023 IEEE

- While CPU cores are optimized for the rapid execution of a single thread of sequential instructions, GPUs are built for the rapid execution of several threads of parallel instructions.
- Vector units in CPUs utilize single instruction multiple data (SIMD), While scalar threads in GPUs employ single instruction multiple threads (SIMT), scalar threads in CPUs do not.
- Multi-threaded processes differ substantially. GPUs can execute up to 1024 threads per multiprocessor, compared to CPUs' 1-2 per core. CPUs take hundreds of cycles to swap threads, whereas GPUs can do it in one [5].
- Large caches and branch prediction help CPUs decrease the time it takes to reach memory. By executing thousands of threads in parallel, GPUs eliminate memory access latencies. The GPU can run another thread without delays while the first waits for data from memory [6].

II. RELATED WORK

Multi-core computers are becoming more common, which is both a chance and a problem when it comes to parallel processing. This work talks about automatic parallel tools based on OpenMP, ROSE, and CUDA. The focus is on ROSE and how it is used. The study also uses automatic parallelization on CUDA and shows a new interface to reduce complexity [7]. In order to improve the efficiency of dynamic programming algorithms, this paper suggests using chip multiprocessor (CMP) techniques that make optimal use of cache. D-CMP, which uses individual caches, S-CMP, which uses shared caches, and Multicore, which uses both individual L1 and shared L2 caches, are all taken into consideration. The research results are then applied to problems such as local dependency dynamic programming, the Gaussian Elimination Paradigm, and parenthesis. The results of the experiments indicate that the algorithms might potentially be greatly sped up by using reduced versions [8].

This paper presents a concurrent method for solving dynamic programming optimisation problems, such as finding the optimal cost-to-go function for robot motion planning in dense environments, which allows for faster solutions while maintaining mathematical equivalence to those found using traditional grid-based motion planning solvers [9].

DAG-based distributed X10 framework DPX10 simplifies parallel programming for distributed DP applications. DAG patterns and DP application logic in the calculate function simplify the procedure. DPX10 has eight popular DAG patterns and a simple API for customization. Job distribution, scheduling, and communication are fault-tolerant. Four DP applications with 2 billion vertices and 240 cores demonstrate the framework's simplicity, efficiency, and scalability [10].

For dynamic programming problems, state-of-the-art cache-blind parallel approaches seldom employ optimum parallelism. By only putting tasks into motion when real dependencies are satisfied and were able to get rid of the false dependence that was holding us back. Asymptotically optimal work and cache complexity were maintained while Floyd-Warshall's technique, stencil computations, and LCS were extended to a wide variety of dynamic programming problems. There is a three- to five-fold increase in absolute running time, a tenfold increase in encumbered span, and a similar decrease in L1/L2 cache misses while using PAPI because of Cilkview [11].

Non-serial polyadic dynamic programming (NPDP) and matrix chain multiplication (MCM) are two examples of dynamic programming algorithms that may be parallelized on multicore and manycore GPUs to handle combinatorial and optimization problems. This study demonstrates that static scheduling is inappropriate in NPDP settings. Performance is enhanced by 14% when using adaptive parallelization as opposed to fixed block-size methods. A generic approach to scheduling NPDP subproblems on multicore and manycore is introduced to maximize hardware and software utilization [12].

The use of time-sensitive scheduling by comparing the rates of cache misses with the rates of workloads running in a core, Cache Aware Dynamic-Earliest Deadline First (CAD-EDF) for SMT multicore processors attempts to limit deadline misses. The method guarantees efficient scheduling and resource allocation, and it is used as the basis for the LITMUS-RT kernel [13].

Important applications may coexist with less crucial ones on multicore systems, as suggested by the study. In addition to measuring interferences from other cores, this technology allows for continuous monitoring of the progress of the program without needing any changes. Important slowdowns between 1.8 and 3 percent are detected using this technology, allowing for rapid identification of abnormal or unacceptable lags and the execution of appropriate countermeasures [14].

A new approach for estimating the power consumption of multicore CPUs is presented here. Embedded systems with finite power supplies need to be energy efficient. Virtual platforms may be built, and instruction tracing and profiling can be performed, all with the help of the Instruction Accurate Simulator Imperas. The OR1K and MIPS32 energy consumption models both employ a CPI of 1 [15].

III. PERFORMANCE STRATEGIES

The four cornerstones of performance optimization are:

- Convert CUDA C code in parallel to reduce time execution.
- Increase efficiency by maximizing the use of parallel processing.
- Try to convert recursion code to Serial code and further parallelize it.
- Remove dependencies in CUDA C code.
- Maximize instruction throughput by making the most of each one.

The performance limiters of a given subset of an application dictate which tactics will offer the highest performance improvement for that subset; for example, Improving the instruction consumption of a kernel that is mostly constrained by memory accesses will not result in a significant performance benefit. Constant monitoring of the performance bottlenecks, for instance with the CUDA profiler, is essential for focusing optimization efforts in the right places.

A. Maximum Utilization

To get the maximum utilization of the available resources, application should be parallelized in such a way that application keeps Several parts of the system are typically in use.

1) Application Level: The programmed should optimize parallel execution across the host, the devices, and the bus between the host and the devices by using asynchronous function calls and streams. It needs to make sure that the host computer is handling the serial workloads and the devices are handling the parallel ones. There are two scenarios in which data needs to be shared between threads in a parallel-execution program:

- Sharing data across threads inside the same block necessitates the use of sync threads() and shared memory.
- If the threads come from different sections. Here, writing to and reading from global memory necessitate two distinct kernel invocations.

2) Device level: The program, at a more fundamental level, should make the most of the device's multiple processors by doing multiple tasks simultaneously. Devices with computing capability 1.x only allow a single kernel to run at a time, As a result, when starting the kernel, it is critical to give it with as many thread blocks as the device's multiprocessors. Maximum utilization can also be attained for computing capability 2.0 devices in which many kernels can operate concurrently on a device through the use of streams to enable sufficient numbers of kernels to run in parallel.

3) Multiprocessor Level: At a higher level, the application should optimize for parallel execution across the many functional units that comprise a multiprocessor. A GPU multiprocessor uses thread-level parallelism to improve efficiency. Therefore, the amount of use is proportional to the total number of local warps. At each instruction issue time, a warp scheduler selects a ready warp to execute, then transmits the next instruction to the warp's active threads. Full utilization is accomplished when the latency of each warp is entirely concealed by the delay of other warps, or when the warp scheduler always has some instruction to issue for some warp at every clock cycle throughout that latency period. [16]. The amount of clock cycles it takes for a warp to be ready to execute its next instruction is referred to as latency. The number of instructions needed to mask delay is proportional to the throughput of those instructions. There will be delays due to register dependency if all input operands are also registers. The warp scheduler must plan distinct warps when an instruction's latency equals the time it takes to execute the instruction before it because some input operand is written by the previous instruction.

B. Maximize Instruction Throughput

It is considered to be excellent practice to apply optimization at a lower level after all optimization at a higher level has been performed. In order for the program to get the most out of its instruction throughput:

- Reduce the amount of low-throughput arithmetic instructions you employ by, for example, utilizing single-precision rather than double-precision arithmetic or de-normalizing numbers to zero if doing so won't influence the final output [17].
- Reduce as much as possible the number of divergent warps that are created by control flow instructions.
- Cut down on the amount of instructions by, for instance, eliminating as many synchronization points via optimization or making use of limited pointers.

IV. NVCC Compilation

A source file written in the expanded CUDA language is converted into an ANSI C source file during the CUDA phase. It may then continue the compilation and linking process with the help of a general-purpose C compiler. In Figure-2, the specific procedures that must be followed are shown.

A. Compilation Flow

Depending on the NVCC-code option, the CUDA compilers/assemblers transform device code into CUDA binary (cubin) or intermediate PTX code. The host code includes a device code descriptor with this code. The CUDA runtime system uses this descriptor to choose a GPU load image when the host programmed calls the device code. [18].

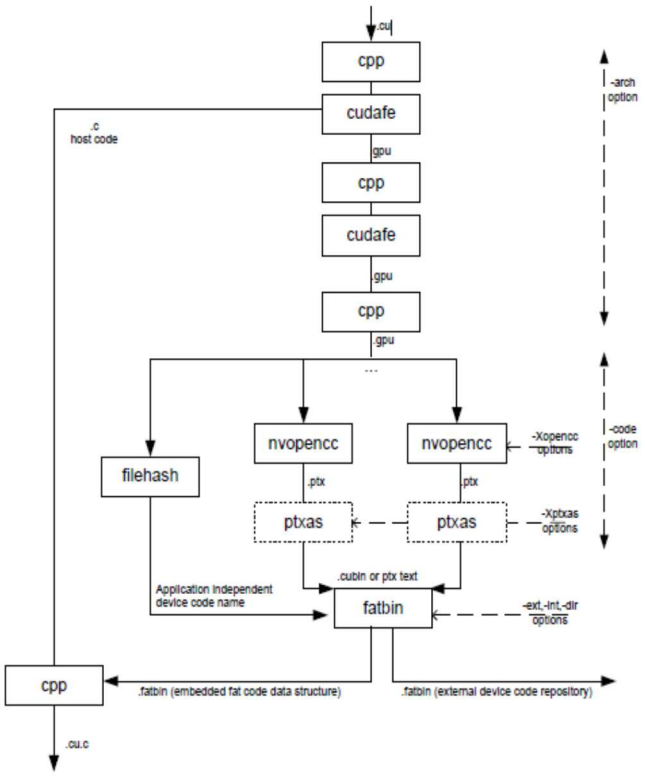

Fig. 2 CUDA compilation from .cu to .c [3]

V. Dynamic Strategy

The core concept of dynamic programming may be summed up in a few words. To solve any given issue, first solve its constituent pieces (sub problems), and only then integrate their

answers to arrive at a comprehensive answer. Many of these smaller issues boil down to the same core issue. To minimize the total amount of calculations, the dynamic programming method prioritizes finding a unique solution to each subproblem. This comes in handy when there is an exponentially high number of repeated sub issues.

Since the finished calculation is a sub-problem of a larger computation, the results of the calculation may be stored and reused in subsequent calculations using top-down dynamic programming. To do a complicated calculation using bottom-up dynamic programming, a recursive succession of smaller computations must be formulated.

A. Dynamic Programming in Computer Engineering

Dynamic programming requires the issue to have an optimum substructure and overlapping sub problems in order to be useful. In contrast, the term "divide and conquer" is more often used than "dynamic programming" to describe the approach used when the overlapping issues are far less severe than the initial problem. [9]. This is why finding all matches of a regular expression, merge sort, and rapid sort are not considered dynamic programming issues. In an optimum substructure, the solution to the overall optimization issue is derived from a union of the solutions to the individual subproblems that make up that problem. The first step in developing a dynamic programming solution is, then, to determine whether or not the issue displays such optimum substructure. To describe such ideal substructures, recursion is often used. If a graph G = (V, E), then the optimum substructure is shown as the shortest route p from a vertex u to a vertex v, where w is an intermediate vertex. If p is the shortest path, then the routes p1 from u to w and p2 from w to v are the shortest paths connecting the pertinent vertices, according to the CLRS's straightforward cut-and-paste argument. As a result, the Bellman-Ford method may be seen as a recursive formulation of the solution for finding shortest routes.

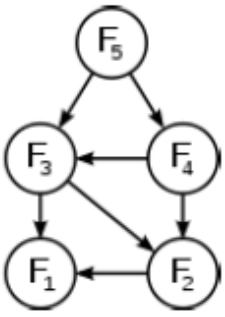

Fig. 3 The sub problem graph for the Fibonacci sequence. The fact that it is not a tree indicates overlapping sub problems [9]

- Top-down approach: This follows necessarily from posing any problem in a recursive fashion. It is easy to memorize or store in a table the solutions to a set of subproblems if the solution to the main problem can be written recursively using the solution to the subproblems, and if the subproblems overlap. Each time to tackle a new subproblem, it looks up in the table to see if it has already been solved. If a solution has already been recorded, apply it without solving the subproblem first and then recording the result in the table.

- Bottom-up approach: This is the situation that intrigues the most. Attempts to reformulate an issue using a bottom-up technique after a recursive formulation of the answer in terms of its subproblems, i.e., by solving the smaller subproblems first and then using those solutions as a foundation to solve the larger subproblems. This is also typically done in a tabular format, with solutions to progressively larger subproblems being generated by applying the solutions to smaller subproblems.

VI. SPEEDUP PERFORMANCE

The algorithms executed on the CPU as well as GPU and took two benchmark conditions:

- Considering memory transfer
- Without considering memory transfer

The sole time-consuming part of graphics processing units (GPUs) is memory transfer between the host and device, and vice versa. In both the benchmark conditions, got the best results tending to produce high speedup performance as compared to that of CPU and made use of NVIDIA Guide for Performing Executions [2].

In this experiment, two GPUs are used and their specification is as follows:

- GTX 480 consisting processor clock of 1401 MHz, Memory transfer rate: 1848 MHz
- Tesla C2070 consisting processor clock of 1494 MHz, Memory transfer rate: 2988 MHz

From the above specifications, better performance is obtained on both the above GPUs rather than CPU which is used Intel Core i3 consisting of 3.14 GHz clock frequency. The following performance table I & II shows the execution time and graph shows the speedup achieved in milliseconds considering both benchmarks defined.

TABLE I
SPEEDUP CONSIDERING MEMORY TRANSFER

Algorithms	**Time Considering Memory Transfer (CMT)**		
	CPU Time(ms)	**GTX 480 Time(ms)**	**Tesla C2070 Time(ms)**
Binary search	592	103	73
Knapsack	153	107	90
LCS	321	154	115
Kruskal	833	493	411
Preliminary Sorting Algorithms			
Insertion Sort	859	91	83
Selection Sort	666	105	49
Bubble Sort	1416	534	497

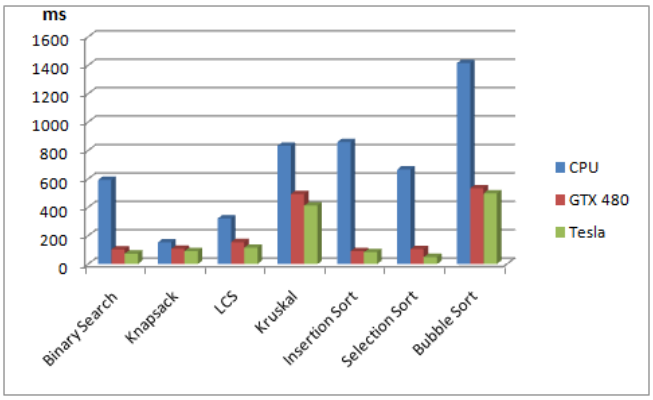

Fig. 4 Speedup performance gain considering memory transfer

Execution of binary search algorithm which obtained speedup gain of 5.7X times on GTX 480 and 8.1X times on Tesla C2070 as compared to CPU by executing the algorithm in parallel [8]. Execution of Knapsack algorithm which obtained speedup gain of 1.42x times on GTX 480 and 1.7X times as compared to that of CPU. Also, execution of Longest Common Subsequence algorithm, achieved a speedup of 2X times on GTX 480 and 2.8x times on Tesla C2070 than CPU. By executions of Kruskal's algorithm, achieved a speedup gain of 1.68X times on GTX 480 and 2X times on Tesla C2070 as compared to that of CPU. Also, executions of preliminary sorting algorithms, achieved more speedup performance on GPU. Hence it proved practically that more performance speedup obtained on GPUs.

TABLE II
SPEEDUP CONSIDERING WITHOUT MEMORY TRANSFER

Algorithms	**Time Considering without Memory Transfer (CWMT)**		
	CPU Time(ms)	**GTX 480 Time(ms)**	**Tesla C2070 Time(ms)**
Binary search	11	0.053	0.043
Knapsack	17	11.53	9.97
LCS	214	96.21	75.43
Kruskal	739	253	213
Preliminary Sorting Algorithms			
Insertion Sort	502	38.44	32.76
Selection Sort	590	27.55	12.63
Bubble Sort	1016	149	117

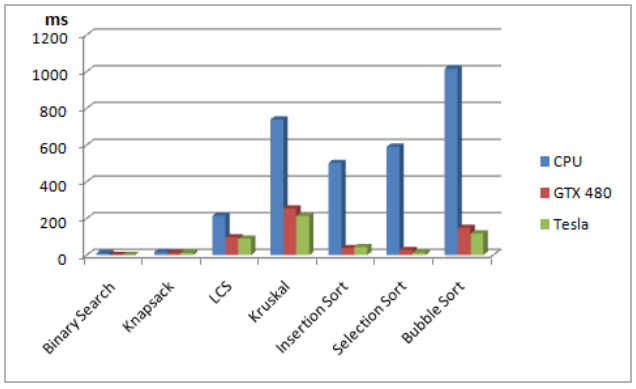

Fig. 5 Speedup performance gain considering without memory transfer

By Execution of binary search algorithm which obtained speedup gain of 207X times on GTX 480 and 256X times on Tesla C2070 as compared to CPU by executing the algorithm in parallel. Execution of Knapsack algorithm which obtained speedup gain of 1.47x times on GTX 480 and 1.8X times as compared to that of CPU. Also, execution of Longest Common Subsequence algorithm, achieve speedup of 2.22X times on GTX 480 and 2.9x times on Tesla C2070 than CPU. By executions of Kruskal's algorithm, achieve speedup gain of 2.92X times on GTX 480 and 3.43X times on Tesla C2070 as compared to that of CPU. Also, executions of preliminary sorting algorithms, achieved more speedup performance on GPU. Hence it proved practically that more performance speedup obtained on GPUs.

VII. FUTURE DIRECTIONS

The future direction can focus on the following points:

Advancements in GPU Technology: GPUs with a lot of cores and a fast clock speed are more effective in increasing performance. Therefore, progress in GPU technology, such as the creation of GPUs with even more cores and greater clock rates, may be in the cards for the future. This may result in even faster and more efficient execution.

Algorithm Optimization: The optimized method would run faster since fewer external factors would slow it down. In the future, researchers may investigate more refined optimization strategies that are optimized for GPU architectures. To fully take use of the power of high-end GPUs, algorithms may need to be redesigned to increase their parallelism and decrease their dependencies.

Integration of GPUs with other Technologies: In the future, it may be possible to achieve even bigger speedup benefits by integrating GPUs with other technologies. Synergistic performance increases in certain domains or applications may be achieved, for instance, by combining GPUs with specialized hardware accelerators like tensor processing units (TPUs) or field-programmable gate arrays (FPGAs).

Combinatorial and Complex Execution Load: Future work may focus on discovering and creating algorithms tailored to these kinds of workloads, making the most of GPUs' parallel processing capabilities. Computational mathematics, scientific simulations, machine learning, and data analytics are all possible examples.

Energy Efficiency: The research community might focus on optimizing the speed and energy efficiency of graphics processing unit (GPU) designs and techniques. This is especially crucial considering the prevalence of GPU-based systems in low-power settings like data centres and mobile devices.

VIII. CONCLUSION

By execution of the complex algorithms and obtaining its statistics, the GPU's having a greater number of cores with high clock frequency achieves gain in speedup. The above implementation results concludes that algorithms implemented on GTX 480 and Tesla C2070 takes less time to execute as compared to CPU leading to speedup gains in microseconds. Even when these algorithms are optimized by removing

dependencies, this enhanced algorithm may achieve more speedup. Also, executions of such combinatorial and complex execution load on high end GPU's leads to more efficient speedup gain.

REFERENCES

[1] CUDA Education & Training. (2011, March 21). NVIDIA Developer. https://developer.nvidia.com/cuda-education-training

[2] CUDA C++ Programming Guide. (2023, June 1). CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] Open Computing Language OpenCL. (2013, April 25). NVIDIA Developer. https://developer.nvidia.com/opencl

[4] Morrison, A. (2022, December 15). Nvidia CUDA Architecture | Cloud2Data. Cloud2Data. https://cloud2data.com/nvidia-cuda-architecture-3/

[5] B. R. Neha Patil, Department of Electrical & Computer and System Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, "Fast and Parallel Implementation of Image Processing Algorithms Using Cuda Technology on GPU Hardware".

[6] NVIDIA Research Homepage. (n.d.). Research, Publications & Journals | NVIDIA. https://www.nvidia.com/en-us/research/

[7] Yang, C. T., Chang, T. C., Huang, K. L., Liu, J. C., & Chang, C. H. (2012). Performance evaluation of openmp and cuda on multicore systems. In Algorithms and Architectures for Parallel Processing: 12th International Conference, ICA3PP 2012, Fukuoka, Japan, September 4-7, 2012, Proceedings, Part II 12 (pp. 235-244). Springer Berlin Heidelberg.

[8] Chowdhury, R. A., & Ramachandran, V. (2008, June). Cache-efficient dynamic programming algorithms for multicores. In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (pp. 207-216).

[9] Cossell, S., & Guivant, J. (2014). Concurrent dynamic programming for grid-based problems and its application for real-time path planning. Robotics and Autonomous Systems, 62(6), 737-751.

[10] Wang, C., Yu, C., Tang, S., Xiao, J., Sun, J., & Meng, X. (2016). A general and fast distributed system for large-scale dynamic programming applications. Parallel Computing, 60, 1-21.

[11] Tang, Y., You, R., Kan, H., Tithi, J. J., Ganapathi, P., & Chowdhury, R. A. (2015, January). Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 205-214).

[12] Diwan, T., & Sathe, S. R. (2017). A generalized strategy for parallelization of non-serial polyadic dynamic programming on multicore and manycore. Advanced Science Letters, 23(4), 3802-3807.

[13] Sijili, S., Balakrishnan, P., & Rashid, M. E. (2018, July). Cache Aware Dynamic Scheduler for Real Time Task in Multicore Processors. In 2018 International Conference on Emerging Trends and Innovations In Engineering And Technological Research (ICETIETR) (pp. 1-5). IEEE.

[14] Freitag, J., & Uhrig, S. (2017, September). Dynamic interference quantification for multicore processors. In 2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC) (pp. 1-6). IEEE.

[15] Irfan, M., Masud, S., & Pasha, M. A. (2018, May). Development of a High Level Power Estimation Framework for Multicore Processors. In 2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC) (pp. 1-1770). IEEE.

[16] Ch., S., Upadhyay, D. N., & Govardhan, D. A. (2017, October 20). A NOVEL AUTOMATIC C TO NVIDIACUDA CODE OPTIMIZATION FRAMEWORK | Ch. | International Journal of Advanced Research in Computer Science. https://doi.org/10.26483/ijarcs.v8i8.4738

[17] Tech blog: https://www.spiceworks.com/tech/devops/articles/what-is-dynamicprogramming.

[18] Karunadasa, N. P., & Ranasinghe, D. N. (2009). On the comparative performance of parallel algorithms on small GPU/CUDA clusters. In International conference on high performance computing.