HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF MECHANICAL ENGINEERING

PROJECT REPORT

Bellman-Ford Algorithm on CPU vs GPU

Group 12

LÊ VŨ ĐỨC HƯNG 20195781

TRƯƠNG QUỐC VIỆT 20195828

TRẦN LONG NHẬT 20195803

Supervisor: PhD. Nguyễn Đại Dương

Hanoi, 22/01/2024

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF MECHANICAL ENGINEERING

PROJECT REPORT

Bellman-Ford Algorithm on CPU vs GPU

Group 12

LÊ VŨ ĐỨC HƯNG 20195781

TRƯƠNG QUỐC VIỆT 20195828

TRẦN LONG NHẬT 20195803

Supervisor: PhD. Nguyễn Đại Dương

Hanoi, 22/01/2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This project presents a comparision between Central Processing Unit(CPU) and Graphics Processing Unit (GPU) implementation of a Bellman-Ford (BF) routing algorithm. In the proposed GPU-based approach, multiple threads run concurrently over numerous streaming processors in the GPU to dynamically update routing information. Instead of computing the individual vertex distances one-by-one like on CPU, a number of threads concurrently update a larger number of vertex distances, and an individual vertex distance is represented in a single thread. This project compares the performance of the GPU-based approach to an equivalent CPU implementation while varying the number of vertices. Experimental results show that the proposed GPU-based approach outperforms the equivalent sequential CPU implementation in terms of execution time by exploiting the massive parallelism inherent in the BF routing algorithm. In addition, the rising in energy consumption by using the GPU is a notable trade-off between performance gains and energy efficiency underscores the need to strike a balance when choosing between CPU and GPU implementations of algorithms like Bellman-Ford.

CHAPTER 1. INTRODUCTION

The Shortest Path Problem (SPP) is a fundamental challenge in graph theory and network optimization, aiming to find the most efficient route between two nodes in a given graph. One of the classical algorithms designed to address this problem is the Single-Source Shortest Path (SSSP) algorithm, and within this category, the Bellman-Ford algorithm holds a significant place. Originally proposed by Alfonso Shimbel in 1955 and independently by Richard Bellman in 1956, and Lester Ford, Jr. in 1956, the Bellman-Ford algorithm efficiently computes the shortest paths from a single source to all other nodes in a directed, weighted graph.

As the demand for faster and more scalable algorithms has grown, there has been a surge in exploring hardware acceleration options to enhance the computational efficiency of algorithms like Bellman-Ford. In this report, we delve into the implementation and performance comparison of the Bellman-Ford algorithm on two distinct computing architectures: Central Processing Unit (CPU) and Graphics Processing Unit (GPU).

The CPU, being the traditional workhorse of computing, has long been the primary choice for executing algorithms. On the other hand, the GPU, originally designed for graphics processing, has emerged as a parallel processing powerhouse capable of handling massive data sets in parallel, offering the potential for significant acceleration in certain computational tasks. This report explores how the Bellman-Ford algorithm, known for its iterative nature, performs on both CPU and GPU architectures. Through a comparative analysis, we aim to elucidate the strengths and weaknesses of each implementation, shedding light on the trade-offs involved and providing insights into the optimal utilization of hardware resources for SSSP algorithms.

CHAPTER 2: BACKGROUND STUDY

2.1 NVIDIA's GPU architecture and CUDA

Initially, the GPU was developed for real-time and high-definition 3D graphics applications. However, the use of GPUs for high performance computing has been gaining attention due to the massive parallelism possible [8]. A GPU uses a parallel programming scheme in which thousands of threads concurrently execute operations on different data chunks. Commonly used GPUs belong to NVidia's G100, GeForce 400, Fermi, and Tesla series [21].

A GPU usually consists of a number of streaming multiprocessors (SMs), which are grouped in pairs into general processing group blocks and is connected to a host (CPU) through a high speed PCI-Express I/O bus. For example, each Tesla multiprocessor has 30 SMs with 240 cores and the Fermi has 16 SMs with 512 cores. Within each SM, warps, or a group of threads, are concurrently executed using a single instruction. A thread is a basic element of the data to be processed in the GPU, and all cores in the same group execute the same instruction at the same time.

Figure 2.1 shows a single precision (SP) GPU platform [5] where different threads are processed on a number of blocks B1,B2,B3...Bn. The special function unit (SFU) executes one instruction per thread per clock, and each thread has a unique identifier within the block (threadIdx) that is used to select the data to operate on. A thread has a relatively large number of registers and also has a private area of memory known as local memory, which is used for register file spilling. Any large automatic variable is referred to as a warp, which is a set of threads. The warp size depends on the hardware platform. For example, the warp size is 16 for Tesla and 32 for Fermi. If there exist 128 threads in a block, then threads 0–31 will be in one warp. The block is then the group of threads that execute together in a batch.

NVIDIA introduced CUDA technology to enable users to solve many complex problems on their GPU cards [14]. CUDA is a proprietary API and set of language extensions that work only on NVIDIA's GPUs [20]. It works by providing parallelism whereby multiple threads concurrently execute the same set of instructions on different chunks of data.

A programmer can control the number of threads to be executed. If the number of threads is more than the warp size, they share time internally on the multiprocessors. A block runs on a multiprocessor at a given time, and multiple blocks can be assigned to a single multiprocessor with their execution being time-shared [12]. A single execution

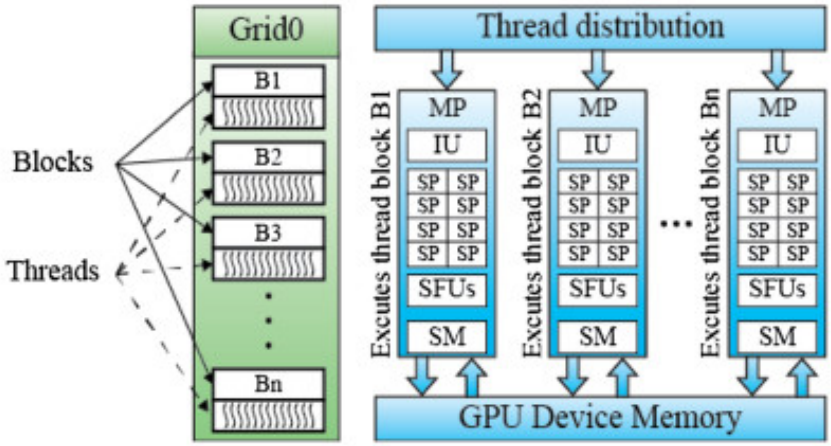


Figure 2.1 GPU multi-threading arrangement.

is called a grid where all threads of different blocks execute on a single multiprocessor and share resources equally. In a manner similar to threads, a unique ID is also assigned to each block during the execution. Using the thread and block IDs, each thread can perform the kernel task on different sets of data [15]. Since the device memory is available to all the threads, the task can access any memory location. Therefore, the hardware architecture allows multiple instruction sets to be executed on different multiprocessors.

Within a CUDA application, data may be placed in global memory, shared-memory, constant memory or texture memory [4]. Global memory is persistent and accessible to all threads of a given application. Shared memory is accessible to threads within a thread block and has a lifetime equal to that of the threads within the block. In addition, it is a read-only memory that provides cached data access. Finally, the data bandwidth of the texture memory tends to be much higher than that of the global memory or constant memory.

The basic CUDA program flow [18] is depicted in Figure 2.2. Initially, the host (CPU) loads data from a source file and stores it into a data structure in the host's memory. The host then allocates the device memory for the data and copies the data to the allocated memory. The device kernels are launched to process the data and produce results. Finally, the results are copied back to the host for further processing [11].

2.2 A Bellman Ford distance vector routing algorithm

The Bellman-Ford (BF) algorithm [1] computes the shortest paths from a single source vertex to all of the other vertices in a weighted graph, as shown in Figure 2.3. Although this algorithm is more versatile than Dijkstra's link-state algorithm, the main limitation of this algorithm is its high computation time. The BF algorithm is based on the principle of 'relaxation', in which an approximation to the correct distance is

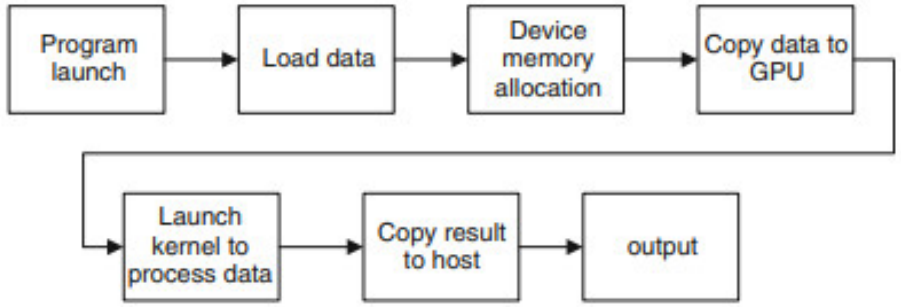
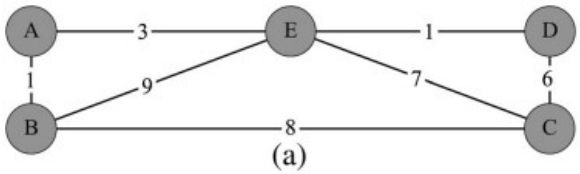


Figure 2.2 Flow diagram of CUDA program.

gradually replaced by more accurate values until eventually the optimum solution is reached. This algorithm utilizes the cost of the edges as a route metric and finds the shortest simple path if no negative cycle exists.



(a)

Bellman-Ford Algorithm

```
1:  Procedure Bellam Ford(list vertices, list edges, vertex source)
2:  STEP 1: initialize graph
3:  for each vertex v in set of vertices : if v is source then v.distance = 0, else v.distance = infinity
4:  set v.predecessor = null
5:  Step 2: relax edges repeatedly
6:  for i from 1 to size (vertices)-1:
7:  for each edge uv in edges: uv is the edge from u to v
8:  u = uv.source
9:  v = uv.destination
10: if u.distance + uv.weight <  v.distance:  v.distance =u.distance + uv.weight
11: v.predecessor = u
12: Step 3: Check for negative-weight cycles for each edge uv in edges:
13: u = uv.source
14: v = uv.destinationt
15: if u.distance + uv.weight <  v.distance: error, then output a negative-weight cycle message.
```

(b)

Figure 2.3 Connected routers with different link weights: a) a pictorial example and b) the Bellman-Ford algorithm.

Given routers A, B, C, D and E, let $\exists i$, which is a label for the routers A, B, C, D and E, respectively. Then, let $D(i, j)$ be the distance for a known route from i to a router location j, and let $d(i, j)$ be the distance from router i to neighbor j, where $d(i, j)$ is set to infinity if $i = j$ or if i and j are not immediate neighbors. The BF equation that can find the best route between routers A and E in Figure 2.3(a) is as follows. $D(i, j) = min(d(i,k) + D(k, j) \forall i \neq j)$. The overall BF algorithm is shown in Figure 2.3(b) [21].

The BF algorithm solves the BF equation by calculating the $D(i, j)$ node, for example, nodes A and E in Figure 2.3(a) by assuming that node i needs distances $d(i,k)$ and $D(k, j)$ from the neighbor nodes. The associated problem with this is that the $D(k, j)$ from the neighbors which are more than one hop to the node are not known by the considered node. Thus, the algorithm resolves this by finding the distance with the least weight link from i to j using single hop links $D(i, j)$ for each node i until node E is reached from node A.

Parameter	Value
Processor	Intel(R) Core(TM) i7-6820HQ
Number of cores	8
Threads	16
Clock frequency	2.7 GHz

Table 2.1 Hardware specifications of the CPU.

2.3 Mapping a parallel Bellman-Ford algorithm on a GPU

This section presents a detailed implementation of the parallel BF algorithm using a heterogeneous model[6]. In a heterogeneous model, the CPU is utilized to initialize a network scenario consisting of thousands of vertices and edges in which the parallel routes update using relax edges in the kernels of a GPU.

We utilize a NVIDIA Quadro M400M GPU using a CUDA driver, CUDA ToolKit version 12.3, to run CUDA C code in the nvcc compiler. The detailed configurations of the CPU and GPU used in the experiment are presented in Tables 2.1 and 2.2. The calculation of the relaxed and negatively weighted edge cycles is an important aspect of the implementation of the BF algorithm, where the recurring loop problem is addressed. Step 2 in Figure 2.3(b) guarantees the shortest distances if the graph does not contain a negative weight. In step 3 of the BF algorithm, there is an attempt to terminate the iteration if a negatively weighted edge cycle is encountered. The iteration is terminated if the graph contains a negative weight cycle for any of edge $u - v$ by the following condition: $dist[v] > dist[u] + weight$ of edge uv [9].

In a GPU, one kernel is executed at a time and a certain number of threads execute in each kernel. The mode of instruction execution on the GPU and the CPU are shown in Figure 2.4(a) and (b), respectively. As shown in Figure 2.4(b), a number of threads concurrently execute the instructions necessary for calculating weight itself, and this is what results in the massive parallelism achieved by the GPU-based approach. The entire graph with all the nodes and edges can be relaxed in a parallel operation by the GPU. After the initialization operation, the GPU can relax the edges and check the

Device	NVIDIA Quadro M4000M
GPU Memory	8 GB GDDR5
Memory Interface	256-bit
Memory Bandwidth	192 GB/s
NVIDIA CUDA Cores	1664
System Interface	PCI Express 3.0 x16
Max Power Consumption	120 W
Thermal Solution	Active
Form Factor	4.4" H × 9.5" L,Single Slot, Full Height
Display Connectors	4x DP 1.2
Max Simultaneous Displays	4 direct, 4 DP 1.2 Multi-Stream
Max DP 1.2 Resolution	4096 × 2160 at 60 Hz
Max VGA Resolution	2048 × 1536 at 85 Hz
Graphics APIs	Shader Model 5.0, OpenGL 4.5, DirectX 12.0
Compute APIs	CUDA, DirectCompute, OpenCL

Table 2.2 Hardware specifications of the GPU.

negative cycle operations in a fully parallel manner. On the other hand, the CPU-based implementation is executed for each node until the entire graph is traversed, as shown in Figure 2.4(a). Since the execution pattern using the CPU is serial, it takes much larger execution time than the parallel GPU-based approach. The total execution time of the CPU depends on the initialization of the vertices and edges and on the completion of the node-by-node and edge-by-edge BF execution. Figure 2.5 shows a flow diagram of the BF implementation using a CPU and heterogeneous GPU system.

In the GPU-based BF algorithm, a graph is divided evenly amongst the threads. Instead of computing the individual vertex distances one-by-one, a thread grid computes all the possible distances between the sequences for each pair of groups concurrently in the GPU. A detailed pictorial explanation of the algorithm execution pattern in both CPU and GPU is presented in Figure 2.6 and 2.7, respectively.

Figure 2.6 illustrates the serial input of vertices into the CPU in a sequential manner, which then results in a serial output of data. In contrast, a group of vertices is launched and processed concurrently on the GPU, as shown in Figure 2.7. All threads run the same code in parallel, and more specifically, each thread of the GPU with an ID computes using data from memory address and makes a control decision. In addition, the number of edge examinations is performed concurrently, reducing the computational complexity from $O(E)$ of in the sequential implementation to $O(log(E))$ in the proposed parallel implementation.

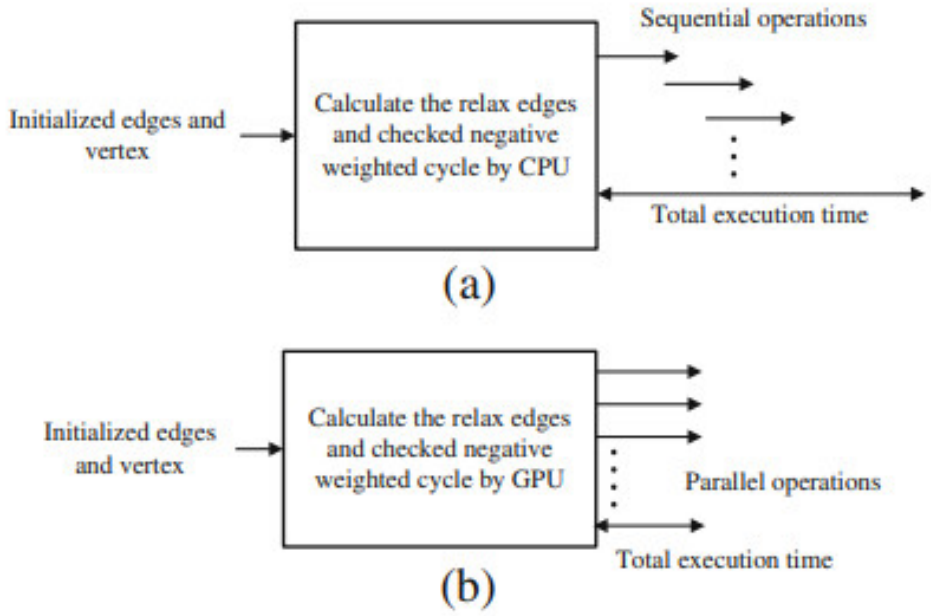


Figure 2.4 The Bellman-Ford algorithm using the a) CPU and b) GPU.

A detailed implementation of the GPU-based BF algorithm is shown in Figure 2.8, where edges $s-r1, s-r2$ and $s-r3$ are considered to belong to thread1, thread2 and thread3, respectively. As the groups of vertices are launched concurrently in the GPU, the output of the different blocks is calculated much faster by using the GPU than by using the CPU. In the GPU-based BF implementation, we allocate memory from the global memory of the GPU, and then transfer the detailed scenario information from the CPU to the GPU. After executing the parallel threads of the BF algorithm in different streaming processors of the GPU, the updated scenario information returns to the CPU.

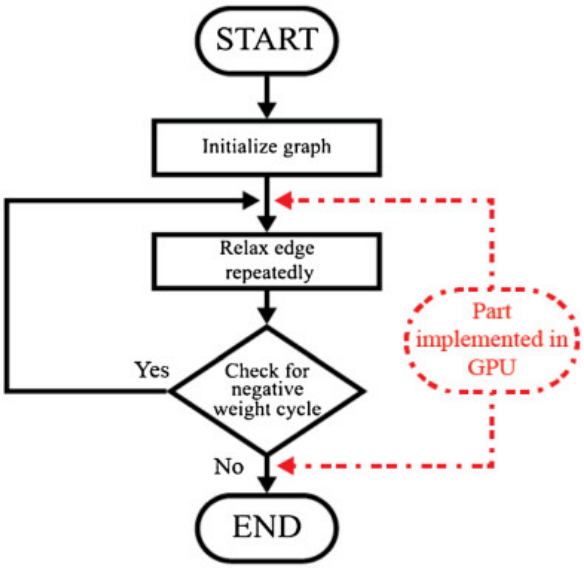


Figure 2.5 A CPU/GPU implementation of the Bellman-Ford algorithm.

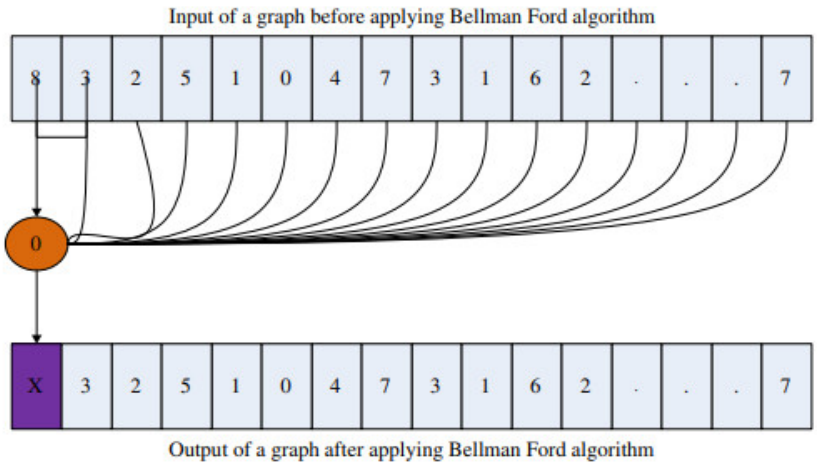


Figure 2.6 Sequential operation of the Bellman Ford algorithm in CPU.

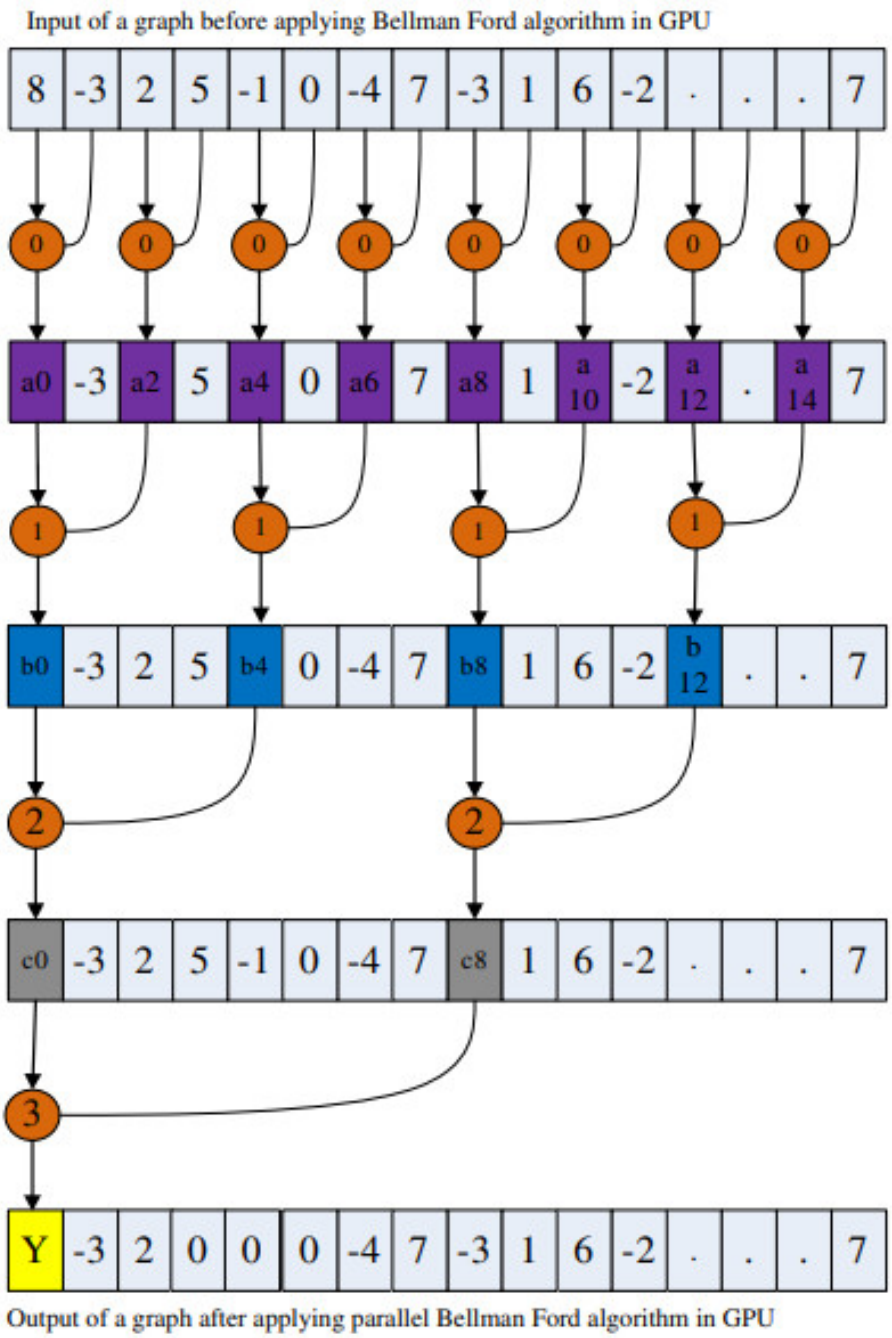


Figure 2.7 Parallel implementation of the Bellman Ford algorithm using a GPU.

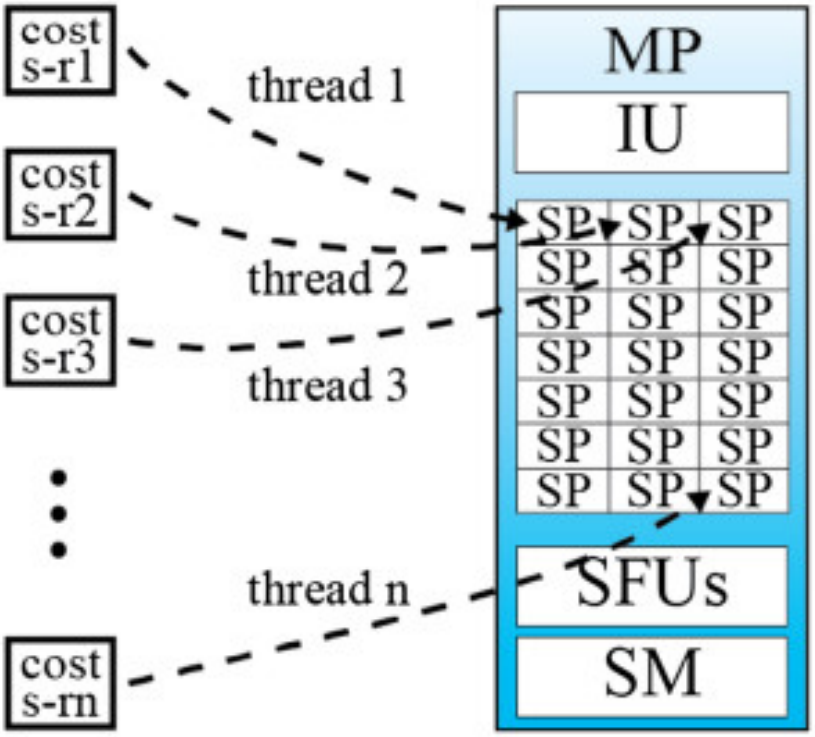


Figure 2.8 A parallel implementation of the Bellman Ford algorithm using a GPU.

CHAPTER 3: PERFORMANCE EVALUATION

This project evaluates the performance of the proposed GPU and CPU implementation of the BF algorithm in terms of execution time and memory usage.

Fig. 3.1 show the speedup of the proposed GPU implementation over that of the CPU implementation for different numbers of nodes and vertices. We observe that the GPU-based approach achieves much higher speedup over the CPU when the number of nodes is greater than 10000, even with CPU OpenMP implementation version. One of the main reasons for this is that the ratio of the data transfer time from the CPU to the GPU and vice versa at $V > 10000$ over the corresponding total execution time of the GPU-based implementation is much smaller than that at $V < 10000$, where the data transfer time is one of the performance bottlenecks for GPU computing.

The data transfer time between the device (GPU) and the host (CPU) is an important factor that limits performance. As shown in Fig. 3.2, the memory usage of GPU increases with much higher over the CPU when the number of nodes is large. This trend should be a concern as a trade-off between performance gains and memory usage.

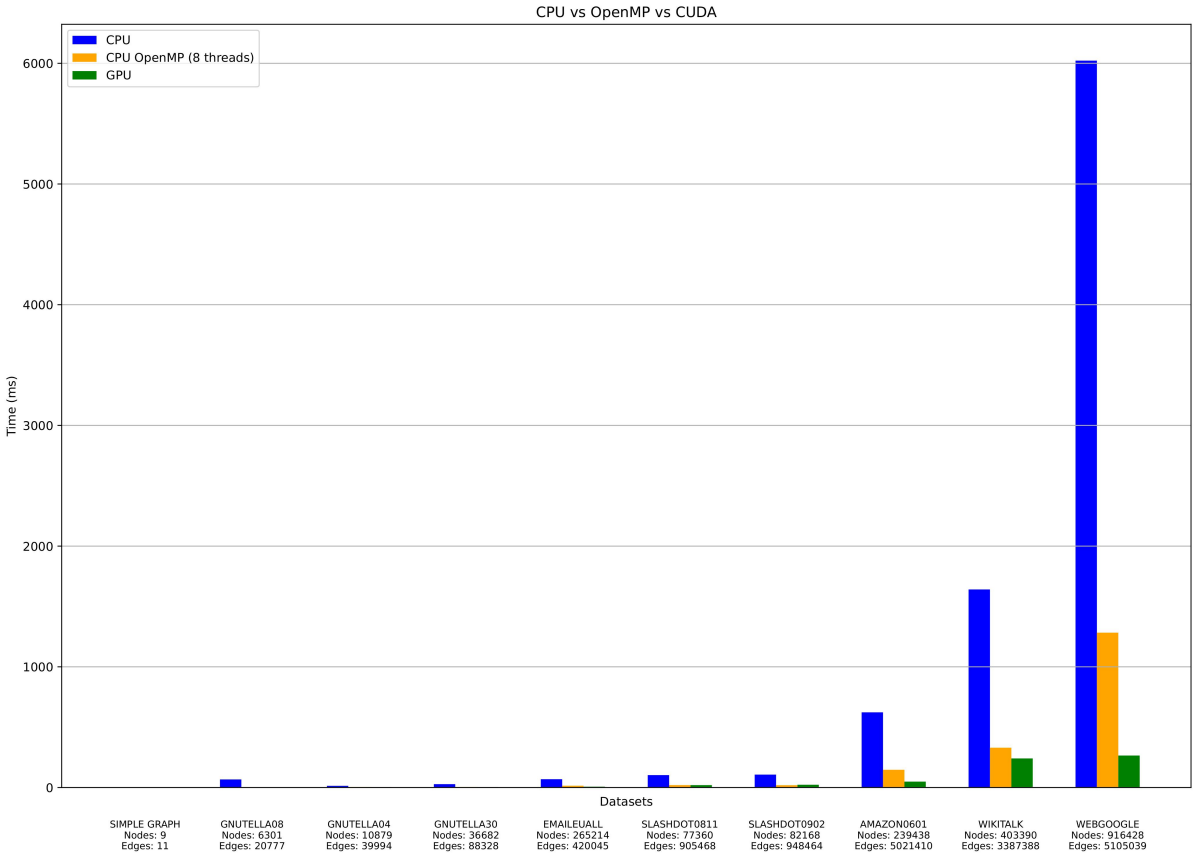


Figure 3.1 Execute time comparision between CPU and GPU.

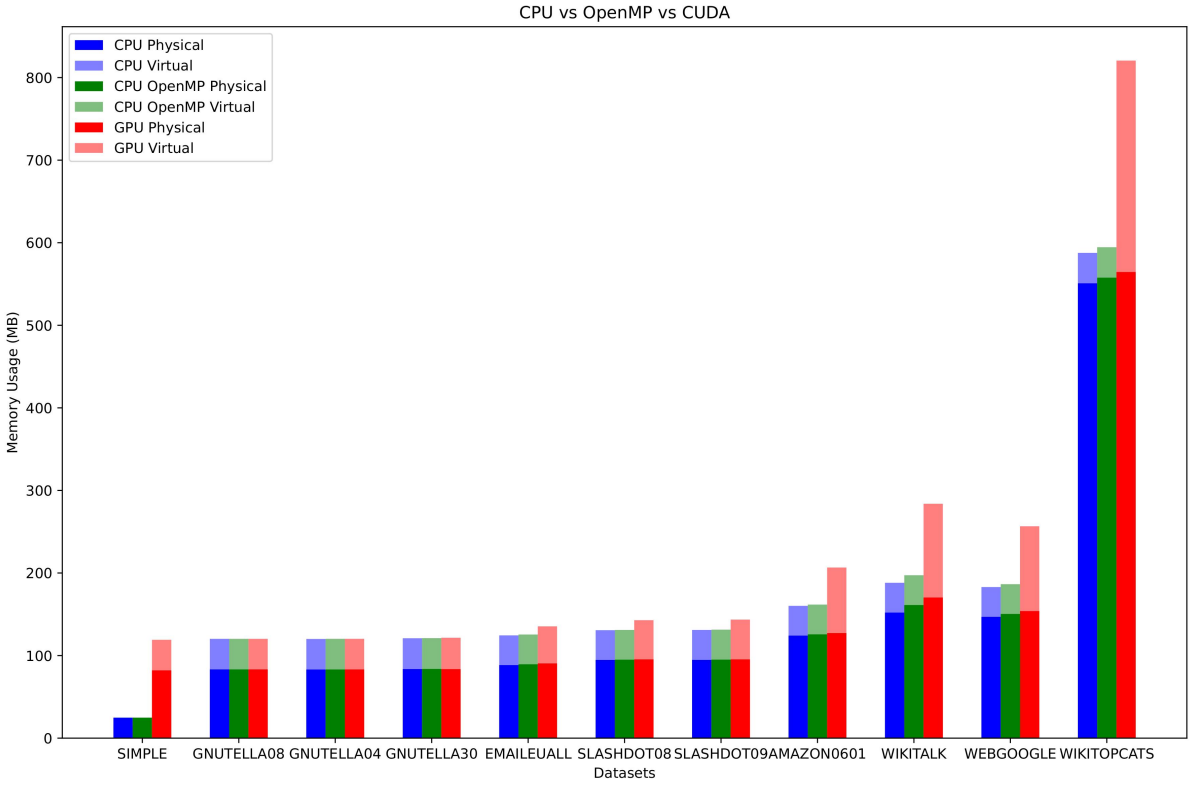


Figure 3.2 Memory usage comparision between CPU and GPU.

CHAPTER 4: CONCLUSIONS

This project presented a parallel implementation of the BF algorithm using a GPU. The performance and memory consumption of the proposed GPU approach were compared to those of an equivalent sequential CPU implementation. The experimental results showed that the proposed GPU approach is significantly faster than the CPU-based implementation, where the GPU achieved a speedup of multiple times that of the CPU implementation when the processed routing graph became denser. Moreover, the rising in memory consumption by using the GPU is a notable trade-off between performance gains and energy efficiency underscores the need to strike a balance when choosing between CPU and GPU implementations of algorithms like Bellman-Ford.

REFERENCES

[1] Bellman R (1958) On a routing problem. Q J Appl Math 16:87–90

[2] Bhattacharya A, Wu W, Yang Z (2012) Quality of experience evaluation of voice communication: an affectbased approach. Human-centric Comput Inf Sci 2(7):1–18

[3] Chang KD, Chen CY, Chen JL, Chao HC (2010) Challenges to next generation services in IP multimedia subsystem. J Inf Process Syst 6(2):129–146

[4] Chao J, Huang X, Wang X, Dang Y, Fu X (2011) Exploiting graphic processors for high performance IP lookup in software routers. IEEE INFOCOM, Shanghai, pp 301–305

[5] Chikkagoudar S, Wang K, Li M (2013) GENIE: a software package for gene-gene interaction analysis in genetic association studies using multiple GPU or CPU cores. BMC Res Note (20, 6). [online]: http://www. biomedcentral.com/1756-0500/4/158/figure/F1?highres=y

[6] Choi H, Son D, Kang S, Kim J, Lee H, Kim C (2013) An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. J Supercomput 65(2):886–902

[7] Chorianopoulos K (2013) Collective intelligence within web video. Human-centric Comput Inf Sci 3(10):1–16

[8] Divya Udayan J, Kim HS, Lee J, Kim JI (2013) Fractal based method on hardware acceleration for natural environments. J Converg 4(3):6–12

[9] Dynamic Programming, Set 23, Geeksforgeeks (2013, 27, 7). [online]: http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/

[10] Gong C, Liu J, Chen H, Xie J, Gong Z (2011) Accelerating the Sweep3D for a graphic processor unit. J Inf Process Syst 7(1):63–74

[11] Harvey JP (2009) GPU acceleration of object classification algorithms using NVIDIA CUDA. Master thesis, Rochester Institute of Technology, pp 1–76

[12] Huang J, Ponce S, Park S, Cao Y, Quek F (2008) GPU accelerated computation for robust motion tracking using the CUDA framework. 5th International Conference on Visual Information Engineering, Xian, pp 1–6

[13] Kang K, Dang Y (2011) Scalable packet classification via GPU metaprogramming. Design, Automation and Test in Europe Conference and Exhibition, Grenoble, pp 1–4

[14] Lalami ME, El-Baz D, Boyer V (2011) Multi GPU implementation of the simplex algorithm. IEEE International Conference on High Performance Computing and Communications, Banff, pp 179–186

[15] Luebke D (2008) CUDA: scalable parallel programming for high performance scientific computing. 5th IEEE International Symposium on Biomedical Imaging From Nano to Macro, Paris, pp 836–838

[16] Mu S, Zhang X, Zhang N, Lu J, Deng Y, Zhang S (2010) IP routing processing with graphic processors. IEEE Design, Automation Test in Europe Conference and Exhibition, Dresden, pp 93–98

[17] Nie DH, Han KP, Lee HS (2009) GPU-based stereo matching algorithm with the strategy of populationbased incremental learning. J Inf Process Syst 5(2):105–116

[18] NVIDIA (2013) CUDA programming guide, CUDA driver, toolkit, and SDK code samples (08, 07). [online]: http://www.nvidia.com/object/cuda get.html

[19] Sharma MJ, Leung VCM (2012) IP multimedia subsystem authentication protocol in LTE-heterogeneous networks. Human-centric Comput Inf Sci 2(16):1–19

[20] Shi L, Chen H, Sun J, Li K (2012) VCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans Comput 61(6):1–11

[21] Tian Y, Zhou B, Zhang YT, Chan KW (2011) Investigation on the use of GPGPU for fast sparse matrix factorization. J Int Counc Electr Eng 1(1):116–122

[22] Truong TT, Tran MT, Duong AD (2012) Improvement of the more efficient and secure ID-based remote mutual authentication with key agreement scheme for mobile devices on ECC. J Converg 3(2):1–10

[23] Tsai JC, Yen NY (2013) Cloud-empowered multimedia service: an automatic video storytelling tool. J Converg 4(3):13–19

[24] Zhang Y, Deng Y, Chen Y (2011) Hermes: an integrated CPU/GPU microarchitecture for IP routing. IEEE Design Automation Conference, New York, pp 1044–1049