# Graphics Processing Units (GPUs): Architecture and Programming: Final Project
# Comparing CUDA and OpenMP for GPU friendly applications

**Meghana Kankanala**
mk7747@nyu.edu

**Hyejin Kim**
hk3342@nyu.edu

**Zicheng Ren**
zr2050@nyu.edu

## 1 Introduction

OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming. OpenMP is gaining more and more popularity for shared memory architectures and recently started to support GPUs. However, CUDA has been the most popular parallel computing platform and application programming interface for GPUs. Given these different interfaces, it would be important for a programmer to use the right interface for different GPU friendly applications because it directly impacts the performance and efficiency. In this project, we analyze the different aspects of OpenMP and CUDA on various applications such as breadth first search, matrix multiplication, and sorting. We compare two versions–OpenMP and CUDA–of each application focusing on the programmability, runtime overhead, scalability, etc. Moreover, based on the experimental results and analysis, we study how to identify when to use OpenMP and when to use CUDA.

## 2 Experiment

We conducted experiments on a linux server equipped with Intel Xeon E5-2660 CPU@2.6GHz, GeForce RTX 2080 Ti GPU which has 11 GB memory, and CentOS 7 operating system. All code is written in C and compiled using gcc 11.2 and CUDA 10.2. Nvprof profiling tool, linux time command and OpenMP "omp get wtime" command are used for collecting results. We measured total execution time and profiled ratio of different behaviors of GPU for both CUDA and OpenMP version. For OpenMP version, we manually measured the time consumed only for computation part on the GPU threads since there was no profiling tool such as nvprof was provided.

### 2.1 Breadth-First Search (BFS)

BFS is an algorithm for traversing or searching tree or graph data structures. It explores all the nodes at the present depth prior to moving on to the nodes at the next depth level. BFS is helpful in many areas such as state space searching, graph partitioning, automatic theorem proving, and is one of the most used graph algorithm [2].

We choose the problem size of BFS based on the number of vertices in the graph ranging from 30 vertices to 30000 vertices. We didn't use larger data sets because the server didn't allow us to use our graph generator to generate larger graph (the job of generating 300000-vertex graph was killed). The CUDA version of BFS is based on the implementation from ispass2009-benchmarks/BFS which implemented an algorithm described in "Accelerating Large Graph Algorithms"[1,2]. To run this program successfully with CUDA 10.2, We removed some functions which are no longer available on CUDA since version 5.0. We implement the OpenMP version of BFS by imitating the functionality of

CUDA version as close as possible while using OpenMP directives to offload work to GPUs instead of using CUDA.

### 2.2 Matrix Multiplication

Matrix multiplication is a parallel algorithm to calculate the product of two square matrices. It is one of the time consuming problems that require huge computational resources to improve its speedup. A comparison between parallel OpenMP version and CUDA algorithm is carried out to evaluate the performance.

In the CUDA version of the implementation, inside the main function memory allocation and deallocation of host and device variables occurs and it takes user input data as matrix sizes. matrixMulKernel is kernal function that computes the matrix multiplication and it employs multiple threads to do the computation simultaneously. The aim of this comparison was to determine which parallel programming results in better performance of runtime speed in matrix multiplication. For analysis, we have performed the experiment on different matrix sizes, from 10*10 to 1000*1000.

### 2.3 Quicksort

QuickSort is a divide and conquer algorithm which recursively picks an element as pivot and partitions the given array around the picked pivot. At each process of partitioning the array, it puts pivot element at its correct position in sorted array and put all smaller elements before the pivot, and put all greater elements after the pivot.

This process of sorting a partition of array given a pivot value can be done faster by leveraging multiple threads. Also, each partition of array can be processed in parallel way. For a parallelized quicksort utilizing GPUs, we call the kernel function with multi-threads and inside of the kernel function it recursively calls two kernel functions for each partitioned array. Each kernel function runs in parallel and returns asynchronously.

The CUDA version of implementation initially launches a kernel function and creates asynchronous stream using 'CUDAStreamCreateWithFlags' when making recursive kernel function calls. After the initial kernel function returns, 'CUDADeviceSynchronize' is called to wait for all the threads finish and then write back to the original array. The openmp version initially defines the number of threads and call the initial function inside of 'pragma omp target data' and 'pragma omp parallel' clauses. Inside of this function it makes asynchronous and parallel recursive function calls using 'pragma omp task' clause.
We conducted experiments for different length of array, from 10 to 100,000,000.

## 3 Result

### 3.1 Breadth-First Search

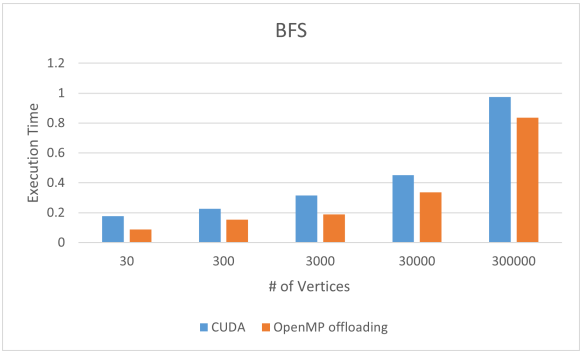


Figure 1: total execution time OpenMP vs CUDA graph

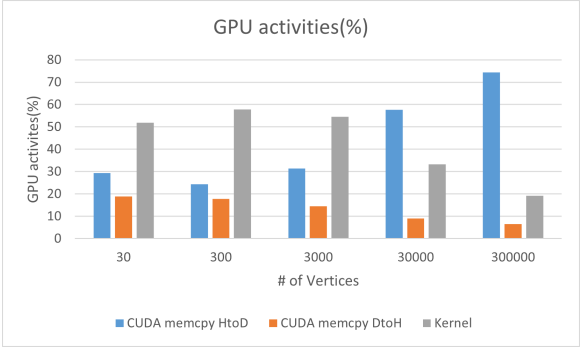


Figure 2: nvprof profiling of CUDA program

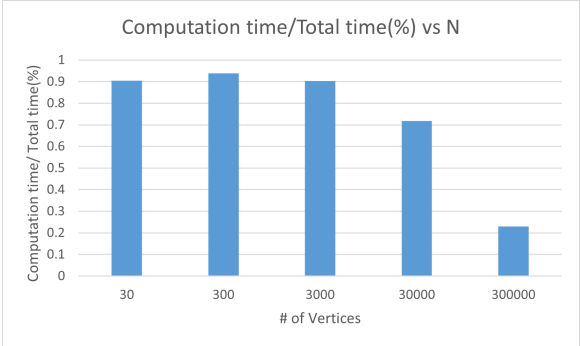


Figure 3: Computation-only ratio of OpenMP program

### 3.2 Matrix Multiplication

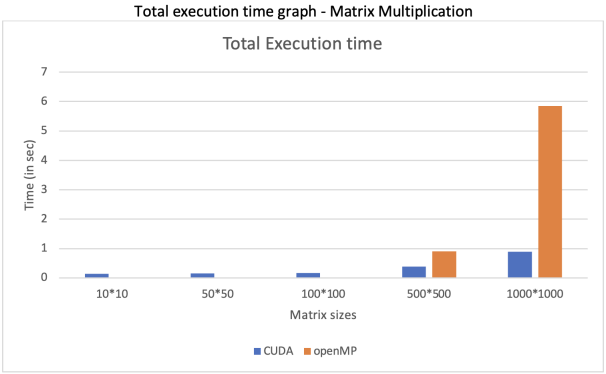


Figure 4: total execution time OpenMP vs CUDA graph

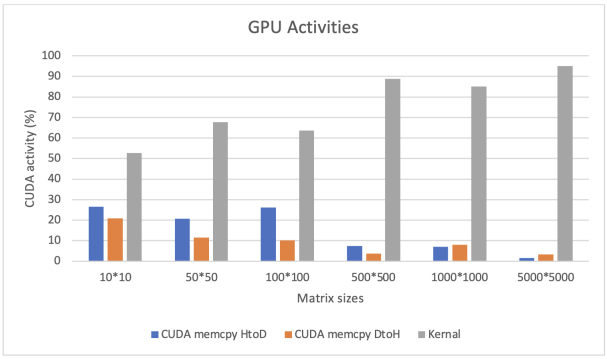


Figure 5: nvprof profiling of CUDA program

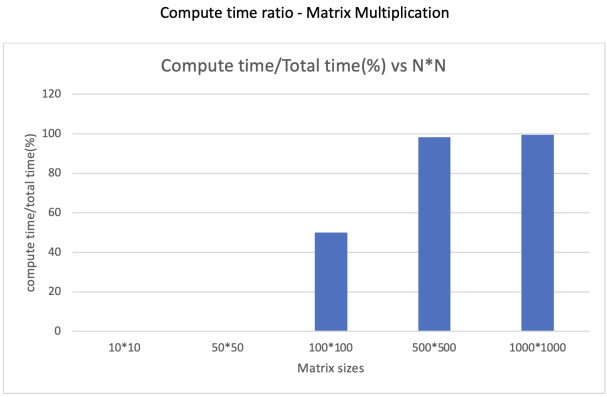


Figure 6: Computation-only ratio of OpenMP program

### 3.3 Quicksort

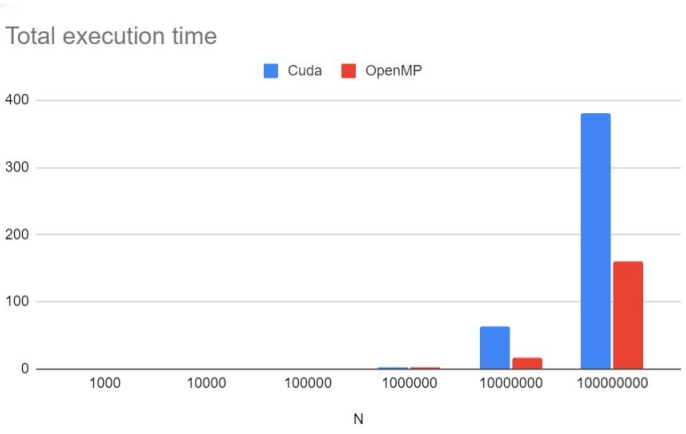


Figure 7: total execution time OpenMP vs CUDA graph

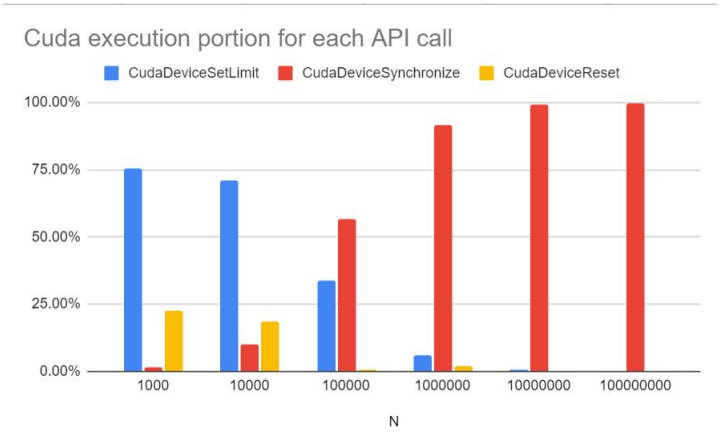


Figure 8: nvprof profiling of CUDA program

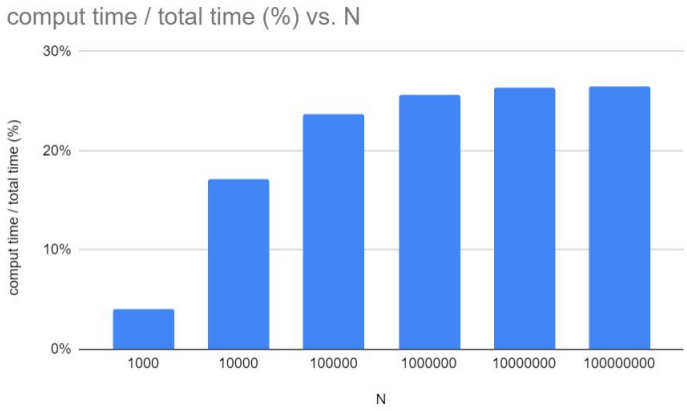


Figure 9: Computation-only ratio of OpenMP program

# 4 Analysis

## 4.1 Programmability

Regarding code structures, one big difference between OpenMP and CUDA is that OpenMP follows a top-down model while CUDA follows a bottom-up model. Coding with CUDA is more like micro-controlling what each thread will do in the GPU using the kernel functions, while OpenMP is a high-level language that distributes work in a for loop to each thread in the GPU using the target clause.

The way of treating data variables is also different. In short, CUDA treats the host and the devices equally, but OpenMP is host centered. CUDA establishes a clear boundary between the host and the devices that memory has to be allocated on the sides where data variables touch. Data variables representing the same thing but sit in different sides need to have different names and treated differently. As a high-level language, OpenMP treats data variables differently that data variables used in both sides but represent the same data structure are considered as one single data variable. Movement of data variables between host and devices can be easily defined by a map-type clause such as "tofrom" in OpenMP, but for CUDA, it requires "CUDAMemcpy" along with direction for each transfer between host and devices.

As a result, for programming GPU, OpenMP target offload is easier to start with as it inherits the good abstraction and programmability from OpenMP. However, when it comes to sophisticated thread management, CUDA has its advantages.

### 4.2 Runtime overhead

#### 4.2.1 Cuda

Overhead comes from different places for different applications. Based on the nvprof results in Figure 2 and 5, as problem size grows, most of the overhead in GPU activities comes from "CUDA memcpy HtoD". But for larger problem size, the major overhead for matrix multiplication was computation part on kernel function, while BFS overhead caused by "CUDA memcpy HtoD" increased.

For quick sort, according to Figure 8, the overhead from "CUDADeviceSynchronize" dominates. "CUDADeviceSynchronize" is called on a host to wait for all the running threads on the device kernel to finish. This means each kernel function(sorting each partitioned array) takes longer time than same function on OpenMP.

#### 4.2.2 OpenMP

For BFS, it is clear that the major bottleneck comes from memory access(or other potential parts) for large problem size since the time consumed for computation decreased. For matrix multiplication and quicksort, major bottleneck was computation part since Figure9 and Figure6 shows saturating tendency of computation-only part.

### 4.3 The quality of the final code

#### 4.3.1 Speed

The speed of the program showed different tendency for different applications. According to Figure 1 and Figure 7, OpenMP version of codes were faster than CUDA version for BFS and Quicksort, while CUDA version was faster for matrix multiplication. This is probably caused by the fact that matrix multiplication doesn't do branching at all, while the other two applications need a lot of if statements to determine the next stage of the computation for each thread, causing large amout of overhead for CUDA. OpenMP version achieved better performance than CUDA version for smaller input sizes in Matrix multiplication. As the input size increased, CUDA version was faster.

#### 4.3.2 Executable Size

CUDA executable files were bigger than OpenMP executable files for all three applications. Especially for BFS, the CUDA executable file is 20X larger than the OpenMP executable file.

#### 4.3.3 Scalability

For the small problem size, OpenMP version showed faster speed than CUDA version with the same problem size in all three applications. However, as we increased problem size, CUDA version was much faster than OpenMP version with the same problem size for matrix multiplication while BFS and Quicksort showed opposite result.

Thus, for matrix multiplication CUDA version scales better than OpenMP version. According to Figure 5(nvprof result of matrix multiplication CUDA version), as the problem size increases the ratio of computation part of operation on the kernel keep increasing, while on the OpenMP version the computation-only ratio saturated after the problem size 500*500. This means OpenMP version suffers from cost of memory access or other potential bottlenecks when increasing problem size. CUDA version of matrix multiplication will scale better when we add more hardware resources.

For BFS, OpenMP version scales better than CUDA version. According to Figure 2(nvprof result of BFS CUDA version), as we increased the problem size the time consumed by CUDAMemcpy increased to over 70 percent. As we discussed in 4.2, this bottleneck caused by memory access cannot be mitigated by adding more compute resources. On the other hands, Figure 3 shows the computation time consumed on each kernel for OpenMP version decreased as we increased the problem size. This means OpenMP version benefits more from parallelism on a GPU and will scale better when used more hardware resources.

Also for quicksort, OpenMP version scales better than CUDA version. According to Figure 8(nvprof result of quicksort CUDA version), as we increased the problem size about 99 percent of time was

consumed by CUDADeviceSynchronize. As we discussed in 4.2, waiting for all the threads to finish is the major bottleneck for this implementation. However, this synchronization cost would not be resolved by leveraging more hardware resources.

## 5 Conclusion

Comparative analysis has shown that CUDA based parallel matrix multiplication algorithm runtime speed is better than OpenMP matrix multiplication algorithm speed for bigger problem sizes, whereas OpenMP version of BFS and quicksort achieves better performance than CUDA version. Irrespective of the applications, openmp performs better than the CUDA version for smaller problem sizes. For larger problem sizes, CUDA is preferred when there is no branch divergence and openMP is favored when there is a complicated implementation required and involves more branch divergence.

## 6 References

1. https://github.com/gpgpu-sim/ispass2009-benchmarks/tree/master/BFS

2. P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In HiPC, pages 197–208, 2007.

3. Fazlul Kader Murshed Nawaz, Arnab Chattopadhyay, Kirthan G J, Girish D Mane1, and Rohith N Savanth. Comparison of Open MP and CUDA.

4. Diaz, Jose Manuel Monsalve et al. "Analysis of OpenMP 4.5 Offloading in Implementations: Correctness and Overhead." Parallel Comput. 89 (2019): n. pag.