Comparative Analysis of CUDA and OpenMP for GPUs

Anoushka Gupta
New York University
ag8733@nyu.edu

Arunima Mitra
New York University
am13018@nyu.edu

***Abstract*— As deep learning models have grown in complexity and size, the need for accelerated and high-performance computing is paramount. Most cloud services rely heavily on the parallel processing ability of the Graphics Processing Unit for computationally heavy workloads. In the last two decades, NVIDIA-developed CUDA has served as the foundation of GPU programming and is the most widely adopted language. However, in the last few years languages like OpenMP and OpenCL have also started supporting GPU offloading. With the choice of multiple languages and frameworks, it is necessary for each developer to discern what is the optimal choice for their specific requirements. In this paper, we conduct a comprehensive comparative analysis of CUDA and OpenMP on the basis of their programmability, scalability, performance and overheads for five diverse and varied algorithms: Sieve of Eratosthenes, Convolution Operation, Bellman-Ford, N-Queens and Kmeans Clustering. The findings highlight OpenMP's user-friendly programming environment, while CUDA excels in scalability and has faster kernel launch times. CUDA affords greater control to programmers, but with larger executable files. With the continued growth of the OpenMP community, we foresee its performance catching up to CUDA, providing developers with an increasingly attractive option.**

I. Introduction

The evolution from single-core CPUs to multi-core CPUs and the subsequent transition to GPUs represents a response to the challenges and limitations encountered in traditional processor architectures. Initially, attempts to enhance performance through higher clock speeds encountered issues with power consumption and heat dissipation. As transistor sizes neared limits, the industry embraced multi-core CPUs for parallel processing, alleviating clock speed constraints. While multi-core architectures addressed certain parallelism challenges, they were not optimal for highly parallel tasks due to factors such as shared caches and limited scalability. In contrast, GPUs originally designed for graphics rendering, which were revolutionised with efficient cores, excel at massive parallelism. Overcoming SIMD limitations in CPUs, GPUs parallel architecture and efficient heat management positioned them as potent accelerators, marking a computational paradigm shift.

There are several programming languages utilised for GPU programming today, reflecting the diverse needs and preferences of developers. Introduced in 2007 by NVIDIA, CUDA or Compute Unified Device Architecture, stands out as a preeminent and widely adopted GPU framework, leveraging the capabilities of NVIDIA GPUs. OpenCL, established in 2009 by the Khronos Group, represents a prominent open and royalty-free standard for cross-platform parallel programming. OpenMP, originating in 1997, expanded its support to accelerators in 2013 with the release of version 4.0, accommodating multiple languages and diverse hardware platforms.

II. Related Work

Extensive research and effort have been dedicated to the comparison of various GPU programming languages. The paper by Akihiro Hayashi et al.[1] concentrates on evaluating the performance of OpenMP's target constructs on GPUs with the aid of compiler optimizations. They assess the performance of six benchmark algorithms. Ganesh G Surve et al.[2] implemented the CUDA implementation of the Bellman Ford algorithm and despite the substantial performance improvements observed, this study overlooks the runtime overhead and lacks adequate code profiling.
Several comparative studies have also been done in this field. The paper by Memeti, Suejb, et al.[3] compare OpenMP, OpenCL, Cuda and OpenACC on their performance, productivity and energy consumption. They employed their own tool to determine the percentage of code lines that were required to parallelize the code using a specific framework. x-MeterPU was used to gauge energy consumption and performance was evaluated on Intel Xeon E5 Processors. Authors[4] adopt different performance metrics to compare the GPU languages. They evaluate the languages on the basis of runtime performance,speedup and portability of applications between languages. Mikhail, Alexey[5] benchmarked the matrix multiplication algorithm by analysing the impact of the programming language on achievable GPU memory bandwidth.

III. Objectives

The primary aim of this study is to gain insights into the most suitable language for a given workload. We focus on CUDA and OpenMP, and analyse them on the basis of the following paradigms-

1) Programming Effort: Assessing the ease of programmability i.e. simplicity and user-friendliness of programming with each language.

2) Performance Overhead: Examining the impact on overall system performance associated with the use of CUDA and OpenMP.

3) Kernel launch times- Analysing the speed and efficiency of launching GPU kernels in both languages.

4) Program size: Evaluating the size of programs developed using CUDA and OpenMP.

5) Scalability: Investigating the ability of each programming approach to scale with increasing workloads or computational demands.

6) Learning Curve & Control : Assessing which programming language provides any developer a more accessible entry point for parallel programming.

IV. Problem Definition & Methodology

A. Sieve of Eratosthenes

The Sieve of Eratosthenes, devised by the ancient Greek mathematician Eratosthenes, is an efficient algorithm to identify all prime numbers up to a certain limit. Its primary use lies in efficiently generating prime numbers, crucial for cryptography algorithms, number theory studies, and optimization problems. The algorithm's structure allows for parallelization, making it suitable for distributed computing environments.

We start by defining an array of size N, presuming all numbers to be prime initially. Subsequently, we take these numbers one by one, systematically marking their multiples as composite numbers. The first prime number is 2, so we mark all multiples of 2, ranging from 4 to N, as non-prime. Subsequently, the next prime number, 3 is picked up and all multiples of 3, from 9 to N are marked non-prime, and this pattern continues. The time complexity of the algorithm, given an input of N, is $O(N * log(log(N)))$. Notably, each segment of the sieve (which performs marking composite numbers) can be processed independently, presenting an opportunity for parallel processing.

Algorithm 1 Sieve of Eratosthenes Algorithm

```
for i = 0 to N do
    composite[i] ← FALSE
end for
for i = 2 to √N do
    if not composite[i] then
        j ← i × i
        while j ≤ N do
            composite[j] ← TRUE
            j ← j + i
        end while
    end if
end for
```

Parallelization Strategy: The inner loop which is responsible for marking composite numbers is an independent task and hence can be parallelized. On the other hand, the iterations in the outer loop are dependent on successful completion of previous iterations and therefore, cannot be offloaded to the GPU. For CUDA, the inner loop execution is carried out by each kernel. We establish the number of blocks based on the number of multiples of the prime number up to N, that is the number of inner loop iterations. For OpenMP, we use the *#pragma omp target teams distribute parallel for* directive to offload the execution to the device.

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Fig. 1: Sobel Filter

B. Convolutional Operation

The convolution operation plays a fundamental role in various fields such as signal processing, gradient computation, image analysis and deep learning. It is a type of matrix operation, consisting of a filter, which is a small matrix of weights and an input matrix. The filter slides over the input matrix performing element-wise multiplication with the part of the input it is on, followed by summing the results into an output matrix. The output matrix is also called an output feature map. A filter in a convolution operation is used to extract specific features or patterns of the input data. Here,we have used the Sobel Filter 1 as our convolution filter. The Sobel filter is predominantly used for vertical edge detection in images. The time complexity of the convolution operation is $O(N^2)$ where N is the size of the square input feature map or matrix.

Algorithm 2 Convolutional Operation Algorithm

```
for i = 0 to N − 2 do
    for j = 2 to N − 2 do
        sum ← 0
        for m = 0 to K − 1 do
            for n = 0 to K − 1 do
                sum ← sum + matrix[i + m][j + n] × filter[m][n]
            end for
        end for
        output[i][j] ← sum
    end for
end for
```

Parallelization Strategy: In both the OpenMP and CUDA code we have parallelized the two loops which iterate over a 3x3 local neighbourhood of the input feature map. In OpenMP the loops are parallelized by using the *#pragma omp distribute parallel for collapse(2)* directive after offloading the data to the GPU. In the CUDA code, a kernel is launched to parallelize these two loops. The computation for each of these local neighbourhoods is done independently which helps increase the speed compared to the sequential

code. Each thread is thus responsible for a linear complexity workload.

C. *Bellman-Ford Algorithm*

Bellman-Ford algorithm is an algorithm used to determine the shortest path between a given source vertex and every other vertex of the graph. It is immensely useful due to its versatility- it can be used for negative and positive weights whereas other shortest path algorithms like Dijkstra's Algorithm can only be used when the graph contains positive edge weights. Due to this versatility it is used extensively in network routing, optimization and transportation planning. Bellman-Ford algorithm also detects and reports negative cycles. A negative cycle in a graph is a cycle where the edge weight sum is negative or less than zero.

The algorithm works by iteratively relaxing edges in the graph. In each iteration, it updates the distance of each vertex from the source vertex. For relaxation, the distance estimate of vertex v from the source is compared to the sum of the distance estimate of vertex u and the edge weight between vertex u and v. If this sum is less than the distance estimate of vertex v it indicates that a shorter path has been found and thus the distance estimate of vertex v is updated accordingly. This relaxation step is done a maximum of $N-1$ times where N is the number of vertices in the graph because a shortened path between source and destination vertex can have at most $N-1$ edges. The time complexity of this algorithm is $O(N*E)$ where N is the number of vertices and E are the number of edges in the graph.

Algorithm 3 Bellman-Ford Algorithm

```
procedure BELLMAN-FORD(graph G, source vertex s)
    negative_cycle ← false
    for each vertex v in G do
        dist[v] ← ∞
    end for
    dist[s] ← 0
    for i ← 0 to |V(G)| − 1 do
        for each edge (u, v) in G do
            if dist[u] + weight(u, v) < dist[v] then
                dist[v] ← dist[u] + weight(u, v)
            end if
        end for
    end for
    for each edge (u, v) in G do
        if dist[u] + weight(u, v) < dist[v] then
            negative_cycle ← true
        end if
    end for
    if not negative_cycle then
        for each vertex v in G do
            print "Distance to vertex v: dist[v]"
        end for
    end if
end procedure
```

Parallelization Strategy: In each relaxation iteration, the update of the distance estimate is independent and can be parallelized. Thus, the loops which iterate over all pairs of vertices are parallelized in both OpenMP and CUDA. In OpenMP the *#pragma omp target map* directive is first used to offload data to the device and *#pragma omp teams distribute parallel for collapse(2)* directive is used to parallelize the loops. We also make use of the *#pragma omp atomic write* directive while updating the distance estimate to prevent race conditions. The *#pragma omp barrier* directive is used after each relaxation iteration to ensure that all threads complete one iteration of updates before moving to the next iteration. In CUDA, each relaxation iteration is executed by launching a kernel. After each kernel launch we use *cudaDeviceSynchronize()* which enables the host to block further execution (execution of next iteration) until all GPU operations are completed.

D. *N-Queens Algorithm*

The N-Queens problem is a classic problem in chess and computer science that involves placing N chess queens on an $N \times N$ chessboard in such a way that no two queens threaten each other. As queens in chess have the ability to attack horizontally, vertically, or diagonally, the task is to strategically place the queens on the board in a manner that avoids any shared rows, columns, or diagonals. The N-Queens puzzle finds applications across various domains beyond academia where there is a requirement for generating multiple permutations. It can be analogously applied for scheduling, constraint satisfaction problems, resource allocation, network design, and even cryptography. To optimise memory usage, our algorithm accommodates only one queen per row, thereby, simplifying the need to examine conflicts solely in the diagonal and horizontal directions. It is evident that there exist N! permutations for the queens. The subsequent phase of the algorithm entails individually evaluating these N! chessboards and tallying the number of valid chessboard configurations.

Algorithm 4 N-Queens Algorithm

```
procedure PLACEQUEENSANDCHECKVALIDITY(N)
    Initialize N × N board as a blank matrix
    for row = 1 to N do
        for col = 1 to N do
            PLACEQUEENROWWISE(board, row, col)
        end for
    end for
    for queen = 1 to N do
        if HASVERTICALCONFLICT(queen) then
            return false
        else if HASDIAGONALCONFLICT(queen) then
            return false
        end if
    end for
    return true
end procedure
```

Parallelization Strategy: The task of generating different chessboard configurations is sequential. However, we are able to divide the search space and distribute different chessboard configurations among the GPU cores. Hence, the task of checking the validity of a particular board configuration is parallelized. In the CUDA code, the kernel's logic involves checking the validity of solutions, and the calculations are parallelized across multiple threads to explore different possibilities concurrently. We use CUDA's *atomicAdd* operation to avoid the race condition of simultaneous updates to the counter value by more than one threads. After each kernel launch we also call the method *cudaDeviceSynchronize()*, to enforce a synchronisation point. In the OpenMP code, we use the directive *#pragma omp parallel for* to parallelize the loop which checks validity of different boards. We also use *#pragma omp atomic* to enforce atomicity for the increment solution, should we find a valid board configuration. This ensures that only one thread can increment the counter at a time.

E. *K-Means Algorithm*

K-Means clustering is an algorithm used in unsupervised machine learning to partition data into distinctive groups based on some similarity. It is often used for image segmentation to create boundaries between different objects. K-means clustering is based on an iterative approach of clustering data points into clusters followed by refining the centroids. Initially K centroids are chosen at random from the dataset, where K represents the number of classes or clusters we want to cluster the data into. In each iteration, we calculate the distance between each data point and centroid, assigning points to the cluster with the closest centroid. The centroids of the clusters are recalculated based on the mean of the data points in the cluster. This process continues until convergence where assignments of points to clusters don't change and the centroids stabilise.

Algorithm 5 K-Means Clustering

for $k \leftarrow 1$ to K **do**
    $\mu_k \leftarrow$ random point
end for
while not converged **do**
    for $n \leftarrow 1$ to N **do**
        $c_n \leftarrow \arg\min_k \|\mu_k - x_n\|$
    end for
    for $k \leftarrow 1$ to K **do**
        $\mu_k \leftarrow mean(\{x_n : c_n = k\})$
    end for
end while
return c

Parallelization Strategy: Each iteration consists of two parts: assigning points to clusters and recalculating centroids. For each data point, the task of assigning it to a cluster is independent and can be distributed. While assigning clusters, the loop which iterates over each data point is parallelized. The number of centroids(K=10) are very small compared to the number of data points and thus loops iterating over centroids are not parallelized. In OpenMP, the data is offloaded to GPU using the *#pragma omp target teams num_teams(512) map* directive followed by *#pragma omp distribute parallel for* to parallelize the loop. While assigning clusters we also keep track of the size of each cluster, i.e the number of data points assigned to each cluster. This is used later while recalculating the centroids. The *#pragma omp critical* directive is used while updating the size of cluster to ensure that only one thread accesses the data at a given time to maintain data integrity. In CUDA a kernel is launched for assigning new points to clusters, followed by a *cudaDeviceSynchronize()*, followed by a centroid recalculation kernel. This continues until the centroids converge.

V. Experimental Setup

We implement three versions of the above algorithms -

1) Sequential
2) CUDA
3) OpenMP GPU version

A. *Setting up the environment*

We conduct our experiments on the CIMS cuda3 machine. The CPU and GPU specifications are shown in Table I and Table II respectively. We use CUDA *v11.6*. We use the GCC compiler *v4.9.2* for CUDA code compiling with *nvcc* and we use GCC *v12.2* for OpenMP compiling.

Specification	Value
Model:	Intel(R) Xeon(R) Gold 5118 CPU
Architecture:	x86_64
CPU(s):	48
Thread(s) per core:	2
L1d cache:	32K
L1i cache:	32K
L2 cache:	1024K
L3 cache:	16896K

TABLE I: CPU Specifications

Specification	Values
GPU Model:	NVIDIA TITAN V
Number of GPUs available:	2
Memory Size:	12GB

TABLE II: GPU Specifications

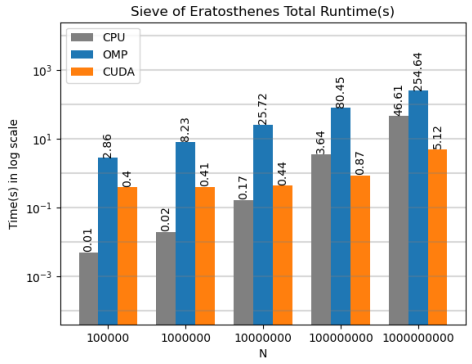
B. *Profiling*

We use *nvprof* for profiling our parallel codes and capturing important metrics. *nvprof* is a command-line profiler for GPU programming offered by NVIDIA. It is part of the NVIDIA CUDA Toolkit, well suited GPU accelerated applications. While *nvprof* is commonly associated with CUDA, it supports profiling and analysing OpenMP applications on GPUs as well. The primary purpose of *nvprof* is to help developers understand the performance characteristics of

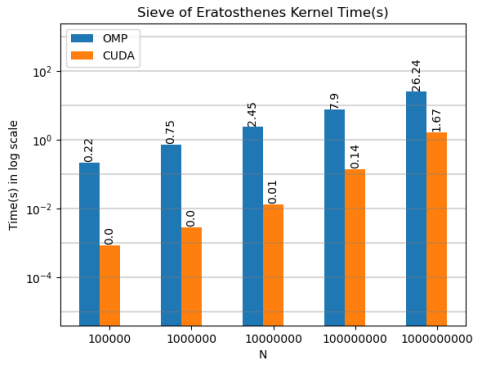
their applications by providing detailed profiling information. For our study, we use *nvprof*'s profiling information to analyse four key metrics - kernel launch times, total running times, kernel execution times, and the memory transfer times between device and host.

VI. Experiments and Analysis

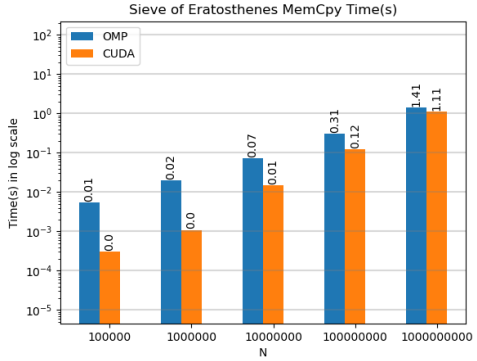
A. *Sieve of Eratosthenes*



(a) Sieve of Eratosthenes Total Runtime



(b) Sieve of Eratosthenes Kernel Time



(c) Sieve of Eratosthenes MemCpy Time

Fig. 2: Sieve of Eratosthenes

We consider problem sizes from $N = 10^5$ to 10^9 and compute all prime numbers between 2 to N. We profile the code and analyse various metrics such as the run-time, kernel-time and MemCpy time for each problem size. Refer Figure 2.

Total Runtime: The OpenMP implementation appears to be the least efficient when compared to the sequential and CUDA versions. During code profiling, it became apparent that the OpenMP implementation experienced significant overhead due to *cuMemAlloc* and *cuMemFree*. For instance, with a parameter value of $N = 10^9$, the combined duration of these two metrics amounted to approximately 226 seconds. This represents a figure eight times higher than the kernel time and constitutes over 85% of the overall runtime and it appears to be the primary reason for longer OpenMP running time. For small problem sizes, there is very small difference between the two speedups. However, as the problem size increases, we achieve a speed-up of nearly 9x for CUDA but up to 18x slowdown for OpenMP.

Kernel time: Generally, the CUDA kernel outperforms the OpenMP counterpart. For larger problem sizes, the kernel time is comparable to the total sequential running time (within order of 10).

MemCpy time: The MemCpy times exhibit similar performance for both OpenMP and CUDA.

B. *Convolution Operation*

We perform the convolutional operation using the Sobel Filter on input matrices of varying sizes. We experimented with N=10^3, 10^4, 2×10^4, 3×10^4 and 3.5×10^4. All the input matrices were unit matrices. We used various metrics like kernel run time,total run time and MemCpy time to analyse the experiment. Refer Figure 3.

Total Runtime: Initially for smaller problem sizes ($N = 10^3$ and 10^4) CUDA, OpenMP and sequential have similar total runtime which are comparable. For bigger problem sizes, the sequential version becomes increasingly slower while CUDA and OpenMP remain comparable. OpenMP and CUDA are ~ 4.5x faster than sequential code.

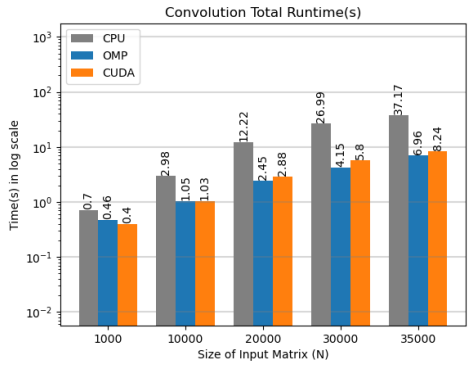
Kernel Runtime: In general, CUDA kernel runtime is faster than OpenMP kernel runtime. For bigger problem sizes, ($N = 2 \times 10^4$,3×10^4, 3.5×10^4) CUDA is about $\sim 10^6$x faster than OpenMP.

MemCpy Time: The MemCpy time which includes Device to Host, Host to Device and *cudaMemcpy* API call is very similar for both CUDA and OpenMP.

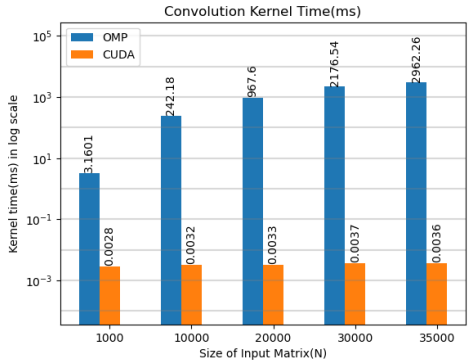
C. *Bellman-Ford Algorithm*

We perform the Bellman-Ford algorithm for graphs which have different total number of vertices. We experimented with N=100, 1000, 2000 & 3000.We profile the code and analyse various metrics such as the run-time, kernel-time and MemCpy time for each problem size. Refer Figure 4

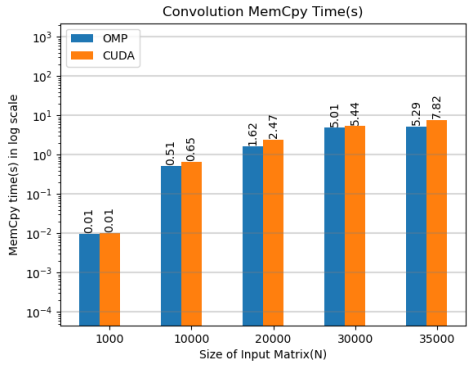
Total Runtime: For a small input size ($N = 100$), the sequential version is much faster than OpenMP and CUDA due to the large memory overheads associated with GPU offloading. For bigger problem sizes, ($N = 1000$,2000,3000) CUDA outperforms both sequential and OpenMP versions. For N=3000, CUDA is 100x faster than its corresponding OpenMP version. OpenMP and sequential versions are comparable even for bigger problem sizes and there isn't much speedup observed.



(a) Convolution Operation Total Runtime



(b) Convolution Operation Kernel Time

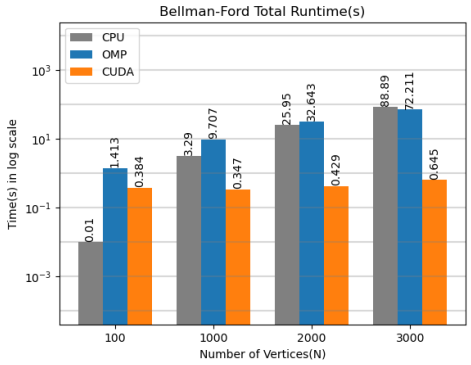


(c) Convolution Operation MemCpy Time

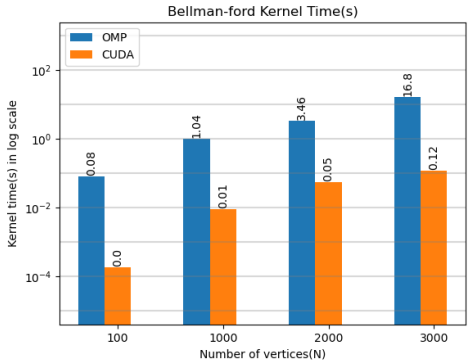
Fig. 3: Convolution Operation

Kernel Runtime: Generally CUDA kernel time is significantly smaller than the OpenMP kernel time. For $N = 3000$, the CUDA kernel is ∼140x faster than the OpenMP kernel.

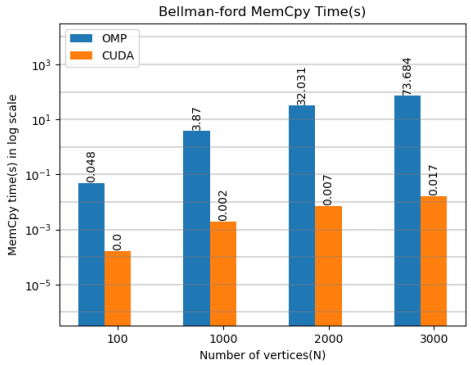
MemCpy Time: Generally, CUDA is quicker when transferring data from Device to Host and from Host to Device for all problem sizes for the Bellman-ford algorithm.



(a) Bellman-Ford Total Runtime



(b) Bellman-Ford Kernel Time



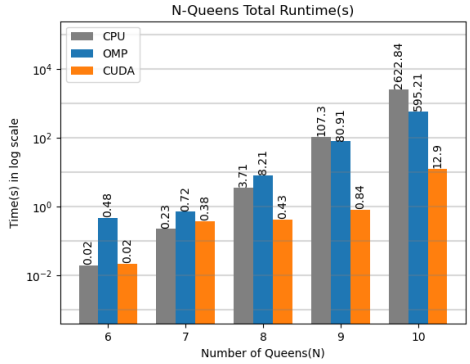
(c) Bellman-Ford MemCpy Time

Fig. 4: Bellman-Ford Algorithm

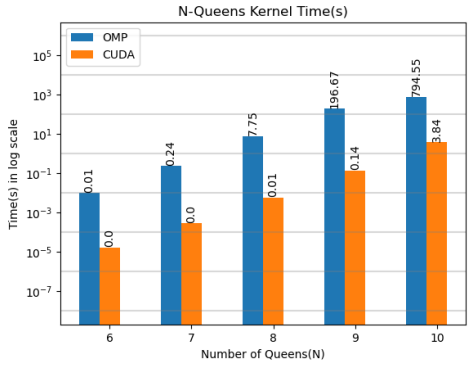
D. N-Queens

We consider different number of queens in the board, from $N = 6$ to 10 to compute the number of valid configurations

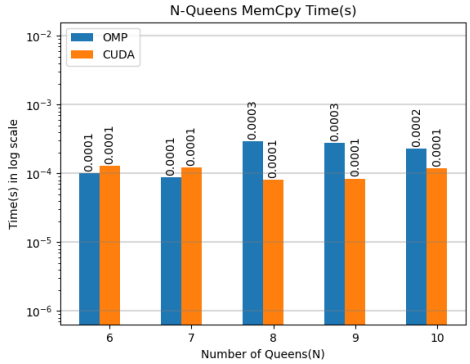
for placing N queens on an NxN chessboard. We profile the code and analyse various metrics such as the run-time, kernel-time and MemCpy time for each problem size. Refer Figure 5 .



(a) N-Queens Total Runtime



(b) N-Queens Kernel Time



(c) N-Queens MemCpy Time

Fig. 5: N-Queens Problem

Total Runtime: Initially, when the problem size is small (5x5 chessboard or 5^5 board configurations), there is no speedup i.e. OpenMP and CUDA both perform worse than sequential. This is due to the memory copy overheads. But as the problem size becomes larger (N = 9, 10; board configurations = 9^9 , 10^{10}), OpenMP and CUDA both perform better than the sequential.

Kernel time: In general, the kernel time of CUDA is significantly lesser than that of OpenMP. For larger problem sizes (N = 9, 10; board configurations = 9^9 ,10^{10}), CUDA kernel time is 200x faster than OpenMP.

MemCpy Time: The MemCpy time which includes Device to Host, Host to Device and cudaMemcpy API call exhibit similar performance for both OpenMP and CUDA.

E. Kmeans Clustering

We perform Kmeans clustering for K=10 clusters for varying number of input coordinates(N). We experimented with N=$1k, 10k, 100k, 1000k$ & $2000k$. We profiled the codes using *nvprof* and analysed the total runtime, the kernel runtime and the memcpy runtime. Refer Figure 6.

Total Runtime: Initially for smaller problem sizes (N=$1k$ and $10k$), the sequential version performed better than CUDA and OpenMP code. For bigger problem sizes (N = $100k, 1000k, 2000k$) CUDA starts performing significantly better than the sequential version. For N=$1000K$, CUDA is ∼12x faster than the sequential code. OpenMP performs worse than both CUDA and sequential versions for bigger problem sizes. This is due to the large OpenMP kernel and memcpy time. For N=$1000k$, CUDA is ∼600x faster than OpenMP.

Kernel Runtime: Generally,CUDA kernel time outperforms OpenMP for all problem sizes.For N = $1000k$,CUDA is 2900x faster than OpenMP.

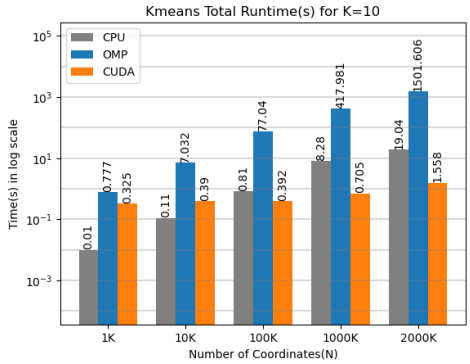
MemCpy Time: MemCpy time includes Device to Host, Host to Device and cudaMemcpy API calls. CUDA is significantly faster than its OpenMP counterpart. For bigger problem sizes,(N = $100k$ and $1000k$), CUDA carries out memory transfers ∼50x faster than OpenMP.

VII. Observations

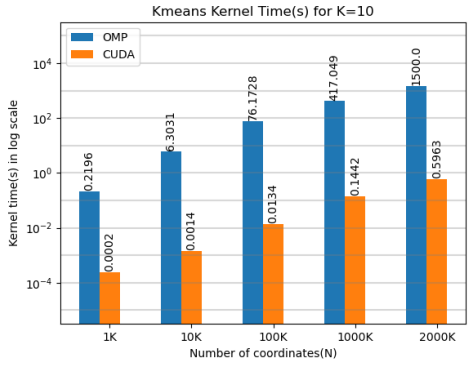
A. Programming Effort

According to our analysis, it takes more knowledge and effort to code in CUDA than in OpenMP.
Before comparing the effort required between CUDA and OpenMP programming, it is essential to note that CUDA provides developers with more authority to define structures, necessitating advanced knowledge of GPU programming. The developer is tasked with responsibilities such as allocating and deallocating memory on the device, determining the execution grid's geometry, crafting kernels (functions designated for execution on the device), and ensuring synchronisation. In contrast, OpenMP's pragma-based directives offer a higher-level abstraction. These directives are generally concise one-liners added prior to the region intended for parallelization. The primary focus of an OpenMP programmer is seen in the algorithm phase, often utilising for-loops—in a manner conducive to parallelization. OpenMP handles all other aspects, such as memory allocation and deallocation,

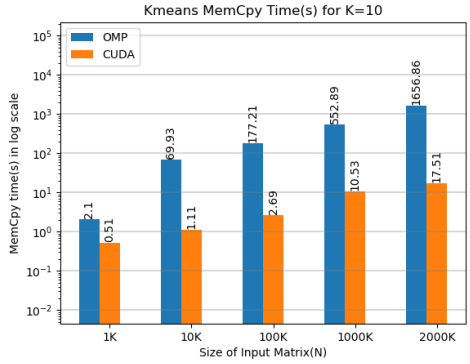
streamlining the programming process beyond the initial algorithmic design. Shifting from CUDA to OpenMP was not particularly easy for us. Having had most of the control for defining different structures, it was difficult to debug OpenMP errors in the initial phase. It got easier once we became familiar with the language.



(a) Kmeans Clustering Total Runtime



(b) Kmeans Clustering Kernel Time



(c) Kmeans Clustering MemCpy Time

Fig. 6: Kmeans Clustering Algorithm

B. *Performance Overhead*

Having employed *nvprof* for profiling the two languages, we were able to capture four main time metrics - average kernel launch time, total runtime, kernel runtime and MemCpy time. It's evident that CUDA consistently outperforms OpenMP in minimizing overheads, contributing to more efficient execution. Specifically, the memory allocation and deallocation operations, *cuMemAlloc* and *cuMemFree*, appear to be notably expensive in certain algorithms when implemented with OpenMP. In contrast, CUDA exhibits minimal overhead in these operations. For example, in the Sieve algorithm, the two metrics combined occupied over 80% of the total running time for larger problem sizes. Such overheads are minimal in CUDA. In terms of the MemCpy time, both the languages have similar performance across algorithms. Total kernel runtime was lower, hence better in CUDA.

C. *Kernel launch time*

We also report the average kernel launch times observed for CUDA and OpenMP across all the algorithms. As reported in Table III , CUDA launches kernels much faster than OpenMP. For the convolution operation CUDA kernel launch time is $\sim$40x faster than OpenMP. For the N-Queens algorithm, CUDA launches the kernel in 0.003% of the time taken by OpenMP to launch its kernel. CUDA kernel launch is extremely quick.

Algorithm	CUDA	OpenMP
Sieve of Eratosthenes	5.85μs	22.85μs
Convolution Operation	9.36μs	4168.8μs
Bellman Ford	29.28μs	68.84μs
N-Queens	15.17μs	4368.2μs
K-means clustering	6.46μs	127.76μs

TABLE III: Kernel Launch Time

D. *Program Executable Size*

As observed from Table IV, we can see that the executable size is maximum for CUDA, followed by OpenMP and then the sequential version.

Algorithm	CUDA	OpenMP	Sequential
Sieve of Eratosthenes	741KB	552KB	8.4KB
Convolution Operation	738KB	562KB	15KB
Bellman Ford	642KB	569KB	15KB
N-Queens	748KB	523KB	13KB
K-means clustering	663KB	571KB	21KB

TABLE IV: Program Executable Size

E. *Scalability*

The key insight from our experiments is that, for a given program, as the problem size increases, a CUDA implementation offers more substantial speedup compared to OpenMP. The runtime and kernel times are comparable for smaller problem sizes. However, with the increase in the problem size, CUDA demonstrates significantly faster performance, hence it is more scalable than OpenMP.

F. *Learning Curve & Control*

OpenMP, while offering parallelism through pragmas, may handle certain aspects like memory management more automatically. On the other hand, CUDA provides more fine-grained control to the developer, hence it involves more explicit coding for memory management and thread control. We felt the learning curve is steeper for CUDA programming. OpenMP also demands certain precision to execute proper offloading, but when compared with CUDA, it is definitely a good starting point for any developer new to parallel programming.

VIII. Conclusions & Future Scope

1) In this research, we conducted a comparison between two GPU languages, namely OpenMP and CUDA, using five distinct programs.

2) Additionally, we assessed their speed-ups in comparison to sequential programs, and profiled the parallel codes to capture various metrics such as kernel launch time, total runtime, kernel runtime, and memory copy times.

3) Our findings indicate that CUDA programs ran faster than OpenMP for all the algorithms. Hence, despite requiring more programming effort, it offers enhanced scalability, flexibility and performance.

4) Conversely, OpenMP is more user-friendly, requiring less programming effort. An added advantage of writing accelerator code with OpenMP is that it can work on any accelerators.

Therefore, a programmer needs to weigh in all the factors above and adapt to the programming architecture that suits best for their specific programming needs.

Looking ahead, we aim to delve deeper into the challenges limiting the scalability and performance of OpenMP. Furthermore, we plan to extend our comparative analysis to include other GPU languages such as OpenCL and OpenACC, encompassing a broader range of programs in our study.

References

[1] Performance evaluation of openmp's target construct on gpus-exploring compiler optimisations. Int. J. High Perform. Comput. Netw., 13(1):54–69, jan 2019.

[2] Ganesh G Surve and Medha A Shah. Parallel implementation of bellman-ford algorithm using cuda architecture. In 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), volume 2, pages 16–22, 2017.

[3] Memeti, S., Li, L., Pllana, S., Kołodziej, J., & Kessler, C. (2017, July). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (pp. 1-6).

[4] Cleverson Lopes, Dominique Ledur, Carlos Zeve, and Dos Anjos, *Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms*, *WSPPD 2013*, August 2013.

[5] Mikhail Khalilov and Alexey Timoveev, *Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU*, *Journal of Physics: Conference Series*, **1740**, 2021. https://api.semanticscholar.org/CorpusID:234169213

[6] Vincent Dumoulin and Francesco Visin, *A guide to convolution arithmetic for deep learning*, *arXiv:1603.07285 [stat.ML]*, 2018. https://arxiv.org/abs/1603.07285

[7] Cuda toolkit documentation. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html