GPUs - Final Project: Comparison between OpenMP and CUDA

Ashay Changwani
New York University
ac8832@nyu.edu

Avinav Goel
New York University
ag7644@nyu.edu

Shubham Jha
New York University
sj3549@nyu.edu

Swapnil Gupta
New York University
sg6665@nyu.edu

Abstract

CUDA has long been the de-facto framework for building multiprocessing applications that utilize GPUs. Recently, OpenMP has gained a lot of interest in the community due to it's portability and scalable model. It provides a generic API built for shared-memory multiprocessing programming and can run on a variety of hardware. OpenMP supports GPUs through a mechanism called accelerator offload. Given that OpenMP and CUDA are fundamentally different frameworks with different programming models, and performance characteristics, it can be challenging for computer scientists to determine the most appropriate framework for a given use case. In this work, we compare OpenMP to CUDA along multiple dimensions such as performance, ease of programming, and overhead. Through experiments and analysis, we aim to determine the situations where it's beneficial to use OpenMP instead of CUDA or vice versa.

1 Introduction

1.1 Problem Definition

CUDA is one of the most mature GPU languages. However, it is not the only one used to program GPUs. There is OpenMP (starting from version 4.5), OpenCL, OpenACC, etc. In our experiments we make use of OpenMP and compare it to CUDA in terms of the following factors:

1. Programmability: How is it easy to write code with OpenMP vs CUDA?
2. Overhead: resources used by OpenMP vs CUDA for do the same task.
3. The quality of the final code: which is faster? bigger in executable size?
4. Scalability: How does the framework scale as problem size increases?

We have picked distinct programs built for GPUs and implemented them using both CUDA and OpenMP and run them on GPUs and compare the performance and scalability. We also analyse the behavior of both versions and find why one is better than the other and what are the factors affecting whether CUDA or the other will be better.

1.2 Experimental Survey

In this work we mainly compare OpenMP and CUDA. They are both frameworks to write multi-processing programs but they follow significantly different programming philosophies - OpenMP follows a to-down paradigm, whereas CUDA follows a bottom up philosophy.

CUDA is a proprietary framework built by NVIDIA in 2007. It was specifically created for NVIDIA GPUs and it doesn't support GPUs built by other manufacturers.

OpenMP is managed by the OpenMP Architecture Review Board. It was created in 1996 and supports multiple languages and hardware platforms.

2 Programs

We perform experiments by implementing multiple algorithms using both CUDA, and OpenMP. We've ensured that CUDA and OpenMP implementations follow a similar logic in order to facilitate a fair comparison. For example, we ensure that the implementations use similar data types and data structures, similar input/output formats, similar logging, etc.

2.1 N-Queens Algorithm

In this classic backtracking problem based on the Divide and Conquer logic, we attempted to leverage different strengths of OpenMP and CUDA in order to better represent the pitfalls of one vs the other. The aim of the algorithm is to place N queens from the game of chess on an NxN board. The queens can move vertically, horizontally and diagonally. All N queens must be safe from each other. The algorithm can broadly be divided into two parts: The first is creating all permutations of the chess board. To simplify the space requirements, we implicitly ensure that only one queen can lie in every column. Hence, we only need to check for diagonal and horizontal conflicts. It is trivial to see that there are a total of N^N permutations for the queens. The second part of the algorithm is to individually processes these N^N chess boards and count the number of viable solutions.

OpenMP's strength seems to lie in distributing the work of for loops, and hypothetically should gain an advantage since both parts can be parallelized. In CUDA, however, we are unable to parallelize the generation of the chess boards since it is inherently a sequential operation to produce all permutations. However, in the second part of the algorithm, CUDA should theoretically have an advantage for being able to process each permutation in parallel.

Algorithm 1: N Queens

Input: $N : NumberofQueens$
Output: $Valid = PlaceQueensandcheckifboardisvalid$

Initialize N^N board as blank matrix of N*N elements

for $row = 1$ *to* N **do**
  for $col = 1$ *to* N **do**
    placeQueen(board, row, col)
  end
end
for $queen = 1$ *to* N **do**
  if *hasHorizontalConflict(*$queen$*)* **then**
    return false;
  end
  else if *hasVerticalConflict(*$queen$*)* **then**
    return false;
  end
  else if *hasDiagonalConflict(*$queen$*)* **then**
    return false;
  end
  return true;
end

2.2 Color Quantization using KMeans

Color quantization refers to diminishing the quantity of particular tones in a picture while attempting to keep up with the visual appearance of the image. It observes its application in many undertakings in compression and picture handling like segmentation, texture analysis and watermarking. Our strategy tries to reduce the number of distinct colours by clustering pixels with comparable colours. The closeness between two pixels is estimated by utilizing the euclidean distance between the RGB values. The clustering converges utilizing the K-Means calculation which guarantees that there is high intra-group and low inter-group pixel similarities. The mean RGB values of every one of these clusters are put away and every pixel in the quantized picture has a place with one of the cluster group and will show the mean value of this cluster, referred to as it's centroid. In this work we explore various parallel programming techniques and platforms such as CUDA and OpenMP to achieve a higher efficiency. This algorithm is highly suitable for parallel implementation as it involves two phases of calculations as mentioned below:

1. **Clustering** of the Pixels to the nearest centroid

2. **Recentering** the Centroid based on the pixels that are part of the cluster

These steps are repeated until convergence where the cluster assigned to a pixel does not change with subsequent iterations.

Algorithm 2: Color Quantization Sequential

Input: $Pixels = \{p_1, p_2, p_3...p_N\}$, $K : Clusters$
Output: $Clusters = \{c_1, c_2, c_3...c_K\}$

Initialize all cluster centroids to random $p_i \in Pixels$

while *Cluster Centroids keep Changing* **do**
  `/* Phase 1: Finding Clusters */`
  for $j = 1$ *to* N **do**
    $pixel.Cluster[j] = argmin_{K \in 1,2,3..K} EuclidDistance(p_j, c_k)$;
  end
  `/* Phase 2: Recenter the Clusters based on the included Pixels */`
  for $j = 1$ *to* N **do**
    $c_k = \frac{1}{|S_k|} \sum_{p_j \in S_k} p_j$;
  end
end

2.3 Floyd Warshall Algorithm

Floyd Warshall is a popular dynamic programming algorithm for finding all pairs shortest paths in a graph. The algorithm compares all possible paths throughout the graph between each pair of vertices. It does this with by picking an intermediate vertex, checking the distance between every pair of vertices via the intermediate point. This results in n^3 comparisons in a graph in total. n^2 comparisons with each vertex as intermediate. In this manner, it incrementally improves an estimate on the shortest distance between two vertices, until the estimate is optimal.

Algorithm 3: Floyd Warshall

Input: $W = DistanceMatrix$
Output: W = Distance Matrix with minimum distances

```
for k = 1 to N do
    for i = 1 to N do
        for j = 1 to N do
            if W[i][j] > W[i][k] + W[k][j] then
                W[i][j] = W[i][k] + W[k][j]
            end
        end
    end
end
```

2.4 Merge Sort

Mergesort is a sorting algorithm based on the Divide and Conquer principle. The input to the algorithm is a list of elements in arbitrary order, and the output of the algorithm is a sorted list containing all the input elements. The Mergesort algorithm consists of two phases:

1. (Divide) Recursively break down the input list into two halves, and independently sort them.

2. (Merge) Take the two sorted arrays of size n/2 from the previous step, and merge them into a sorted array of size n.

The Mergesort algorithm is quite efficient, with a time complexity of O(nlogn). If the input data can be accessed serially, Mergesort can be more efficient than Quicksort for some type of lists. It is a popular algorithm in languages such as Lisp, where sequential data structures are ubiquitous. The second phase (merge phase) of this algorithm can be parallelized. In our parallel implementations of the Mergesort algorithm, we break down the input list into chunks, and then iteratively merge them. Each thread is responsible for the sorting of a chunk. The chunk sizes grow by a factor of 2 every iteration, which causes the neighbouring chunks to me merged. We conduct experiments with integer arrays of lengths ranging from 1 to 10^6.

3 Experimental Setup

We create 3 implementations for each of the aforementioned algorithms:

1. Sequential implementation

2. CUDA implementation

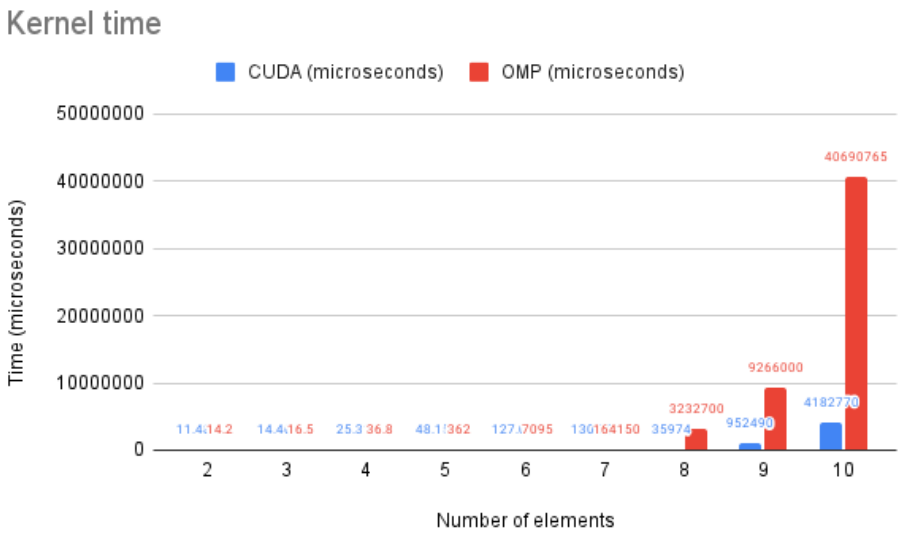
3. OpenMP implementation

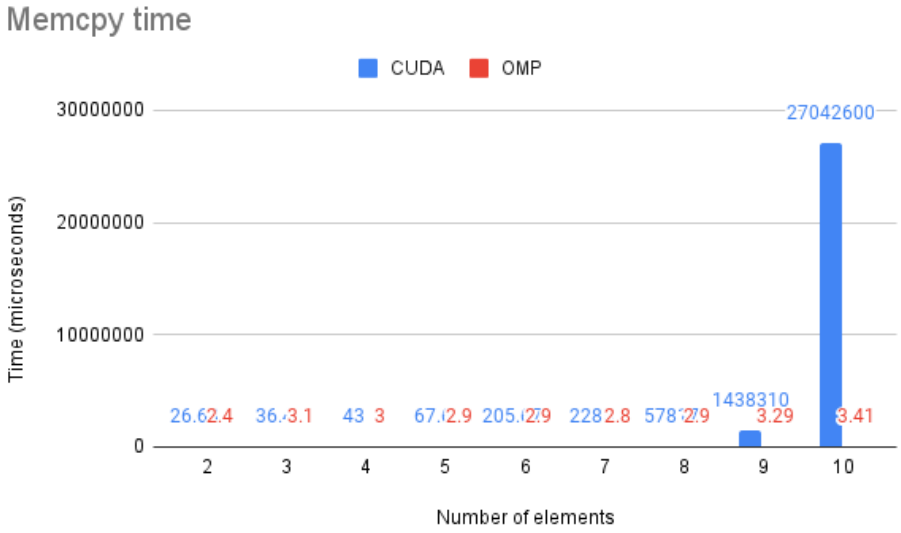
We use C and C++ for our implementations. The CUDA code was compiled using nvcc version 10.1. Total program runtimes were measured using the Linux time utility. The profiling was done using nvprof version 10.1.243 (21).

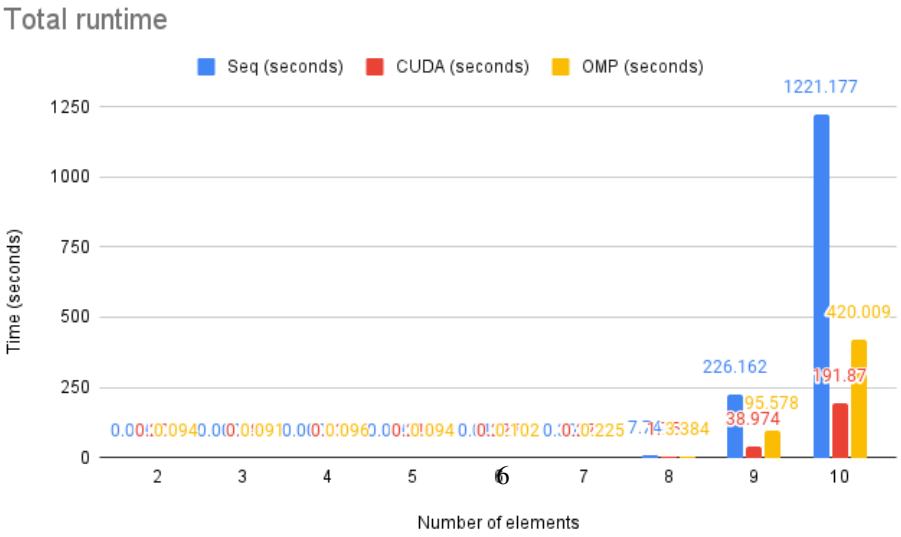
The experiments were run on the NYU Greene cluster. Hardware specifications: 8GB RAM, 8xIntel(R) Xeon(R) Platinum 8268 CPUs, Tesla V100 GPU.

4 Results

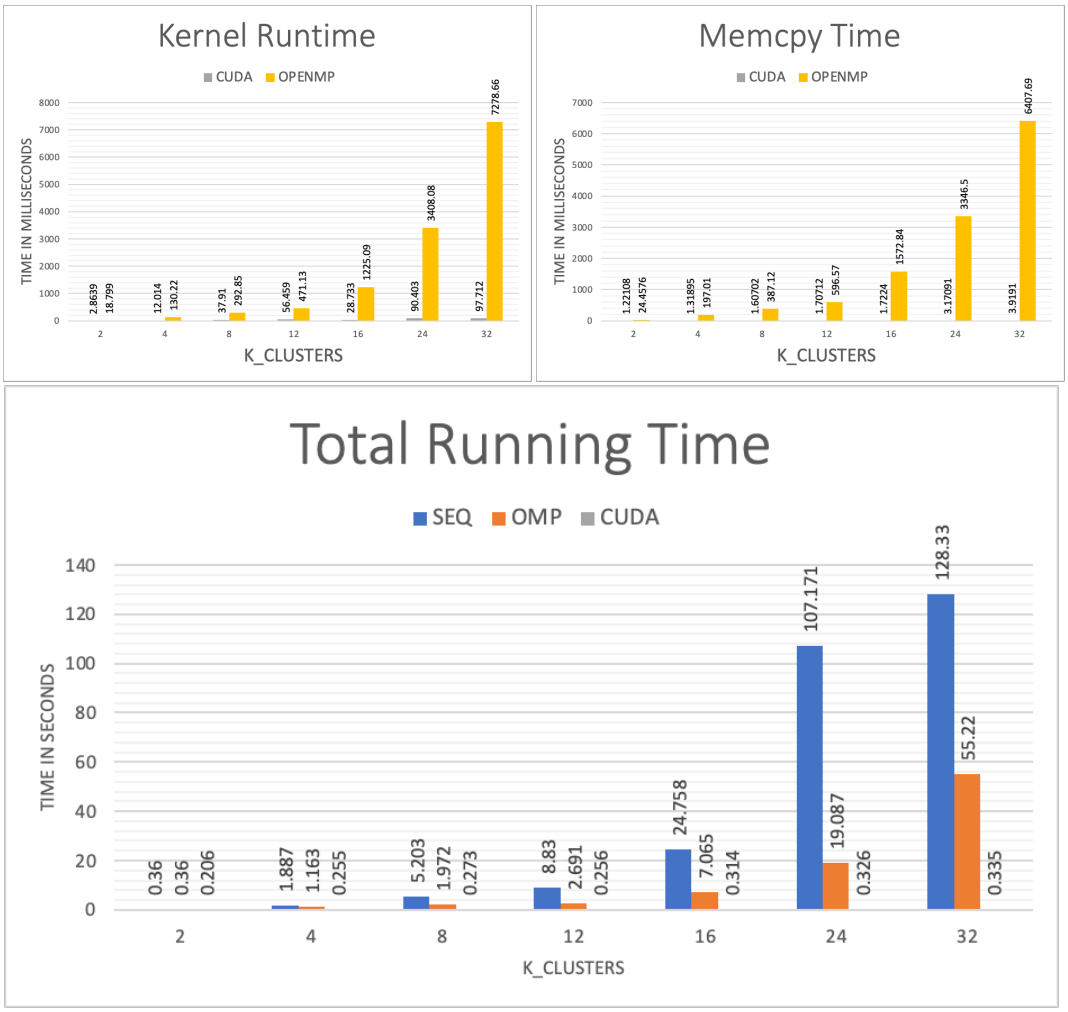
4.1 NQueens







4.2 Color Quantization using KMeans



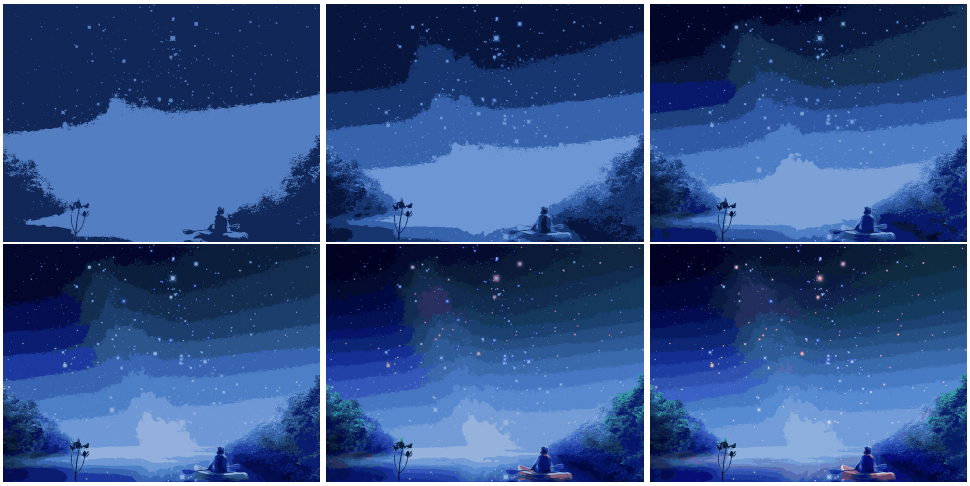
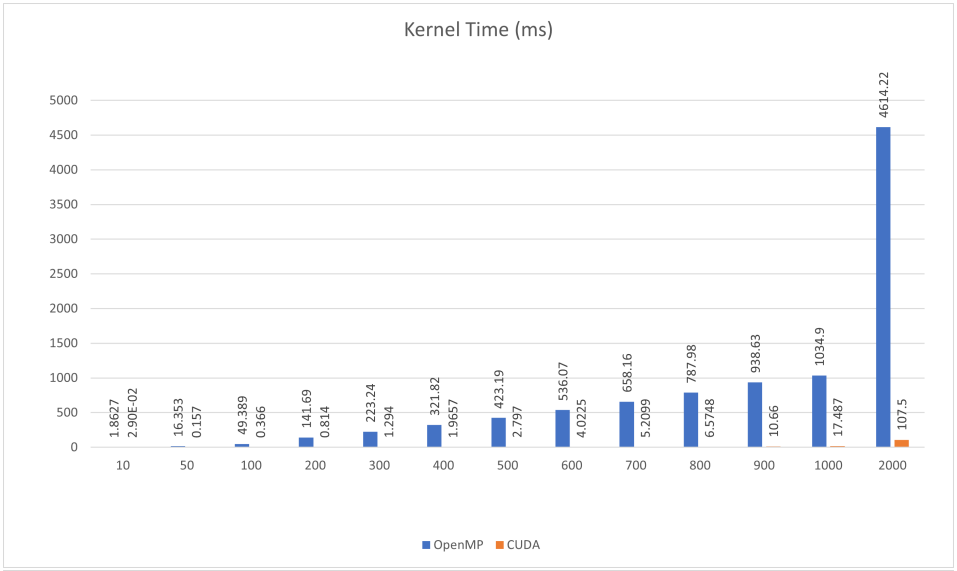
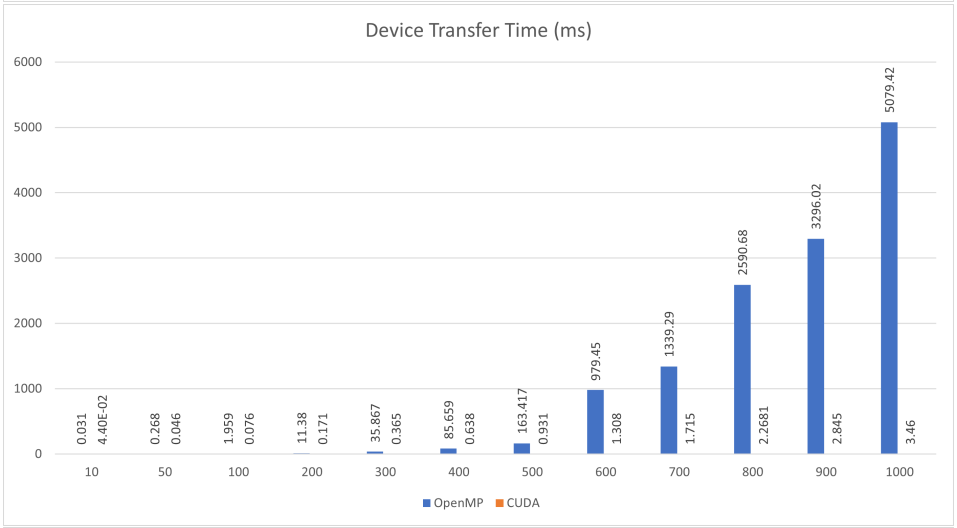
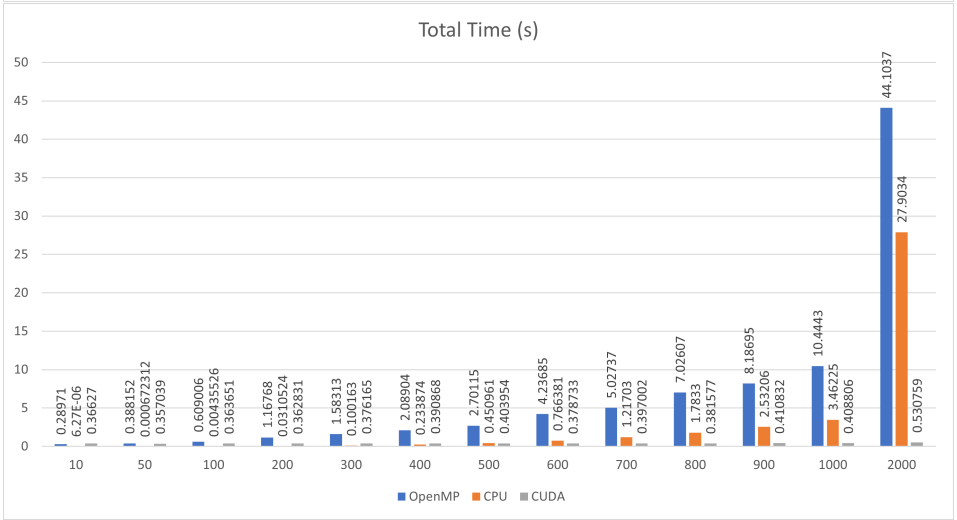


Figure 1: Color Quantized Output Images with Cluster Size $\in \{2, 4, 8, 12, 24, 32\}$

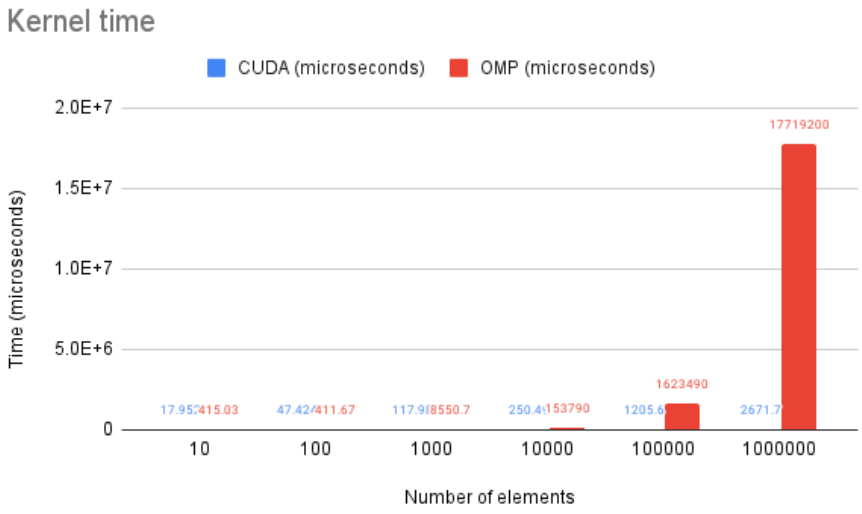
4.3 Floyd Warshall Algorithm

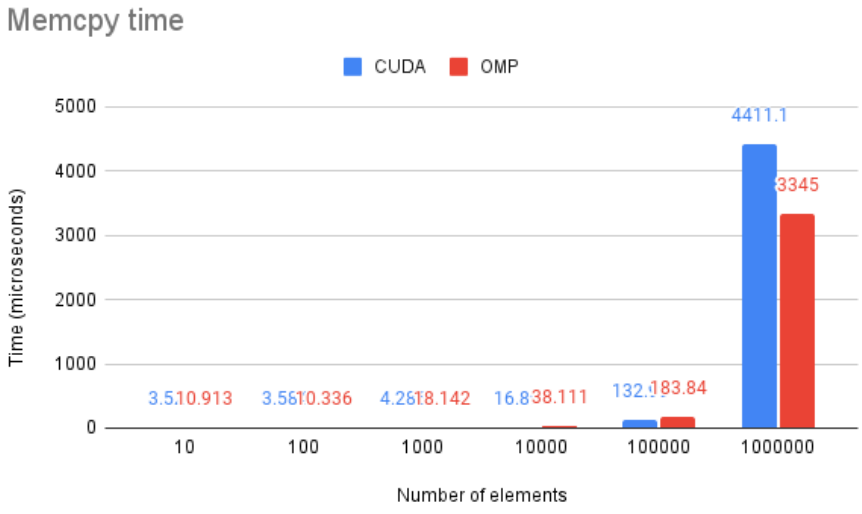


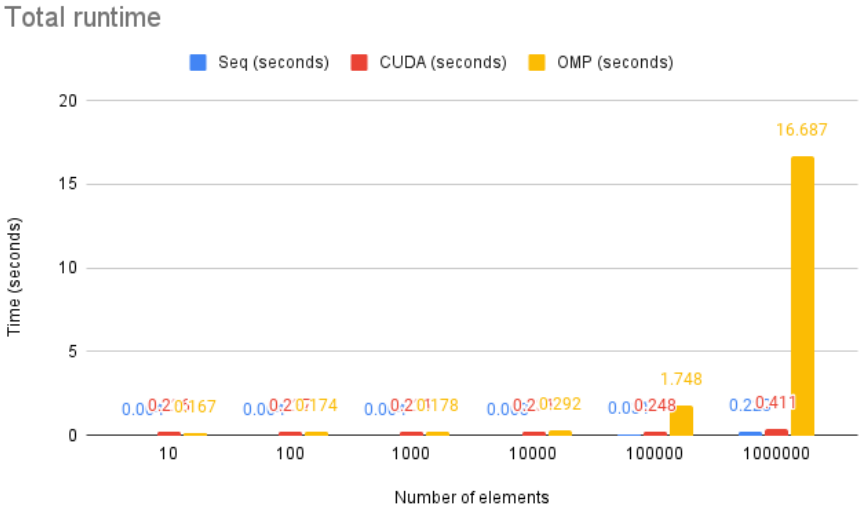




4.4 Mergesort algorithm







5 Analysis

5.1 Ease of programming

CUDA and OpenMP differ significantly in their programming paradigms - OpenMP takes a top-down approach while CUDA adopts a bottom-up approach.

When programming using CUDA, one needs to explicitly copy data between host and device (using cudaMemcpy), decide on how the computation gets distributed between threads, and implement kernels which run on the device. CUDA breaks down the computation into a grid of blocks. Each block contains multiple threads (basic unit of computation).

In contrast, OpenMP allows users to configure the behaviour of the program using compiler directives. While programmers do not have to explicitly move data around, or explicitly distribute computation among threads, they have the option to do so. Movement of data between host and device can be customized using the map directive, and the organization of threads can be controlled using the teams directive.

OpenMP often allows programmers to create multiprocessing application using significantly fewer lines of code compared to CUDA.

OpenMP also contributes towards the intricacy of designing different rules for individual loops by the distribute, parallel and teams keywords. In applications where not all loops have to be executed in a similar fashion, this can play a massive part towards ease of programming. In CUDA by contrast, there would need to be a different kernel for each different iteration along with changes to the environment to support the different parameters.

However, due to the recentness of OpenMP offering support for GPUs, there is very little documentation and QnA threads on using it. This made it difficult to debug errors as compared to CUDA for which there is a large amount of documentation available on the internet.

5.2 Overheads

Overhead is generally observed in two sections namely Kernel Execution time and Memcopy time. In light of the nvprof brings about in the above analysis, as issue size develops, the majority of the overhead in GPU exercises comes from CudaMemcopy. But for some compute heavy applications such as KMeans which operates on 473,600 pixels, the kernel execution takes more time as each pixel is processed individually in the clustering algorithm. And we observe using nvprof results that the calls made to the kernel execution far exceed the calls to memcopy and hence for Color Quantization the major overhead becomes computation in contrast to the memory. Similar overheads are observed as part of both OpenMP and CUDA and can be seen from the figures. In the case of Floyd Warshall, there is not much data transfer overhead seen in the CUDA impmlementation as there are only 2 calls made to MemCpy, one each to move W to and from the GPU. But with OpenMP, we have $N * 2$ calls being made to MemCpy which produces a large overhead when N increases. The data transfer overhead is 42 seconds compared to the 4 second kernel time for $N = 2000$. For CUDA, this overhead is negligible for larger N.

In the N-Queens Algorithm, we can see that as the value of N grows, N^N also grows at a very high rate. This exposes the inability of OpenMP to keep up as the input scales, since the kernel execution times skyrocket compared to those of CUDA.

5.3 Quality of Final Code

5.3.1 Speed

We compare the total execution time of the OpenMP and CUDA versions of the same algorithm. The general trend indicates programs optimized with CUDA are faster than programs optimized with OpenMP. In the case of Floyd Warshall, we see that OpenMP performs well for smaller inputs (<1 sec for N<200) but as we increase the N, the total time increases exponentially becoming slower than the sequential algorithm itself. This is due to the high number of memCpy operations occuring in the OpenMP implementation. CUDA is over 80x faster for an input $N = 2000$. Even if we look at just the kernel times for both, CUDA is upto 40x faster than OpenMP.

In the case of NQueens, the graphs for kernel time and memcpy times tell a very intricate story. What can be seen is that for smaller values of N, it is clear that OpenMP easily beats CUDA in terms of total run time, however, as the value of N grows, the kernel execution time for OpenMP and the Memcpy time for CUDA increases exponentially. Most notably, the former increases at a rate much faster than the latter. This is likely due to the fact that the N^N permutations need to be transferred to and from the GPU in CUDA at the end of the execution, however in OpenMP they reside on the device itself. This means that while memory transfer times are far lower, the kernel execution time shows a massive difference in favor of CUDA.

In the case of Mergesort, we the trend holds - the CUDA implementation is significantly faster than the OpenMP implementation, even if we compare just the kernel execution times.

Incase of Color Quantization with KMeans, we obseve the visulaizations of the same as part of the bar graphs above. We see that as the numbers of clusters grow, the Sequential Algorithm takes a lot of time, followed by OpenMP and then CUDA. CLearly CUDA displays the best speed of 426x for 32 clusters while OpenMP displays only 3x speedup and is not able to scale. The main reasons for the same are because the phase 1 for recentering runs on pixel for every thread. Thus CUDA GPU scales better in such case as the number of cuda cores is larger. The mean of the cluster are calculated as part of recentering phase 2 and are highly optimized and calculated in Parallel.

We also observe that while the memory copy speed remains similar between OpenMP and CUDA, in code cases OpenMP performs more memory copy operations, thereby increasing the overhead and execution time.

5.3.2 Scalability

OpenMP does not scale very well as the problem size increases. As mentioned in the previous section, the data transfer overhead increases exponentially as the problem size increases. CUDA on the other hand does not have a very large overhead and hence performs relatively faster as the problem size increases.

5.3.3 Executable Size

The executable sizes (in KB) are given in the below table. As seen, the executable sizes for OpenMP programs are slightly smaller than CUDA programs.

Program	OpenMP	CUDA
KMeans	594	775
Floyd-Warshall	640	768
NQueens	545	778
Mergesort	556	746

6 Conclusion

In this work, we implement multiple algorithms in CUDA and OpenMP, and perform comparisons between them along multiple dimensions. We observe that OpenMP is very beginner-friendly and can be used to parallelize even some sequential programs with minimal code changes. It also does not require specialized knowledge of the Device hardware. We also observe that OpenMP can be quite competitive with CUDA in terms of performance in some cases, but in other cases, may require some extra optimizations on the programmer's end. We also observe that online documentation for OpenMP offloading is quite sparse and it can be challenging for beginners to setup and get started with OpenMP. In contrast, online documentation and support for CUDA is plentiful.

Overall, we believe that OpenMP is a solid alternative for CUDA since it is very beginner friendly and allows programmers to parallelize serial programs with minimal code changes while achieving performance comparable to CUDA.

A References

1. OPENP 4.5 DEVICE OFFLOADING DETAILS, COLLEEN BERTONI

2. https://www.admin-magazine.com/HPC/Articles/OpenMP-Coding-Habits-and-GPUs
3. https://sc18.supercomputing.org/proceedings/workshops/workshop$_files/ws_waccpd109s2-file1.pdf$
4. https://on-demand.gputechconf.com/gtc/2018/presentation/s8344-openmp-on-gpus-first-experiences-and-best-practices.pdf