GPUs: Architecture and Programming , Spring 2022

Comparing CUDA and OpenMP for parallelizable applications

Bharath Surianarayanan (bs4224), Sripranav Suresh Kumar (ss14432), and Diya Ilinani (di2078)

New York University

Abstract

While initially GPUs were primarily used for graphics rendering, in recent decades its usage has become profound in various applications. Developers however are still faced with the challenge of understanding the CUDA syntax and the architectural complexity associated with GPUs. OpenMP offers a relatively simpler method of programming and the associated changes to the serial code are very minimal. In this paper, we have considered six different GPU friendly applications. We develop code for accelerators using OpenMP and compare it to CUDA code for all four applications. To analyze their performance we use various criteria such as ease of memory allocation and computation, running times and overall quality of code.

1 Introduction

The usage of single core CPU's had been prevalent for a large period of time for most general purpose applications. CPUs have been able to offer performance gains over the past few decades primarily due to its significant improvement in hardware. However, this is no longer primarily the case. Due to physical restrictions on the size of the transistors and their heating capacity, the size of transistors can no longer be made smaller to fit in more of them in a single core. This gave rise to multicore systems where as the name suggests there were multiple cores and each core could run different threads. However, this architecture was an MIMD architecture which, while definitely better than a single core, is very generic and its performance for SIMD problems isn't the best. To tailor the architecture to the type of problems being solved, we now have accelerators like GPUs, TPUs and FPGAs to get the maximum possible performance. GPUs are data-parallel and throughput oriented processors that hide relatively expensive global memory accesses with extensive use of parallel threads[1].

Although GPUs were primarily used for Graphics related applications, their importance to various other applications have grown in the last few decades. About the same time, 3D graphics application programming interfaces (APIs) became popular, notably OpenGL and Direct3D component of Microsoft DirectX exposed several programmable features to the programmers. The direction towards more programmable processors was driven by the fact that programmers wanted to create more complex visual effects on the screen than those provided by the fixed-function graphics hardware[2].

With this in mind, Nvidia developed CUDA which is a parallel programming model for GPU applications. CUDA was generic enough to allow developers to build any application on GPU. As one of the earliest in the field, CUDA is still very much popular as it gives programmers a lot of power over how the gpu is used. CUDA however is a low level framework and requires a significant level of understanding about the architecture of the GPU itself.

OpenMP is a powerful parallel programming model that has become extremely popular due to it's easy to use syntax, the ability to use simple directives to parallelise any serial code and its portability. OpenMP started off as a model for multicore programming and was later enhanced to include accelerators. OpenMP is very high level and the programmer doesn't need more than a cursory understanding of GPU architecture to write code that runs on the GPU. Another aspect is that while CUDA is specifically for GPUs, OpenMP works with any type of accelerator. In this paper, we compare and contrast GPU applications written in CUDA to GPU applications written using OpenMP. We use the following criteria for comparison.

- Ease of Memory Allocation and Computation.
- Running Times.
- Overall quality of code.

2 Literature Survey

There have been a lot of research on the performance of various algorithms on GPU with some algorithms performing much better than others. Uditi et al. [3] performed a comparative analysis of various shortest path algorithms using OpenMP, which provided the inspiration to choose Dijkstra and Bellman Ford as two of the algorithms we aim to parallelize. Bakhoda et al. [4] chose to compare over 12 applications based on their performance using CUDA and it is from here we draw inspiration for implementing Breadth First Search and the N-Queens Problem.

Ganesh G Surve et al.[5] implemented the CUDA implementation of the Bellman Ford algorithm in his paper. While this paper offers significant performance gains over the serial version, it fails to consider the run time overhead and lacks appropriate code profiling.

In his paper[6] Sabne et al. explores the effects of compiler optimizations on the translation of code from OpenMP to CUDA. Similarly, Akihiro Hayashi et al.[7] also focuses on performance evaluation of OpenMP's target constructs on GPUs using compiler optimizations. They consider six benchmark algorithms and evaluate their performance. Although the techniques explored in this paper are beyond the realm of our work, their proposal to overcome the inefficiencies of previous translators such as OpenMPC and their work on tuning in the presence of runtime variations offer us significant insights.

Implementing Breadth First Search on GPU has been done quite frequently.Merrill et al.[8] implemented the Breadth First Search algorithm on GPU using CUDA and came up with better performance than the traditional CPU implementation and Luo et al.[9] implements the Breadth First Search algorithm using a more efficient queue structure and multiple frontiers to optimize performance. A lot more people have come up with more specialized optimizations for GPU performance. Recently, Dong et al.[10] proposes a warp aligned adjacency list among other methods to optimize the GPU implementation of Breadth First Search further.

With its wide applications in various fields, it is no surprise that there has been a lot of work in the are of Fourier Analysis. Computing Discrete Fourier Transforms have always been a very inefficient and time consuming process and one of most used algorithms which makes computing DFTs much quicker is the Cooley - Tukey Fast Fourier Transform algorithm[11]. Puchała et al.[12] talks about the effectiveness of GPU implementations over CPU implementations. Zhang et al.[13] also talks about a more optimized solution where texture memory is used for the computation of the twiddle factor. Even though we do not implement this, it is a fascinating approach which can be looked into going forward.

Although GPUs have great computational power in terms of the simultaneous usage of threads, it also comes with some extra cost such as the cost of context-switching for threads. [14] evaluates all pairs shortest paths using floyd-warshall algorithm which requires V^2 threads and SSSP which requires V threads. Floyd Warshall algorithm showed an improvement over the CPU version with a factor of 3 while SSSP showed an improvement of 17.This implies that while choosing to use a GPU will enhance performance, the actual improvement highly depends on an implementation that allows us to launch kernels that are few in number while using GPU memory.

3 Problem Definition

In this section, we define every problem we have and the potential challenges that arise when we attempt to paralleize them.

3.1 Bellman Ford Algorithm

The Bellman-Ford Algorithm is used to compute the shortest path from a single source to all vertices in a given graph. It differs from the Dijkstra algorithm due to its ability to handle negative weights. The Bellman-Ford algorithm runs in $O(VE)$ time, where V is the number of vertices in the graph and E the number of edges. The existence of a negative weight cycle is also reported by the algorithm.

The idea behind the algorithm is that length of the path from the starting/source vertex is overestimated to each of the other vertices in the graph. Now this estimation is recomputed by iterating across every path whose length is shorter than the previous path lengths.

3.2 Dijkstra Algorithm

The Dijkstra algorithm, like the Bellman-Ford is used to find the shortest path between nodes in a graph. However the optimality of the algorithm is only for positive edge weights. The primary reason is that if all the weights are non- negative then adding a new weight cannot make the path shorter, whereas negative edge weights add the problem of negative edge cycles. The best possible run time of Dijkstra is $O(E + Vlog(V))$, which requires the usage of appropriate data structures. In our implementation we do not use these data structures, since we believe for our comparisons this implementation does not add any additional value.

The logic behind the Dijkstra algorithm is to assign the initial distances from the source to every other vertex in the graph(except itself) to infinity. From this source node, edges are *relaxed*, i.e the shortest path from the current node to every adjacent node is updated. Now the adjacent node with the minimum shortest path from the source is selected and the relaxation is done from this node. Once all the nodes are visited, the solution is obtained.

3.3 Breadth First Search

The two most commonly used graph traversal algorithms are the Breadth First Search and the Depth First Search. Depth First Search works by exploring all the nodes in a branch across the entire depth before going to the next branch whereas Breadth First Search works by exploring all the nodes at a particular depth before moving onto the next level. Keeping this in mind, a stack is used for the Depth First Search Algorithm whereas a queue is used for the Breadth First Search algorithm.

The Breadth First Search algorithm being a graph algorithm requires a graph and there are two ways to represent a graph - using an Adjacency Matrix or an Adjacency List. The adjacency matrix is an $n \times n$ matrix filled by boolean values where it is true if an edge exists between the nodes represented by the indices and false if there exists no edge between the said nodes. Adjacency list is where we have a 1D array mapped to each node and the 1D array contains all the nodes that have an edge with the node they have been mapped to. The main difference between the two methods is in terms of the memory they require. The adjacency matrix takes up n^2 space even if there are very few very few edges whereas the space taken up by an adjacency list entirely depends on the number of edges. However, in order to implement on GPU, the adjacency matrix is much more easier to implement. It is also to be noted that Breadth First Search is a fundamentally hard problem to solve on GPU due to the highly irregular nature of the memory accesses which definitely brings down the efficiency of the GPU.

3.4 Fast Fourier Transform

Fourier Analysis converts a signal from its original domain to a representation in the frequency domain and vice versa. The Discrete Dourier Transform (DFT) is obtained by decomposing a sequence of values into components of different frequencies. Computing the DFT is required frequently in many fields. Computing it directly however is an expensive operation and is in the order of $O(n^2)$ which is highly ineffective.

Fast Fourier Transform(FFT) is an algorithm which aims to speed up this process of computing the DFT. This is achieved by factorizing the DFT matrix into a product of two sparse matrices and this is done repeatedly. This brings down the time complexity down from $O(n^2)$ to $O(n \log n)$ which is a major speed up especially for large values on n. This is very realistic as in the real world applications the data size, n is usually really large. This optimisation has made the computation of DFT much easier in real life applications. We implement the Cooley - Tukey algorithm which is one of the most popular FFT algorithms for the calculation of DFT. This particular implementation works only for values of n which are powers of 2.

The main challenge faced during the implementation of FFT in GPU is mapping the iterations of the algorithm to the thread ids. This is the case because the algorithm has a nested for loops whose indices both had to be mapped to the thread id.

3.5 N-Queens

In the game of chess, a queen has the ability to move across the same row, column, and even diagonally. If there were 2 queens on the same chessboard, to facilitate a situation in which they can't attack each other, the 2 queens should not be placed in the same row, column, or diagonal. N Queens is the problem of determining a valid arrangement for N such queens in an N * N chessboard such that they don't attack each other. We know that there are at least N! possible configurations for any N if no queen is on the same row or column. As N increases, the time complexity to check the validity of every permutation increases rapidly.

Finding an efficient way to solve n queens would directly translate to many real-world applications where it is required to generate many permutations such as cryptography, parallel memory storage schemes, VLSI testing, traffic control, and deadlock prevention.

The method that I have used to evaluate N queens is to consider all NexpN permutations and weed out the configurations where there is a queen in the same row, column, or diagonal. While, in a sequential code we can consider N! possibilities, while using openMP or GPU, it is much simpler to assign a position in every row to an individual thread. The asymptotic growth rate of the number of solutions for N queens is $(0.143n)^n$. Since the growth rate is extremely high, we only the exact number of n-queens solutions up to n=27. In the GPU implementation of N Queen, there is a massive advantage that increases the efficiency as a single position that is assumed on the board can be evaluated simultaneously, greatly improving the rate of evaluation.

3.6 Floyd-Warshall Algorithm

Floyd-Warshall algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. The result of the algorithm is a list of lengths of shortest paths between all pairs of vertices. The floyd-warshall algorithm has a complexity of O(V^3). Floyd-Warshall uses dynamic programming where a graph is broken up into smaller pieces to evaluate, the nature of which allows us to conduct these operations in parallel using a GPU. Unlike Dijkstra's algorithm, it also works for cycles with negative edges. Its format allows us to implement Floyd-Warshall for usage in large distributed systems such as maps.

In the implementation of floyd warshall, we can see that both time taken to execute memory transfers and the time needed to execute kernels are rising exponentially, while the rate at which the time taken to execute kernels is rising much faster until N=5000. This indicates that there would be a gain in performance for a higher number N when compared to the CPU version until the limited capacity of the GPU poses a problem.

4 Experimental Setup

4.1 Dataset

- For the Bellman-Ford and Dijkstra algorithm the weights of the adjacency matrix are generated at random between 1 to 10.
- For Breadth First Search, with the number of vertices and edges given, edges are created at random between the vertices.
- For Fast Fourier Transform, the input is randomly generated into a file once we give the size of the input we want. The file is then passed into the Fast Fourier Transform which reads the file to generate a list of complex numbers as input.
- For the Floyd-Warshall Algorithm, a graph of the specified number of vertices is generated, the initial distance between any two vertices is randomly produced and stored in an adjacency matrix.

4.2 Machine configuration

We have executed all of our codes on a CPU with the following configurations. The specification of the GPU in the machine also follows.

4.2.1 CPU configuration

- Architecture: `x86_64`
- CPU(s): 4
- Thread(s) per core: 1
- Core(s) per socket: 1
- Socket(s): 4
- NUMA node(s): 1
- L1d cache: 32K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 8192K

4.2.2 GPU configuration

- Number of devices: 2 (both the devices have the following configuration)
- Name : NVIDIA GeForce GTX TITAN Black
- CUDA Driver Version / Runtime Version : 11.4 / 10.2
- Compute capability : 3.5
- total global memory(KB): 6084 MBytes
- shared mem per block: 49152 bytes
- regs per block: 65536
- warp size: 32
- max threads per block: 1024

4.3 Experimental configuration

1. All the code was written in C.
2. All the code was compiled using gcc-4.8.5.

4.4 Challenges Faced

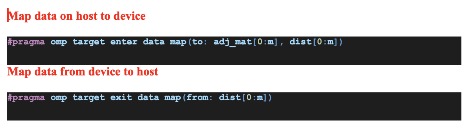
1. Lack of profiling information for OpenMP version.
2. While we were able to profile the CUDA code using nvprof, the same was not possible for the OpenMP version.
3. Hence comparison of OpenMP and CUDA using profiling was not possible.

5 Results and Performance

5.1 Ease of Programmability

5.1.1 Memory Allocation

The figures below indicate the memory allocation and data movement in OpenMP and CUDA.



Map data on host to device

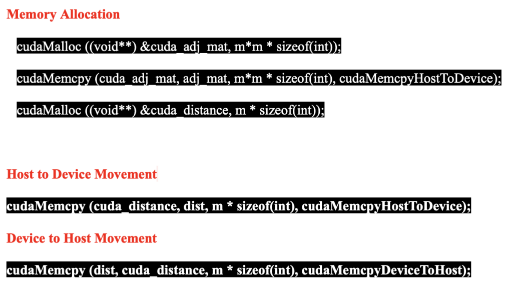
```
#pragma omp target enter data map(to: adj_mat[0:m], dist[0:m])
```

Map data from device to host

```
#pragma omp target exit data map(from: dist[0:m])
```

Figure 1: Memory allocation in OpenMP

The above figures clearly show that Memory allocation and data movement is much easier in OpenMP than in CUDA. Hence from a programming point of view the syntax for OpenMP is much easier to understand and write compared to CUDA.



Memory Allocation

```
cudaMalloc ((void**) &cuda_adj_mat, m*m * sizeof(int));

cudaMemcpy (cuda_adj_mat, adj_mat, m*m * sizeof(int), cudaMemcpyHostToDevice);

cudaMalloc ((void**) &cuda_distance, m * sizeof(int));
```

Host to Device Movement

```
cudaMemcpy (cuda_distance, dist, m * sizeof(int), cudaMemcpyHostToDevice);
```

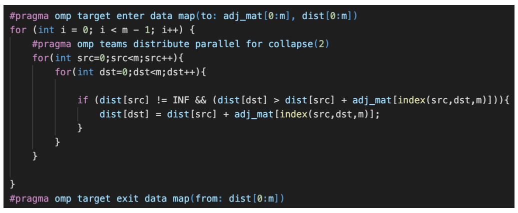
Device to Host Movement

```
cudaMemcpy (dist, cuda_distance, m * sizeof(int), cudaMemcpyDeviceToHost);
```

Figure 2: Memory allocation and Data Movement in CUDA

5.1.2 Computation

The below figures show the ease of computation and kernel launch involved in OpenMP and CUDA.

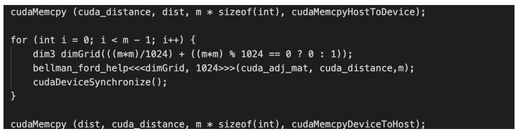


```
#pragma omp target enter data map(to: adj_mat[0:m], dist[0:m])
for (int i = 0; i < m - 1; i++) {
    #pragma omp teams distribute parallel for collapse(2)
    for(int src=0;src<m;src++){
        for(int dst=0;dst<m;dst++){

            if (dist[src] != INF && (dist[dst] > dist[src] + adj_mat[index(src,dst,m)])){
                dist[dst] = dist[src] + adj_mat[index(src,dst,m)];
            }
        }
    }

}
#pragma omp target exit data map(from: dist[0:m])
```

Figure 3: Ease of Computation in OpenMP



```
cudaMemcpy (cuda_distance, dist, m * sizeof(int), cudaMemcpyHostToDevice);

for (int i = 0; i < m - 1; i++) {
    dim3 dimGrid(((m*m)/1024) + ((m*m) % 1024 == 0 ? 0 : 1));
    bellman_ford_help<<<dimGrid, 1024>>>(cuda_adj_mat, cuda_distance,m);
    cudaDeviceSynchronize();
}

cudaMemcpy (dist, cuda_distance, m * sizeof(int), cudaMemcpyDeviceToHost);
```

Figure 4: Ease of Computation in CUDA

Figure 3 shows the OpenMP syntax for executing a *for loop* and the associated computation.
Figure 4 shows the CUDA syntax for launching the kernel and associated device to host variable transfer.

5.1.3 Conclusion

Clearly OpenMP offers an easier ability to program as compared to CUDA due to its ease of memory allocation and computation. The syntax for CUDA particularly those for *cudaMemcpyDeviceToHost* are prone to errors and hence may not be ideal.

5.2 Performance

5.2.1 Bellman-Ford Algorithm

- The performances for the Bellman-Ford algorithm on the **serial**, **OpenMP** and **CUDA** version are shown in Table 2.

- For vertices=100, the performance of the serial version is clearly better than the openMP and CUDA version.

- As the vertices increases, the performance of OpenMP becomes better than serial version and that of CUDA is much better than those of serial and OpenMP.

- For the CUDA implementation nvprof was used to obtain profiling information about the code. The columns in Table 3 are the **vertices**, **cudaMalloc**, **cudaLaunchKernel** and **Global Memory Load Efficiency**. From Table 2, it can be seen that the overhead for memory allocation dominates that of kernel computation for small number of vertices. As the vertices increases the usage of GPU becomes a lot more effective.

Vertices	Serial	OpenMP	CUDA
100	0.018	1.326	0.145
500	1.125	6.858	0.335
1000	8.362	14.152	0.975
5000	378.285	126	20.42
10000	>20m	>15min	81.627
20000	-	-	359.415
30000	-	-	>15m
32000	-	-	-

Table 1: Time taken for Bellman-Ford

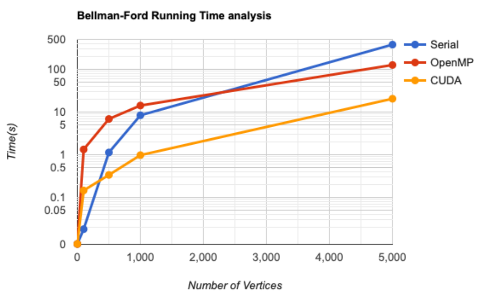


Figure 5: Graph for Bellman Ford analysis

Vertices	cudaMalloc	LK	gld
100	94.28%	4.29%	80.95%
500	47.72%	50.99%	84.95%
1000	16.28%	82.73%	90.5%
5000	0.84%	98.90%	94.5%

Table 2: Profiling Information for Bellman-Ford

5.2.2 Dijkstra Algorithm

- The performances for the Dijkstra algorithm on the **serial**, **OpenMP** and **CUDA** version are shown in Table 3.

- For small vertices the performance of serial version exceeds that of CUDA and OpenMP greatly.

- As the number of vertices increases the performance of OpenMP is comparable to that of serial version. However the performance of CUDA is still poor.

Vertices	Serial	OpenMP	CUDA
100	0.006	0.014	0.167
500	0.026	0.049	0.809
1000	0.065	0.111	4.459
5000	1.182	1.042	476.628
10000	4.555	4.268	>20m
20000	17.713	17.039	-
30000	39.684	38.96	-
32000	45.288	41.955	-

Table 3: Time taken for Dijkstra

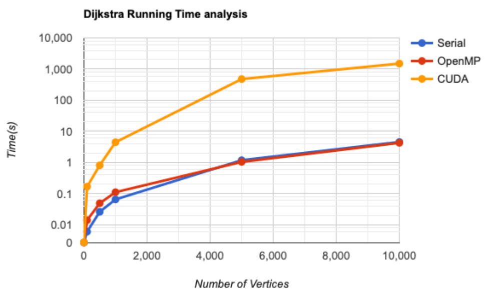


Figure 6: Graph for Dijkstra analysis

5.2.3 Breadth First Search

- The performances for the Breadth First Search algorithm on **serial**, **OpenMP** and **CUDA** can be seen in Table 4.

- We can see clearly that for smaller graphs, the serial version performs the best.

- However, something to note is that OpenMP catches up pretty quickly and starts performing better than the serial code. In fact, even though OpenMP is slower than the serial implementation till then, it is still much faster compared to the CUDA implementation.

- However, with increase in graph size, the CUDA implementation performs much better than both the serial implementation and the OpenMP implementation.

- The same trend is observed when the number of vertices is kept constant and the number of edges is increased.

Vertices/Edges	Serial	OpenMP	CUDA
1000/500	0.005	0.006	0.105
1000/10000	0.007	0.01	0.128
1000/100000	0.045	0.046	0.137
1000/200000	0.057	0.041	0.144
10000/5000	0.039	0.039	0.16
10000/100000	0.388	0.329	0.273
10000/1000000	0.532	0.476	0.446
10000/2000000	0.698	0.671	0.624

Table 4: Time taken for BFS

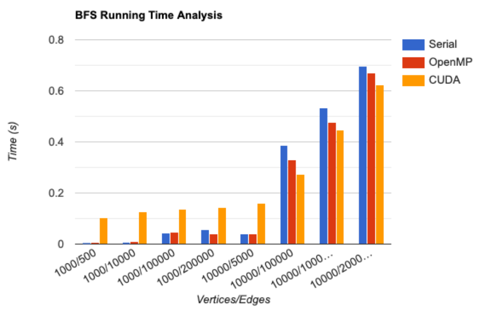


Figure 7: Graph for BFS analysis

5.3 Fast Fourier Transform

- The performances for the Fast Fourier Transform algorithm on **serial**, **OpenMP** and **CUDA** can be seen in Table 5.

- We can see clearly that for smaller input sizes, the serial version performs the best.

- However, something to note is that OpenMP catches up pretty quickly and starts performing better than the serial code when the input size is 512. In fact, even though OpenMP is slower than the serial implementation till then, it is still much faster compared to the CUDA implementation.

- However, with increase in input size, the CUDA implementation performs much better than both the serial implementation and the OpenMP implementation.

- It can be seen that even though the OpenMP implementation performs much faster for smaller input sizes, for larger input sizes, it is nowhere near the performance level of the CUDA implementation.

Input Size	Serial	OpenMP	CUDA
8	0.003	0.006	0.097
32	0.004	0.006	0.099
64	0.004	0.006	0.105
256	0.008	0.011	0.107
512	0.013	0.011	0.107
65536	0.099	0.096	0.140
131072	0.175	0.162	0.177
262144	0.255	0.241	0.240
524288	0.505	0.466	0.357
1048576	0.923	0.862	0.589
2097152	1.846	1.715	1.079
4194304	3.619	3.500	2.066
8388608	7.223	6.731	3.939

Table 5: Time taken for Fast Fourier Transform

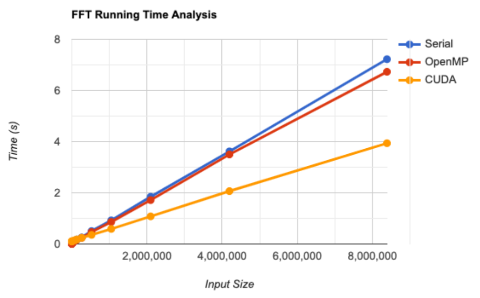


Figure 8: Graph for FFT analysis

5.4 N Queens

- The performances for the N Queens algorithm on **serial**, **OpenMP** and **CUDA** can be seen in Table 6.

- We can see clearly that for smaller input sizes, the serial version performs the best.

- As the value of N increases the performance of OpenMP becomes better than that of the serial version and the performance of CUDA dominates over that of the OpenMP version.

- For N=10 the speedup of CUDA as compared to the serial version is 823.322, while that of OpenMP is 21.276 which clearly highlights the gulf in performance of CUDA as compared to OpenMP.

N	Serial	OpenMP	CUDA
5	0	0.010	0.15
6	0	0.08	0.15
7	0.0625	0.023	0.16
8	1.203125	0.34	0.16
9	30.593750	9.041	0.19
10	782.156	36.761	0.98

Table 6: Time taken for N Queens

Vertices	Serial	OpenMP	CUDA
10	0.001756	0.000023	0.000362
100	0.101487	0.016610	0.001101
500	0.5599	0.510955	0.010049
1000	3.781112	3.638753	0.027323
2000	29.575	30.028	0.120463
5000	467.04	470.9079	265.369

Table 7: Time taken for Floyd-Warshall

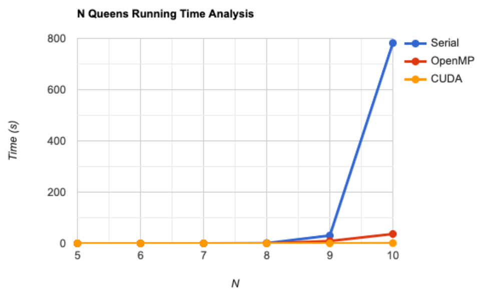


Figure 9: Graph for FFT analysis

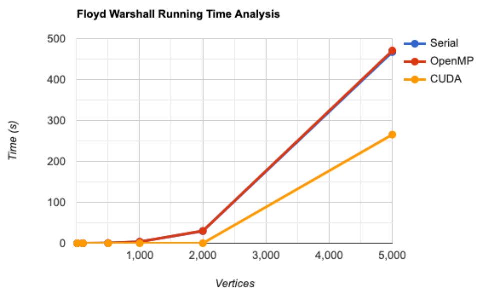


Figure 10: Graph for Floyd-Warshall analysis

5.5 Floyd-Warshall Algorithm

- In the openMP version, parallelism is taken advantage of only in the first loop, which is distributed among concurrent threads, while in the GPU version, work is distributed for the inner loops, giving a better performance as N increases.
- The sudden departure of efficiency for the openMP version after N=10 is explained by the larger N without the capacity to distribute computational power for the inner loops.
- The method implemented in GPU works in loops of the number of vertices, for a small number of vertices, there would be a lot of overhead for relatively small work done (V^2) by each loop.
- However, for a large number of vertices, although the number of loops increases it also results in better usage of the computational power of the GPU because every launched kernel does O(V^2) work. This explains the better performance of the CUDA program as the number of vertices increases.

5.6 Size of the executable

From Table 8, it can be seen that the size of the executable for OpenMP is much smaller than that of CUDA.

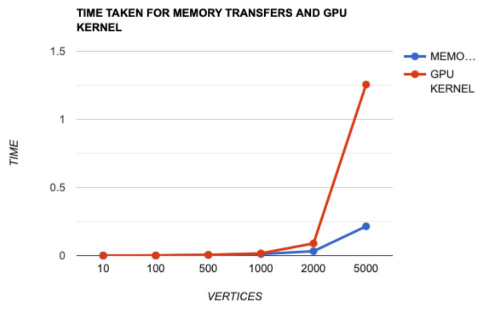


Figure 11: Time Taken to transfer memory and execute the GPU kernel.

Algorithm	Serial	OpenMP	CUDA
Dijkstra	8592	13216	647432
Bellman-Ford	8704	8704	650184
BFS	8752	8728	639032
FFT	12888	12896	647424
N-Queens	8960	8712	655264
FWM	13184	13184	643696

Table 8: Size of executables(in bytes)

Conclusions

- In terms of performance CUDA performs better than OpenMP for all algorithms except Dijkstra.
- In terms of ease of programmability OpenMP has the clear advantage, since it is simpler to write error free code in OpenMP as compared to CUDA.
- An added advantage of writing accelerator

code with OpenMP is that it can work on any accelerators.

- Hence the programmer needs to consider the factors above and adapt to architecture specific programming models.

References

[1] Marko J. Mišić, Đorđe M. Đurđević, and Milo V. Tomašević. Evolution and trends in gpu computing. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 289–294, 2012.

[2] David Luebke. Chapter 2 - data parallel computing. In David B. Kirk and Wen mei W. Hwu, editors, *Programming Massively Parallel Processors (Third Edition)*, pages 19–41. Morgan Kaufmann, third edition edition, 2017.

[3] Uditi and M. Arun. Comparative analysis of parallelised shortest path algorithms using open mp. In *2017 Third International Conference on Sensing, Signal Processing and Security (ICSSS)*, pages 369–378, 2017.

[4] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, page 163–174. IEEE, 2009.

[5] 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.

[6] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Effects of compiler optimizations in openmp to cuda translation. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, page 169–181, Berlin, Heidelberg, 2012. Springer-Verlag.

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

[8] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Trans. Parallel Comput.*, 1(2), feb 2015.

[9] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Design Automation Conference*, pages 52–55, 2010.

[10] Rongyu Dong, Huawei Cao, Xiaochun Ye, Yuan Zhang, Qinfen Hao, and Dongrui Fan. Highly efficient and gpu-friendly implementation of bfs on single-node system. In *2020 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 544–553, 2020.

[11] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, 19:297–301, 1965.

[12] Dariusz Puchała, Kamil Stokfiszewski, Mykhaylo Yatsymirskyy, and Bartłomiej Szczepaniak. Effectiveness of fast fourier transform implementations on gpu and cpu. In *2015 16th International Conference on Computational Problems of Electrical Engineering (CPEE)*, pages 162–164, 2015.

[13] Fan Zhang, Chen Hu, Qiang Yin, and Wei Hu. A gpu based memory optimized parallel method for fft implementation. *arXiv preprint arXiv:1707.07263*, 2017.

[14] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing – HiPC 2007*, pages 197–208, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.