# A* and Dijkstra's algorithm simultaneously on NVIDIA GPU using CUDA C++

Shishir Govinda M[1], Thrisha V S[2]
[1]Department of Computer Science Engineering, Sir M Visvesvaraya Institute of Technology, Bengaluru, India.
[2]Assistant Professor Department of Computer Science Engineering, Sir M Visvesvaraya Institute of Technology, Bengaluru, India.

***Abstract: A* algorithms is the best known path finding algorithm which is guided by a heuristic function to reach the goal node. Dijkstra's path finding algorithm works on greedy method i.e., it checks all the neighbouring nodes and is not guided by a heuristic function. In this paper, we try to achieve a much faster way of computing the path by combining both A* and Dijkstra's algorithm, where in we will be using A* algorithm from the start node and Dijkstra's algorithm from the goal node and at the point of intersection we will have our path. To further increase the speed of computation we will be making use of Nvidia's GPU computation via CUDA C++. We will also see the computational speed difference between serial CPU computation and parallel GPU calculation and comparing the results learned with Amdahl's Law in Parallel Computing. We will also look at some of the hindrances faced while enabling computation with Nvidia GPU using CUDA C++, which are non-existent with GPU computation.***

***Keywords: CUDA, C++, A*, Dijkstra, GPU, CPU, Path finding.***

## 1. INTRODUCTION

In this paper we will go in depth on how we use the computational power of GPU's in order to solve path finding algorithms such as A* and Dijkstra's, the two most widely known path finding algorithms [24]. We will also discuss the implementation, analysis, memory usage, performance metrics. For this path finding method we use the A* algorithm [18] from the starting node with the Manhattan distance as a heuristic function and simultaneously using the Dijkstra algorithm from the goal node until there exists a common node that is discovered by A* and Dijkstra [21][15]. This type of searching method were we use 2 same either same or different algorithms is often referred to as Bi-directional search[37][22].

This paper will also go in depth in relating and comparing the results and working of this algorithm with Amdahl's Law of Parallel Computing. [34] which states that in computer design, a system with improved resources should theoretically achieve a speedup in task execution latency at a fixed workload.

The law can be stated as:
*"The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"*.[35] [33]

A typical consumer level CPU consists of 2 to 24 cores, and about 2 threads per core. Where as consumer level Nvidia GPU has thousands of CUDA Cores, a mid tier Nvidia RTX 4060 has 3072 CUDA cores and the high end Nvidia RTX 4090Ti has 16384 CUDA Cores. This hardware level advantage allows GPU's to perform multiple tasks in parallel. But the main difference being CUDA cores are a type of floating point compute units where as CPU cores are general purpose compute units. A CPU core can handle all sorts of tasks given to it which require complex branching and decision making, but a CUDA core cannot perform such tasks they are designed to perform massive multitasking floating point mathematical calculations[30].

TeraFLOPS is the unit of measurement for the number of floating point operations completed in a second (TFLOPS - Tera floating point operations per second). A consumer level Intel Core i7 - 13700HX CPU has a TFLOPS value of 0.12, where as the Nvidia RTX 4060Ti has a TFLOPS rating of 22, which is significantly higher [31]. While a direct speed comparison between the CPU and GPU is not possible, this measure illustrates how we can leverage the GPU to expedite mathematical processes in comparison to a sequential CPU. [29].

Up until a few years ago GPU's were mainly used for gaming, which often require intense mathematical calculations for 3D and 2D image rendering. But now with the introduction of CUDA from Nvidia their GPU's are now also called as GPGPU's (General

Purpose GPU) [26]. This allows us to perform computations that are much more complex, with the help of CUDA developer kits provided by Nvidia.

## 2. MATERIALS AND METHODS

In this work, we will be comparing the same path finding Bi-directional algorithm [22] that is run on the same system and same compiler but both on CPU and GPU, with necessary changes made in order to support the data structure supported by the CUDA GPU. The main difference being since CUDA C++ does not directly support the use of C++ STL [32], we need to employ pointers and linearly arranged 2 dimensional vectors instead of just a 2D array that is supported by CPU and standard C++. It is imperative that we take into account the duration required for data to be transferred from the CPU memory to the GPU memory following computation, as external direct allocation to and from the CPU is prohibited[28].

| Item | Description |
|---|---|
| CPU | Intel Core i7 - 13700 HX |
| GPU | Nvidia RTX 4060 |
| CPU Memory | 16 GB 4800 MT/s |
| GPU Memory | 8 GB |
| Shared Memory | 16 GB |

Table 1: Hardware Specifications.

| Item | Description |
|---|---|
| Operating System | Windows 11 |
| Development Environment | Visual Studio 2022 |
| C++ version | C++ 17 |
| CUDA version | CUDA 12.5 |

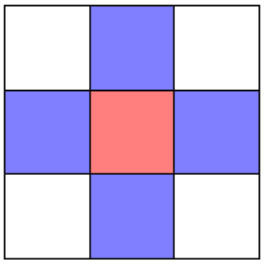
Table 2: System Specifications.

The above mentioned Hardware Specifications is the hardware specification and System Specifications is the system specification of the computer system used for development and testing of this particular Bi-directional path finding Algorithm.
This paper will look at mainly four different combinations of evaluating the computational speed.

| Device | Obstacles | Directions |
|---|---|---|
| CPU | 50% | Octagonal |
| CPU | 30% | Quad |
| GPU | 50% | Octagonal |
| GPU | 30% | Quad |

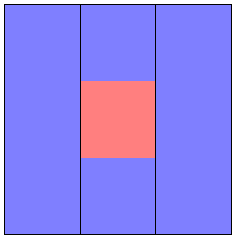
Table 3: System Specifications.

For the Quad Direction method we use the following coordinates:



{-1,0}, {1,0}, {0,1}, {0,-1}

For the Octagonal Direction method we use the following coordinates:



{-1,0}, {1,0}, {0,1}, {0,-1}, {-1,-1}, {1,1}, {-1,1}, {1,-1}

### 2.1 Bi-directional Algorithm on CPU

The A* algorithm is implemented using *unordered map* and *priority queue* available in the STL of C++ which is guided by *Manhattan Distance* as a heuristic function.

The Dijkstra's algorithm is also implemented using *unordered map* and *priority queue* available in the STL of C++ but there is no need of a heuristic function as Dijkstra's is a greedy algorithm and all the adjacent nodes are visited.

The combination of A* and Dijkstra as a bidirectional path finding algorithm happens inside a *while loop* where each iteration of the loop A* algorithm from the start node takes a step and Dijkstra algorithm from the goal node takes a step and before the start of next iteration A* map and Dijkstra's map are checked to see if a common *meet node* is available, In case a *meet node* is found then the process execution stops and path is found, and after all the iterations if there is no *meet node* then there exists no path between the start and goal node. The below Bi-directional Algorithm on CPU describes the framework of our algorithm that is run on the CPU with the use of *unordered maps, and priority queue*.

This algorithm uses 2 priority queues, 2 unordered maps for storing node from start and goal respectively, and 2 unordered maps for storing heuristic values of nodes from start and node respectively. The heuristic values are guided by Manhattan distance function which is the most well known heuristic function. For heuristic in case of Dijkstra it is set as the actual distance from the goal node.

This Bi-directional Algorithm on CPU works on a step by step basis, where each step moves forward by one iteration of each A* and Dijkstra algorithms, starting from A* algorithm. The initial node gets inserted into the unordered maps, then Dijkstra algorithm proceeds by one iteration checking if the node from A* is repeated during the A* search, in the subsequent iteration the reverse happens if any of the previously visited nodes from Dijkstra are again visited by the A* algorithm. This process goes on until a common node is found, or until all nodes are visited. If a common node exists then a path is found or else no path exists [23].

Algorithm 1: Bi-directional path finding on CPU

1: Procedure bi-directional(grid, start, goal):
2: *Find the shortest path from start to goal*
3: Let $Q_1$ be the priority queues of the start list
4: Let $Q_2$ be the priority queues of the goal list
5: Create a MAP $M_1$ for nodes from start
6: Create a MAP $M_2$ for nodes from goal
7: Create a MAP $G_1$ for heuristic values from start
8: Create a MAP $G_2$ for heuristic values from goal
9: PUSH($Q_1$, *start*)
10: while $Q_1$ is not empty and $Q_2$ is not empty do
11: {A* algorithm from the start}
12: *currentStart* = $M_1$.*TOP*
13: *POP* $M_1$
14: if *currentStart* in $G_2$ then
15: *PATHFOUND*
16: RETURN
17: end if
18: for *d* in Directions do
19: if $d_x$,$d_y$ is out of bounds then
20: continue
21: end if
22: if $G_1$ of *neighbour* is better than $G_1$ of *this* node then
23: PUSH ($Q_1$, *neighbour*)
24: UPDATE ($M_1$ of *neighbour* to *currentStart*)
25: end if
26: end for
27: {Dijkstra's algorithm from the goal}
28: *currentGoal* = $M_2$.*TOP*
29: *POP* $M_2$
30: if *currentGoal* in $G_1$ then
31: *PATHFOUND*
32: RETURN
33: end if
34: for *d* in Directions do
35: if $d_x$,$d_y$ is out of bounds then
36: continue
37: end if
38: if $G_2$ of *neighbour* is better than $G_2$ of *this* node then
39: PUSH ($Q_2$, *neighbour*)
40: UPDATE ($M_2$ of *neighbour* to *currentStart*)
41: end if
42: end for
43: end while=0

By assigning priority queues, it is ensured that the best available node is picked and is moved on to further iterations, and by assigning unordered maps, it is ensured that only feasible nodes are passed on the next iteration and only these maps are searched for a common meet node, if no such node exists then the map is popped out of memory on to which a new map will be inserted during the current iteration.

a. Bi-directional Algorithm on GPU

This method makes use of *CUDA (Compute Unified Device Architecture)* from Nvidia specifically made for Nvidia GPU's[8]. CUDA is a parallel computing platform and application programming interface. CUDA allows software developers to use a CUDA-enabled graphics processing unit (GPU)[19] for general purpose processing, an approach known as General Purpose GPU (GP GPU) computing [27].

To implement Bi-directional Algorithm for the computation on GPU, we have to follow a specific methods because we cannot directly implement the C++ STL functions such as *queues, maps, vectors and more*, and CUDA Kernel functions do not return any values, they are void only functions. Hence there is a need to prepare an algorithm that complies with the capabilities of CUDA functioning.

The primary step to make an algorithm support CUDA is to move memory allocated to *CPU RAM* to the *GPU VRAM*, which is a more robust memory device that is built to support the parallel processing capabilities of CUDA[7].

The CUDA architecture works on three basic parts - *grids, blocks and threads*. *Grids* contains multiple *blocks* and they contain multiple *threads*. For this implementation we have used a block size of 256, which is the standard block size used for most CUDA algorithms. The computation is done by threads and these threads utilize shared memory onto which data is sent through the CPU using CUDA kernel functions. The CUDA functions used in this algorithm are *cudaMalloc, cudaMemcpy, cudaDeviceSynchronize, and cudaFree* these are the basic CUDA functions that are used to allocate CPU

memory to GPU memory and back again to CPU memory[3].

CUDA architecture uses three different memories that have different scopes they are *global, host and device* [25].

- *Global scope*: This refers to the GPU memory that is accessible to all of the GPU's threads and blocks.
- *Device*: The Device refers to the GPU and it's associated memory. The lifetime of this memory is the CUDA Kernel context.
- *Host*: Host refers to the CPU and it's memory. This is the main memory which is inaccessible directly by the GPU.

Due to CUDA's lack of support for 2D arrays or vectors, there is a need to convert the 2D grid into a 1D array. This array will be passed on to the GPU with the use of pointer. These are some of the utility functions that are need for the efficient conversion of CPU compute able data to GPU compute able data in the CUDA API framework. This conversion of 2D array to 1D is done in the following way:

Conversion of 2D array to 1D array:
*grid* is the 1D array *map* is the 2D array for $i = 0$ to *mapSize* do
for $j = 0$ to *mapSize* do
$grid[i * mapSize + j] = map[i][j]$
end for
end for=0

Following conversion, the GPU receives the pointer to the first element of the 1D array as well as the total number of nodes, or elements, in that specific array[36].
The method of implementation here: this implementation makes use of two GPU CUDA kernels namely Bi-directional Algorithm on GPU and Bi-directional Algorithm on GPU since CUDA kernels cannot return values both these kernels are of return type *void*. Both these kernels are of the following block size and grid size, these are the required parameters while calling a CUDA kernel:

$$blockSize = 256$$

$$gridSize = \frac{(numNodes + blockSize - 1)}{blockSize}$$

Both these kernels use the same way of selecting a new node for the next iteration, here *tid* is the index of the next node, and *nodes* is the pointer that is used to access all nodes sent to GPU as 1D array. *tid* = *blockIdx.x* ∗ *blockDim* + *threadIdx.x currentNode* = *nodes*[*tid*]

The below are The algorithms employed in the implementation of *A* kernel* and *Dijkstra Kernel*, both these kernels use the same data pointers in order to ensure both these algorithms are computing the same data simultaneously. Since CUDA can only handle 1D arrays, unlike the CPU approach that allowed us to use STL data structures, the data must be converted and verified to be compute able as a 1D array. And the pointers to these 1D arrays will be passed on the GPU with the help of CUDA kernel functions as mentioned above.
The algorithm must verify that the appropriate memory architecture was utilized for the particular kernel functions, because in this specific algorithm two kernels are working on the same set of pointer data, and the memory architectures are scope and lifetime constraint.

In both Bi-directional Algorithm on GPU and Bi-directional Algorithm on GPU, a new node is picked based on the tid (thread ID) of that memory block and a new *neighbourNode* is calculated to see if it is closer compared to the *currentNode* regarding distance to the goal node, and if it is better then then *neighbourNode* will be set as the *currentNode* and in the Bi-directional Algorithm on GPU algorithm this is calculated using the Manhattan heuristic function at each *neighbourNode* and in the Bi-directional Algorithm on GPU all the nodes that surround the *currentNode* based on how many directions there are, either 4 or 8 of the surrounding nodes will be calculated to see if the connect to the node from Bi-directional Algorithm on GPU and if a common node exists then a path is found or else there is no path from start to goal nodes. These two algorithms run concurrently on the GPU using the same set of pointer data[16].

Algorithm 1 A* Kernel for the GPU

1: Input grid, nodes and data pointers
2: *tid* = *blockIdx.x* ∗ *blockDim* + *threadIdx.x*
3: if *tid* is out of bounds then
4: return
5: end if
6: *currentNode* = *nodes*[*tid*]
7: for *d* in Directions do
8: if $d_x$,$d_y$ is out of bounds then
9: continue
10: end if
11: *newX* = *currentNode.x* + *directions*
12: *newY* = *currentNode.y* + *directions*
13: if *newX,newy* is out of bounds then

14: continue
15: end if
16: Initialize: *neighbourNode*(*newX,newY* )
17: if *neighbourNode* is visit able and *currentNode* is better than *neighbourNode* then
18: *neighbourNode.heuristic* = *heuristic*(*neighbourNode,numNodes* − 1)
19: *neighbourNode.parent* = *currentNode*
20: *fromStart* = *currentNode*
21: end if
22: end for=0

Algorithm 2 Dijkstra Kernel for the GPU

1: Input grid, nodes and data pointers
2: *tid* = *blockIdx.x* ∗ *blockDim* + *threadIdx.x*
3: if *tid* is out of bounds then
4: return
5: end if
6: *currentNode* = *nodes*[*tid*]
7: for *d* in Directions do
8: if $d_x,d_y$ is out of bounds then
9: continue
10: end if
11: *newX* = *currentNode.x* + *directions*
12: *newY* = *currentNode.y* + *directions*
13: if *newX,newy* is out of bounds then
14: continue
15: end if
16: Initialize: *neighbourNode*(*newX,newY* )
17: if *neighbourNode* is visit able and *currentNode* is better than *neighbourNode* then
18: *neighbourNode.parent* = *currentNode*
19: *fromStart* = *currentNode*
20: end if 21: end for=0

## 3. RESULTS

The performance metrics recorded for both the control algorithm run on CPU and the proposed algorithm run on the GPU using CUDA[1][14], the execution time are measured from the C++ built in library function *CLOCK()* which is available in the *time.h* header file. The performance is measured from the time of either CPU or GPU function call and timer is stopped after the function call returns a result value which is *NULL* for the GPU as CUDA kernels do not return any value and for CPU functions the *Path* is returned as a *BOOL* meaning either a path is found or not.In short the time be measured before the function call and after the function execution is completed, and the time taken for all the other operations such as memory transfer from CPU RAM to GPU VRAM and vise-versa are not measured as they are out of the scope of this paper. The test is an average of 10 trials per set of nodes, the sets of nodes are *10.000, 1.000.000, 25.000.000 and 56.250.000* nodes. From the above tables *50% CPU vs GPU* and *30% CPU vs GPU* the difference in performance is obvious as the number of nodes increase from 10.000 to 56.25 million nodes, the difference is also evident in the computation with number of nodes either 4 or 8. The below are the time taken performance metrics tabulated as below.

| Nodes | CPU | GPU |
|---|---|---|
| 10000 | 0.0056 | 0.0019 |
| 1000000 | 0.0262 | 0.002 |
| 25000000 | 0.1932 | 0.0027 |
| 56250000 | 0.23 | 0.0079 |

Table 4: 50% Obstacles with Octal Directions measured in seconds

| Nodes | CPU | GPU |
|---|---|---|
| 10000 | 0.0138 | 0.003 |
| 1000000 | 0.6217 | 0.0029 |
| 25000000 | 3.5718 | 0.0077 |
| 56250000 | 18.4276 | 0.0121 |

Table 5: 30% Obstacles with Quad Directions measured in seconds

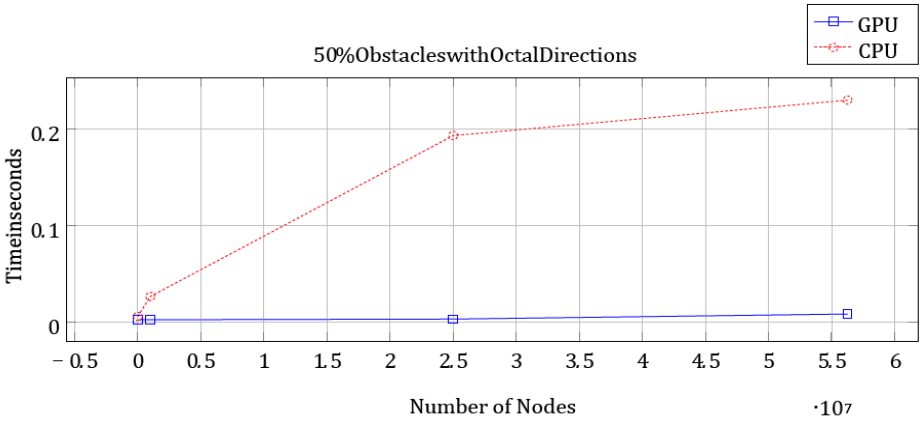



Figure 1: CPU vs GPU performance measured in seconds

## 4. DISCUSSION

The aforementioned graphs demonstrate how much better the GPU method performs than the CPU algorithm, this is more evident when looking at tests with higher number of nodes [17]. This paper tests these algorithms up to 56.25 million nodes from 10.000 nodes, both these algorithms take almost the same amount of time at small number of nodes this is because the efficiency CUDA parallel programming makes a significant difference when large number of operations are needed to be performed, with a small sample size it takes almost the same time as CPU even though CPU operates sequentially[13].

At *56.25 million* nodes with 30% obstacles and only quad direction CPU takes about 18 seconds to find the path, where as the CUDA GPU is able to find the path in under 0.012 seconds this as increase of *1500x* at it's peak, this sort of computing performance is borderline impossible to achieve with sequential CPU computation. And with 50% obstacles and octal directions the CPU took about 0.23 seconds and the GPU took about 0.0079 seconds at 56.25 million nodes this is an increase in performance of around *30X*[10].

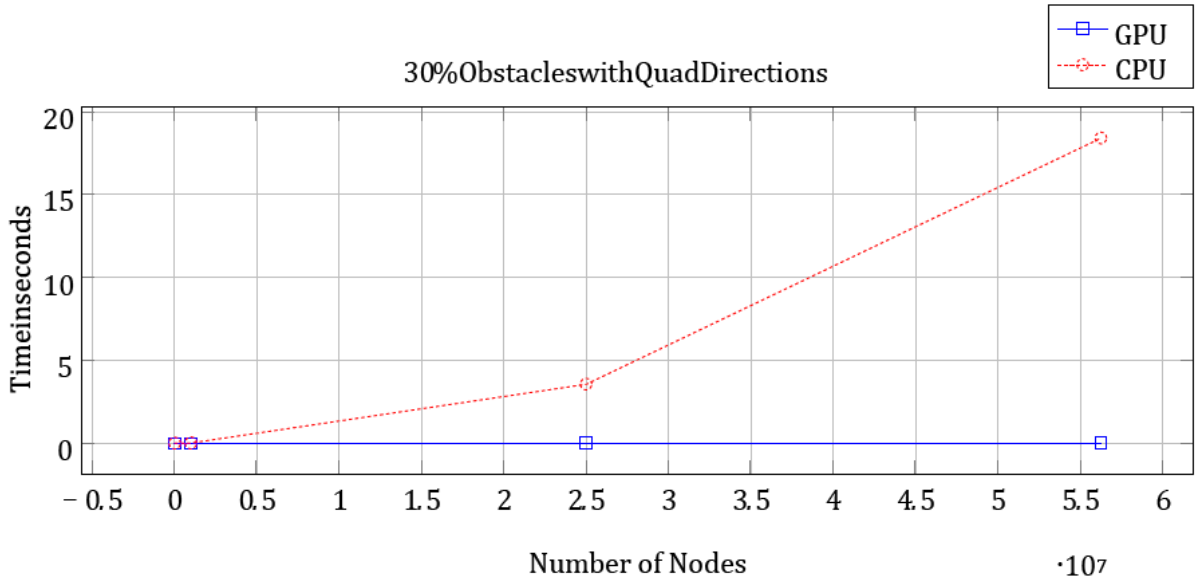
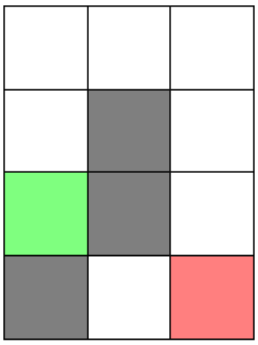


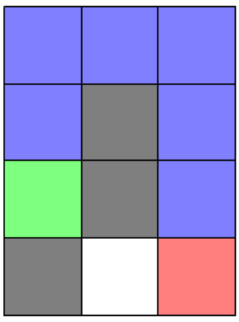

Figure 2: CPU vs GPU performance measured in seconds

The performance again is so evident in the set with 30% obstacles because the probability of finding a better node reduces by half when compared to octal directions, and since the direct diagonal path is unavailable with quad directions each step increases complexity by a varying factor depending on the map type.

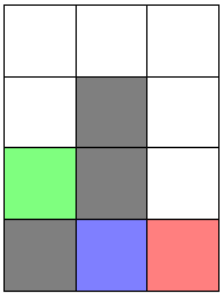


Example start to goal instance

From the above diagram, let *GREEN* be the start node and *RED* be the goal node. With *QUAD* and *OCTAL* distance the path will have to be as below:



Path with Quad directions



Path with Octal directions

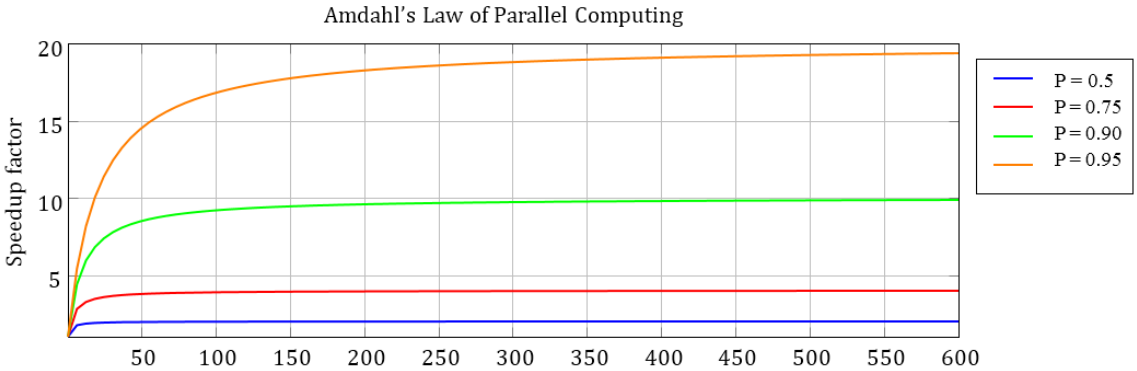
As the number of nodes that need to be calculated increases at a much higher range as compared to octal route finding, it is evident from the preceding diagrams why quad path finding takes a significantly longer time. In the above given scenario the QUAD path had to calculated for 6 nodes excluding the start and goal nodes where as in the octal path only 1 node had to be calculated for, with multiple such scenarios the number of nodes that will have to be calculated for increases significantly even though the map contains only 30% of obstacles which is 20% less compared to the map tested with octal which has 50% obstacles. And when expanded to multiple millions of nodes the complexity adds on resulting in slower computation. But with the aid of CUDA parallel programming it becomes much faster to compute as the nodes get distributed among multiple threads of the GPU.
With a small set of nodes the performance is not very evident because GPU takes around the same amount of time to parallely process a small set of nodes that the CPU with it's intense sequential processing power takes to process the set of nodes, but as the quantity of distinct path node operations increases the time taken by the CPU also increases significantly as parallel computation is much faster compared to sequential processing.

## 5. CONCLUSION

This paper contains the implementation of A* and Dijkstra's path finding algorithms simultaneously from the start node and goal node, this method of implementing algorithms from two directions is also called as Bi-directional path finding algorithm. Testing of the implementation has been done using the CPU as a control while the main goal was to measure the performance gains made by implementing the algorithm parallely on the GPU using Nvidia's CUDA methods.

The parallel implementation of the algorithm on GPU with CUDA has proven to by far superior in terms of raw computational performance of the path finding from start node to goal node, with multiple parameters such as percentage of map that is filled with obstacles and the number of directions each node can move forward to. While with a small number of nodes the performance gains is not visible because the superior computational power of the CPU is able to calculate the path for such paths while taking the same time as the GPU would, but more than a million nodes the superiority of parallel computation is very significant.

Comparing our results with Amdahl's Law of Parallel Computing [33], it is visible that the Amdahl's theoretical representation of the computing speed in parallel processing is similar to the graph of results available from the testing of our A* and Dijkstra Bi-directional path finding[9]. The below graph is a typical representation of Amdahl's law of parallel computing[12]. Amdahl's Law only applies in situations where the problem size is fixed. As a result, up until a certain point, adding more processors in parallel will not result in a greater overall performance improvement. [6][20]. The following is a formulation of Amdahl's Law.[11]:




$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

- $S_{latency}$: refers to the potential acceleration of the task's execution.
- $s$: is the acceleration of the task's portion that gains from enhanced system resources.
- $p$: is the percentage of the execution time that was initially consumed by the component that benefited from better resources.

Number of processors

Although CUDA C++ programming is complex as there is no support of C++ STL functions inside the CUDA kernels and all the that ought to be parsed has

to be as a kind of 1D arrays and only pointer of that 1D array can be parsed. Variables allocated in CPU RAM cannot be directly used to parse to the CUDA Kernels, It is necessary to use specific CUDA memory functions to allocate data to the GPU's VRAM. Additionally, after the CUDA kernel has executed, the memory from the CUDA kernel must be parsed back into CPU RAM memory locations using specific CUDA functions. The memory allocated on the CUDA GPU must also be freed because the GPU does not have garbage collection available.

The algorithm's CPU implementation is comparatively much simpler because we can use any of the current C++ STL [4] [2] functions, which make it easy to manage data in memory and enable parsing of the data rather than converting it all to 1D and parsing the pointers to those 1D arrays. However, there is an obvious distinction in the processing capability of the CPU and GPU as the number of nodes. [5].

This paper was able to successfully show that when implemented effectively and with proper management of data it is much faster to compute on the GPU parallely when compared to the CPU sequentially, although CUDA has it's limitations in terms of availability of standard C++ STL data structures, passing only pointers, memory transfer from CPU to GPU, it is possible to develop a system that can leverage the benefits of parallel programming that CUDA-enabled GPU's offer.

## REFERENCES

Bibliography


[1] Hojin Choi, SeongJun Choi, and SeogChung Seo. "Parallel Implementation of Lightweight Secure Hash Algorithm on CPU and GPU Environments". In: Electronics 13.5 (2024), p. 896.

[2] Patrick Diehl, Steven R Brandt, and Hartmut Kaiser. "C++ Standard Library". In: Parallel C++ Efficient and Scalable High-Performance Parallel Programming Using HPX. Springer, 2024, pp. 17–32.

[3] Pranay R Kommera, Suresh S Muknahallipatna, and John E McInroy. "Optimized CUDA Implementation to Improve the Performance of Bundle Adjustment Algorithm on GPUs". In: Journal of Software Engineering and Applications 17.04 (2024), pp. 172–201.

[4] Ruben Laso, Diego Krupitza, and Sascha Hunold. "pSTL-Bench: A Micro-Benchmark Suite for Assessing Scalability of C++ Parallel STL Implementations". In: arXiv preprint arXiv:2402.06384 (2024).

[5] Marcos Nogueira Lobo de Carvalho et al. "Performance analysis of distributed GPU-accelerated task-based workflows". In: Proceedings 27th International Conference on Extending Database Technology (EDBT 2024): Paestum, Italy, March 25-March 28. OpenProceedings. 2024, pp. 690–703.

[6] Guido Schryen. "Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis". In: Journal of Parallel and Distributed Computing 187 (2024), p. 104835.

[7] Neda Seifi and Abdullah Al-Mamun. "Optimizing Memory Access Efficiency in CUDA Kernel via Data Layout Technique". In: Journal of Computer and Communications 12.5 (2024), pp. 124–139.

[8] Kohei Yoshida et al. "Analyzing the impact of CUDA versions on GPU applications". In: Parallel Computing 120 (2024), p. 103081.

[9] Kirk W Cameron. "Adventures Beyond Amdahl's Law: How Power-Performance Measurement and Modeling at Scale Drive Server and Supercomputer Design". In: Journal of Computer Science and Technology 38.1 (2023), pp. 80–86.

[10] Marwan Abdelatti, Manbir Sodhi, and Resit Sendag. "A multi-gpu parallel genetic algorithm for large-scale vehicle routing problems". In: 2022 IEEE High Performance Extreme Computing Conference (HPEC). IEEE. 2022, pp. 1–8.

[11] Donald Ene and Vincent Ike Anireh. "Performance Evaluation of Parallel Algorithms". In: CoRR (2022).

[12] Mike Bailey. "Parallel Programming: Speedups and Amdahl's law". In: Oregon State University (2021).

[13] Qunsong Zeng et al. "Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing". In: IEEE Transactions on Wireless Communications 20.12 (2021), pp. 7947–7962.

[14] Mouna Afif, Yahia Said, and Mohamed Atri. "Computer vision algorithms acceleration using graphic processors NVIDIA CUDA". In:

Cluster Computing 23.4 (2020), pp. 3335–3347.

[15] Ade Candra, Mohammad Andri Budiman, and Kevin Hartanto. “Dijkstra’s and a-star in finding the shortest path: A tutorial”. In: 2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA). IEEE. 2020, pp. 28–32.

[16] Mayez A Al-Mouhamed, Ayaz H Khan, and Nazeeruddin Mohammad. “A review of CUDA optimization techniques and tools for structured grid computing”. In: Computing 102.4 (2020), pp. 977–1003.

[17] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. “A study of the fundamental performance characteristics of GPUs and CPUs for database analytics”. In: Proceedings of the 2020 ACM SIGMOD international conference on Management of data. 2020, pp. 1617–1632.

[18] Sumit Sharma, Shashwat Srijan, and Vidhya JV. “Parallelizing Bidirectional A* Algorithm”. In: Intelligent Systems and Computer Technology. IOS Press, 2020, pp. 558–562.

[19] Reyyan Tekin, Houssam-Eddine ZAHAF, and Giuseppe Lipari. “Programming in NVDIA GPUs using CUDA”. In: HAL Open Science (2019).

[20] Ashur Rafiev et al. “Speedup and power scaling models for heterogeneous many-core systems”. In: IEEE Transactions on Multi-scale computing systems 4.3 (2018), pp. 436–449.

[21] D Singh and N Khar. “Modified Dijkstra’s algorithm for dense graphs on GPU using CUDA”. In: Indian Journal of Science and Technology 9.33 (2016), pp. 1–9.

[22] Lalinthip Tangjittaweechai et al. “Fast bidirectional shortest path on GPU”. In: IEICE Electronics Express 13.6 (2016), pp. 20160036–20160036.

[23] Amadou Chaibou and Oumarou Sie. “Improving global performance on GPU for algorithms with main loop containing a reduction operation: case of Dijkstra’s algorithm”. In: Journal of Computer and Communications 3.8 (2015), pp. 41–54.

[24] Yichao Zhou and Jianyang Zeng. “Massively parallel A* search on a GPU”. In: Proceedings of the AAAI conference on artificial intelligence. Vol. 29. 1. 2015.

[25] Panagiotis D Michailidis and Konstantinos G Margaritis. “Accelerating kernel density estimation on the GPU using the CUDA framework”. In: Applied Mathematical Sciences 7.30 (2013), pp. 1447–1476.

[26] Jayshree Ghorpade et al. “GPGPU processing in CUDA architecture”. In: arXiv preprint arXiv:1202.4347 (2012).

[27] Michael McCool, James Reinders, and Arch Robison. Structured parallel programming: patterns for efficient computation. Elsevier, 2012.

[28] Marko J Mi˘si´c, Dor~~d~~e M Dur~~d~~evi´c, and Milo V Toma˘sevi´c. “Evolution and trends in GPU computing”. In: 2012 Proceedings of the 35th International Convention MIPRO. IEEE. 2012, pp. 289–294.

[29] Randal E Bryant and David Richard O’Hallaron. Computer systems: a programmer’s perspective. Prentice Hall, 2011.

[30] Rob Farber. CUDA application design and development. Elsevier, 2011.

[31] Petr Pospichal et al. “Acceleration of grammatical evolution using graphics processing units: computational intelligence on consumer games and graphics hardware”. In: Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. 2011, pp. 431– 438.

[32] Martin Reddy. API Design for C++. Elsevier, 2011.

[33] Mark D Hill and Michael R Marty. “Amdahl’s law in the multicore era”. In: Computer 41.7 (2008), pp. 33–38.

[34] Yuan Shi. “Reevaluating Amdahl’s law and Gustafson’s law”. In: Computer Sciences Department, Temple University (MS: 38-24) (1996).

[35] John L Gustafson. “Reevaluating Amdahl’s law”. In: Communications of the ACM 31.5 (1988), pp. 532–533.

[36] David P Rodgers. “Improvements in multiprocessor system design”. In: ACM SIGARCH Computer Architecture News 13.3 (1985), pp. 225–231.

[37] Ira Pohl. Bi-directional and heuristic search in path problems. Tech. rep. SLAC National Accelerator Laboratory (SLAC), Menlo Park, CA (United States), 1969.