Accelerating Breadth-First Search via GPU-Driven Dynamic Parallelism

Niklas Mäckle
Hochschule der Medien
Stuttgart, Germany
nm067@hdm-stuttgart.de

***Abstract*—This paper examines whether GPU-driven dynamic parallelism is capable of increasing the performance of the breadth-first search algorithm (BFS), an inherently sequential algorithm used for finding the shortest path from a starting vertex towards a target vertex in a graph.**
Due to its sequential nature, implementations on the GPU require its kernel to run multiple times until all nodes in a graph have been visited and a path can be found.
Dynamic parallelism is used to eliminate GPU idle times in between these kernel launches which are caused by them being executed on the CPU.
Performance tests are run on graphs that mimic the setup of flow fields, graphs whose vertices are aligned in a regular grid with zero to four connections to their neighboring vertices.
The resulting data shows that dynamic parallel increases the performance of BFS by up to six times in the tested setups, contradicting the statements of some older papers on dynamic parallelism that claim that large overheads caused by its usage cause a reduction in performance instead of an increase.
Overall, the measured data highlights, how dynamic parallelism is able to reduce required run times for kernels that have to run multiple times to reach their final state.

***Index Terms*—cuda, bfs, dynamic parallelism**

I. Introduction

The **breadth-first search (BFS)** is one of the most important algorithms used to find paths in graphs alongside others such as the Dijkstra and A* algorithms. [1] [2]
These graphs are made up of entities, referred to as vertices, and the relations between these vertices, referred to as edges. BFS assigns each vertex in the graph a level ranging from zero at the target vertex up to however many levels are required to cover the full graph.
These levels can then be followed in descending order beginning at a start vertex to find a path to the target vertex consisting of the lowest number of edges possible.
An example use case of BFS are **flow fields**, regular grid-based graphs. Here, the vertices represent the cells in the grid connected to their neighboring vertices by edges representing whether the cells are neighbors or not. They are used in e.g. video games where large crowds of units have a common destination. [3]
Typically, BFS is implemented on the CPU as it is an inherently sequential algorithm. The main reason for this is, that each vertex in the graph has exactly one assigned level that it can only receive if neighboring vertices assigned to a previous level have already been covered. Have a look at figure 1 for an example representation of this.

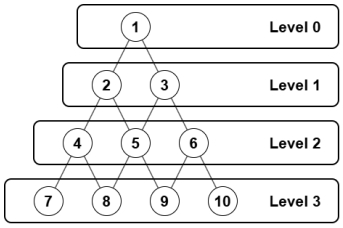

Fig. 1: BFS algorithm on the CPU. Vertex numbers represent loop iteration they are handled in. Level 0 is the target vertex

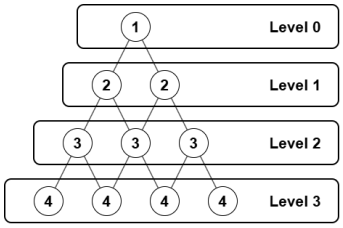

Fig. 2: BFS algorithm on the GPU. Vertex numbers represent kernel launch they are handled in. Level 0 is the target vertex

But when graphs become very large consisting of multiple thousands or millions of vertices and edges, moving BFS from the CPU onto the GPU still becomes a good option to improve the performance of path calculations in these large data sets.
Even though the algorithm mainly runs in a sequential order, this is possible, by e.g. assigning one thread for each vertex included in a given level of the BFS. That way, all vertices in that level can check their neighboring vertices for whether they have been assigned a level yet in parallel instead of in sequence, creating potential performance improvements, especially when many vertices are part of the same level.This approach is called **vertex-based BFS**. An example representation of this can be seen in figure 2.

Similar to this an edge-based BFS approach is possible, but this paper will focus on the former.
While the algorithm can cover each level of the graph in parallel this way it still has to be executed once per level, which can slow it down as the CPU has to execute each new

kernel for the levels in the graph separately.
This problem can be tackled by moving the launch of new GPU kernels from the CPU into the kernel itself. In CUDA, this practice is called **dynamic parallelism**. [4] [5]
The CUDA Toolkit developed by NVIDIA provides a development environment for GPU programming that includes features such as dynamic parallelism and was used to create all the implementations in this paper. We will explore whether dynamic parallelism is actually capable of speeding up the execution of path calculations in graphs of different sizes.
This will be achieved by testing the theory that dynamic parallelism will create a performance boost by removing GPU idle time between kernel launches normally caused by executing the kernel on the CPU while not creating too much of an additional overhead by its usage.

II. Background

Especially in video games like e.g. Vampire Survivors where large amounts of enemy units all have the same common goal of reaching the player, the game's performance benefits from fast path finding. BFS can be used to generate a flow field that has to be created once before it can be used by all units. This results in units not having to do individual path finding, but simply sampling the flow field for their path. [6] [3]
However, in the past, multiple papers have claimed that dynamic parallelism is unable to yield any relevant performance improvements when used due to added large overhead created by its usage. [7] [8]
If this is no longer an issue in more recent versions of CUDA, dynamic parallelism has the potential to speed up recursive or iterative algorithms that have parallel components.
The observations in this paper will show whether the statements in these papers are still accurate using more recent CUDA versions or whether other, more recent papers that include dynamic parallelism in their implementations that claim performance gains by its usage stand true. [9]

III. Related Work

Due to BFS being a widely used algorithm, it has been implemented and used in many different papers. A comprehensive overview of its implementation along with some performance improvements can be found in the book "Programming Massively Parallel Processors" written by Hwu et al. [1] The usage of dynamic parallelism isn't included though.
As BFS is a path finding algorithm used for the creation of shortest paths by edge count, which can be helpful in many different scenarios such as GPS navigation, there is a given demand for it to run as fast as possible. Multiple papers already discuss potential performance improvements for BFS, including some that also use dynamic parallelism to remove the performance penalty from iterating between CPU and GPU. [10] [11] [12]

The graphs used as test data in this paper are equivalent to the graphs used in flow fields.
Flow fields, however, are a less discussed topic in scientific research compared to BFS itself, as their implementation mainly stems on the BFS algorithm in their core with the difference of not including an early end of the algorithm if a provided start vertex for the path is reached. This is part of the reason why most general information on them can be found in blog posts and other sources like videos instead of papers. [3] [13] [14] [15]
In addition to that, multiple papers on their performance can be found, too. [19] [20]

Dynamic parallelism itself has been used and experimented with in many different algorithms across multiple different fields. [9] [10] A good introduction is given by NVIDIA's Stephen Jones. [5]
Among other sources, dynamic parallelism has also been covered in university lectures and some blog posts by NVIDIA. [16] [17]

IV. Method

A **vertex-based breadth first search** GPU kernel was implemented, based on the information found in the "Programming Massively Parallel Processors" book, including two described improved versions of it using CUDA and C++. [1]
The first improvement to the basic kernel uses two frontier buffers to launch threads equal in size to the number of affected vertices in each level of the BFS algorithm. As the basic GPU kernel does not do this, it launches one thread per vertex regardless of whether the vertex actually has to be checked in a level or not, creating many idle threads unnecessarily occupied by the kernel.
The second improvement is based on the first one and uses shared memory to move the atomic writes that calculate the size of the next frontier of every kernel launch from global memory into shared memory including which vertices are included in that next frontier. In an added final step of the kernel, the locally calculated frontier sizes of each block are added along with the combination of the identified frontier vertices to create the next frontier.
In addition to these three GPU implementations of the BFS kernel, a modified version of the improved algorithm was developed that uses dynamic parallelism to remove the requirement of going back and forth between CPU and GPU to launch the kernels. Listing 1 shows the code added at the end of the kernel that enables this.

Listing 1: Code Snippet BFS Dynamic paralleism

```
1    ...
2   __threadfence();

4   __shared__ bool isLastBlock;
5   if (threadIdx.x == 0) {
6     uint32 arrivalId =
7             atomicAdd(blocksFinished, 1);
8     isLastBlock = (arrivalId == gridDim.x - 1);
9   }

11  __syncthreads();
```

```
13    if (isLastBlock && threadIdx.x == 0) {
14      *blocksFinished = 0;
16      uint32 totalCount = *numCurrFrontier;
18      if (totalCount > 0) {
19        uint32 nextBlockCount = (totalCount +
20                  blockDim.x - 1) / blockDim.x;
22        *numCurrFrontier = 0;
24        bfsDynamicParallel
25              <<<nextBlockCount, blockDim.x,
26              0, cudaStreamTailLaunch >>>
27              (csrGraph, level, currFrontier,
28              prevFrontier, totalCount,
29              numCurrFrontier, currLevel + 1,
30              finalLevel, blocksFinished);
31      }
32      else {
33        *finalLevel = currLevel;
34      }
35    }
```

First, all the blocks executed in the kernel wait for each other to finish their work. The block finishing last then uses a single thread to set up the execution of the next kernel launch until the frontier of vertices is empty ending the recursive kernel launch and finishing the BFS execution.

Two single-threaded CPU-based versions of BFS were also implemented to observe on what grid sizes the CPU performs better than on the GPU.
The first one is a simple translation of the basic BFS kernel used on the GPU from CUDA to C++ for it to be able to run sequentially on the CPU.
The second version is a more suitable method for solving BFS on the CPU using *std::queue*.
Experiments with multithreaded CPU implementations of these algorithms turned out to run slower in the tested scenarios and should be looked at further in the future.

Graphs used in the comparison of these algorithms are stored as a sparse matrix using the **compressed sparse row (CSR)** format.
The tested graphs all have similar characteristics. Their vertices are arranged in a regular grid similar to how they would be layed out in a flow field, meaning that all vertices have zero to four edges connecting them to their surrounding neighboring vertices. Edges allow movement between vertices in both directions.

V. Setup of the Experiment

All six BFS implementations are compared with each other in two different graph setups, one with obstacles in the graph and one without them. The graph's grid sizes include **8 x 8**, **32 x 32**, **128 x 128**, **512 x 512** and **2048 x 2048** vertices.
The test results are measured by running each algorithm 100 times. Kernels on the GPU are run one additional time as a warm up run that normally takes longer than following kernel executions.
Measurements start before any VRAM buffers have been

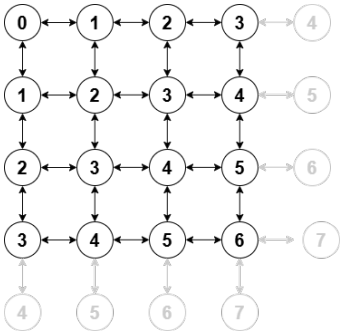

Fig. 3: Basic graph setup including example levels

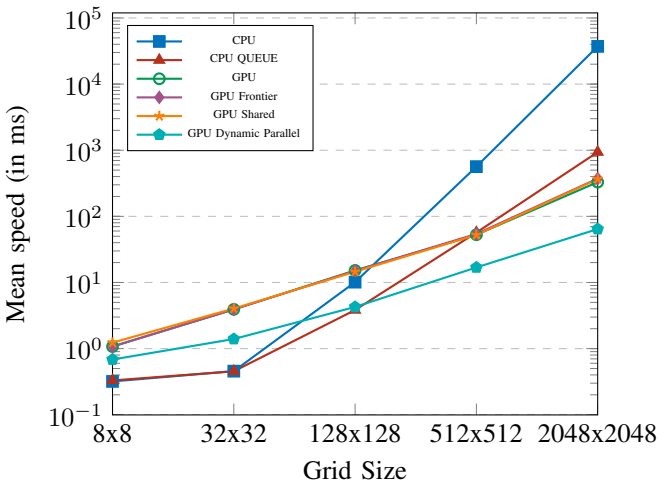

Fig. 4: Measurement results: BFS w/o obstacles

created for the GPU versions and end after all data has been synced with the CPU for access there. All measurements exclude any cleanup steps such as buffer deletion and deletion of dynamically allocated memory.

All results are measured in a test setup using an NVIDIA RTX 3080 graphics card and an AMD Ryzen 7 3800X CPU on Windows 11. The version of CUDA used is V12.6.85.

A. Setup 1: No obstacles

The first setup is similar to that shown in figure 3. It does not include any obstacles, meaning that all vertices in the graph have between two to four edges connecting them to their neighboring vertices:

- **8 x 8:** 64 vertices (224 edges)
- **32 x 32:** 1.024 vertices (3.968 edges)
- **128 x 128:** 16.384 vertices (65.024 edges)
- **512 x 512:** 262.144 vertices (1.046.528 edges)
- **2048 x 2048:** 3.556.119 vertices (12.871.006 edges)

The mean runtime of all six algorithms for this graph setup can be seen in figure 4 and in more detail in Appendix A

B. Setup 2: Including obstacles

The second graph setup includes obstacles created by using noise generated by FastNoiseLite to remove some of the

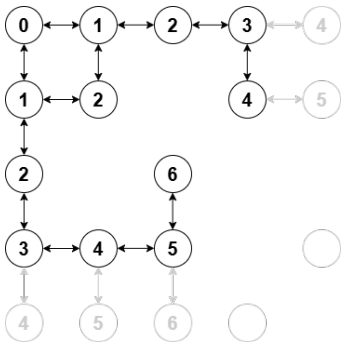

Fig. 5: Graph setup with obstacles including example levels

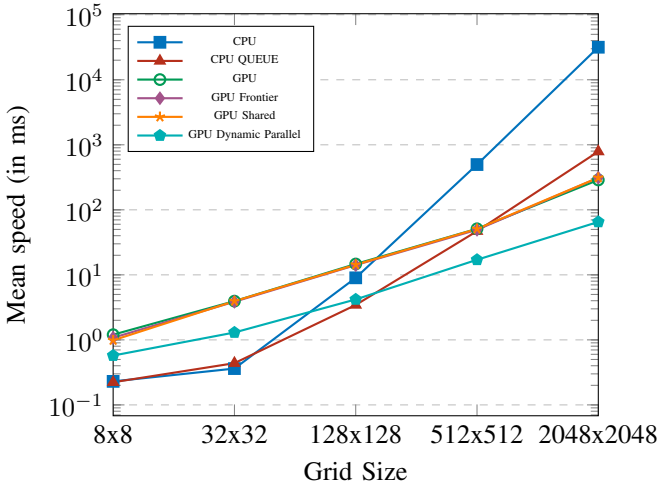

Fig. 6: Measurement results: BFS w/ obstacles

vertices from the graph [18]. This is done to observe whether more variance in edge counts between vertices results in different performance or whether it stays roughly the same due to the overall low outgoing edge count of each vertex that leads to relatively low workloads per vertex. An example of how such a graph can look like can be seen in in figure 5.

In this graph setup vertices have zero to four edges going out towards neighboring vertices resulting in the following vertex and edge counts:

- **8 x 8:** 61 vertices (206 edges)
- **32 x 32:** 884 vertices (3.116 edges)
- **128 x 128:** 13.949 vertices (50,196 edges)
- **512 x 512:** 222.211 vertices (803.748 edges)
- **2048 x 2048:** 4.194.304 vertices (16.769.024 edges)

The mean speed each of the algorithms performs at for this setup can be seen in figure 6. As with setup 1 more detailed measurements are included in the appendices under Appendix B

VI. Evaluation of measured data

Looking at the measured data, we can see that the usage of dynamic parallelism leads to massive performance gains over the other GPU kernel versions of the algorithm running up to six times as fast as the other GPU implementations.

Fig. 7: NVIDIA NSight Sytems Capture of the BFS Dynamic Parallelism Kernel with obstacles at 32 x 32 graph size

Additionally, it starts outperforming the sequential CPU algorithms at around 128 x 128 vertices while only taking twice as long to run for smaller vertex counts. The main reason for this is that the allocation of GPU memory creates a large overhead resulting in slower runtimes than the sequential CPU method. An example of this can be seen in figure 7. This overhead is compensated for by faster runtimes of the algorithm itself in larger graphs.

In general, dynamic parallelism causes a performance gain over similar GPU algorithms, especially at large vertex counts. One major reason for this can be seen when looking at the algorithm's execution layout using software like NVIDIA NSight Systems that displays when the kernels are actually actively running on the GPU and when it is idling as it has to wait for syncs with the CPU.

In figure 8 we can see that dynamic parallelism removes any idle times between kernel launches that occur in the optimized GPU Shared kernel that is not using it (figure 9).

It can be observed that a major portion of the performance loss is caused by this handshake between CPU and GPU where data is moved and synchronized between them that is removed by the usage of dynamic parallelism.

We can also observe that the GPU overhead for all kernels becomes negligible very fast with increasing vertex counts as all GPU kernels quickly outperform the CPU implementations. Dynamic parallelism does not lead to a performance loss, as stated in earlier published papers, causing a performance increase in all test cases over the other GPU implementations. Additionally, there is no clear visible difference between having obstacles in the graph and not having them.

Graphs including obstacles run faster for all implementations, but this is mainly caused by the presence of smaller amounts of vertices. A clear difference between the GPU where larger amounts of vertices should not make as much of a difference and the CPU cannot be observed.

Even though running BFS on the GPU at large vertex counts is faster, it should be mentioned though that it also requires more memory as the graph data has to be copied over onto the GPU. Because the graph is stored in a sparse matrix, this additional memory required remains relatively small in comparison to the amount of data that has to be stored.

Another observation that can be made when looking at the GPU kernel runtime in more detail (See Appendix A and B) is that the creation of frontier buffers between kernel runs for the GPU Frontier and GPU Shared algorithms cause larger amounts of standard deviation between kernel runs. As the dynamic parallel kernel uses two buffers that are big enough for the maximum size of the frontier already, leading to them being allocated only once the standard deviation is massively reduced as the data is able to stay on the GPU in between

Fig. 8: NVIDIA NSight Systems Capture of the BFS Dynamic Parallelism Kernel

Fig. 9: NVIDIA NSight Systems Capture of the BFS Shared Kernel

kernel runs.
Looking at the data of the CPU kernels we can observe that both versions run at relatively the same speed for 8 x 8 and 32 x 32 vertex graphs. However, their performance slows down faster after these smaller sizes, possibly due to the usage of slower but larger L2 or L3 CPU cache.
Apart from that, all six algorithms show a relatively linear performance slow down as graphs get bigger at the same rate. However, the CPU implementations slow down faster than the GPU versions, starting at grid sizes of 32 x 32 vertices. To get a clearer picture on when exactly this occurs, future work could include measuring the data at even more grid sizes.

Similarly to more recent papers on dynamic parallelism, an overall performance increase can be seen. This highlights that there have been improvements to the technology over the years, either through updated versions of CUDA or because of more recent hardware.

VII. Future work

To dive deeper into the advantages of dynamic parallelism for BFS edge-based BFS could be examined in a similar fashion as this paper has done for vertex-based BFS. Along with this, similar inherently sequential algorithms like prefix sum could also be tested with the usage of dynamic parallelism along with having a look at the advantages of dynamic parallelism for algorithms that already run well in parallel without it.
As all results were measured on the same hardware and the same version of CUDA, it remains unclear why dynamic parallelism has become faster than it was in the past. A case study on this comparing different hardware and versions of CUDA would yield further knowledge on this.
Additionally, CPU implementations of BFS optimized for multithreading could prove to have their own use case and could be further explored than in small experiments as was done for this paper.
Further improvements for the CPU implementations such as code optimization compiler flags could also be improved in future work as there were some issues with getting them to work correctly in the CMake setup used in this paper's experiments.
Finally, the results could be compared with additional BFS implementations that include additional performance improvements such as running small frontiers like the first few levels of the graph in one kernel launch instead of multiple.

VIII. Conclusion

We discovered that dynamic parallelism is capable of increasing the performance of BFS by up to six times due to its ability to remove the handshake of CPU and GPU between kernel runs.
We also discovered that the usage of frontier buffers in GPU based kernels causes a large increase in standard deviation that should be remembered in projects that may require it to be low.
Furthermore, no big performance difference between graphs that include obstacles and those that do not was observable apart from runtime changes caused by varying vertex counts.

IX. Acknowledgment

The implementations of the algorithms compared in this paper were created in part with the support of Google Gemini 3. Any code generated by the AI was manually reviewed for its correctness and adjusted to work correctly before usage in the project.

REFERENCES

[1] Wen-mei W. Hwu, David B. Kirk, Izzat El Hajj, "Programming Massively Parallel Processors: A Hands-on Approach", Fourth Edition, Morgan Kaufmann, 2022, ISBN 978-0323912310

[2] Steven M. LaValle, "Planning Algorithms", Cambridge University Press, 2006. Available: https://lavalle.pl/planning/ [Accessed: February 2, 2026]

[3] E. Emerson, "Crowd pathfinding and steering using flow fields", Game AI Pro 360: Guide to Movement and Pathfinding, 2019

[4] NVIDIA, NVIDIA CUDA Toolkit 13.1. [Download Page] Available: https://developer.nvidia.com/cuda/toolkit [Accessed: February 2, 2026]

[5] Stephen Jones, "Introduction to dynamic parallelism", 2012. [Presentation]. Available: https://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0338-GTC2012-CUDA-Programming-Model.pdf [Accessed: February 2, 2026]

[6] poncle, Vampire Survivors, 2022. [Video Game]. Available: https://store.steampowered.com/app/1794680/Vampire_Survivors/

[7] Xulong Tang et al., "Controlled Kernel Launch for Dynamic Parallelism in GPUs", IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017

[8] Jim Wang, Sudhakar Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications", IEEE International Symposium on Workload Characterization (IISWC), 2014

[9] Liwen Xue, Shenglong Gu, Songdong Shao, "Enhancement of GPU-accelerated smoothed particle hydrodynamics (SPH) method with dynamic parallelism, Results in Engineering 27, 2025

[10] Dominik Tödling, Martin Winter, Markus Steinberger, "Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU", IEEE High Performance Extreme Computing Conference (HPEC), 2019

[11] Avi Bleiweiss, "GPU Accelerated Pathfinding", 2008

[12] Hao Wen, Wei Zhang, "Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs", IEEE High Performance Extreme Computing Conference (HPEC), 2019

[13] Deep Dive Dev, "Pathfinding Hordes of Enemies with Flow Fields", 2025. [Online Video]. Available: https://www.youtube.com/watch?v=tVGixG_N_Pg [Accessed: February 2, 2026]

[14] Deep Dive Dev, "Forcing My GPU to Compute Flow Fields Faster", 2025. [Online Video]. Available: https://www.youtube.com/watch?v=E89FvoFLav4 [Accessed: February 2, 2026]

[15] Leif Erkenbach, "Flow Field Pathfinding", 2013. [Blog Post]. Available: https://leifnode.com/2013/12/flow-field-pathfinding/ [Accessed: February 2, 2026]

[16] G. Zachmann, "Massively Parallel Algorithms Dynamic Parallelism", University of Bremen, 2024. [Presentation]. Available: https://cgvr.cs.uni-bremen.de/teaching/mpar/folien/06%20-%20Dynamic%20Parallelism.key.pdf [Accessed: February 2, 2026]

[17] Andy Adinets, "CUDA Dynamic Parallelism API and Principles", NVIDIA, 2014. [Blog Post]. Available: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/ [Accessed: February 2, 2026]

[18] Jordan Peck et al., FastNoiseLite v1.1.1, 2024. Available: https://github.com/Auburn/FastNoiseLite [Accessed: February 8, 2026]

[19] Graldy Tirta Kumala, Wirawan Istiono, "Comparison of Flow Field and A-Star Algorithm for Pathfinding in Tower Defense Game", International Journal of Multidisciplinary Research and Analysis Volume 05 Issue 09, 2022

[20] Jonathan Helsing, Alexander Bruce, "A Scalability and Performance Evaluation of Precomputed Flow Field Maps for Multi-Agent Pathfinding", 2022

APPENDIX A
FULL MEASUREMENT RESULTS: BFS W/O OBSTACLES

TABLE I: Performance Comparison (w/o obstacles)

Size	CPU	CPU Queue	GPU	GPU Frontier	GPU Shared	GPU DP
8 x 8	Mean: 0.319ms Min: 0.195ms Max: 0.622ms Med: 0.288ms SD: 0.084ms	Mean: 0.330ms Min: 0.181ms Max: 0.777ms Med: 0.243ms SD: 0.108ms	Mean: 1.074ms Min: 0.836ms Max: 1.890ms Med: 0.971ms SD: 0.231ms	Mean: 1.076ms Min: 0.779ms Max: 2.421ms Med: 1.269ms SD: 0.285ms	Mean: 1.232ms Min: 0.828ms Max: 2.076ms Med: 1.849ms SD: 0.302ms	Mean: 0.684ms Min: 0.457ms Max: 1.269ms Med: 0.973ms SD: 0.158ms
32 x 32	Mean: 0.456ms Min: 0.297ms Max: 0.899ms Med: 0.492ms SD: 0.121ms	Mean: 0.454ms Min: 0.363ms Max: 0.706ms Med: 0.441ms SD: 0.073ms	Mean: 3.925ms Min: 2.883ms Max: 5.950ms Med: 3.678ms SD: 0.716ms	Mean: 3.914ms Min: 2.870ms Max: 5.333ms Med: 4.691ms SD: 0.604ms	Mean: 4.036ms Min: 2.837ms Max: 6.103ms Med: 3.812ms SD: 0.679ms	Mean: 1.396ms Min: 1.061ms Max: 2.526ms Med: 1.499ms SD: 0.346ms
128 x 128	Mean: 10.121ms Min: 9.370ms Max: 12.658ms Med: 9.659ms SD: 0.654ms	Mean: 3.834ms Min: 3.441ms Max: 4.681ms Med: 3.861ms SD: 0.215ms	Mean: 15.216ms Min: 11.664ms Max: 25.495ms Med: 16.019ms SD: 2.717ms	Mean: 15.024ms Min: 11.673ms Max: 24.379ms Med: 13.978ms SD: 2.560ms	Mean: 14.537ms Min: 11.743ms Max: 21.854ms Med: 13.379ms SD: 1.969ms	Mean: 4.267ms Min: 3.616ms Max: 5.223ms Med: 4.241ms SD: 0.388ms
512 x 512	Mean: 564.741ms Min: 558.693ms Max: 585.028ms Med: 572.146ms SD: 4.970ms	Mean: 56.848ms Min: 53.268ms Max: 78.547ms Med: 55.326ms SD: 3.199ms	Mean: 52.725ms Min: 44.616ms Max: 80.787ms Med: 61.262ms SD: 7.124ms	Mean: 54.234ms Min: 45.552ms Max: 82.147ms Med: 62.987ms SD: 8.781ms	Mean: 52.371ms Min: 45.251ms Max: 84.541ms Med: 51.784ms SD: 7.162ms	Mean: 16.886ms Min: 15.443ms Max: 19.001ms Med: 16.626ms SD: 0.708ms
2048 x 2048	Mean: 37160.2ms Min: 36646ms Max: 38887.5ms Med: 37204.7ms SD: 345.15ms	Mean: 936.665ms Min: 917.288ms Max: 986.784ms Med: 927.151ms SD: 14.586ms	Mean: 329.463ms Min: 295.72ms Max: 404.592ms Med: 328.547ms SD: 26.669ms	Mean: 372.371ms Min: 184.323ms Max: 606.148ms Med: 425.977ms SD: 135.04ms	Mean: 366.207ms Min: 180.917ms Max: 611.074ms Med: 447.771ms SD: 135.49ms	Mean: 64.288ms Min: 62.543ms Max: 79.654ms Med: 63.927ms SD: 2.268ms

APPENDIX B
FULL MEASUREMENT RESULTS: BFS W/ OBSTACLES

TABLE II: Performance Comparison (w/ obstacles)

Size	CPU	CPU Queue	GPU	GPU Frontier	GPU Shared	GPU DP
8 x 8	Mean: 0.231ms Min: 0.161ms Max: 0.562ms Med: 0.196ms SD: 0.066ms	Mean: 0.224ms Min: 0.159ms Max: 0.433ms Med: 0.161ms SD: 0.059ms	Mean: 1.203ms Min: 0.759ms Max: 2.249ms Med: 1.595ms SD: 0.341ms	Mean: 1.079ms Min: 0.746ms Max: 2.190ms Med: 0.965ms SD: 0.303ms	Mean: 0.985ms Min: 0.733ms Max: 2.204ms Med: 0.836ms SD: 0.284ms	Mean: 0.575ms Min: 0.439ms Max: 1.217ms Med: 0.461ms SD: 0.135ms
32 x 32	Mean: 0.362ms Min: 0.281ms Max: 0.657ms Med: 0.348ms SD: 0.072ms	Mean: 0.438ms Min: 0.330ms Max: 0.665ms Med: 0.396ms SD: 0.068ms	Mean: 3.943ms Min: 2.961ms Max: 6.614ms Med: 3.615ms SD: 0.661ms	Mean: 3.863ms Min: 2.878ms Max: 5.030ms Med: 3.403ms SD: 0.466ms	Mean: 3.939ms Min: 2.861ms Max: 6.868ms Med: 3.826ms SD: 0.689ms	Mean: 1.299ms Min: 1.071ms Max: 1.889ms Med: 1.128ms SD: 0.212ms
128 x 128	Mean: 9.003ms Min: 8.159ms Max: 11.203ms Med: 9.221ms SD: 0.549ms	Mean: 3.443ms Min: 3.139ms Max: 5.242ms Med: 3.343ms SD: 0.265ms	Mean: 14.694ms Min: 11.342ms Max: 23.187ms Med: 13.855ms SD: 2.376ms	Mean: 14.128ms Min: 11.802ms Max: 23.576ms Med: 15.704ms SD: 1.830ms	Mean: 14.327ms Min: 11.774ms Max: 23.148ms Med: 14.434ms SD: 2.116ms	Mean: 4.189ms Min: 3.681ms Max: 5.566ms Med: 3.897ms SD: 0.355ms
512 x 512	Mean: 496.916ms Min: 478.296ms Max: 616.678ms Med: 494.34ms SD: 18.334ms	Mean: 47.025ms Min: 44.120ms Max: 51.349ms Med: 46.665ms SD: 1.768ms	Mean: 51.101ms Min: 44.350ms Max: 76.115ms Med: 63.599ms SD: 6.438ms	Mean: 49.390ms Min: 44.402ms Max: 67.717ms Med: 48.720ms SD: 4.501ms	Mean: 50.822ms Min: 44.542ms Max: 65.366ms Med: 50.771ms SD: 4.250ms	Mean: 17.062ms Min: 16.233ms Max: 19.041ms Med: 18.139ms SD: 0.509ms
2048 x 2048	Mean: 31664.7ms Min: 31033.9ms Max: 33271.3ms Med: 31551.7ms SD: 346.54ms	Mean: 787.565ms Min: 766.818ms Max: 830.685ms Med: 771.961ms SD: 13.561ms	Mean: 288.092ms Min: 276.145ms Max: 352.281ms Med: 288.468ms SD: 13.892ms	Mean: 311.443ms Min: 180.01ms Max: 537.4ms Med: 288.044ms SD: 100.014ms	Mean: 311.906ms Min: 187.004ms Max: 492.694ms Med: 349.861ms SD: 94.417ms	Mean: 65.202ms Min: 63.471ms Max: 70.585ms Med: 64.104ms SD: 1.968ms