

Computer Games Development

Project Report

Year IV

Robin Meyler

C00231699

04/05/2021

# GPU acceleration of A\* pathfinding with the Vulkan API

**Contents**

[Acknowledgements 3](#_heading=h.30j0zll)

[Project Abstract 4](#_heading=h.1fob9te)

[Project Introduction and Research question](#_heading=h.3znysh7) 7

Background 8

[Literature Review 10](#_heading=h.tyjcwt)

Project Description 12

[Evaluation and Discussion 15](#_heading=h.3dy6vkm)

Project Milestones 19

Major Technical Achievements 21

Project Review 22

Conclusions 23

References 24

Appendices 25

# Acknowledgements

I would like to thank the following people who assisted in completing this project including;

IT Carlow and all of the lectures that I have had over the last 4 years for providing me with the knowledge and foundation to be able to take on a project like this.

To my supervisor Philp Bourke for assisting me through the process of creating this project and providing guidance where possible.

# Project Abstract

**The Motivation:**

Since the late 1990’s the GPU has become an extremely important component of a video game experience and in many cases a more crucial part of the experience than the CPU as the term “GPU” was popularized by Nvidia in 1999. It uses many threads, which differ from the thread on a CPU, to execute many invocations of code at once in parallel. This parallel execution fits perfectly with the current most popular form of graphics rendering of triangles called polygons. Polygons consist of vertices and each of these vertices are operated at the same time using shader scripts. Shader scripts are programmable scripts that take information specified by the creator and apply transformations. A vertex shader operates on vertices and a fragment shader operates on pixels but there is another shader the GPU allows for called a compute shader. The compute shader allows the user to specify buffers with sets of information that can each be operated on independently and is regularly used in modern games as a means of calculating particles information or image pre/post processing. The information can then be directly sent to a vertex shader or accessed again via SSBO (Shader Storage buffer object).

Pathfinding is present in almost all modern games in one form or another and is very commonly used as a means of navigating AI agents around obstacles to a goal. Determining the fastest route to a goal in the shortest amount of time is extremely valuable to the developer as the number of AI agents increases, the execution time increases. A\* Pathfinding is considered one of the best and fastest pathfinding algorithms in modern programming as it uses a distance heuristic and a priority queue to ensure the best nodes are processed next. Since the GPU executes in parallel, if the correct information is provided to it theoretically could process the path for specific AI agents in tandem and the execution time would be the same as the slowest single execution for each warp and the overall execution matches the combination of the slowest in each warp. The main motivation of this project is to determine what speed of execution benefit would be achieved by this approach compared to general CPU unthreaded approach.

Vulkan is a modern graphics API released for public use in 2016 by the group Kronos, the creators of OpenGL. Vulkan was designed as a cross platform abstraction over the GPU that requires the programmer to explicitly declare every part of the rendering pipeline and memory allocation on modern multicore computers. The API is much more verbose than OpenGL but it unifies the graphics and compute functionality into a single API as it expects general purpose usage.

**The Aim:**

My goal for this project is to examine the speed of execution of the various amounts of agents pathfinding on various amounts nodes using A\* pathfinding on the GPU via a compute shader. I will be using the Vulkan API to synchronize the compute operation with general rendering, for controlling the exact memory allocation needed for each A\* execution and to map to, write to and read from GPU device local memory that is optimized to operate as fast as possible. Upon capturing the speed of execution of the GPU approach, I will attempt to determine if and when this approach might be viable as a pathfinding solution in a modern game.

**The Method:**

I began research into this area by looking into Vulkan with the intention of getting something drawing to the screen such as a triangle. I very quickly realised that the verbose and explicit nature of the API meant that a much stronger understanding of the Rendering pipeline was needed to even get going. In an attempt to broaden my knowledge on the topic I acquired the book “Real Time Rendering” and read the relevant chapters on the Graphics rendering pipeline and GPU architecture.

After studying the basics of the pipeline I returned to Vulkan with a much better understanding of the API. I began working on getting a triangle rendering to the screen. After a lot of online research, referencing the Spec and 1000 or so of specific code, I had a triangle rendering to a window. The process of getting to this stage taught me a great deal about the API and how each of its elements must interact to create a rendering environment. The process also crystallised my understanding of the Graphics pipeline I got from my textbook by explicitly declaring configurable parts of the pipeline and by fitting the jigsaw of parts together.

Once I had gotten to this point, I attempted to create a 3D object of a cube, I leaned back on previous knowledge I had learned in previous college years on vertex manipulation in 3D and indexing. Using this I was able to create Vertex Buffer Objects, VBO, cubes and mapping them to GPU VRAM during initialization. I then looked into way to ways to update the cubes during run time and found that creating new buffer on HOST-VISIBLE memory with the cubes new positional information as staging and then copying the buffer to the DEVICE-LOCAL memory declared at initialization was a performant method of vertex updating.

After I felt comfortable with the creation of many cubes and updating them during run time, I determined I had learned enough to demo the results of the project and began work researching the use of Compute shaders. Since there is very little resources on Compute shaders in Vulkan, I returned to the OpenGL implementations and began to research the basics of GLSL scripting. Vulkan uses bytecode format, SPIR-V, for its shading but the SDK provided by LunarG allows for compiling GLSL code to SPIR-V. After more research online I came across methods of creating Storage buffer objects, declaring appropriate memory and mapping it to the GPU. I could then execute the compute shader by dispatching it in a command buffer and then re-map to the memory after the Vulkan fence has declared the execution had finished.

Now that I knew how to create a compute pipeline and knew how to get memory to the GPU and back I began working on creating my compute shader. I started with a simple operation adding vectors and then by debugging the resulting information I could test whether it was successful, with some trial and error mixed with changes to the memory allocation I was able to get it to work. Now that I knew how to get more than 1 buffer to and from the GPU and felt comfortable with the basics of writing compute shaders in GLSL I began work on the A\* implementation.

Firstly I needed correct struct creation for the Node data for each invocation of the GPU, this struct needed to match the memory struct on the CPU side so it would match up. After some experimenting I found the memory was a bit off and after research this was a memory alignment issue that is common in GPU usage, I altered the CPU side by adding some padding to sync up each instance of memory allocation. A\* pathfinding requires the use of a priority queue, PQ, to order the nodes to save time. Since GLSL scripts have no access to the Standard library in C++, I created my own using an array of integers for the NodeIDs of the next node to be processed, I could then use this queue ID as the array index for the “top” of the queue. I created a “push” that did a check for the total accumulated value and searched the PQ till I found the correct position for the new node and placed it there. I create a function for “pop” and “queueSize”

Now that I had a way to organize the nodes, I wrote the rest of the shader using reference from my own project from previous years and adapting it to work in GLSL. After the pathfinding had reached its goal I broke from the loop and wrong the previous IDs from the goal back into the “Paths” buffer for the CPU side.

After the information had been processed and returned I went about integrating the compute pipeline with the graphics pipeline. They both use separate pipeline configurations explicitly declared, are executed with separate command buffers from different command pools and use different descriptor sets to organise the information. I organised the integration of both pipelines and insured the synchronization of both pipelines via fences.

Finally I created a grid and movable cubes for the demoing of the pathfinding results with updating of the cubes over time to show it working and I then recorded the times of executing of the memory mapping with dispatching of the shader as well as just the dispatching.

# Project Introduction and Research question

*“GPU acceleration of A\* pathfinding with the Vulkan API”*

**Question:** “What execution speed increase can be achieved from A\* pathfinding with GPU acceleration compared to traditional execution?”

This project objective is to use GPU accelerated execution of the A\* pathfinding algorithm to determine the performance benefits that may occur compared to a traditional non CPU threaded approach.

I am using the A\* Pathfinding algorithm because pathfinding is one of the most common problems to be solved with all modern games big and small and A\* is considered one of the best and fastest methods of finding the best path from one position to another in an obstacle rich environment. Using a distance heuristic to evaluate and set specific node information during execution that, with the help of a priority queue, it allows for fast resolving of paths.

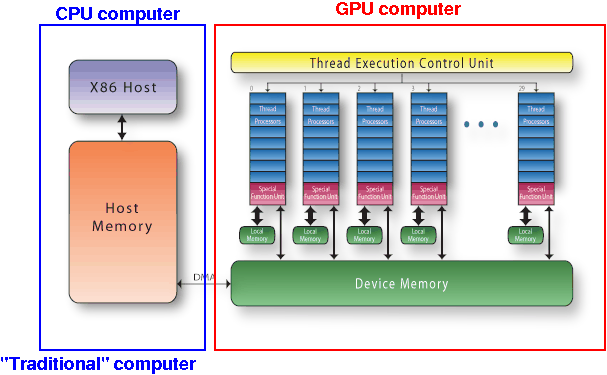
The reason I am using the GPU to accelerate the pathfinding is because the design of GPU architecture means that it executes many invocations at the same time in parallel. This will theoretically mean a drastic improvement in execution time and the difference, if any, is what I wish to measure.

I am using the Vulkan API to interface with the GPU as it is the most modern, verbose and explicit GPU that allows for specific memory allocation. Vulkan unifies the rendering and compute functionality to allow the developer to both render to a window as well as use compute execution in tandem by synchronising the GPU with fences. Vulkan’s general approach to rendering is to decide if you are using double or triple buffering and then create a swap chain with many images, it is then your job to synchronize each of these images so that while you are presenting one image to the screen you are writing to another image. Each stage of drawing and presenting happen at indeterminate speeds so semaphores and fences allow for minimizing time lost. This synchronizing fundamental allows for the use of compute shading operations that don’t interrupt the rendering.

The potential impact of this project would drastically increase pathfinding execution time in such a way that would leave CPU threads available for more dynamic processing such as networking that may change the nature of its execution more often and need a lot more custom creation. Pathfinding can simply be a brute force execution that happens as fast the CPU will allow it.

**Background**

**GPU:**

****

The GPU arose from the necessity of having to do a large number of texture lookups, vertex manipulation and testing Z-depth very frequently. This led to dedicated hardware for the operations which over time grew into the modern GPU architecture we see today. A GPU generally runs at a slower clock speed that a CPU since a CPU has many methods of operating quickly such as caching. The benefit is speed at calculating and it’s parallel design as it contains dedicated silicon to allow it to access textures and buffers quickly.

The GPU is a stream processor meaning it processes ordered sets of similar data in turn. It contains thousands of shader cores that can compute independently of one another. Each specific execution is called an invocation and in terms of the compute shader has a specific ID that can be query in the shader script. Each shader invocation runs on a thread, which is not like a CPU thread. In modern GPUs, threads are executed in groups of 8-64 based on the architecture. These are called Warps on Nvidia and Wavefronts on AMD. Nvidia warps generally contain groups of 32 threads per warp, that means if there are 2000 threads to be computed, it will be split into 2000/32 = 62.5 warps which will round up to 63 warps with 62 full with 32 threads and 1 warp with 16 threads. Each of the threads in these warps execute in lockstep. If the warp comes to a point in the program that requires a slow call such as a texture look up, the warp begins the lookup and swaps out the warp being executed for another warp and begins computing that warp from the last point it left off. This method of latency reduction employed by GPU is to make up for a slower clock speed.

One concern to be considered with threads is that they have a small amount of memory register space to work with, the more memory needed per invocation, the less threads are available for computation, the fewer the threads the fewer the warps. Warps available are called “in flight” and the number of these is called the “occupancy”. High occupancy means there are warps idle and ready but low occupancy leads to poor performance. Another fact valuable to the project is “thread divergence”. Thread divergence is when “if” statements and loops appear in shaders, if one of the threads in a warp has a different value to the rest at this point each thread in the warp has to execute both outcomes and discard the unwanted one, this can have an affect on general performance of a shader, but in this projects case it’s a trade worth taking.

**A\* Algorithm:**

*Pseudo code:*

**Q = Queue**

**T = top of queue**

**C = Child**

**G = Goal**

While(Q not empty and T != G)

For (each C node of T)

If(C is not passable)

Continue

**CV** = C’s cost from start in distance + T’s cost from start + T->C’s Arc weight

If(CV < C’s total accumulative cost)

C’s total accumulative cost = **CV**

C’s total cost from start = T total cost from start + T->C’s Arc weight

C’s previous ID = T’s ID

If(C is not marked)

Q push C

C marked

Pop T from Q

A\* was created as part of a project called “The Shakey project” in 1968, this project had the aim of having a mobile robot plan it’s actions and traverse its surroundings. A\* uses a distance heuristic and a priority queue to minimize the nodes traversed by using the distance value mixed with distance from start to order the nodes, it follows the best path until it meets the goal or a dead end. Upon meeting a dead end it then retreats through it’s queue to the next best path to follow.

A\* has become one of, if not the best, algorithm used for pathing of agents or AI in video game development. The execution of A\* can be unpredictable and once many agents are looking to path find at the same time to a goal/goals then the CPU may not be able to keep up with the framerate of the program causing stutters which is poor for performance of the product. This proposed project looks to alleviate this issue by outsourcing the many computations of many agents to the GPU where the path values can be determined in parallel.

# Literature Review

The use of the GPU for general purpose programming came into experimental use about a decade ago as GPU architecture quality began to hit a threshold point that made the trade offs worthwhile. Since then there haven’t been many papers on the topic of using a one of a few APIs designed for a developer to interact with the GPU in a non-rendering aspect. CUDA is a common API created by NVIDIA that is used to compute general purpose calculations in a massively parallel way. NVIDIA are the manufactures of a lot of the worlds GPU’s and can optimize their drivers to work with the API to give it a boost in performance, CUDA does not work on AMD cards for business reasons which makes it much less usable in a commercial game setting as the consumer of a developers game could have any GPU hardware.

Another issue with this use of CUDA is that it is a separate API to the API required for rendering. So if a developer wishes to use the GPU’s computational power for something such as pathfinding, they would have to provide the information, retrieve the results and they map the resulting information into a rendering API such as OpenGL, this can reduce execution speeds as compatibility issues and alignment can cause issues. This downside can mean that while the GPU can compute information such as pathfinding for the developer the trade-off of getting it to work with another API makes it not worthwhile.

Vulkan offers a potential solution to this issue, Vulkan unifies the Rendering and Compute functionality under one API, that allows the developer to map memory in a completely compatible way. Vulkan allows the user to create separate pipelines for the compute operations from the Rendering pipeline, it allows you created command buffers for each and also has inbuilt functionality to set flags on every pipeline to wait for specific flags from the GPU, a semaphore can be assigned to the rendering pipeline to tell it to wait for all compute operations to complete before trying to render an image. The value of this is that almost no time is lost between the actions as they are synchronized and that for image processing or other dependent compute operations, the input for the vertex shader could be mapped to the output of a computer shader.

***Other papers on the area:***

I have come across three different but similar approaches to Pathfinding on the GPU to showcase the use of its massively parallel design to execute intensive algorithms.

**“GPU Accelerated Pathfinding”** (Bleiweiss, 2008)

* This was built in house by the creators of CUDA and the manufacturer of graphics cards where the creator has intimate experience and knowledge into the architecture, it example of the power and capability of the GPU for non-rendering tasks
* The paper also uses A\* as it needs to process the least amount of nodes compared to many other pathfinding algorithms, which is important for keeping loop count lower. While the GPU is great for parallel computation it actually operates at a lower clock speed than a CPU so it’s not suited well to very long loops.
* The A\* employed here differs from my approach, while it applies a heuristic it keeps track of the node cost in a different manner.
* The project uses CUDA for its computation and doesn’t output the results to any visual representation. This differs from my project as I aim to use the Vulkan API to use compute shaders instead of direct access to the GPU and also to have the compute work alongside the rendering pipeline with synchronization.

**“GPU Accelerated Multi-agent Path Planning based on Grid Space Decomposition”** (Caggianese, 2012)

* This paper is very similar to the paper above, it uses CUDA API to interface with the GPU using a single thread for each pathfinding.
* For the A\* implementation it uses built in CUDA functionality to act as the priority queue which is not an approach that is available to Compute shaders.
* The resulting data from the test was evaluated on small map sizes with low number of agents but it was also tested 9 years ago when GPU architecture was less advanced.
* This study also does not use any visual representation to show the results

**“Dynamic Search on the GPU”** (Anon, Unknown)

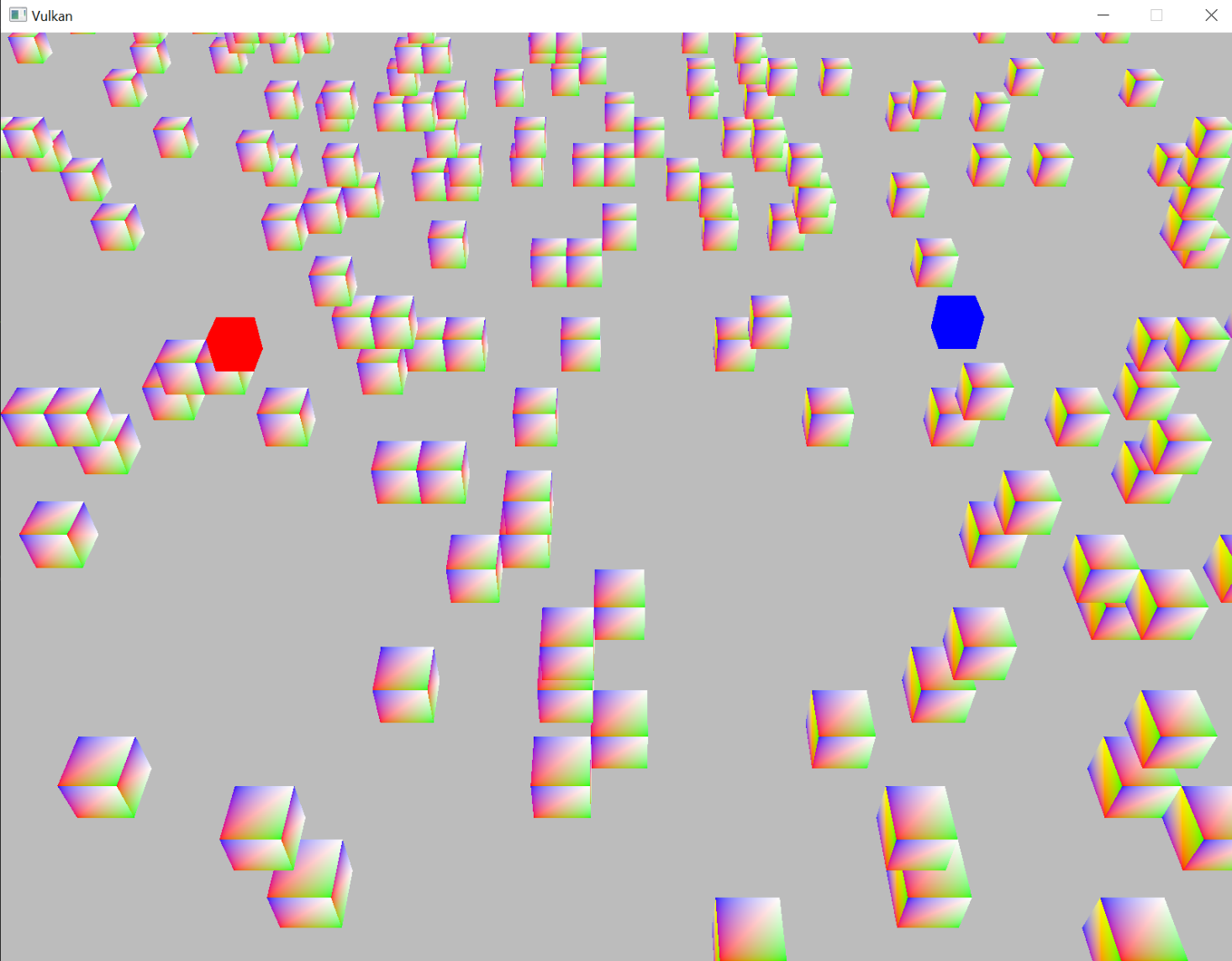
This paper is different from the other 2 as it attempts to use the GPU for Dynamic searching but does share similarities to my proposed approach. It uses the GPU’s parallel design to execute intensive computations that would normally have to be managed.

* This paper also uses CUDA as its way of interfacing with the GPU but is undated so it’s hard to determine what other options were available to the developer at this time.
* This paper seems to have a visualisation of parts of its searching which indicated possible usable rendering.
* Contains a lot of GPU specific information and data charts which do a good job of showing memory to input size.

**Project Description**

Before starting the program, the code must be adjusted to decide how many nodes and how many agents are wanted for the demonstration. The “Astar.comp” compute shader must be updated to expect the same amount of nodes for the memory to match on both the CPU and GPU. Once the shader is updated, you will need to compile GLSL shader code into SPIR-V using the provided compiler.

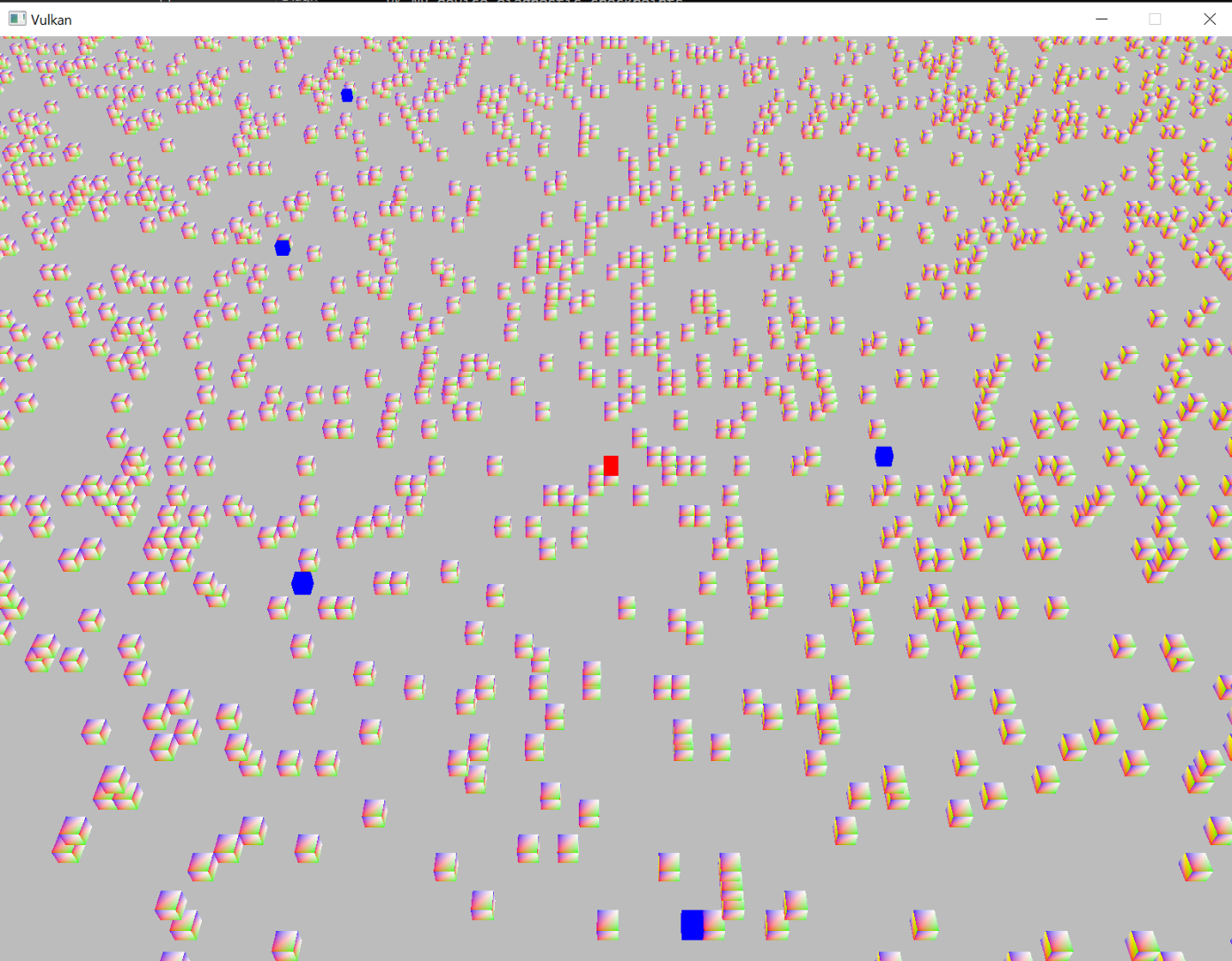
Once the user decides on the amount of nodes and agents the program will attempt to create the requirements and allocate specific GPU memory to accommodate the nodes in a storage buffer, at the same time the program will create Vertex buffer objects of cubes for the outer walls of the grid. The program will create a cube for agents of the number selected and then randomly place them on a passable node on the map. The program will create a Goal cube at the selected goal position.



*Up close view of a grid with obstacle cubes, the goal cube and one of the many agent cubes.*

Once all of the storage and buffer object objects are allocated on HOST\_VISIBLE memory as staging memory and copied to DEVICE\_LOCAL memory on the GPU, the program runs and the user is shown an above view of the grid of the desired size. The rendering pipeline renders to images and swaps them for presenting for triple buffering using synchronization fences to ensure the GPU is finished with a draw and that it is finished presenting the last image before we try to draw the next image. Via the nature of Vulkan, it is possible to start drawing to an unused image while the last image is still presenting, this can provide performance gains.

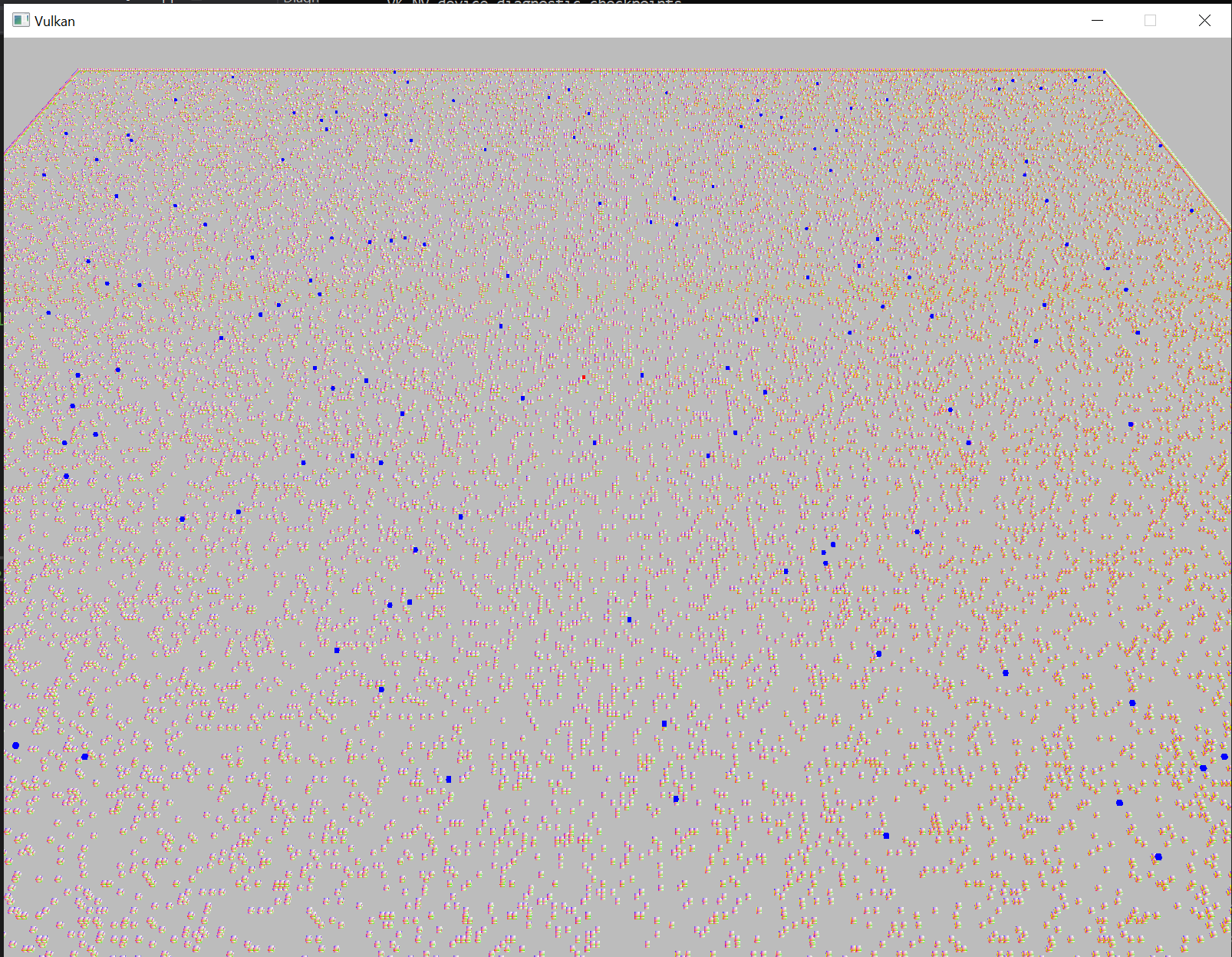
The user has the ability to move the camera forward, back, left and right with W,S,A,D. In addition to this the user has the ability to zoom in to the grid and back out with Q,E. Finally the user has the ability to change the perspective to be at an angled view seen in the example images with F,C. The user has the ability to scan over the whole grid of their chosen size and observe any specific agent moving along their path to ensure they are avoiding obstacles or zoom out and watch all agents move along their path together toward the goal.



*Somewhat far out image to show the scale of the grid and angled to demonstrate the 3D nature of the elements*

The user then can press the space key to activate the pathfinding, where all the agents that are drawn to the grid will have their paths computed on the GPU with the compute shader by submitting the compute command buffer specified at initialization. This shader begins a series of calls which prepares the dispatch using the Compute pipeline created in initialization. This pipeline is specified to avoid the rendering aspects of the GPU by avoiding vertex shader, rasterization and each unneeded stage that is required to bring an image to fruition. This allows faster and direct connection to the GPU.

The speed of the dispatch is measured by using the Standard Library’s Chrono time management include, starting a precise clock just before dispatch and using a fence that waits for the GPU to be idle. Once the GPU returns it is idle, the time is taken and compared. At this point, the blocks will begin to move at a slow speed from block position to block position along their path resolved by the compute shader. The speed of the moving blocks can then be increased and decreased with UP and DOWN keys. Once the blocks have reached their final location, the user can press R to return the positions to their starting places and pressing Space will begin the Dispatch again.



*200 AI agents pathfinding on 250,000 grid of nodes zoomed out*

# Evaluation and Discussion

***Define the testbed:***

***Specs:***

All the tests were conducted on my personal computer with specs:

* Nvidia RTX 3070 graphics card
* Intel i7 CPU
* 16GB of RAM (Duel 8GB)
* 1TB SSD

***Conditions:***

The specific conditions were such that no other taxing software was running on my PC other than Visual Studio itself in an attempt to maximize GPU availability. The impact of Windows 10 itself is hard to measure.

***Execution:***

The Compute shader dispatches are called in isolation such as, the CPU waits for a fence (flag) to indicate that the GPU has finished presenting the last frame. Then, before rendering the next frame, it begins a timer using Chrono in the STD library and dispatches the A\* shaders, it waits for the GPU to become idle. Once idle it records the time taken, maps the results to CPU memory and continues drawing the next frame. The resulting time is written out to a file. The test will use A\* pathfinding on a square grid of up to 4 neighbours per node. The grid also has an obstacle incidence rate of 10%. Priority queue of size 5 (ints) was used.

***To Test:***

Testing will be conducted to establish the time in microseconds it takes for the GPU to calculate the exact path of all the agents given using the grid of nodes it is given. The size of the Grid and the number of Pathfinding agents will be varied to study the difference of each. The resulting times will be represented in seconds for ease of understanding and considered the speed of execution. 4 Grid sizes will be measured (50x50, 100x100, 250x250, 500x500), the upper limit of which was decided by failure to allocate buffer memory on the GPU for higher node counts for any reasonable testing. The number of agents will also be varied (10, 50, 100, 250, 500, 1000) to understand the impact of large amounts of calculations of the GPU.

***To Establish:***

The first thing the testing wishes to establish is how fast the GPU can execute the Pathfinding at a certain agent count relative to other agent counts, for example, does 100 agents take twice as long as 50 agents to execute on a set grid size. This is to determine how much of an effect the parallel nature of the GPU has on the computation. Another thing the testing wishes to establish is the difference between grid sizes relative to one another, for example, how much longer does 250x250 grid (62,500 nodes) take compared to a 100x100 grid (10,000 nodes).

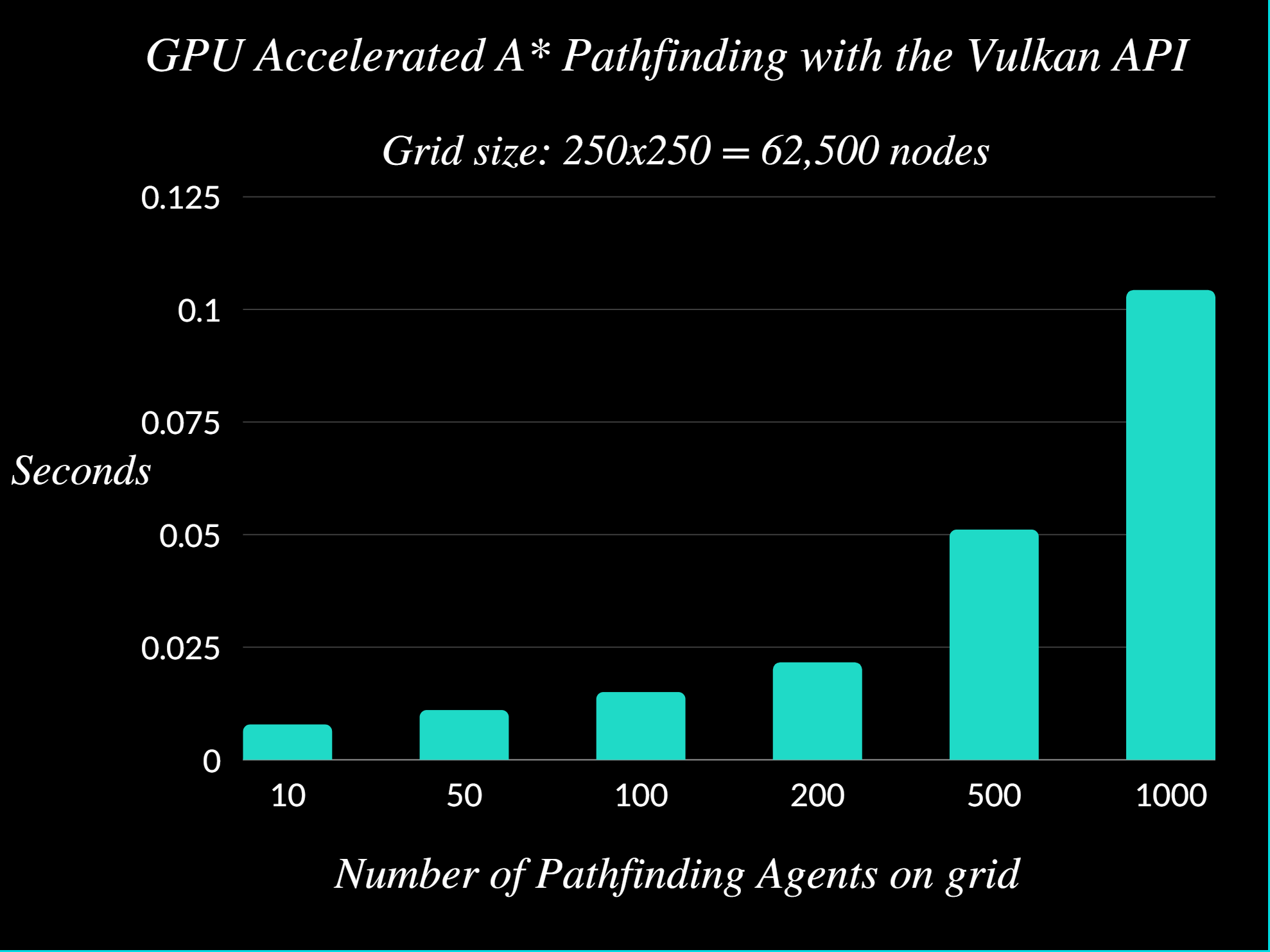
***The Results:***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Grid size** | **Speed of execution represented in seconds** | | | | | | |
|  | 10 agents | 50 agents | 100 agents | 200 agents | 500 agents | 1000 agents |
| 50 x 50 | 0.0023s | 0.0029s | 0.0037s | 0.0060s | 0.0131s | 0.0255s |
| 100 x 100 | 0.0055s | 0.0075s | 0.0086s | 0.0112s | 0.0220s | 0.0443s |
| 250 x 250 | 0.0077s | 0.0109s | 0.0149s | 0.0214s | 0.0509s | 0.1040s |
| 500 x 500 | 0.0168s | 0.0247s | 0.0351s | 0.0516s | N/A | N/A |

***Conclusions:***

One overall note is that every set of agents and grid size executed in less than 0.1 seconds bar 1000 pathfinding agents on 250x250 nodes, which immediately speaks to the viability of GPU accelerated A\* pathfinding. The cells with “N/A” were sets that could not be tested due to GPU memory allocation issues of the setup. Some conclusions from the data:

1. The difference between 10 and 50 agents on all grid sizes is only marginal, considering 5 times more executions were required for this calculation but the resulting time was very far from 5 times slower, we can conclude that the GPU was successfully calculating the paths for many of these agents in parallel, reducing compute time.
2. Conversely the difference between 500 agents and 1000 agents is roughly double the time taken. While this initially suggests that they aren’t being executed in parallel, it’s important to note that the GPU executes in Warps(groups) of 32 thread invocations, this means while the increase was equal to the agent increase, they are both decreased on a base level since they are computing 32 at once.
3. The best results seems to be in the 200 agents range on all grid sizes, the jump in time from this result is very far from double 100 agents but after this point the increase in time seems to be directly proportional to agent increase.
4. One thing to note is that the 250x250 grid has 62,500 nodes while the 500x500 grid has 250,000 nodes, this increase is roughly 4 times higher yet the increase at every agent level is just over double the time taken. This could be related to the grid’s square layout.
5. The size of the priority queue inside the A\* Compute shader is the greatest limiting factor for executions size, a doubling of this priority queue increases the speed of execution exponentially. Greater obstacle incident rate requires a larger queue size but at a 10% incidence rate used for these tests, all agents reached their target.

*Chart displaying the speed of GPU dispatch at 62,500 nodes on 250x250 grid*

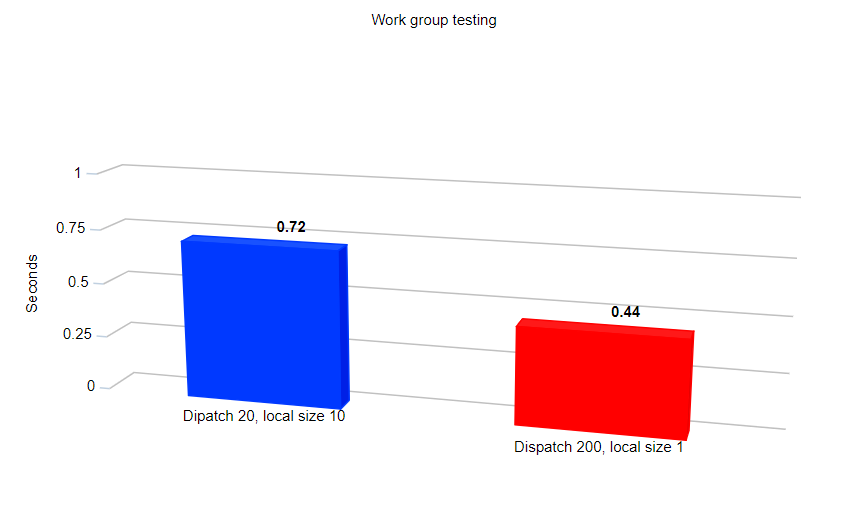
***Specific Case Study:***

In this one case, the test used a 250x250 grid consisting of 62,500 nodes. The first most striking thing to note is that the increase in computation time starts off increasing at a small curved rate until it hits somewhere between 200 and 500 agents. At this point, the increase in execution time becomes linear with the number of agents pathfinding to the goal node.

* The jump from 10 agents to 50 is very marginal though the amount of paths to be resolved increased 5 fold. This suggests that there is a base level of time that the process itself takes and that is encompassed by both agent numbers. Considering a warp size of 32 threads, 10 agents could be done at once while 50 agents could be done in 2 warps, which essentially increases the execution to 2 runs.
* 50 agents to 100 and to 200 all increase along the same pattern, which is the same as observed from all grid sizes as per the chart on the previous page. Suggesting the 200 – 300 agent range is the sweet spot for optimal performance and time saved during execution.
* Finally the jump from 500 to 100 is roughly double the time taken, 500 agents could fit into 16 warps, that is at worst the execution time of 16 pathfinding runs while 1000 agents would require 32 warps. So while it is executing in parallel, since no time consuming calls are being made, they execute from start to finish and are unable to save time by swapping warps. Since 16 warps to 32 warps is double increase, it makes sense that time would be double, overall, this is still saving significant time.

***Extra test:***

*Result from a small test on where separating the dispatch into smaller work groups*



One other small element I tested was the difference in execution times taken from dispatches that used different configurations. As opposed to dispatching 200 agents in 1 work group, I dispatched 20 workgroups of Local size of 20, this local size is determined in the compute shader with *“ layout (local\_size\_x = 10)”* with this you can set the local size of each workgroup for all 3 dimensions. Unfortunately at many different configurations as well as the one shown above, the execution time decreases by a noticeable margin.

***Overall Conclusion:***

Overall, the results of these tests show that significant time can be saved with GPU accelerated A\* pathfinding. The parallel architecture allows for warps of 32 thread innovations to use the compute shader to pathfinding together. While the relative speed difference between say 500 and 1000 is linear, its saves in execution speed at a base level. The best spot for numbers of agents pathfinding is roughly 200-300 to receive the best speed results relative to higher and lower numbers of agents but the speed gained at all counts are noteworthy. The increase in grid size to time taken for execution also increases linearly even though the number of nodes that can be calculated increases exponentially.

The size of the priority queue was the strongest limiting factor in execution time, all results used a queue size of 5 and successfully reached the target with a 10% obstacle incidence rate without fail but a priority size of about 10 was required for obstacle rates of 20% to 50% which does increase execution time. All of these conclusions leads to the final claim that GPU accelerated A\* in Vulkan is a viable option for a project where the creator has some reliability over the map/grid they are creating, in such cases the time saved in execution is very significant. In other situations, the specifics of parts of the layout might make the usage not worthwhile.

**Project Milestones**

**October 2020:**

I began work on my original idea for the project. I knew I wanted to use Vulkan to explore and learn about explicit rendering and use some of Vulkan’s strengths over its competitors in the graphics API field. My original idea was to use the transferring memory operation to load content to a small 3D environment while rendering it at the same time because I had read that they don’t overlap and it was possible but I quickly realised that the scope of that project idea was much too large for me at my current knowledge level. I began researching and coding in Vulkan as I knew it was the technology I wanted to do my project with. I then came to the idea that I could attempt to possibly use the GPU to do some CPU intensive operation. My initial thought was to use the Min-Max algorithm for chess AI.

**November 2020:**

I continued working on trying to get a triangle drawing in Vulkan, realising it took more understanding that I expected and that it was much more complicated than OpenGL, I went back to basics and spent a lot of my time reading about the rendering pipeline and GPU architecture from the textbook “Real Time Rendering”. Using this and online research I made progress with the program, the Compute shading part of the project was left aside because I needed to see if I could get a working program before I could proceed.

**December 2020:**

I finally after lots of trial and error mixed with a lot of reading on forums, using the Vulkan Spec, reading example code from Sasha Willems on Github and consuming any resource on the API I could find I got a triangle rendering to the screen. It took about 1000 lines of Vulkan specific code but from what I had read it was the hardest and most verbose part of the API as Vulkan is considered “Ahead of time” API where every possible configuration is declared at compile time and the correct configuration is used when needed as oppose to OpenGL which is considered a “Just in time” API. With the boilerplate code working I presented the idea to the supervisors with the intention of the Min-Max algorithm being used.

**January 2021:**

It was at this point I began research on Compute shading in Vulkan and decided on the scripting language GLSL as it was the standard for the API’s precursor of OpenGL and the Vulkan SDK had a compiler to turn the GLSL into SPIR-V byte code used for execution making it fully cross platform. Upon research and thought I determined that the Min-Max algorithm would not suit the idea I was hoping to achieve with the project. It is not likely to be run enough times to have the time benefits be worth the implementation. I also could not envisage a way to fragment the algorithm up without inter invocation communication on the GPU being needed which was not ideal and in some cases not possible on the GPU. It was at this point I decided that A\* would suit the project as it was the type of that could have hundreds of agents executing on a very large map. While researching and deciding to take this approach I implemented cube objects into Vulkan and had to alter different parts of the code to accommodate this change. I also added in methods for creating Vertex buffer objects and Uniform buffer objects which are essentially global information for all the shaders that are updated frequently and work very well with a camera so I implemented the MVP model.

**February 2021:**

Now that the project had enough base code to allow me to demonstrate a demo and synchronize the pipelines I began work on compute shaders. I began by creating a very simple project to add vectors to explore the creation of a Compute pipeline and everything needed to get such a program to work. After research on line and looking at the few other examples of compute shading I could find I managed to get the shader to pass up SSBO to the GPU, execute the shader on each instance and I learned how to remap the memory after the dispatch and examine if the memory was as I expected it.

**March 2021:**

Now that I had a working compute pipeline I worked on integrating the project with the rendering pipeline that I already had working in the projects, since Vulkan is an ahead of time API, both pipelines and everything needed for each pipelines such as different buffers, different descriptors sets, different pipeline layouts and different command buffers. While working on getting these two pipelines to work together I began refreshing myself with the general concept of A\* by reviewing a project I had done last year.

**April 2021:**

I began work on the A\* compute shader GLSL script modelling it off of my previous implementation. I wrote the basic logic for the calculating of the heuristic. I then created my own priority queue inside the shader using as little memory as I could and as few loops as possible to not affect performance. I then organised the storage buffers and the Structs needed in GLSL and on the CPU side so they matched up with memory alignment. I abstracted the rendering code away from the main loop in the project to make it clearer which code was game logic and what was simply rendering operations/setup. I then integrated the compute pipeline fully from a separate C++ file I was working with so that both pipelines could be dispatched during draw calls. After bug fixing and trial and error I got the A\* working with the project. I got a demo working with a simple grid of cubes with obstacle cubes. I then had the cubes updating with the returned paths over time to properly demo the pathfinding. I then began to vary the grid/node size and the number of agents pathfinding from 1 compute A\* dispatch. Once I observed large numbers working on large node counts, I used the standard library chrono library to measure the time of execution of the mapping the memory to the GPU and the time taken to dispatch the A\* and return path information back. At this point I just continued trying to optimize the program.

**Major Technical Achievements**

The main objective of my project was to execute an algorithm that would usually be a CPU intensive operation on the GPU using its architecture to reduce execution time using the explicit nature of memory allocation of the Vulkan API and it’s coexistence of the compute and rendering functionality. I believe I have achieved this tasked as I have managed to mix compute and rendering, I have managed to use the compute shader to execute A\*, I have use the GPU to execute 100+ of agents of A\* on 100,000+ nodes in parallel, I have been able retrieved path information back from the compute shader and I have managed to create a 3D environment to showcase the A\* working in real time while rendering. I have also gathered information to evaluate and draw conclusions on my hypothesis.

More specific examples of technical achievements I have reached throughout the project would be:

* I have learned a lot of knowledge on the use of GLSL shading language that is used to program parts of the Rendering pipeline. Not only have I gained knowledge on how to use and interact with Compute shaders through inputs, outputs, structs, device memory and general GLSL syntax. Through my reading I have also become a lot more comfortable with Vertex shaders and Fragments shaders.
* I have made strong headway into understanding the nuances of the Vulkan API and am more than a component enough to begin making programs in the API going forward as I hope to do in the near future. The explicit nature of this API means that all the skills I have learned from this I can take into learning an API like OpenGL or DirectX as they are both more controlled versions of the same principles.
* I have learned a lot more information about the Rendering pipeline, compute pipeline the GPU architecture in general through online research, textbook reading and in creating the project itself. This will help me with any other project or engine work in the future as even if I am using Unity or Unreal, I am still using the same GPU for the similar processes and this will help me optimize programs at every level I work on them.
* I have learned a lot more about memory management at a very low level as both Vulkan and the nature of the project itself has forced me to consider the size of the exact information I wish to allocate and the stride and the fragmentation of many allocations. Through working with very large amounts of nodes, I have found that in many cases with this project memory and memory allocation became a limiting factor with the GPU.
* I have become more comfortable with the A\* algorithm as having to port it into such a limited script space forced me to think more about priority queues.
* I learned about Instanced rendering technique where you render many versions of an object in 1 draw call by setting different model information for each and creating a separate pipeline in Vulkan to expect it.

**Project Review**

In this section I plan to reevaluate the project as a whole and discuss areas that I am happy with how they turned out and other areas that could have gone better. I will also state things I may change if I was to start over again.

Firstly, one element of the project that I thought went well was the usage of Vulkan, while it was a very steep learning curve, I was able to get some handle on all the working parts of the machine and through a lot of reading and a lot of trial and error I managed to get the program rending to a window using double buffering. I managed to get in the ballpark of 100,000 cubes drawing to the screen in a grid, I managed to get vertex shader and fragment shaders working in a pipeline I could work with and change to help me demo the project.

Another area I felt went quite well was the GLSL shading itself, I had very little ability to working with GLSL before beginning the project but after scouring the specs and through usage of the compute shader I was able to get the shader up and running without any major roadblocks that held me up for any significant period of time. Writing shaders is a skill that will be good for me in any game developing endeavour I may have in the future so I’m very happy with the results here.

One area I was drawing bulk cubes, in my current implementation the program creates the desired amount of cube VBO objects and prepares them for rendering but when the number gets up to the region of 100,000+ the amount of allocating needs is too much. The solution to this issue at face value would be “instancing” this is a method of batching all renders of a similar VBO into one very large buffer, this will avoid fragmentation of the memory, speed up loading time at the start of my program and allow me to have a lot more cubes for demoing the project. I experimented with instancing and I’m confident it can be done but since it didn’t actually affect the core purpose of my research it took a back seat.

Another thing I thought went quite well was the implementation of A\* in a compute shader, while I reckoned it would be possible I was unsure how it would perform since using loop sin shaders isn’t as common but after testing on relatively large numbers I concluded that as long as I kept the priority queue low enough that it didn’t have massive arrays to search through countless times and I made sure it was large enough that a needed node isn’t lost that the performance is surprisingly good.

One part about my Compute shader I dislike less is that each invocation of the A\* agent running requires node data for each and every node, so if there is a large amount of nodes, it requires some doubling up on code. I spent time thinking about solutions to this such as making the nodes as a uniform buffer which would make them available to all invocations but since each node’s data needs to be changed for the total accumulated cost, it would be overwriting the data. When I get to large amounts of nodes such as 1 million nodes, this can cause issues with memory allocation as there is only so much memory on the GPU to be used, this is an issue of optimization that I believe could be fixed with more time spent on it.

One note I would say on the concept of doing this project over again is that I do believe the action of having A\* on the GPU would be easier done with OpenGL and would have saved me a lot of time that I spent learning about Vulkan with sparse resources available. On the contrary to this I’m unsure how easy it is to use compute shading with rendering as the API is less geared toward general purpose GPU processing. I also must add that using Vulkan was a personal preference and drive for me so while if I was to start the project over again I would still use Vulkan but if I was suggesting it to another student who didn’t care about the rendering detail, I would suggest maybe they should use OpenGL because it will be a lot faster for them to interface with the GPU.

If I had more time to work on this project I would spend it optimizing memory allocation, buffer allocation and storage buffer size. I would spend more time trying to think of more intricate ways to pass the node data without it needing such large amounts of nodes per agent, with this optimization I could get larger numbers of agents pathfinding on larger node counts.

# Conclusions

Overall I’m very happy with the results of my project, I hoped to see a massive increase in speed of execution over normal uses of A\* pathfinding in bulk with very fast times and based on the evidence displayed in the evaluation and discussion section above, the results suggest given the specific hardware used that the performance benefit is very noticeable and I would go as far to say it would be a viable option for pathfinding in an actual game project.

At the highest level this project aimed to prove that the GPU can be used to execute taxing computations off of the CPU and execute them in parallel to improve overall performance of a program. Based on the results from this I can imagine on paper how a very similar approach could be used to investigate using the GPU to execute collision detection between an extremely high number of objects in a scene, given Vulkan’s ahead of time design, the memory of many object positions, size and information could be mapped to storage buffers for compute shaders and updated dynamically then checked in parallel on the GPU.

Another conclusion I can take from this is the explicit design of Vulkan will make it a powerhouse in the coming years. From my personal research it seems that developers have only scratched the surface of the API and what it can do, I can envisage in the coming years brand new rendering/computing practices coming to fruition once talented developers spend time experimenting about ways they can optimise their rendering pipeline.

# References

|  |  |  |
| --- | --- | --- |
| **Referenced Publication** | **Citation** | **Reference** |
| Website | (Bleiweiss 2008) | Bleiweiss A. (2008), GPU Accelerated Pathfinding, NVIDIA Corporation  <https://folk.idi.ntnu.no/elster/tdt24/tdt24-f12/presentations/lars-espen-GPUAcceleratedPathfinding.pdf> (Date Accessed: 12 February 2021) |
| Website | (Caggianese, 2012) | Caggianese G. (2012) GPU Accelerated Multi-agent Path Planning based on Grid Space Decomposition, SciVerse  <https://core.ac.uk/download/pdf/82615384.pdf> (Date Accessed: 3 March 2021) |
| Website | (Anon, Unknown) | Anon . A (n.d) Dynamic Search on the GPU, PeopleCS  <https://people.cs.umass.edu/~fmgarcia/Papers/Dynamic%20Search%20on%20the%20GPU.pdf> (Date Accessed: 12 April 2021) |
| Book | (Akenine-Moller, 2018) | Akeine-Moller A. Haines E. Hoffman N. Pesce A. Iwanicki M. Hillaire S. (2018), Fourth Edition, *Real Time Rendering.*  CRC Press |
|  |  |  |

# Appendices