Bachelor of Science in Computer Science
May 2019

Asynchronous Particle Calculations on Secondary GPU for Real Time Applications

Tobias Pogén

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Tobias Pogén
E-mail: Topo@student.bth.se

E-mail:

University advisor:
Stefan Petersson & Hans Tap
Department of DIDA

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Particles have been a part of modern games for a long time and have always been a computationally heavy process. There is a lot of different ways of approaching the problem of particle calculations, but in the latest version of DirectX 12, there is support for multi GPU rendering. This would allow the secondary GPU that exist in may Intel GPUs to be utilized for particle calculations and hence moving over the processing work to a secondary often lesser GPU.
Objectives. The objective of this thesis is to see if there are any performance gains of running the particle calculations on a secondary GPU and later rendering them on the primary GPU. This compared to calculating and rendering on the same GPU.
Methods. The particles will be executed in a complete engine to give a representative result as this is how it is intended to be used. This means that other then particles there will be multiple other rendering passes taking place on the GPU. Different performance metrics will be measured on the same computer and compared.
Results. The raw data show a great performance increase in calculating the particles on a secondary GPU which allowed for more particles compared to only running a single GPU. It also gives a more stable and efficient execution to all of the affected rendering techniques.
Conclusions. There is a significant amount of performance that could be gained from utilizing a secondary GPU, but it still a complicated system to implement and can be hard to maintain and test. And to answer the research questions, there is a performance gain and most of the performance gain comes from the particle execution time.

Keywords: Multi-GPU, Particle, Rendering

Contents

Abstract **i**

1 Introduction **1**
1.1 The modern problem 2
1.2 Particle System 2
1.3 Why use a secondary GPU 2
1.4 Research Questions 3
1.4.1 What is the Problem? 3
1.4.2 Research Questions 4
1.5 Theory 4
1.5.1 Execution Time 4
1.5.2 Data Movement 4
1.5.3 Wait Time 5

2 Related Work **7**

3 Method **9**
3.1 Multiple Graphics Adapter 9
3.1.1 Creating multiple adapter 9
3.1.2 Resources and Unordered Access Views 10
3.1.3 Synchronising 11
3.2 Engine Structure 11
3.2.1 Improvements 13
3.3 Particle Structure 13
3.3.1 Emitter Data 13
3.3.2 Particle Pass Pipeline 14
3.4 CPU or GPU 14
3.5 Emitter Settings 15
3.6 Creating Particles 16
3.7 Particle Movement 16
3.8 Creating the Vertex Buffer 17
3.9 Wait Times 18
3.10 Experiment 19
3.10.1 Data Collection 19
3.10.2 Comparing results 20
3.10.3 Test Machine 20

4 Results **21**
4.1 Differences . . . 23
4.2 Particle Amounts . . . 24
4.3 Stability . . . 24

5 Analysis **27**
5.1 Did the performance of the particle system increase? . . . 27
5.2 Bottleneck . . . 28
5.3 Compared to the null hypothesis . . . 28
5.4 Scaling the System . . . 28
5.5 Frame Time & GPU offloading . . . 29

6 Discussion **31**
6.1 Why should anyone even care?! . . . 31
6.2 What is the impact of this? . . . 31
6.3 The Future of Dual GPU Rendering. . . . 32
6.4 Is the cost of implementing a dual GPU adapter system worth when constructing a DirectX 12 engine. . . . 33
6.5 Problems . . . 33
6.5.1 Development of the Application . . . 33
6.6 Improvements . . . 34
6.6.1 Copy Times . . . 34
6.7 Other Possible Uses . . . 34
6.7.1 Shadow mapping . . . 34
6.7.2 HUD . . . 34

7 Conclusions **35**
7.1 Future Work . . . 35

References **37**

List of Figures

3.1 Engine Execution Path. 12

4.1 256 Particles executed on both a dual and single GPU system. 21
4.2 65536 Particles executed on both a dual and single GPU system. . . . 22
4.3 1048576 Particles executed on both a dual and single GPU system. . 22
4.4 Results of an increasing amount of particles. 22
4.5 Results of an increasing amount of particles. 23
4.6 Particle execution times over 100 frames with 1 million particles. . . . 25

List of Algorithms

1 Particle Spawn 15
2 Particle Movement 16
3 Creation of Vertex Buffer for Particle Emitter 17
4 Original Wait 18
5 New Wait 18
6 Calculate Total Time 20

Chapter 1

Introduction

Now with Moore's law coming to an end [1] and it is getting harder to manufacture smaller and faster processors. Manufacturers have moved over to produce processors with more cores and encouraging programmers to create more parallel workloads. With modern `Intel`® CPU's coming with an integrated GPU that is frequently overlooked, why not use the secondary GPU for parallel compute work.

With the new `DirectX 12` API, there is native support to create a secondary Device that can be explicitly referenced for parallel compute work. This is because there is native support in the `DirectX 12` API to create multiple devices. There was no native support for initializing multiple devices in the older `DirectX` API's as it was the drivers job to allocate work to the different GPUs but now it is the developer's job.

A task that can be executed in parallel on a secondary GPU [4] is a particle system. A particle system can be highly parallelized because every particle in the system has to be updated and this is why it excels in parallel environments. This is as a result of every particle being able to have a dedicated hardware thread to update, so with more cores, more particles can be executed at any single moment. This can either be performed on a CPU that usually has around 2-8 cores, while a modern GPU can have up to 2500 cores. When running the particle system on the CPU, every particle has to wait for a core to be open before it can be updated, this leads to a lot of waiting time for the particles before they can do any computational work. But if the particles are run on the GPU there is a lot less time that the particles have to wait before a core is open for computation. This is why particles are great for parallel work and why they will be utilized for stressing the secondary GPU in parallel to the main GPU.

Lots of gaming ready computers nowadays have a secondary GPU that is not usually utilized in gaming environments. This is a result of Intel® market share and there integrated GPU's[1]. This article will demonstrate the potential benefits of running a particle system in parallel on a secondary GPU.

[1]These statistics are based on https://www.userbenchmark.com/, which is a crowd gather statistics database on computers. These results might be a bit biased but should still give a good picture of the average computer.

1.1 The modern problem

There is a large amount of computation that takes place on the GPU [5] in modern games, everything from graphical rendering pipeline to particles. All of these tasks are fighting for time on GPU on gaming focused graphics adapters produced by `NVIDIA` and `AMD`, with `NVIDIA` currently holding a majority market share over `AMD`. Async compute is still a new technique that has little to no support with gaming graphics adapters. So a reasonable alternative for asynchronous compute is to use a secondary GPU for the asynchronous compute parts like the particle system. Often what is done with graphics adapters is that they are connected in `SLI` or `Crossfire`™ mode, but this can only be done if the graphics adapters are identical. This then requires the user to purchase two identical graphics cards, this is expensive and it is not a huge number of users that do this. But developers still want more performance out of computers so they can squeeze in more rendering techniques on the GPU. Moving over rendering work to the secondary GPU frees up the main GPU to execute more rendering techniques while the secondary GPU does asynchronous compute tasks.

1.2 Particle System

Particle system [6] is a system that exists in almost every modern AAA game, and that is often executed on the GPU taking up valuable GPU time. This GPU time could be used for other rendering techniques using the direct queue and reducing the number of different queues executing on the GPU allowing for a more streamlined pipeline as `NVIDIA` GPU's do not do asynchronous compute very good.

Executing particles on the GPU is much more effective than executing on the CPU, because a particle system is a highly parallelized workload. This makes it a prime candidate to be moved on over to a secondary GPU for execution. This is why particles will be used to set the benchmark in this article. Particles will be run on the primary and secondary GPU while other rendering passes are taking place like shadow mapping, Screen space ambient occlusion or deferred rendering.

1.3 Why use a secondary GPU

With the shift that has been happening in the CPU and GPU market where processors have gone from getting faster and faster threads to suddenly create more and more threads. This an indication that processes should be able to utilize more cores and run more parallel [8] to other tasks that are happening in the program.

To use the previously unutilized hardware that exists in the form of an integrated GPU in the CPU. This is hardware that already exists in most gaming ready systems that are just waiting to be used. With AAA games utilizing 100% of modern GPU and still needs to find room to add more features, they need to expand to secondary hardware that might exist in the systems. This would allow developers to clear up the primary GPU for new features and or more smooth running games.

The secondary GPU has often been used as a power saving feature, due to always running the primary GPU uses a lot of power [5] compared to the integrated GPU,

so for tasks that do not require the power of the primary GPU, the secondary GPU can be utilized. This is probably one of the reasons why the secondary GPU has not been used for rendering purposes.

1.4 Research Questions

The goal of this research is quite simply to move work from the primary GPU to a secondary GPU. This will allow for a bit more free use of the primary GPU and hopefully better the overall performance. Due to previous experience the expected result would be that the execution time for the particles will take longer as the secondary GPU is slower then the primary, but the fact that it is executed in parallel with the main graphical rendering pipeline will give it an edge and give a slightly improved performance by maybe 10 - 25%. This could give a whole new way of utilizing hardware, where the task that can be done asynchronously is done asynchronously and does not affect the performance of the primary GPU. This would allow for greater scaling when it comes to game performance, where computers with more cores would run faster than computers with fewer cores.

1.4.1 What is the Problem?

In modern AAA games, developers are constantly trying to optimize current algorithms and push modern hardware to their limits. But instead of pushing close to the limits of the primary graphics adapter, but instead, move the limits by adding a secondary GPU. This could be done by using the secondary integrated GPU that can be found in many `Intel®` CPU's. This could give a performance boost of up to 10 - 25%[2] in overall performance.

Now with `DirectX 12` giving the developer full control over the adapter usage, there is no excuse for not using other available hardware adapters that could be utilized for reducing the overall frame time. One problem is assuming that there is a second hardware adapter, but this can be checked for and the code has to be designed so that it can run on multiple GPUs or a single GPU. This can be done by having the secondary GPU be the primary GPU if there does not exist a secondary GPU. This will trick the code that there always are two GPU's but there might only be one. This is a hack and could potentially lead to performance and debugging trouble in the future as one GPU is seen as two.

[2]The percentage is based on the differences in results between an Intel® HD Graphics 630 and an NVIDIA GeForce® GTX 1080 with a certain benchmark.

1.4.2 Research Questions

The questions for this thesis are;

- RQ1 : Is it possible to increase particle system rendering performance by utilizing a separate GPU?
- RQ2 : What and where are the performance differences when using a separate GPU?

1.5 Theory

For the particle system, the expected outcome is that there is going to be a range in the number of particles where the secondary GPU is faster than the primary. This is since previous experiences say that the time it takes to move the data is going to be the bottleneck. A bottleneck is a point in a system that takes the most time to execute.

But other processes in the system could potentially be the bottleneck, perhaps the time it takes to do all of the particle calculations will be the bottleneck. If the particle execution is the heaviest part, then moving the executions to a secondary GPU will probably be beneficial as long as the secondary GPU is fast enough. If the secondary GPU is too slow compared to the primary GPU, then it might not be worth doing the particle calculations on a secondary GPU. This is as a result of the wait time on the primary GPU, this is due to the primary GPU having to wait for the secondary GPU to complete its task.

1.5.1 Execution Time

The execution time is the time it takes for the particles when all of the data is on the GPU to be calculated. When the particles are run on a faster GPU, the execution time will be lower and when it is executed on a slower GPU it will be longer. Lower times are better as the particle emitters should execute as fast as possible.

What is done during this time is that the particles are getting new positions and the vertex buffer is created for the particle emitters.

1.5.2 Data Movement

When using a secondary GPU the particle data has to be moved from the secondary GPU to the primary GPU. This is a slow operation considering that the data have to be moved from the secondary GPU to the CPU when the data have reached the CPU it can be processed and uploaded to the primary GPU for rendering. This is done after the execution is done.

1.5.3 Wait Time

Previous experiences say that the wait time is going to be the big bottleneck, and the wait time is the time when one of the GPU's is not doing any work. This can be seen at the utilization of the GPU, if the GPU is only used 55% there is a great chance that it will have to do a substantial amount of waiting where it is not doing anything. Wait times are something that should be reduced to zero, but it is not an easy task with multiple GPUs with different executions times, that may change depending on what GPUs are in the system.

One way to get around the wait times is simply not to wait and just continue with the execution with the previous data. This will give the particles another frame to complete the calculations. This might make the particles look very rough and uneven, this is as a result of that there is not an even time between particle updates.

Chapter 2
Related Work

There is a lot of previous work with particle physics running on multiple GPU's [2] or utilizing a large amount of GPUs for server work [4] over even creating async particles [7]. There is even research for running a particle system in real-time mobile environments [3]. No published research utilizes a different lesser secondary GPU in a nonserver context, like an integrated GPU for the particle calculations in real time environments or games. This is a gap in the research that this thesis intends to explore. But before the course, the research and implementation of a dual GPU compatible system have already been completed that will be used for measuring the difference between using one and two GPU's.

Chapter 3

Method

To be able to test if running particle calculations on a secondary GPU is beneficial, a testing environment had to be established. These tests were built from a general purpose rendering engine. The engine was not designed for running on multiple GPUs initially, but it was made with asynchronous compute and threading in mind. What is meant with this is that multiple passes can be executed as simultaneously as the hardware allows. All that had to be done, was to move over the particle calculations to a secondary GPU and make sure that it all syncs up.

3.1 Multiple Graphics Adapter

In `DirectX 12` the access to the hardware adapters have been exposed up to the developer, this allows the developer to explicitly access the optimal GPU. This was previously done by the graphics drivers, but a significant downside with this is that the graphics drivers only consider other graphics adapters that are connected in `SLI` or `Crossfire`™. This limits multi-GPU systems to GPU's that are identical to each other, with this setup there is no way of utilizing a GPU of a different kind.

This is both a good and a bad thing as the developer now has full control and responsibility. It can be very useful for an experienced developer to have full control over the code and where it is executed. This will give a smoother experience and easier for the developer due to there is no need to rely on the drivers to do its thing not to change.

3.1.1 Creating multiple adapter

To create two different hardware adapters in `DirectX 12`, the developer simply needs to create two `ID3D12Devices` objects, where they both refer to a specific but different GPU. How both the GPU's are accessed afterward is through the corresponding `ID3D12Device` object.

But if the graphics adapters are connected in `SLI` or `Crossfire`™ mode then there is only need for one `ID3D12Device`. The reason for this is that the drivers see the graphics adapters as one adapter. But when creating resources a specific graphics card can still be specified with the `CreationNodeMask` and the `VisibilityNodeMask`, this allows the developer to use the graphics cards as two adapters instead of one adapter. The node mask is specified as an unsigned integer but it is not the actual integer value that is used to reference one of the graphics cards instead, it is the bit value of the unsigned integer. This is because every bit in

the unsigned integer is used as a "gate" to access the graphics card, where the first bit is referencing to the first graphics card and the second reference to the second card. If a bit is set or not, determine if the data is accessible for the graphics card in question. This allows the developer to pool the graphics memory instead of sharing it potentially doubling the graphics memory that is accessible by the developer.

So if the value of the bit mask is 2 then the corresponding bits are 0010, this would give access to the second graphics card in the `SLI` or `Crossfire`™ setup. But if the bit mask value was 3 then the bits would be 0011 and both the graphics cards would gain access.

To create two `ID3D12Devices`, then, first of all, there need to be two available graphics hardware adapters. If there exist two available adapters then it is simply just creating two devices where each device references one of the available hardware adapters.

3.1.2 Resources and Unordered Access Views

Creating a shared resource between hardware adapter can be done as long as the resource is not accessed from the CPU, and what is meant by that is that the CPU should not have access to the resource, it can not write or read from it. Why this is, is a result of the memory bandwidth between the hardware adapters, the hardware adapters need to be able to send the resource between the at a high speed and to avoid slowdowns the data is not moved through the CPU. This gives the resource the disadvantage that it can not be CPU accessible.

The core of the particle system is the unordered access view (UAV) and what makes a UAV so useful is that the GPU can write to that memory. Every resource in `DirectX 12` is seen as a read-only resource on the GPU side and CPU side if other parameters are not explicitly set. This is a result of the CPU being able to quickly transfer the data to the GPU. This data is not allowed to read from the CPU or to write form the GPU, which makes it useless for the particle system. For the particle system to work the GPU needs two write the results in an array of particles, this can only be done if the GPU has access to write to the memory. To give both the CPU and the GPU access to the memory a UAV can be created, this then has the downside that it can not be created as a shared resource between the graphics adapters due to the CPU access.

Movement of the resource between the adapters has to be done manually now because there is no direct link between the graphics adapter for that resource now. To move the resource between the adapters, the resource first has to be downloaded from the secondary GPU down to the CPU, where the CPU can process the data and uploaded the data to the primary GPU.

If a technique like shadow mapping had been used instead of particles, shared heaps might have been an option because there is no need for the CPU to interfere. This would allow for faster memory movement between the different graphics adapters.

3.1.3 Synchronising

To synchronize multiple graphics adapters in `DirectX 12` is done with the native `ID3D12Fence` class, this offers native support for synchronization within and between graphics adapters.

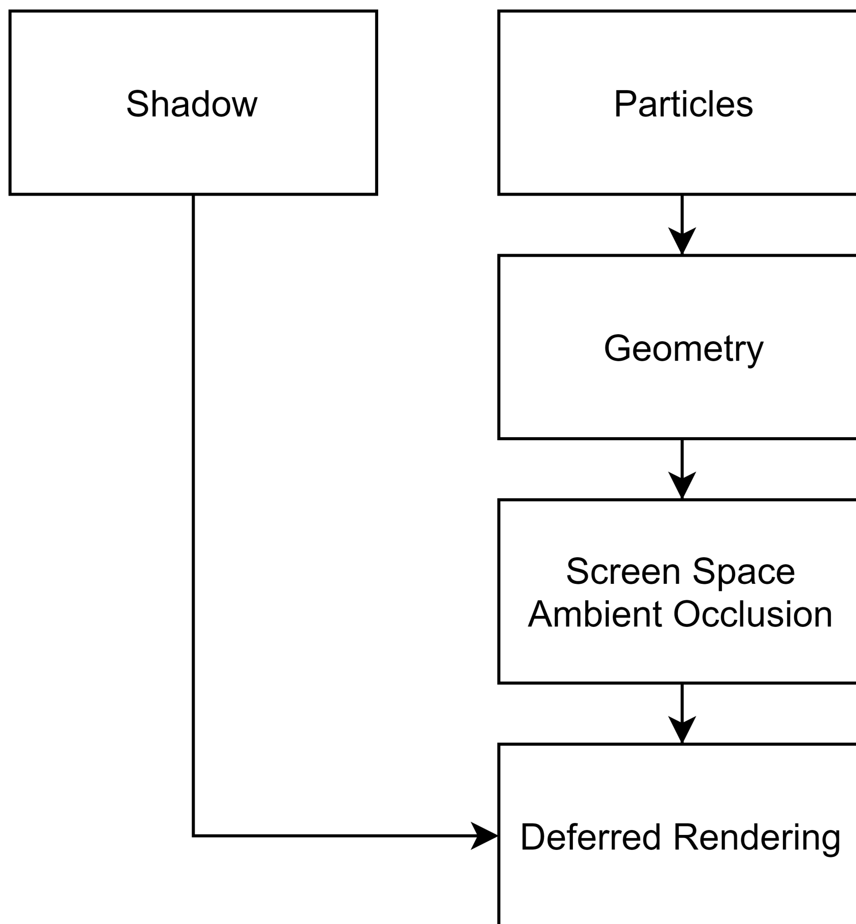
The developer simply needs to signal graphics adapter with a value of increasing magnitude that should be set when the current `CommandQueue` is done with its execution. Then the fence offers different methods of waiting for the `CommandQueue` to finish, one way is to stop the `CommandQueue` from executing the next batch of command until the previous execution is completed. This effectively waits on the GPU side of the execution, but the data for the execution can still be uploaded, this is the most effective way to wait because the as soon as the fence is complete the GPU can continue and the CPU can keep recording command and updating other parts of the engine. The other way of waiting for the fence is to wait on the CPU side for the `CommandQueue` to finish, this will freeze the entire execution of that thread and everything after will have to wait for the previous `CommandQueue` to finish its job before the CPU can start recording the next batch of commands. To wait on the CPU is much slower as there is a time where the GPU does not have commands to execute and waits on the CPU for more commands.

3.2 Engine Structure

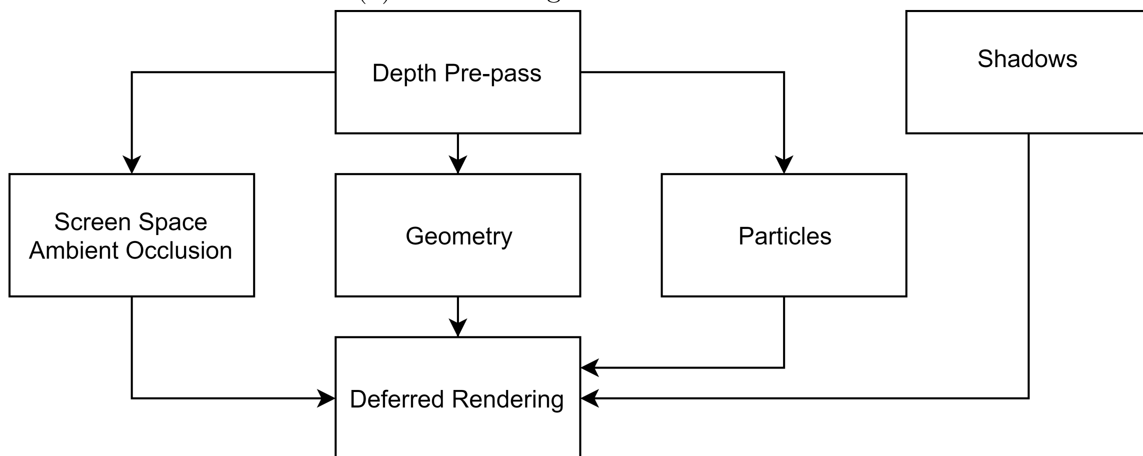
The engine is structured so that not any rendering pass is necessary for the final image and can easily be removed from the final image if there is no need for the effect. The current execution path of the engine can be seen in figure 3.1a, if needed any pass can be removed when rendering but all of the passes will be used for the particle calculations. For the particle calculations not to be the bottleneck, the particles are always at least one frame late, before they are rendered. This gives more time for the GPU to complete the particle calculations.

To draw a single object in the engine all that is required by the developer is to load a mesh and call the function
`Drawable::Draw()`. This will submit the drawable object with the mesh for rendering pipeline. Depending on what setting is set in the Drawable object, the object will be queued to different rendering pipelines. As an example, if the Drawable object is set to cast shadow then the object will be added to the shadow pre-pass rendering pass queue. A drawable object in the context of the engine is an object that all of the positional and rendering information are stored. A drawable object can share meshes and textures with other object reducing the number of meshes that have to be loaded.

The first pass to execute is the particle pass because it is one of the slower passes. After that, the Shadow pass starts in parallel to the particle pass. These are both executed on different command queues, the direct and compute queue. When the particles are finished on the CPU side, the geometry pass will start and render all of the objects to four different textures, the world position, color, normal and the metallic. All of these will be used later in the deferred rendering pass that combines all of the other rendering passes. The particles are also rendering in the geometry pass



(a) Current Engine Execution Path



(b) Alternative Engine Execution Path

Figure 3.1: Engine Execution Path.

as the particles need to be in the same render target and use the same depth buffer. This could be solved by using a Depth pre-pass to render the depth buffer, this depth buffer could be used to also render the particles on the secondary GPU furthermore reducing the load on the primary GPU. After the geometry pass, the required data is moved over to the Screen space ambient occlusion rendering pass. When all of the primary rendering passes are completed it is time for the deferred rendering pass. Here is where all of the light and shadow calculations are done, and the final image is constructed and prepared for presentation to the screen. The deferred rendering pass is also the most intensive when it comes to graphical computation and is the rendering pass that takes the longest to execute on the primary GPU. So for the particles to give the deferred rendering pass more time to execute would give the engine an overall lower frame time and better frame rates.

3.2.1 Improvements

What could be done to improve the rendering time is to do a depth pre-pass where the world position and depth buffer is saved. With this information, the rest of the engine could execute in parallel as seen in 3.1b. This would probably allow the CPU and the GPU to execute more work, and lower the overall frame time. But most importantly is that it would give the particles a bit more time before the process has to wait for the particles before continuing. This engine structure is untested and there is no certainty that it would be faster but due to previous experience doing more work in parallel does improve performance and reduces wait time. So if the GPU always has worked then it will utilize the utmost of the GPU.

3.3 Particle Structure

The particles are sorted into different particle emitters. It is the emitters job to create new particles and storing them. Here is where all the required calculation and vertex data is stored. When all of the data is created it is important for the engine to know if the particles are going to be executed on a secondary GPU or the primary GPU. For the reason that all of the buffers need to be created and be visible to the correct GPU, due to all of the buffers being UAVs as they can not be shared between adapters. This is done by trying to create all of the buffers on the secondary GPU, if it does not work then there is no secondary adapter for all of the data to be created on. If the data creating failed on the secondary GPU the buffers are re-created on the primary GPU and the system continues.

3.3.1 Emitter Data

Every emitter needs three buffer where two of them are UAVs, the first one or the calculations buffer is the particle positions and other vital data that is essential for the execution of the particle emitter. The calculations buffer is an output buffer, this means that the GPU writes to this buffer and later on the CPU the buffer is downloaded and copied to a corresponding CPU accessible array. The calculation buffer is used for the emitter to know if it should create more or delete particles.

The second buffer or the vertex buffer and it is also an output buffer and will later be copied to another vertex buffer that is located on the primary GPU, but this step can be skipped if there is only one GPU. The vertex output buffer is copied over to another vertex buffer that is always on the primary GPU. But if there is only one GPU the vertex output buffer can be directly bound as the vertex buffer when rendering the particles later in the graphical pipeline state. The third buffer is the vertex buffer, this is only needed when there are two GPU's as this is always on the primary GPU. This means that the vertex output buffer needs to be downloaded and copied over to the vertex buffer, this is a step that is not needed when executing the particles on one GPU as all of the data already exist on the same GPU. The emitters also hold an array of pointers to the textures that the emitter needs for the rendering later on and what texture that should be used is decided by how long the particles have been alive. So if there are three textures then the first third of the particle life, it will use the first texture.

3.3.2 Particle Pass Pipeline

The emitters are submitted to the particle compute pass, where the emitters particle data is updated and moved in to buffer that can be uploaded to the GPU. When all of the buffer packing is complete the data can be uploaded to the correct GPU for processing. Depending if the system is running on multiple GPUs or a single GPU the particles are uploaded to the GPU that is going to do all of the calculations, this will be the secondary GPU if there exists one. When the particle data is on the GPU the compute pass can execute the particle movement and create the output buffers that are needed for updating the emitters on the CPU side and later render the particle emitter in the geometry render pass. The particles are downloaded to the CPU and updated, this is if the engine needs CPU access to the particles for something like physics enables particles. The compute shader has two jobs to do, the first is to update all of the particle's position and other vital information, the second is the create a vertex buffer that can later be used in the geometry pass.

To update the buffers on the CPU. The CPU needs to wait for the GPU to complete, but this is an expensive action as it locked up the entire CPU pipeline and grinds everything to a halt. Instead of waiting for the GPU to finish its execution the CPU skips the updating of the particle for the current frame and waits for the next frame. This allows the particle to update at a lower frame rate than the rest of the rendering and does not hold up the rendering pipeline. When a frame is skipped the primary GPU uses the old data from the previous frame to once more render the particles.

3.4 CPU or GPU

Why this is executed on the GPU and not the CPU is due to the fact that a GPU will always have more cores than a CPU. This allows the GPU to calculate a lot more particles at the same time that the CPU would have done it. This comes with its problems, for example when pushing elements to the output buffer. All of the elements need their specified position before the calculations even start. The compute

pass also needs to know what position to place the output information when it is done. This can be done by pre-allocation an array that is precisely big enough for the max number of particles that can be outputted. When recording a `Dispatch(x,y,z)` call to the GPU, the developer needs to specify how many threads that shall be executed, every thread then gets a dispatch thread ID. With this information, the particle output position can be calculated by multiplying the thread ID with the number of elements in the output structure. A flaw with this is that all of the elements in the output structure can only be of types that are the same size in memory. So, for example, a single `float4` takes up 16 bytes in memory while 4 unsigned integers also takes up 16 bytes. These can be pair with each other while something like a `float4` and a `UINT64` cannot be paired with each other, as the `float4` takes 16 bytes and the `UINT64` takes up 64 bytes. The output of the particle is constructed of different `float4` where they are used as either a position or every individual `float` is used for other vital information such as time alive.

This would not be a problem if the particles were done on the CPU side as the particles can be pushed back in a vector. But it would still have to do some waiting if another thread is writing to the same vector, but the wait times would not be anything big, as there would not be a lot of threads that are trying to write to the vector. As a modern CPU only has around 2-8 cores with 4-16 threads, this only allows the CPU to calculate a maximum of 16 particles at a single movement. While meanwhile, a GPU can have upwards of 3,000 cores, this gives a theoretical max of 3,000 particles that could be calculated parallel to each other. If 3,000 threads are trying to write to the same position there is going to be a lot of wait time for every thread.

Algorithm 1 Particle Spawn

1: $SpawnTimer \leftarrow SpawnTimer + DeltaTime$
2: **if** $SpawnTimer >= SpawnRate$ **then**
3:     **while** AmountOfParticles $<$ MaxParticles **do**
4:         $ParticlePos \leftarrow Base + Rand(0,1) * Spread$
5:         $TimeToLive \leftarrow MaxLifeTime * Rand(0,1)$
6:         **if** $TimeToLive < MinLifeTime$ **then**
7:             $TimeToLive \leftarrow MinLifeTime$
8:         **end if**
9:         $PushBackParticle(ParticlePos, TimeToLive)$
10:     **end while**
11: **end if**

3.5 Emitter Settings

The emitter settings define how the particles should be structured and how they shall behave. This is done by setting different variables, such as `Speed`, `Spawn Spread`, `Direction`, `Size`, `Spawn Rate`, `Min/Max life span` and `Max particles`. All of these variables will change how the particle will behave and look like. The `Speed` variable will change how fast the particle will travel, this could be a range instead

of a fixed value for all the particles in the system. This is an aesthetic change and will not affect performance. What the `Spawn Spread` variable does is to define how far from the center of the emitter that a particle can spawn. The `Direction` will determine the direction of all of the particles. The `Size` determines the size of all of the particles, this could also be a range between min and max to get some different sizes on the particles, this is another aesthetic change and will not affect performance. The `Spawn Rate` and `Min/Max life span` will determine how fast the particles spawn and how long they live. The `Max particles` is a hard cap on how many particles can exist in the emitter and this is not anything that can be changed during runtime as it defines how much memory is allocated to the buffers. The particles are spawn according to the algorithm seen in algorithm 1. This algorithm does not allow for a slow "ramp up" of particles, the emitter will always start with the max number of particles. This is done for testing purposes so there is no need to wait for all the particles to be created. This algorithm makes sure that there is always a max amount of particles.

3.6 Creating Particles

The particles are created on the CPU side and are given a direction based on what the emitter settings say. Then they are placed in a vector, but the vector does not store the particle data, it only stores the data that is needed to create a particle. The particle data is stored in the different buffers, and the particles are recreated every frame based on the new information, that is downloaded from the previous execution on the GPU.

Algorithm 2 Particle Movement

1: $i \leftarrow particleIndex$
2: **if** $dT > 0$ **then**
3: $\quad TimeAlive[i] \leftarrow TimeAlive[i] + dT$
4: $\quad$ **if** $TimeAlive[i] > MaxLifeTime[i]$ **then**
5: $\quad\quad Pos[i] \leftarrow SpawnPosition[i]$
6: $\quad$ **else**
7: $\quad\quad Pos[i] \leftarrow Pos[i] + Dir[i] * Vel[i] * dT$
8: $\quad$ **end if**
9: **end if**
10: $vID \leftarrow ThreadID * 2$
11: $Calc[vID + 0] \leftarrow Pos[i]$
12: $Calc[vID + 1] \leftarrow (TimeAlive[i], MaxLifeTime[i])$

3.7 Particle Movement

Moving and displaying the particles require two different buffers, one with the new position of the particles and one with the vertex buffer. To calculate the new position the compute pass use the simple equation seen in algorithm 2. All this does is to add

the direction to the current position with the magnitude of the velocity multiplied with the delta time since the last update. The direction is chosen when the particle is created and is never changed during the entire emitters lifespan once the direction is set it does not change. Once a particle is created, it does not change until the emitter is destroyed. This is a performance optimization as the particles are created on the CPU, and to offload some of the work from the CPU to the GPU, the particles are never destroyed. They are just moved back to the spawn position, and the lifetime is reset, this is why when the particles are created they can not be changed without destroying the emitter and creating a new emitter with new particles.

Algorithm 3 Creation of Vertex Buffer for Particle Emitter

1: $Direction \leftarrow normalize(particlePos - cameraPos)$
2: $Right \leftarrow normalize(cross(Direction, UP))$
3: $Up \leftarrow normalize(cross(Direction, Right))$
4: $UpperLeft \leftarrow particlePos + (-Right * SizeX) + (Up * SizeY)$
5: $UpperRight \leftarrow particlePos + (Right * SizeX) + (Up * SizeY)$
6: $LowerLeft \leftarrow particlePos + (-Right * SizeX) + (-Up * SizeY)$
7: $LowerRight \leftarrow particlePos + (Right * SizeX) + (-Up * SizeY)$
8: $UpperLeftUV \leftarrow (0, 0)$
9: $UpperRightUV \leftarrow (1, 0)$
10: $LowerLeftUV \leftarrow (0, 1)$
11: $LowerRightUV \leftarrow (1, 1)$
12: $vID \leftarrow ThreadID * 12$
13: $VertexBuffer[vID + 0] \leftarrow LowerLeft$
14: $VertexBuffer[vID + 1] \leftarrow LowerLeftUV$
15: $VertexBuffer[vID + 2] \leftarrow UpperLeft$
16: $VertexBuffer[vID + 3] \leftarrow UpperLeftUV$
17: $VertexBuffer[vID + 4] \leftarrow UpperRight$
18: $VertexBuffer[vID + 5] \leftarrow UpperRightUV$
19: $VertexBuffer[vID + 6] \leftarrow LowerRight$
20: $VertexBuffer[vID + 7] \leftarrow LowerRightUV$
21: $VertexBuffer[vID + 8] \leftarrow LowerLeft$
22: $VertexBuffer[vID + 9] \leftarrow LowerLeftUV$
23: $VertexBuffer[vID + 10] \leftarrow UpperRight$
24: $VertexBuffer[vID + 11] \leftarrow UpperRightUV$

3.8 Creating the Vertex Buffer

The creation of the vertex buffer is necessary for rendering the particles, and there are two ways of doing it. The first is to create it during the computation of the particles and save it down to a vertex buffer this is more memory intensive as it requires more memory to store the entire vertex buffer instead of only the positional information, the other is to create the particle vertices on the fly in a geometry shader based on the position of the particles. Creating the particles in a geometry shader would probably be the more effective way of doing making the vertices if all of the particles

were calculated on the same GPU as there would be no memory movements. But in this instance the vertex buffer is created when the particle is calculated, this would move more of the particle calculations over to the secondary GPU.

The particle vertex buffer is created by taking the position of the particle and creating the vertex points based on the camera position. This is to make sure that the particles always look toward the position of the camera. The particles are created using the following algorithm seen in algorithm 3. This will assure that all of the particle vertex data is packed and coming in the right order, then this buffer will be set as the input vertex buffer if the process is running on a single GPU system. If there is more than one GPU then the data will have to be moved and therefore it will be downloaded to the CPU and copied over to a secondary buffer that exists on the primary GPU and then uploaded. This is as a result of the inability to create CPU accessible buffers that are shared between multiple hardware adapters.

Algorithm 4 Original Wait

1: $DispatchParticles(DeltaTime)$
2: $Wait()$
3: $UpdateParticles()$

Algorithm 5 New Wait

1: **if** $PreviousFrameComplete()$ **then**
2: $\quad dt \leftarrow dt + DeltaTime$
3: $\quad UpdateParticles()$
4: $\quad DispatchParticles(dt)$
5: **else**
6: $\quad dt \leftarrow dt + DeltaTime$
7: **end if**

3.9 Wait Times

When the particles are executed on the GPU, the CPU either needs to wait for the GPU to finish or ignore to update the current data and using the old set of data. The data update cannot be done before the particle calculations are completed on the GPU. Instead of waiting after every particle `Dispatch` as seen in algorithm 4. The system now uses the new algorithm 5 where it does not update the particle data if the previous execution is not complete. This will allow the particles to update at a lower tick rate than the rest of the engine, this will also reduce stalls for the primary GPU. As the time the engine has to wait for the secondary GPU is reduced to zero. A flaw with this system is that particles can look rough and uneven, as they are not moving as smoothly over the screen. This is a result of the particles moving the same distance but in a fewer amount of steps. So where had it previously moved one unit in 60 steps over one second, the particles can now one unit in 45 steps over one second. This is even when the engine is updating at a steady frame rate. All of this is done to reduce the wait time for the rest of the engine. This will give the

engine a smoother frame rate when handling input and rendering the scene for the user even when the particles cannot keep up with the rendering of the scene. But as the number of objects that are draw goes up and the particle count stays the same, then the rendering times will equal out. The particles can never be updated at a higher tick rate then what the rest of the engine is running at, this is as a result of the engine is the limitation in this case. The particles will have to wait for the engine to get to the particle execution. This is not a problem in that if the particles are updated faster then the engine can display them then there is a lot of unnecessary work that is being done.

This could result in the CPU having to wait before it can record its command list that should be sent to the GPU and why the command list cannot be recorded and executed before the previous executions are finished on the GPU is as a result of that the number of particles is not known before the previous execution is complete.

3.10 Experiment

The formal experiment will be made up out of a lot of smaller tests, where different amounts of particles are tested. The experiment will test everything from running 256 particles to 1 million particles as this will give a clear representation of where the performance benefits and where the bottlenecks lie if there is any. The test will be done with 256, 1024, 4096, 16k, 64k, 256k, 512k, 1m with both the secondary GPU active and the without the secondary GPU. Comparing the results between the tests will show if there is an increase in performance and in what range with the particle system running on the secondary GPU.

3.10.1 Data Collection

When the tests are executed, there is a lot of data to be collected. For there to be any interesting information to derive from the tests, there needs to be a lot of data. For every frame, five timers are running and measuring different parts of the system. This is to give a good overall view of the entire system as a whole. The first timer measures the time it takes for the first part of the shadow pass to complete, this is measured as it is executed in parallel with the particle pass. Two timers measure the particle pass, the first one measures the entire particle pass as a whole and returns the total time it takes to execute one iteration of the particles. The second time measures only the time it takes for the CPU to download the data from the GPU and update the data on the CPU. If the particles are calculated on a secondary GPU the timer also measures the time it takes to copy over the data from the secondary GPU to the primary GPU. The two last timers measure the total frame time and the time it takes for the rest of the engine to execute its work. As the timers are not timers that increment on a variable, but instead save down the current time of the GPU they do not have a noticeable performance impact. This also allows them to measure over multiple frames.

All of this time data is stored in RAM until 100 frames have been collected when the desired amount of data is collected the timers are printed down to file. This data can be imported to the desired spreadsheet application. The data is stored in RAM

until the collection is completed to make sure that there is no hardware stall when writing to a file that could affect the performance of the entire engine. This will assure that the majority of the hardware is used for the particle calculations and not data collection.

Algorithm 6 Calculate Total Time

1: $Ticks \leftarrow End - Start$
2: $ToMs \leftarrow (1.0/QueueFrequency) * 1000.0$
3: $Total \leftarrow Ticks * ToMs$

The timers in them self are GPU timer, this means that the times that are taken are as accurate as possible. How they work is by query the GPU for the GPU tick at the start and the end of the timer. This will get the total time in GPU ticks. But using GPU ticks as a metric is useless for comparing multiple timers as different GPU and even different queues on the same GPU can have different tick rates. To convert from ticks to milliseconds algorithm 6 is used.

3.10.2 Comparing results

To compare the data, the data first needs to be structured in sheets so the P-Value can be calculated with the `TTEST` function. As the data needed to answer RQ1 is the particle execution time and those are the times that were compared to each other.

3.10.3 Test Machine

The tests will be carried out on a `Intel®` `Core™` `i7-7700K` Processor and an `Nvidia GeForce GTX 1080` graphics processor, where the Intel processor has an `Intel® HD Graphics 630` integrated graphics processor.

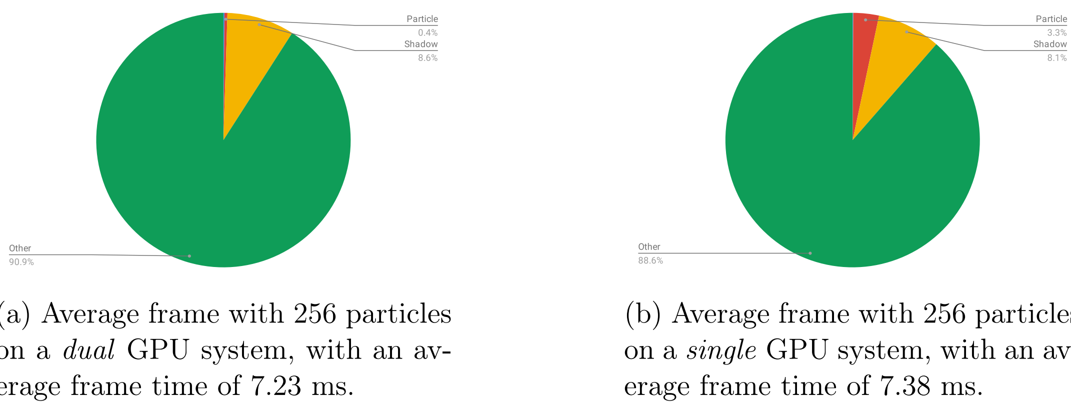
This will give a representative result as the secondary GPU is an integrated GPU, the only problem is that it is only one machine with one configuration.

Chapter 4

Results

Num particles	Percentage diff
256	681.8 %
1024	637.3 %
4096	326.9 %
16384	306.4 %
65536	374.7 %
262144	363.1 %
524288	452.0 %
1048576	494.4 %

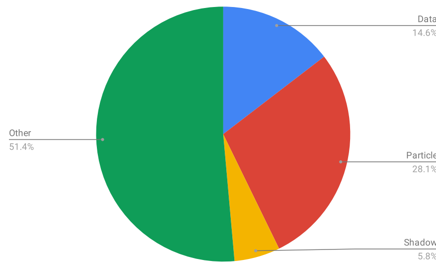
Table 4.1: The percentage difference between running the particles on a dual GPU system vs a single GPU system.



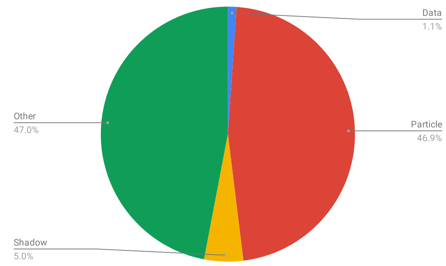
(a) Average frame with 256 particles on a *dual* GPU system, with an average frame time of 7.23 ms.

(b) Average frame with 256 particles on a *single* GPU system, with an average frame time of 7.38 ms.

Figure 4.1: 256 Particles executed on both a dual and single GPU system.

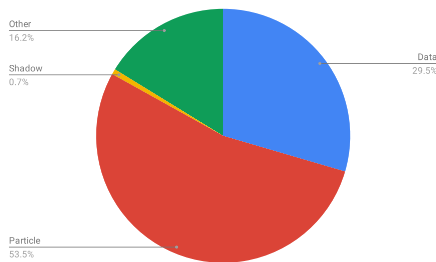


(a) Average frame with 65536 particles on a *dual* GPU system, with an average frame time of 8.86 ms.

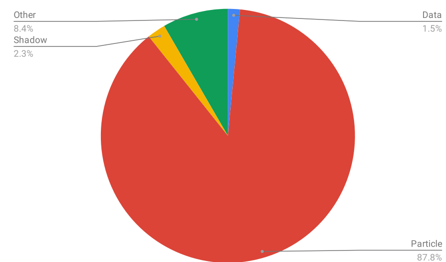


(b) Average frame with 65536 particles on a *single* GPU system, with an average frame time of 19.9 ms.

Figure 4.2: 65536 Particles executed on both a dual and single GPU system.

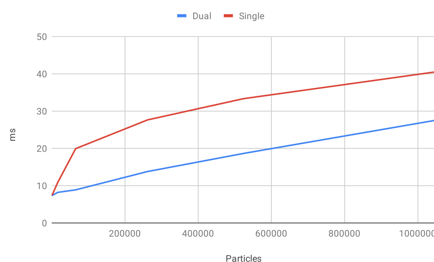


(a) Average frame with 1048576 particles on a *dual* GPU system, with an average frame time of 27.5 ms.

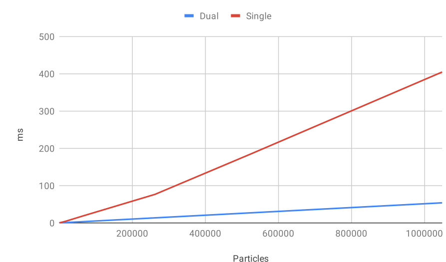


(b) Average frame with 1048576 particles on a *single* GPU system, with an average frame time of 40.5 ms.

Figure 4.3: 1048576 Particles executed on both a dual and single GPU system.

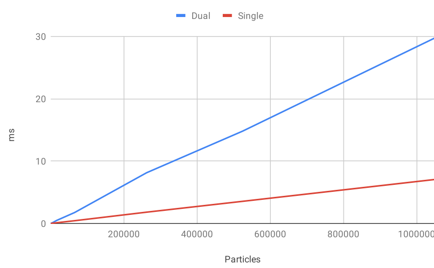


(a) Average *frame time* with increasing amount of particles.

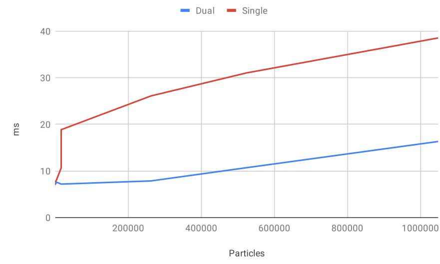


(b) Average *particle execution time* with increasing amount of particles.

Figure 4.4: Results of an increasing amount of particles.



(a) Average *data movement time* with increasing amount of particles.



(b) Average time it takes for the rest of the engine to execute with an increasing amount of particles.

Figure 4.5: Results of an increasing amount of particles.

The results of the test were not as expected, it showed it was major improvements of running the particles on the secondary GPU no matter how many particles were executed. The total execution time for the particles went down and the general frame time went down. This resulted in a smoother execution of the engine and smoother particles movements.

4.1 Differences

The big difference is between the particle system running on a secondary GPU or running on the primary GPU, is that the execution time for the particles has significantly reduced. No matter how many particles were running, the secondary GPU was always faster. Even with the higher loads and data movement required when executing a larger set of particles the secondary GPU was faster in almost every aspect except the data management. When executing a low number of particles the percentage difference was quite significant but the actual difference was not as big, as there was a difference of 0.3 ms. But as the times are so small the percentage difference is enormous.

When executing 1048576 particles on the secondary GPU took an average of 54.4 ms to execute and 29.7 ms to update the data. Meanwhile, on the primary GPU, it took an average time of 408.8 ms to execute the particles and 7.0 ms to update them. This gives the secondary GPU a total of 84.1 ms while to primary GPU to 415.9 ms to execute.

When the particles were executed on the primary GPU the shadow pass took longer to execute when there where more particles in the pipeline with an average execution time of 10.8 ms. This had gone up from an average of 0.7 ms, while on the secondary GPU with a low amount of particles, the shadow pass started at an average execution time of 0.7 ms which is the same as when the particles were executed on the primary GPU. But when it came to a larger number of particles, the shadow pass stayed at 0.7 ms.

As seen in figure 4.3a and 4.3b the particle pass is always the pass that takes the most of the frame time when executing a larger amount of particles. Even when the memory movement times are getting larger, the particle execution time is the part

that takes the longest to execute.

When executing lower amounts of particles as seen in figure 4.1a, 4.1b, 4.2a and 4.2b the secondary GPU is always faster than the primary GPU even when the executing time is very small.

As seen in the table 4.1 the secondary GPU is always faster through the entire range of particles calculated. The secondary GPU is an average 454% faster than the particle system running on the primary GPU.

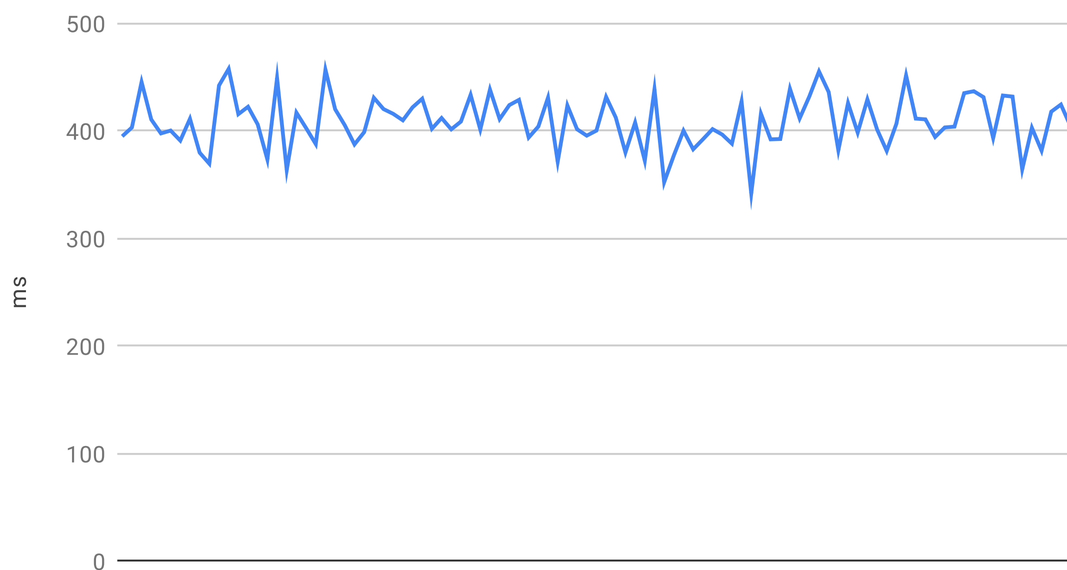
4.2 Particle Amounts

When the number of particles increased the time to execute increased linearly seen in 4.4b, with the secondary GPU being faster to execute over the entire range of particle amounts. But where the secondary GPU flaws are that the data movement times increase steeper for the secondary GPU as seen in figure 4.5a. But this time is quite large for the secondary GPU as it almost doubles the total execution time for the particles, while for the primary GPU this time is small and slowly increases. But even with the data movement times, the secondary GPU is still 494% faster than the primary GPU when doing one million particle calculations.

4.3 Stability

When particles are executed on a secondary the execution time is much more stable. When executing the particles on the secondary GPU the execution time is on average 54.4 ms with a standard deviation of 0.1 ms. Meanwhile, the particles executed on the primary GPU had an average execution time of 408.8 ms with a standard deviation of 23.2 ms. This can be seen in the figure 4.6a and 4.6b where the dual GPU system has a straight line while the single GPU systems line varies a lot. This could be a result of the primary GPU having other rendering techniques to execute and leaving the particles to wait, while the secondary GPU can freely execute the particle calculations.

(a) Particle execution time with a secondary GPU.



(b) Particle execution time without a secondary GPU.

Figure 4.6: Particle execution times over 100 frames with 1 million particles.

Chapter 5

Analysis

When particles that are executed on the secondary GPU is more stable and execute faster than when executed on the primary GPU. This results in overall better frame time.

5.1 Did the performance of the particle system increase?

The performance benefits with running particles on a slower secondary GPU where exceptional. The performance impact on the primary GPU, when execution a significant number of particles where less when the particles were executed on the secondary GPU. This is as a result of the particle calculations being removed from the primary GPU. This allows the primary GPU to focus on one task and not to switch between tasks and this is faster for the GPU. This can be shown on the stability of the particle execution. The particles that were executed on the secondary GPU took almost the same amount of time to execute every time. The secondary GPU had a standard deviation of 0.1 ms. The particles that were executed on the primary GPU where unstable and did not take the same amount of time to execute every frame, this could be a result of the primary GPU having other calculations to do. This limits the time the particles have and results in unstable execution time between frames, this is probably the main reason that executing the particles on the secondary GPU is faster. The particles are moved to a secondary GPU where that GPU can only focus on particles and the primary GPU can have 100% focus on the direct queue instead of the compute queue.

The main performance benefits where the particle execution was faster to execute on the secondary GPU, this was not the expected result as the expected result where for the particles to take longer to execute. But this was not the case, the particle execution time was down to 15% of the original execution time. But the downside of this was that the data movement times were up by an average of 264% compared to the particles being executed on the primary GPU. This was not a problem as the total time when executing the particles on the secondary GPU was 454% faster than when executing on the primary GPU.

The stability of the particle calculation on the secondary GPU is a result of the particles not needing to fight for GPU time. When the particles are executed on the secondary GPU they have the entire GPU for them self and the executing does not have to wait for other queues to finish their work, there is also no switching between

tasks. As the GPU does the same amount of work with the same amount of particles every frame in these and the calculations always take the same amount of time with a variation of 0.1 ms.

5.2 Bottleneck

The bottleneck for both the dual and single GPU system is still the particle calculations. The data movement for the particles on the secondary GPU is not big enough for being a bottleneck in the system. This can be seen in fig 4.3a and 4.3b. The data movement is still a big part of the particles executed on the secondary GPU but the particles on the secondary GPU is still about 500% faster even when the data movement is added to the execution time. Other then the data movement time there was no new time that was introduced to the system when running the secondary GPU at least nothing that was noticed in the measurements.

If the engine was waiting for the particles to finish its execution then the particles pass would still be the bottleneck in the system. This is because the particle calculations take such a long time and then has to be downloaded to the CPU from the GPU. This causes the entire system to wait and not do anything. If over four million particles are run then the data movement time is almost the same as the particle execution time but the time it takes for the particles to execute on the primary GPU goes up four times.

5.3 Compared to the null hypothesis

The null hypothesis for RQ1 would be that there is no difference when running the particles on a secondary or primary GPU. But this is false and there is a statistical significance with a p-value of zero when particle execution time on the two systems are compared. The number of particles does not matter as the p-value is zero through the entire range of tests. This means that executing particles on a secondary GPU is faster than doing the calculations on the primary GPU. None of the parts in the particle system had a p-value that was over 0, but this was not always in favor of calculating the particles on the secondary GPU. As when the particles were executed on the secondary GPU the transfer time or data copy time got significantly worse, while the primary GPU had a data copy time mean of 7.0 ms the secondary GPU had a mean of 29.7 ms. This gave the secondary GPU a worse data copy but this time is made up for as the rest of the system is a lot faster.

5.4 Scaling the System

Everything in the system scales linearly and the performance of the system depends on the power of the hardware. But what's interesting is the frame time scaling seen in fig 4.4a, the average frame time starts in about the same place when there is a small number of particles. Then the single GPU frame time takes a jump at the beginning where none of the systems have to wait for the particles over multiple frames. This makes the difference at the lower number of particles 256 to 4096 so small that there

is no need to implement the particle execution on a secondary GPU as the gains will not outweigh the rewards. But as soon as the particles have to wait for one or more frames to be completed the secondary GPU is much more effective, this can be seen in figure 4.4b where they both start at about the same execution time. But when the number of particles increases the difference between the two systems, also increases. This allows the system with dual GPUs to scale a lot higher and calculated a larger number of particles before the particles rendering time take too long.

5.5 Frame Time & GPU offloading

The frame time when running with the secondary GPU was lower than when running with a single GPU. But as the system is designed where the CPU never waits for the GPU to finish the GPU work, the frame time, in theory, should stay the same, but they did not. This is a result of the particles being offloaded to a secondary GPU and giving more GPU time to the other rendering passes on the primary GPU. This can be seen in figure 4.5b, which shows the time it takes for the rest of the engine to execute. In this graph, they once more start around the same execution time but then when the particles take multiple frames to execute the rest of the engine takes a huge hit on the single GPU system. Meanwhile, the dual GPU system is keeping a steady frame time to about 256k particles, then it starts to rise. If the particles would have been calculated and *rendered* on the secondary GPU then the frame time probably would not rise at all. As the work is done on the secondary GPU does not affect the primary GPU, but the primary GPU still has to render the particles and display them to the screen. The rendering could also be done on the secondary GPU but the primary GPU will always have to present them to the screen, it is also the rendering of the particles that increase the total frame time when the number of particles increases. The increase in the other rendering passes is a result of the particle pass needing a significant part of the GPU resources, this leads to the other rendering passes not having to either wait or execute at a lower capacity. This leads to the higher execution time for the other rendering passes in the engine and this is also why the frame time does not change that much when executing on the secondary GPU.

Chapter 6

Discussion

In summary, it is more effective to execute the particle system on a secondary GPU as it both frees up space on the primary GPU and the particle is faster on the secondary GPU. This also boosts all the other rendering passes and overall resulting in better frame time.

The better frame time is a result of the particles being moved off the primary GPU and being executed faster than when they were executed on the primary GPU. This did free up more time on the primary GPU and allow for the rest of the engine to execute faster then when the particles where on the primary GPU.

6.1 Why should anyone even care?!

As there are so many processors with an integrated GPU and that is not utilized in modern computers, why should modern game engines not use that hardware. This technology could allow for games with better frame time or better graphics. It is the next evolutionary step in game engine development, to use the entire computer to its limits.

The future of computers is also expanding with many threads instead of the speed of the threads, this is a start of what the future could look like. That computational tasks are running on multiple GPUs with lots of threads that execute in parallel. What is the secondary GPU then a bunch of threads that could be utilized for anything, not just for rendering. Any task that could be done on either the CPU or the GPU could be done on a secondary GPU.

6.2 What is the impact of this?

The potential of using a secondary GPU is pretty much endless, but some part of the rendering process fits better to be executed on the secondary GPU. But it could even be used for gameplay features like AI in an RTS game where a lot of AI actors are needed. The secondary GPU might not even be used for the rendering process instead it might only be used for gameplay features. There are many uses for dedicated hardware and now with `DirectX 12` that allows for utilizing that previously unutilized hardware. The secondary GPU could even be used for rendering a part of the screen and share the entire load of the primary GPU.

Anything that the primary GPU could do the secondary GPU can do and it also in the process reliefs the primary GPU of the work. This makes the possibilities

endless for the secondary GPU if only the engine developers will add support for utilizing the secondary GPU.

Using the secondary GPU could help low spec systems like laptops and low-end computers to experience higher graphics or higher frame rates. It is not limited to low en system even high-end systems that already max out the settings could get a smoother experience when workloads are moved over to the secondary GPU. As the secondary GPU could be used for anything there is always a situation where a group of computers will benefit from the secondary GPU and as implementing a dual GPU system in `DirectX 12` is easier than ever.

The dual GPU system could be used for physics calculations if there are many physics object in the scene that needs to interact with each other. This kind of physics calculations can be done on the CPU but does not scale very well when there are a lot of objects. If the CPU is not enough for all of the physics objects then the GPU is usually used, but this is at the cost of rendering time which means that another or multiple primary rendering techniques have to suffer. But with the secondary GPU, there is no other primary rendering technique that has to suffer, as the work is not done on the primary GPU.

Another use for the secondary GPU could be to calculate the visibility of the player from an enemy's perspective on a pixel-perfect level. By rendering the entire scene from the enemy's perspective and counting the number of pixels of the visible player. This could be a feature that is useful in stealth games to give a realistic representation of what sneaking feels like. This is a feature that could be a frame or two late depending on how critical it is for the game.

There are many uses for the secondary GPU but its main utility is to give the developer more flexibility when developing an application or game. The fact that work can be moved from the primary GPU gives the developer a lot more freedom when it comes to utilizing the primary GPU. This could result in games with higher fidelity graphics and or smoother gameplay.

6.3 The Future of Dual GPU Rendering.

The future of dual GPU rendering could be anything, it could be all from most of the game logic running in different threads on a separate GPU, or a part of the screen could be rendered on the secondary GPU.

The future could hold multiple GPUs and just use the CPU to distribute the work between the different GPUs, this is a highly scalable system as all that is needed to scale the system is to add more cores. This relies on developers developing highly parallelizable workloads that could be spread on multiple GPUs.

6.4 Is the cost of implementing a dual GPU adapter system worth when constructing a DirectX 12 engine.

When constructing a new engine, it is worth exploring the possibility of having the engine recognize multiple graphics adapters in the system and offload some of the compute queue over to the secondary GPU. If dual GPU execution is planned from the beginning of the project, it is simpler to design what will happen if there is and is not a secondary GPU that the engine can utilize. There is also a clearer picture of what the secondary GPU could be used for if it exists and how the engine will compensate if there is none.

An example of an object that would need multi-GPU support is a UAV, as the CPU has access to the buffer it cannot be shared between adapters. This means that the developer has the responsibility to transfer the buffer between the graphics adapters if needed. For this to be done there has to support from the engine to the developer to download and upload data between multiple graphics adapters. This is something that can be done with an already complete rendering engine but it is easier when designing an engine from the ground.

6.5 Problems

There is a lot of potential problems with running a dual GPU system, it is an untested area in larger scales. This could lead to performance problems if the secondary GPU is too slow and this is probably the final user should decide if to use the secondary GPU. As there are so many different combinations of GPU's there is no way of testing all of the different combinations, so there needs to be a fail-safe if the secondary GPU executions take too long to execute and do not become a bottleneck in the system. As there is no way of knowing beforehand if the frame time and rendering times are going to improve with every GPU combination there needs to be a way of shutting off the secondary GPU in the settings of the applications and give the responsibility to the end user.

There is also the elephant in the room that all of the users of the system might not have a secondary GPU for particles or other rendering techniques to execute on. There is no use of implementing a dual GPU system if the majority of the users do not have a secondary GPU. But the fact that the performance benefits are so significant the cost of implementing a dual GPU system for the part of the user base that can utilize it might be considered.

6.5.1 Development of the Application

The development of the application will become more complex as the developer needs to decide on the rendering technique should be executed on the secondary GPU or the primary GPU. This would introduce another step in the development and testing of the application, as this step is quite unusual in the current development environments there might be quite a learning curve for what should be executed on the secondary

GPU and how.

6.6 Improvements

There is always a lot of improvements that can be done with the right expertise.

6.6.1 Copy Times

One way to improve the system is to make sure that the data do not pass the CPU. In the current state of the engine, the data have to be downloaded from the secondary GPU to the CPU then uploaded to the primary GPU. This is a slow part of the system, and it becomes a big part of the execution time. If this time could be avoided by maybe having one UAV resource that the secondary GPU can write to and a second shared resource that the UAV can be copied to that then the primary GPU can almost instantly have access to. This could give a performance boost of around 30% just as a result of the decrease in the data movement times.

6.7 Other Possible Uses

Rendering techniques that would fit to run on a secondary GPU are techniques that can be executed asynchronously and is not needed until late in the pipeline or even can be a frame or two late. This would give more time for the secondary GPU to finish the execution and not stall the primary GPU. Stalling the primary GPU is not good because it is the fastest hardware and should always be working at 100%.

6.7.1 Shadow mapping

The reason behind why shadows would be good to do on a secondary GPU is that shadows are made with two rendering passes, in the first rendering pass the depth of every fragment from the perspective of the light is saved to a texture. The second part takes place in the final rendering of a scene where every visible fragment has to be multiplied over to the perspective of the light and then the depth is compared between the current depth and the depth that is sampled from the texture. If the depth of the fragment is greater than the depth in the texture then the fragment if in the shadow and if it less then it is not. The first rendering pass in shadow mapping could be done on the secondary GPU because it can be done asynchronously from other rendering passes.

6.7.2 HUD

Other elements that could be executed on the secondary GPU is perhaps the Heads up display or HUD. If a game has complex HUD elements that take up a lot of time on the primary GPU. They can be executed and rendered on the secondary GPU and later moved over to the primary GPU for presentation. This is also outside the scope of this article. The HUD does not need to be completely synced with the primary GPU, as the HUD could be a frame or two late.

Chapter 7
Conclusions

To answer RQ1 it is much more effective to execute a particle system on a secondary GPU no matter how many particles are executed and the bottleneck in the system is still the particle calculation but the copy time does increase. The increase in the copy time is weighed up by the fact that the particle calculation time only takes a fraction of what it did on the primary GPU. This allows the primary GPU to focus more on the rendering of the scene and not calculating particle movement and the secondary GPU is utilized for asynchronous compute.

But when it comes to the second research question RQ2 that asks what the performance differences in particle system performances are. The main performance boost is the execution time for the particles.

But the secondary GPU could be used for much more the particle calculations, it could be used for rendering a part of the screen to hopefully get a higher frame rate. It could also be used for gameplay features if there is an element of the game that can be highly parallelized, then the secondary GPU could do the calculations.

The only problem with using the secondary GPU is that the engine needs to be designed with dual GPU utilization in mind and if it is not it is hard to implement. Then there is the fact that not everyone does have a secondary GPU that could be utilized, then the developers have double the work and scenarios to code for as there is now a chance that there is a secondary GPU that needs to be utilized. All of this would result in a lot more test cases that need to be considered when debugging and testing an application, and as the hardware needed is immensely big to test for this kind of hardware. There needs to be a generic approach to the rendering engine how it utilizes secondary GPU's as there is a lot of different hardware combinations that could exist in a single system.

7.1 Future Work

Future research would be to answer why the performance difference when utilizing multiple GPU's is so significant.

References

[1] Chris A. Mack. Fifty Years of Moore's Law. IEEE Transactions on Semiconductor Manufacturing. Vol 24. NO 2. 2011

[2] Eugenio Rustico, Giuseppe Bilotta, Alexis Hérault, Ciro Del Negro, and Giovanni Gallo. Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations. IEEE Transactions on Parallel and Distributed Systems, Vol 25, NO 1. 2014

[3] Jin-Chun Piaoa, Jue-Min Lua, Chung-Pyo Hong, Shin-Dug Kim. Lightweight particle-based real-time fluid simulation for mobile environment. Simulation Modelling Practice and Theory. Vol 77. Page: 32 - 48. 2017

[4] J. Nickolls and W. J. Dally. The GPU Computing Era IEEE Micro. Vol 30. NO 2. Page: 56 - 69 2010

[5] Bridges, Robert A. and Imam, Neena and Mintz, Tiffany M. Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods ACM Comput. Surv. Vol 49. NO 3. Page: 41:1 - 41:27 2016

[6] Y. Zhou and Y. Tan GPU-based parallel particle swarm optimization IEEE Congress on Evolutionary Computation. Page: 1493 - 1500 2009

[7] Donev, Aleksandar Asynchronous Event-Driven Particle Algorithms SIMULATION. Vol 85. NO 4. Page: 229 - 242 2009

[8] S. W. Keckler and W. J. Dally and B. Khailany and M. Garland and D. Glasco GPUs and the Future of Parallel Computing IEEE Micro. Vol 31. NO 5. Page: 7 - 17 2011

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden