UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

# Parallel game programming

Jordi Bach Balcells

Director: Cecilio Angulo Bahón

Degree: Videogames

Year: 2021-22

University: CITM

# Índex

# Summary

On this TFG I wanted to find out if parallel programing is and will be used to develop and optimize games. We have this impression because of the fact that nowadays processors improve its power not raising the processing speed but raising the processing units. At the end of the TFG through the development of a windows platform layer we found out that modern low-level APIs give a lot of options to use parallel programming in the form of multithreading of rendering tasks, using SIMD instructions and having the compute pipeline to perform GPUGP tasks. We also found out why the GPU outperforms the CPU at parallel algorithm processing and how we can optimize GPU work focusing mainly on coalesced memory access and memory throughput but also measuring the occupancy and concurrency.

## Key words

Parallelism, Optimization, Synchronization, CUDA, DirectX, GPU, SIMD

## Links

Repository with the programming projects:
https://github.com/bottzo/Parallel_Game_Programming

# Glossary

**CPU:** Central processing unit. The brain of a computer.

**GPU:** Graphics processing unit. It acts as a coprocessor to the CPU helping it with parallel tasks.

**GPUGP:** GPU general purpose programming. Using the GPU for tasks not related to graphics.

**SIMD:** Single instruction multiple data.

# 1. Introduction

## 1.1 Motivation

I have already programmed a couple of game engines but on those I just limited myself to put a bunch of open source libraries together to make them work without optimizing much or knowing how things are working underneath on the platform specific level. Therefore, this TFG is an excuse to learn low-level programing using operating system APIs and general purpose computing on graphics cards (GPUGP) in order to optimize programs on a platform specific level. I think that understanding how things work on such low level will make me a better programmer because I will be able to design better code and spot possible platform specific optimizations even when working with higher level languages.

Also I think that it is important to learn parallel programing because the new hardware every time dedicates more transistors to make more processing units without increasing the instruction processing speed so single thread programs will not be able to take advantage of this new hardware and improve much in performance.

## 1.2 Formulating the problem

Nowadays, new CPUs and GPUs improve its power not raising the instruction processing speed (raising clock frequency) but raising the number of processing units. Therefore, to take advantage of the new hardware in order to improve performance we need to use a multithreaded program. Nevertheless, multithreaded programing adds some extra complexity because threads of the same program share address space and resources like memory. In consequence, we need to synchronize and manage access to memory to avoid problems like race conditions where two or more threads are reading from one memory location while at the same time other threads are writing to the same memory location. Additionally, this synchronization needs to be implemented correctly to avoid more problems like deadlocks where a program can't continue execution because all threads are blocked waiting one another. Furthermore, another problem we can have with bad synchronization is to actually make the program slower than it originally would be with a single thread because of overhead on thread code generation, a lot of context switches between the threads, blocked threads with bad synchronization… This extra complexity generated when troubleshooting these problems makes the program development slower and more difficult to program, debug and profile. Accordingly, I want to see if multithreading is enforced by modern APIs as a possible optimization and how.

Moreover, threads are not the only way to parallelize a program. CPUs have specific instructions extensions that contain specific instructions targeting specific tasks. One of these instruction sets are the single instruction multiple data(SIMD) vector instructions that enable to perform the same instruction like for example an addition to multiple data at the same time effectively augmenting the throughput of the CPU.

For example, we can perform 8 addition instructions on 16 memory locations or we can issue just one instruction that does this same addition but with 2 arrays of 8 entries. Using these instructions can improve the performance of a program greatly however a lot of times they are not used because a program using this instruction sets is not portable to those CPUs that don't support them like the old models or CPUs from other brands that supports them but use a different instruction set architecture (ISA). Consequently, we need to use runtime function dispatching to execute the functions using these instructions depending if the hardware supports them and if it does not, execute the version that doesn't use this instruction sets. This instruction sets are also sometimes avoided because they are not supported on a programming language level, you need to use compiler intrinsics or assembly language to access them. With a program that uses this instruction sets people will be able to enjoy the extra performance that their new hardware supports.

Talking about SIMD, most pcs have a coprocessor alongside the CPU designed with this idea in mind called the graphics processing unit (GPU). The main idea of the GPU design that differentiates it with the CPU is to maximize the instruction throughput instead of focusing on instruction latency. On one hand, CPUs dedicates a lot of transistors on its die with caches, branch predictor, out of order execution etc. trying to execute sequences of instructions as fast as possible, leaving them without a lot of space on the die for a lot of execution units(cores). On the other hand, GPUs have a lot of cores to execute a lot of instructions at the same time and a higher memory bandwidth but they don't have this extra tools and high frequency the CPU has to execute a sequence of instructions as fast as possible. In consequence, GPUs focus on completing the highest number of instructions in a given time excelling at parallel tasks that will take a lot more time to the CPU however, they struggle if the task to perform is not parallelizable. This parallelism makes the GPU very suitable and used for graphics. Although its name (GraphicsPU), it can also be used for other tasks that would greatly benefit from the GPU parallel design against the CPU.

In conclusion, I am trying to implement a simple platform layer that could be the base of a game or a game engine using the most modern APIs provided by the platform to see if it enforces the use of any of the mentioned parallel programming techniques. If they are used, we are going to see their exact implementation and it will mean that the industry also considers parallel programming to optimize in games.

## 1.3 TFG general objectives

Learn exactly how better the CPU is than the GPU performing parallel tasks and why.

Crate a base Windows platform layer for game development using the latest APIs to see if the different parallel techniques mentioned above can be used to optimize modern games. If they can be used also see how they are used.

## 1.4 TFG specific objectives

Use the CUDA platform layer to learn about GPUGP.

Profile how significant the difference is between a parallel algorithm running on the CPU and on the GPU.

Optimize a GPU algorithm to fully understand the GPU hardware.

Use the latest versions of the DirectX API to construct my windows platform layers.

See how DirectX12 implement GPUGP.

See what exactly makes DirectX12 so hyped to get better performance than other older APIs like Opengl.

See if we find an unknown parallel technique while programing the platform layer.

## 1.5 Project Scope

This TFG targets anyone interested in the topic of low level C/C++ game and engine development and platform specific optimization targeting mainly pc games on the windows operating system. It also targets people interested in learning GPU programing.

For the platform layer, we will try to make it as much complete as possible in order to see more areas of the game where parallel techniques can be used. At least it has to be complete enough to render geometry to the screen, use input, textures and audio Those are the features the majority of games use and that configure the core of the DirectX multimedia libraries that I want to test.

For the GPU, the most important thing is to create kernels on CUDA, optimize them and compare them to the CPU times. The number of kernels is not important, what is important is to try to touch the majority of areas of GPU hardware. It will also be interesting to see how different the CUDA kernels are compared to the compute shaders of directX 12.

Finally, I will try to see if any of the APIs uses SIMD instructions.

# 2. State of the art

I have not programmed a Windows platform layer yet. Until now I have been using helper libraries like SDL or GLFW that worked as platform layers. The APIs of those libraries are not specifically designed for a multithreading environment as most of the API is not thread safe. Those helper libraries also use OpenGl as the rendering API. OpenGl works as a state machine. The problem with it is that it is very opaque and tied to a context on a specific thread.

CUDA is the state of the art platform for Nvidia GPU programing. As it is designed by Nvidia themselves, its compilation achieves the best performance possible on Nvidia GPUs. Its API and function intrinsic also offers the possibility to play with all their different GPU platforms. It is designed as an extension to the C++ language. Nvidia also offers the Nsight toolset, very good tools to profile and debug not just CUDA kernels but also any program running on the GPU.

## 2.1 Market study

The two main game engines on the market are Unity and Unreal engine.

On one hand, Unity lets you use C# threads API on scripts, however, it is very limited because you can't use unity API engine functions or resources outside the main thread. Consequently, you are very limited on what you can do in other threads. Despite the poor multithreading, unity has good asynchronous options like corutines but they are just a mean to organize the code not really getting much parallelism. Unity is not the engine you look for to optimize code anyways because it uses managed code and its scripting language is interpreted. It is more focused on getting something done without having to worry much of how it is implemented.

On the other hand, Unreal lets you write C++ scripts which enable the threading capabilities of the language. I'm not familiar enough with unreal to know how they manage threads and their limitations.

Having said so, the two and any engine I find as reference is already built on top of a platform layer which is not directly accessible through the engine. What engines do is abstract all the platform layer onto simple routines that are used to build the different games.

# 3. Project Management

## 3.1 Procedure and tools to follow the project

### 3.1.2 GitHub repository, Git version control system

We will use git as version control System to save the different versions of engine and tests. We will also use github to publish the git repo:

https://github.com/bottzo/Parallel_Game_Programming

## 3.2 Validation tools

For this project it is very important the benchmark of performance to see if we are optimizing with the different parallel techniques. To profile the CPU, we will use the win32 API functions QueryPeformanceFrequency and QueryPerformanceCounter.  T

## 3.3. DAFO

| | **Positive** | **Negative** |
|---|---|---|
| **Internal Origin** | **Strengths**<br>-Having an engine is the best tool to test how suitable is to develop parallelism on a game.<br>-Developing another engine after having already done a couple more.<br>-With an engine we can always keep adding modules or scripts to test the different parallel techniques on a game. | **Weaknesses**<br>-Lack of time to develop a very complete engine, we will have to conform with something basic to test if the techniques appear. |

| | Oportunities | Threads |
|---|---|---|
| **External Origin** | -Learning low level and platform specific optimizations.<br><br>-Learning parallel programing.<br><br>-Learning directX and win32 APIs (Windows OS APIs and libraries for game dev).<br><br>-Learning how the GPU works and general purpose GPU programming. | -Development of the basic engine takes too much time so we can't develop and test well the parallel techniques.<br><br>-Big trouble for the project if the scheduler of the engine doesn't work well or gets complicated.<br><br>-Can't start testing techniques until we have some basic engine. |

## 3.4. Risks and contingency plan

| Threads | Solutions |
|---|---|
| Get stuck developing something on the platform layer. | We can always cut platform layer functionality if it does not lead to any parallelization technique. |
| | |
| | |
| | |
| | |
| | |

# 4. Metodology

**Descripció pas a pas**, les diferents fases i sub-fases del projecte.

To develop the project, we will use an agile methodology with 5 sprints. Each sprint will be of 2 weeks. At the end of this two weeks I will evaluate if we meet the objectives of the week and If we haven't, reschedule everything accordingly.

As mentioned above in the objectives, this TFG looks for parallel programing techniques while creating a windows platform layer. The development of the platform layer will be done at the same time as the CUDA GPU programming in order to have enough GPUGP knowledge when starting to use DirectX12 API.

# 5. Project Development

Now I will explain the different actions I carried during the development of this TFG.

**GPUGP**

As mentioned in 1.2. (Formulacio del problema) GPUs and CPUs are very different because they are designed with different goals in mind (throughput vs. latency). As they are so different, the first task I carried was learning how a GPU works and how to program it. With that goal in mind I decided that the best way to learn it was to make very simple GPU programs using CUDA. CUDA is a C/C++ language extension that allows you to define and execute functions that will run on an nvidia GPU (called kernels). It also provides helper functions to manipulate GPU memory and transferring data between GPU (device) and CPU (host).

The first program I made was the addition of two big vectors.

```
void addCPU(int* c, const int* a, const int* b, int elementsToAdd)
{
    for(int i = 0; i < elementsToAdd; ++i )
    {
        c[i] = a[i] + b[i];
    }
}
```

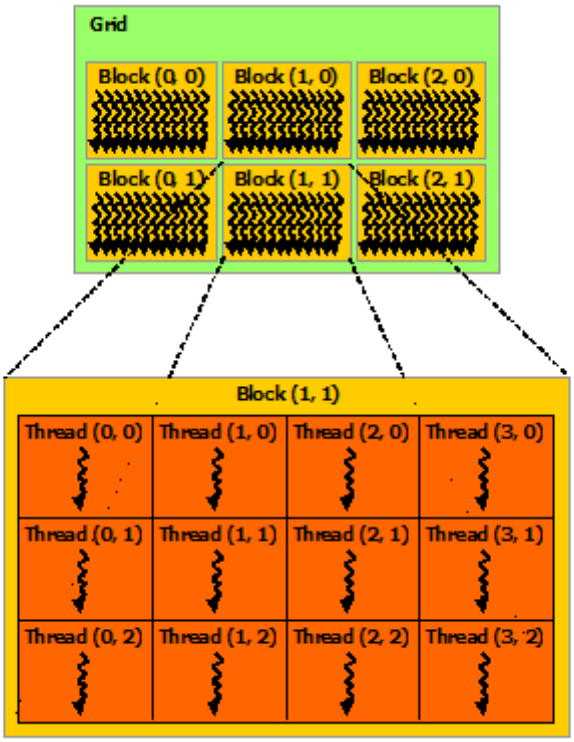
In a CPU this would be done with a for loop incrementing the index on each iteration and with it accessing an array element by element and performing the addition.

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

This is the kernel (marked as a kernel (GPU function) using the __global__ definition) that makes the addition on the GPU. Notice that we perform the same addition operation that we did on the CPU function but we get rid of the for loop because the entire function will run multiple times in different GPU threads. As the GPU has many more cores than the CPU, all this threads can execute the same instruction on different cores at the same time. This model the GPU uses comes from the SIMD (single instruction multiple data) and is called SIMT (single instruction multiple thread). To run a kernel, the GPU just spawns a lot of threads on different cores and each thread runs the first instruction of the function at the same time, and after the first one the next one and the next one … until the function ends. This way, maybe we are slower to perform each instruction (latency) but when we finish performing it, it is like having performed a lot of instructions (the same instruction performed a lot of times in every different thread) achieving an amazing throughput and parallelism. This is great for certain types of tasks called embarrassingly parallel like this vector addition that

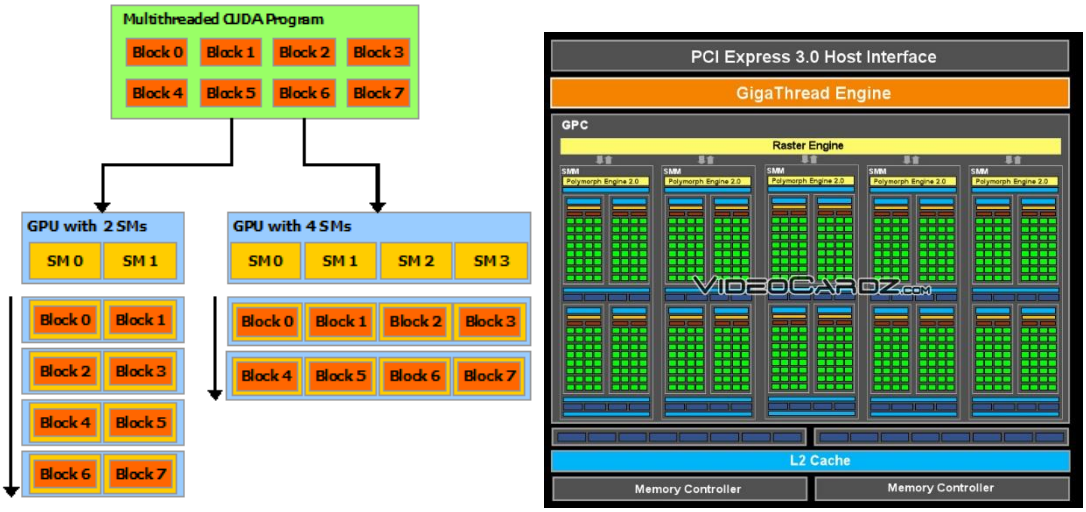
perform the same instruction in different data. Notice on the example kernel that we get the index to the array through a magical variable called threadIdx. This threadIdx variable will automatically get set by the runtime to the index of the current thread running this kernel.

After the vector addition test I did another test with matrix multiplication. In this test I got introduced to the concept of grids, blocks and thread dimensions.




A block is a group of threads and the unit that gets scheduled on the GPU. All the blocks in a kernel form a grid and can be accessed using the blockIdx variable. As the threadIdx variable, it is a 3 dimension variable (x,y,z) because it can be convenient for certain tasks like for example the matrix multiplication to access the data using multiple dimensions.


As mentioned before, the GPU has a lot of cores. All this cores are grouped in different clusters called streaming multiprocessors (SM). Each SM contains cores, a register file with the registers to use for the different cores and cache. To run a certain kernel, the application specifies a grid to the GPU. The GPU then assigns the different blocks to different SM and each block runs until completion. After a block finishes, a SM is freed and another block of threads can get executed on that SM. This way, the more SMs a GPU has the more blocks can run at the same time in parallel and the faster the kernel will complete.




Because of its design, GPUs theoretically can carry parallel tasks a lot faster than CPUs. To see if this is true I also created two programs that do exactly the same task and use the same algorithm but one with the CPU and the other with the GPU. Like this, I am

able to benchmark both and compare how significant the difference in performance is. The program is about image processing. It consists on a simple console app that receives an image and outputs the same image but in black and white. The task is embarrassingly parallel because it fits with the SIMD model commented before as we perform the same operation on each texel of the texture. Consequently, I expect it to run much faster on the GPU than on the CPU.

```
for (unsigned char* p = img, *pg = gray_img; p != img + img_size; p+= channels, pg += gray_channels)
{
    //greyPixels[index] = 0.299f * colorPixels[index].x + 0.587f * colorPixels[index].y + 0.114f * colorPixels[index].z;
    *pg = (unsigned char)((*p) * 0.299f + (*(p + 1)) * 0.587f + (*(p + 2)) * 0.114f);
}
```

```
__global__ void RGBToGreyScale(const uchar3* const colorPixels, unsigned char* greyPixels, int rowElements, int columnElements)
{
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    if (y < columnElements && x < rowElements)
    {
        int index = y * rowElements + x;
        greyPixels[index] = 0.299f * colorPixels[index].x + 0.587f * colorPixels[index].y + 0.114f * colorPixels[index].z;
    }
}
```

To time both functions I used the QueryPerformanceCounter() and QueryPerformanceFrequency() from the Windows API. I also used Nvidia Nsight Compute to get a more accurate time for the GPU kernel. The test was run on an intel i7-10700k CPU and a Nvidia RTX 2080 Super GPU and I used a (3840x2160)4k and (7680 × 4320)8k resolution textures.

As expected, the GPU was A LOT faster than the CPU. For the 4k texture test, while the CPU took 13 ms on average the GPU only needed 0.3 ms on average to complete the same task (0.12915ms on Nsight compute). Continuing with the 8k texture test, the difference was even bigger. The CPU required 53 ms on average to complete the task while the GPU just needed 0.62 ms to complete it (0.50531ms on nsight compute).

A game has to run at high frames rates if it wants the user to not perceive the visual lag between the frames and also to maintain it responsive to user input. The old

standard to make a game playable is 30 fps. Going below that can be noticeable and frustrating to most players. If we want to target that bare minimum of 30 fps we have 33,33 ms to compute each frame. Otherwise, if we target a more acceptable frame rate nowadays like 60 fps we need to compute each frame in 16,66 ms. If for example we want to use a black and white filter when the player is low on health we could use our algorithm on a post processing effect, once the frame has been computed, to turn the output texture to black and white. On one hand, if that output texture is 4k, it would take 13 ms to compute that algorithm on the CPU. Consequently, half of the frame time at 30 fps and nearly all the time at 60 fps would be spent computing the task. Moreover, if this output texture is 8k it takes 53 ms, making the task unachievable even at 30 fps because it nearly doubles the 33,33 ms mark. On the other hand, with the GPU times, the task becomes totally feasible at 4k and also at 8k even at higher frame rates.

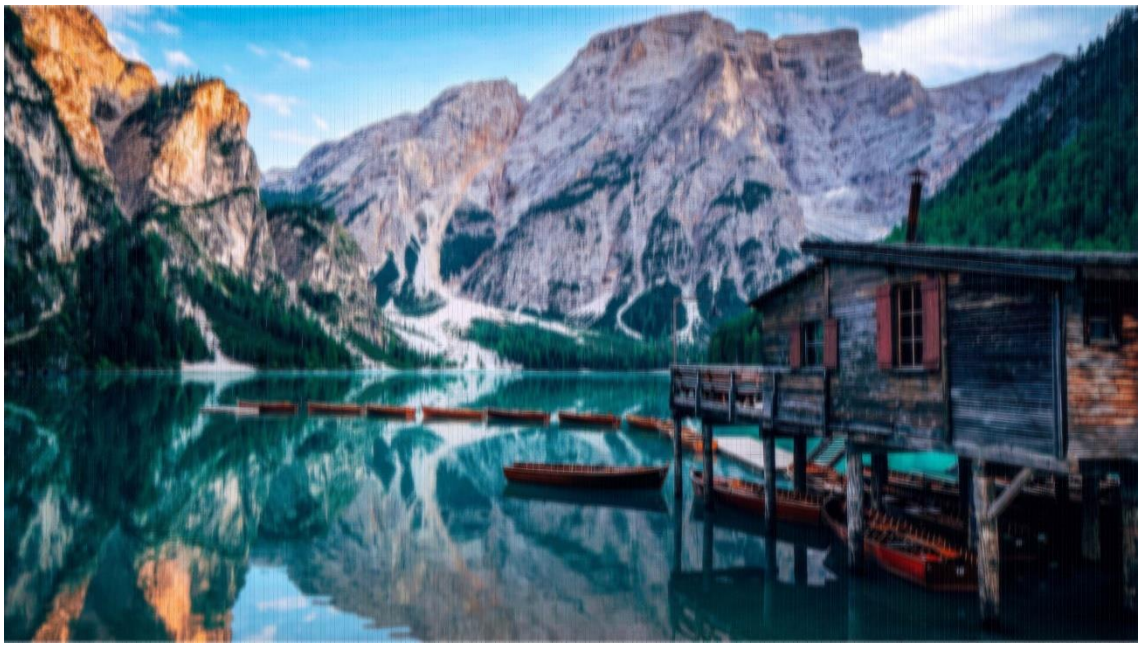
The last test I did with CUDA was to try out the shared memory. Each streaming multiprocessor (SM) has shared memory, which is cache memory close to the cores but that can be accessed by the programmer. Contrary to the CPU, here we can take a portion of the cache and declare it as shared memory. It is very useful to avoid a lot of reads to slow main GPU memory, however, all the data we load on the shared memory can only be shared between the threads of the same block because blocks are what get scheduled to run on the different SMs, each one with its cache. In this test I performed a Gaussian blur on an image. To perform a Gaussian blur each texture texel final color is the average of the sum of all the surrounding texels.

```
__global__ void ChannelUnweightedImageBlurQuad(const unsigned char* const colorPixels, unsigned char* bluredPixels, int rowElements, int columnElements, int quadDim)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int gidx = y * rowElements + x;
    int cacheX = x - (blockIdx.x * blockDim.y) + quadDim;
    int cacheY = y - (blockIdx.y * blockDim.y) + quadDim;
    int cacheDimX = (blockDim.x + quadDim*2);
    int cacheIdx = cacheY * cacheDimX + cacheX;
    //extern __shared__ to allocate the shared memory size at lunch time with the parameters of grid and block sizes <<<grid size,block size,shared bytes>>> kernel(...)
    extern __shared__ unsigned char cache[];
    cache[cacheIdx] = colorPixels[gidx];
    if (threadIdx.y == 0)
    {
        //if (blockIdx.y != 0)
            for(int i = 1; i <= quadDim; ++i)
                if((gidx - rowElements * i) >= 0)
                    cache[cacheIdx - cacheDimX* i] = colorPixels[gidx - rowElements * i];
                else
                    cache[cacheIdx - cacheDimX* i] = 255;
    }
    else if (threadIdx.y == (blockDim.y - 1))
    {
        //if (blockIdx.y != 0)
            for (int i = 1; i <= quadDim; ++i)
                if((gidx + rowElements * i) < (rowElements*columnElements))
                    cache[cacheIdx + cacheDimX * i] = colorPixels[gidx + rowElements * i];
                else
                    cache[cacheIdx + cacheDimX * i] = 255;
    }

    if (threadIdx.x == 0)
    {
        //if (blockIdx.x != 0)
            for (int i = 1; i <= quadDim; ++i)
                if((gidx - i) >= 0)
                    cache[cacheIdx - i] = colorPixels[gidx - i];
                else
                    cache[cacheIdx - i] = 255;
    }
    else if (threadIdx.x == (blockDim.x - 1))
    {
        //if (blockIdx.x != 0)
            for (int i = 1; i <= quadDim; ++i)
                if((gidx + i) < (rowElements * columnElements))
                    cache[cacheIdx + i] = colorPixels[gidx + i];
                else
                    cache[cacheIdx + i] = 255;
    }
    __syncthreads();
    if (y < columnElements && x < rowElements)
    {
        unsigned int numSums = 0;
        unsigned int sum = 0;
        for (int i = 1; i <= quadDim; ++i, numSums += 8)
        {
            sum += (cache[cacheIdx - i]  /* + cache[cacheIdx]*/ + cache[cacheIdx + i] +
                cache[cacheIdx - cacheDimX*i - 1] + cache[cacheIdx - cacheDimX*i] + cache[cacheIdx - cacheDimX*i + 1] +
                cache[cacheIdx + cacheDimX*i - 1] + cache[cacheIdx + cacheDimX*i] + cache[cacheIdx + cacheDimX*i + 1]);
        }
        bluredPixels[gidx] = (1.f / (float)numSums) * sum;
    }
}
```

To perform the final additions and average we do a lot of memory reads so if the memory we are reading is shared memory (cache memory close to the cores) instead of the main GPU memory the reads will be performed a lot faster. Moreover, to load the shared memory each thread of the block loads one texel color to the cache and the corner threads also load the extra surrounding pixels outside of the block.

Before I mentioned that all the threads run at the same time. That is not entirely true. Each block when executing inside a SM divides itself into groups of 32 threads called warps. This warps are the units that get scheduled in each SM and execute each instruction together at the same time. The execution context (program counters, registers, etc.) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. Therefore, contrary to CPUs context switching, switching from one execution context to another, has no cost and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads. This instruction can go to all the threads of the warp if they haven't diverged. For example, if one thread takes an if branch that the others don't take, the instructions of that branch can only be executed on that thread and the rest of threads of the warp are not activated when this instruction is issued for the warp. To achieve the best results, try not to make divergent code with branches or data dependencies because you will be losing the

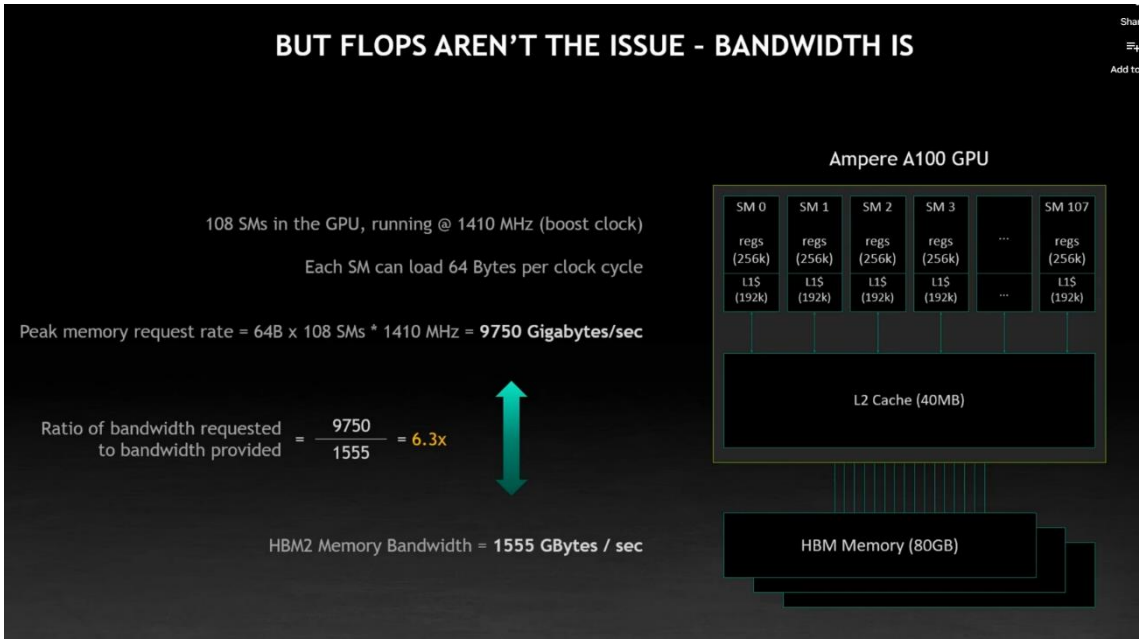
parallelism of 1 instruction for the hole 32 threads at the same time. As some threads in a warp can be executing later kernel instructions than other threads in another warp we might need some synchronization. For example, in the blur test, different threads first load the data to the shared memory cache. After that, the threads start making the average computation reading the data from the cache. If a thread has finished loading its portion of the data to the cache and then starts doing the averages but all the cache still is not filled by the other threads this thread can crush or get bad data trying to access uninitialized data. That is why it exists the intrinsic function __syncthreads() that appears on the blur test. It acts as a barrier that doesn't allow any thread on a block to continue further until all the threads of that block reach that point.



The next step I did was optimizing the kernels I just used. I wanted to optimize them in order to get more insight on GPU hardware. Moreover, although the difference in time between GPU and CPU have already been proven, I was curious if the algorithms I just wrote where actually very slow and not representative of the GPU time when comparing them to the CPU. The possible impactful optimizations on GPU kernels mainly come from 3 things.

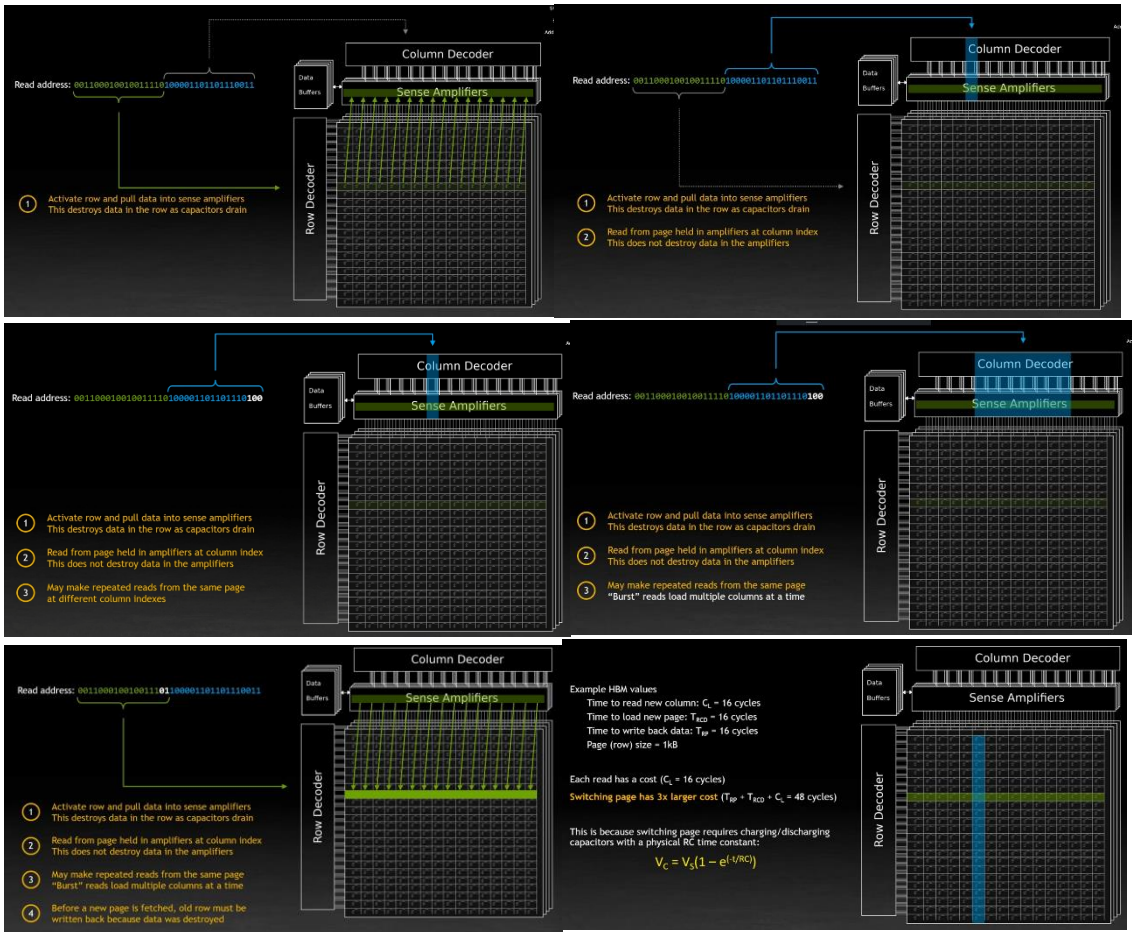
***Memory layout and access patterns***

In order to optimize memory bandwidth. Memory is the highest limiter on performance because the max bandwidth we can achieve is still far from the computational power of all the cores.

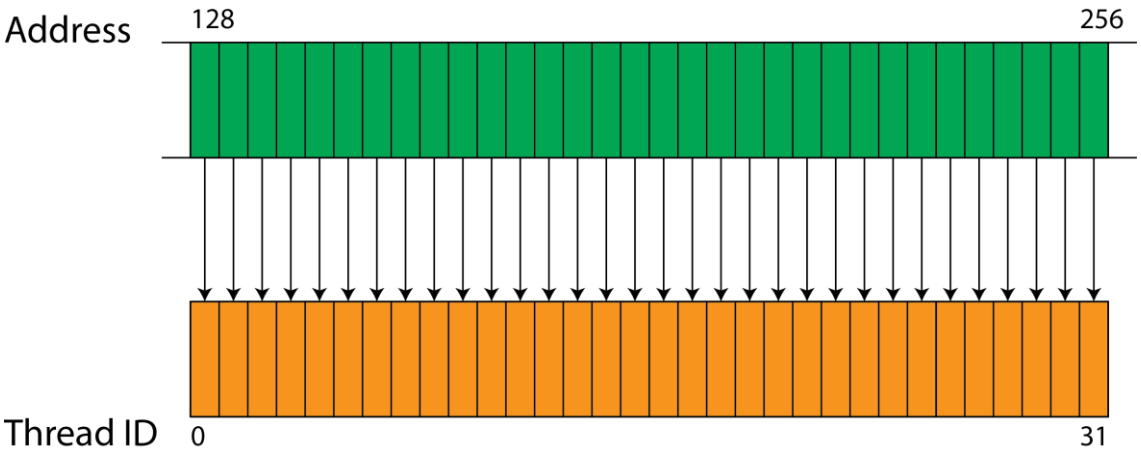



On the image we can see that the peak bandwidth needed to support all the requests to fulfill the computational specs of the GPU is 6 times the actual memory bandwidth of the device. This means that the device has a bottleneck on memory bandwidth, making it the most important spec to target optimizations in order to reach peak device performance.

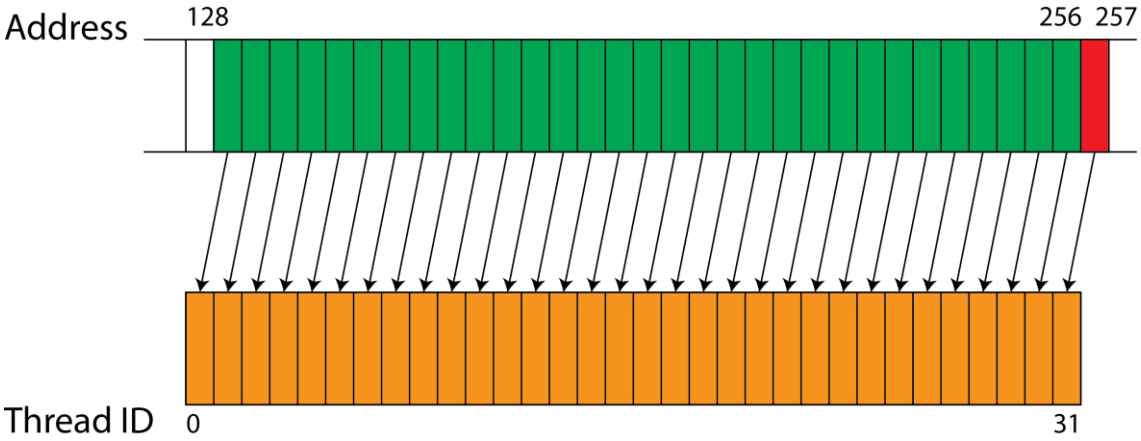
Our objective is to access adjacent memory locations as much as possible to get the peak memory bandwidth. How device global memory works physically is that you can load rows of x bytes (different depending on the device). After loading the row, we can access the values (or columns) of that row and after we write the row back to memory because the row was destroyed when loaded from capacitors for reading. The 10 times speedup comes from the difference in data read/write with just reading/writing 1 value of each row load or reading all the adjacent values(columns) for each road load. Have in mind that switching a row of memory for reading costs 3 times the time to read a column because we have to load the road(page), then read the column(s) and when finished with the operation we have to write the road back.



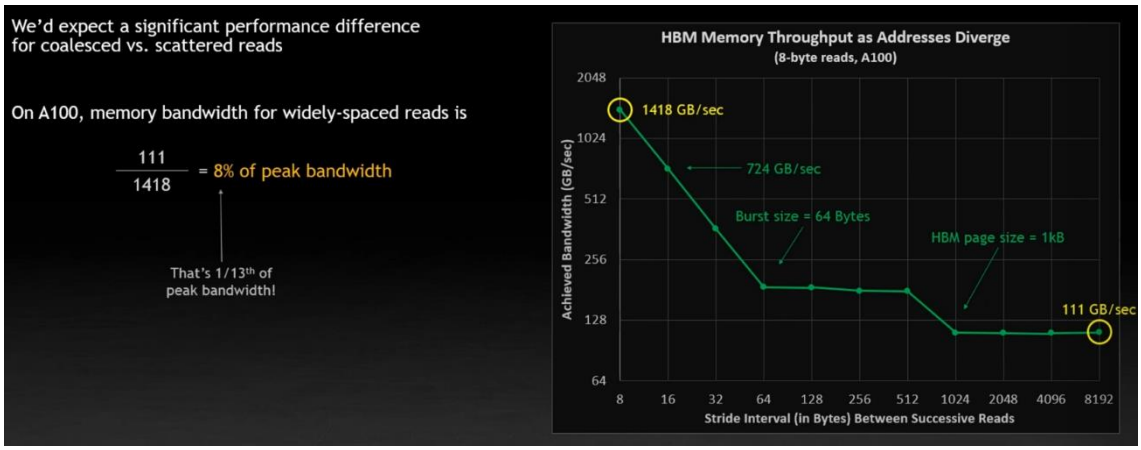

Coalesced memory access: on a warp access global memory instruction (ldg), each of the 32 threads accesses 4 bytes of a memory chunck. In my nvidia RTX 2080 super GPU, each row(page) we read has a size of 128 bytes (that is also the cache line size). This means that if the memory is correctly aligned, we can pack all the 32 reads of the warp into a single memory instruction and get the best possible performance accessing adjacent memory.



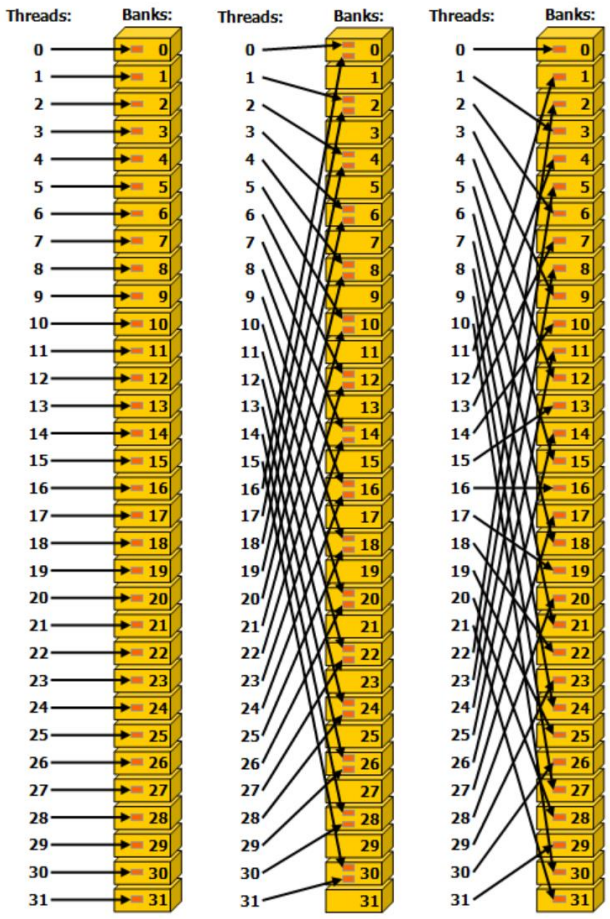

If the memory is not correctly aligned we would need 2 memory read operations. Like this we would read 256 bytes when we just need 128 bytes. It is not that bad if the second 128 byte memory block gets cached and then read by another warp or this one later.




Strided memory access: Each thread of a warp accesses data separated from the next thread data exactly an offset called stride. A typical strided access has an index like [threadIdx.x * stride]. It is not as efficient as the coalesced memory access because we normally will need more than 1 load operation and all the data between the strides will not be used. It gets worse as the stride gets higher because we will need more read operations (read more rows of main memory). Additionally, we will use less of the data loaded as the strides got bigger.




This strided impact can get minimized with caching but it can also generate another not so severe problem when accessing cache memory. Cache memory has lower latency than main device memory because it is closer to the cores. It also has a higher bandwidth because it is divided into equally-sized memory modules called banks, which can be accessed simultaneously. Accordingly, any memory request made on addresses that fall on distinct memory banks can be serviced simultaneously yielding an overall bandwidth that is n times as high as the bandwidth of a single bank. However, if two addresses of a memory request fall in the same memory bank (and are not the same address) the access has to be serialized into different requests, decreasing the throughput by a factor equal to the separated memory requests. This is called a bank conflict. Strided accesses to cache memory can generate bank conflicts.

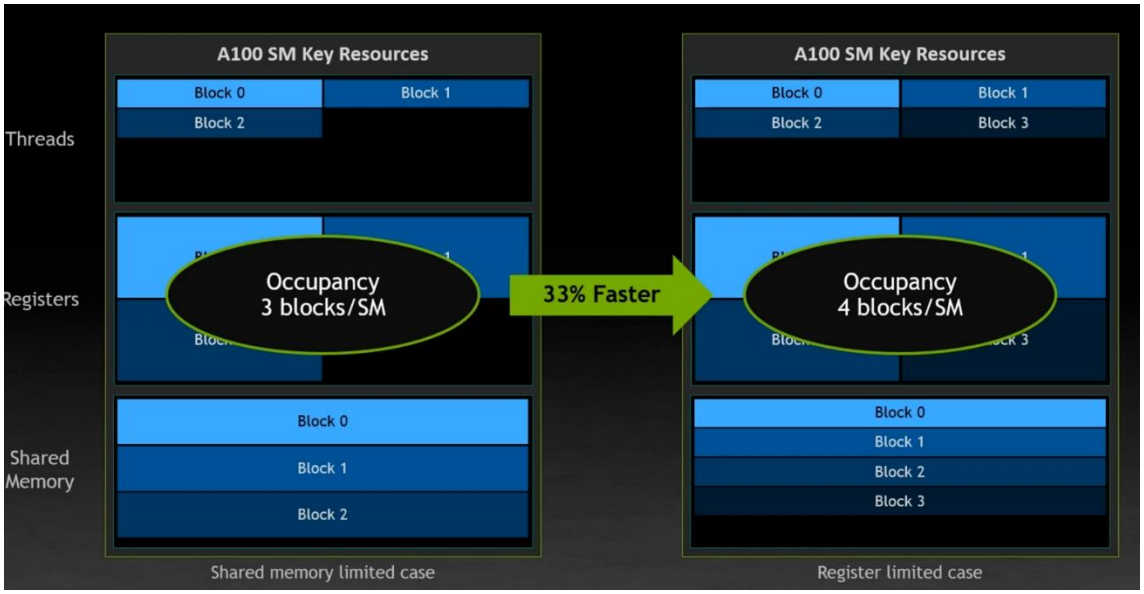



As we can see in the image the first coalesced access does not generate bank conflicts and can be serviced all in one wavefront (number of actual memory operations to satisfy one memory request). However, the second strided access with a stride of two generates bank conflicts and the request cannot be satisfied with one wavefront.

Random memory access:  TODO: cal???

***Occupancy***

How much stuff I can pack into an SM. Each SM has limits in the resources it can hold. This limits determine the number of blocks that can run concurrently on the SM. The main limiters are the max number of threads per SM, the amount of registers in the SM and the total shared memory in the SM. If for example I can feed 4 blocks into 1 SM instead of 3 we get 33% more active threads per SM. Therefore, we can potentially increase performance by 33%.




Occupancy is very important because it is used to hide instruction latencies on the GPU. Each time the scheduler of an SM has to issue an instruction it has to choose one ready instruction from one of the warps of the blocks present on that SM. If for example, we just issued a ldg (load from global memory) to a warp it is executing on the load store cores of the SM and say it will take 4 cycles to complete. The next 4 cycles this instruction is executing, if the GPU has more instructions from other warps issued on cores that are not the load store core we can issue those. This way, we can keep issuing instructions each cycle without having to wait the previous ones to complete. That is why occupancy is very important to have more available instructions to select at issue time and hide the latency of those instructions. It is also very important to keep all the SMs always with blocks because each SM has its own bandwidth, therefore, if we utilize all the SMs we can get the maximum bandwidth for the device. As we have seen before, memory bandwidth is the most precious resource. Consequently, the hardware always first spreads the blocks between the SMs to keep them all active and then fills them with the remaining blocks to hide instruction latencies.

### *Concurrency*

If we tell the hardware the dependencies between the blocks running it is capable to fit more blocks per SM. For example, if we have blocks from 1 grid of a kernel that require shared memory, while filling the SM with those blocks we could be limited by the amount of shared memory in the SM but we still could have space for the number of threads and registers. Consequently, we can fit in blocks from other kernel grids that don't require shared memory. Notice that blocks from the same kernel grid will not fit because they are all the same so they all require shared memory. To be able to perform this concurrency the hardware needs to know the dependencies between the different kernel grids. We can divide a group of kernels in CUDA in different streams of work. This process is called oversubscription.

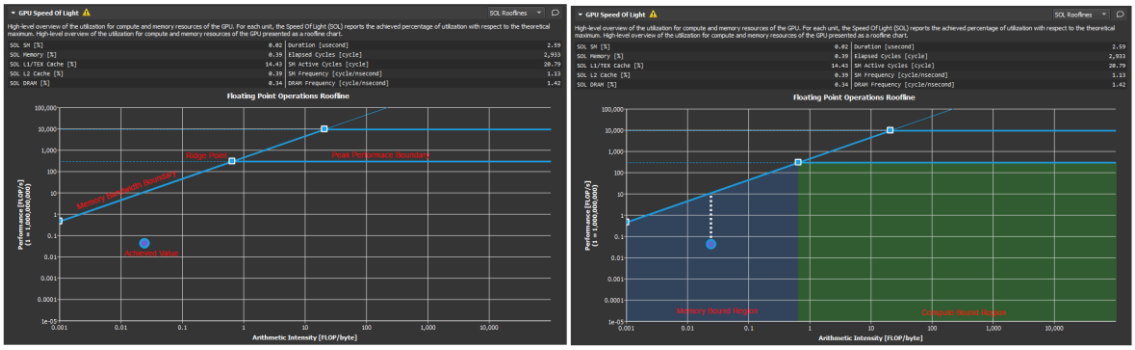
TODO: Link foto: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41487/?playlistId=playList-b07d4d41-377a-4337-bb50-5831cba772fc

The optimization was done on my Nvidia RTX 2080 Super GPU. Compute capability 7.5 which corresponds to the Turing architecture.

> K.6. Compute Capability 7.x
>
> K.6.1. Architecture
>
> An SM consists of:
>
> - 64 FP32 cores for single-precision arithmetic operations,
> - 32 FP64 cores for double-precision arithmetic operations, 34
> - 64 INT32 cores for integer math,
> - 8 mixed-precision Tensor Cores for deep learning matrix arithmetic
> - 16 special function units for single-precision floating-point transcendental functions,
> - 4 warp schedulers.
>
> An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.
>
> An SM has:
>
> - a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
> - a unified data cache and shared memory with a total size of 128 KB (*Volta*) or 96 KB (*Turing*).
>
> Shared memory is partitioned out of unified data cache, and can be configured to various sizes (See Shared Memory.) The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in Texture and Surface Memory.


In order to profile the CUDA kernels I used NVIDIAs Nsight Compute. The program gives us a lot of information about the execution of the different kernels and the hardware they ran on alongside a lot of tips to optimize them based on this info.

Roofline Analysis:




- ***Vertical Axis*** - The vertical axis represents *Floating Point Operations per Second* (FLOPS). For GPUs this number can get quite large and so the numbers on this axis can be scaled for easier reading (as shown here). In order to better accommodate the range, this axis is rendered using a logarithmic scale.
- ***Horizontal Axis*** - The horizontal axis represents *Arithmetic Intensity*, which is the ratio between *Work* (expressed in floating point operations per second), and *Memory Traffic* (expressed in bytes per second). The resulting unit is in floating point operations per byte. This axis is also shown using a logarithmic scale.
- ***Memory Bandwidth Boundary*** - The memory bandwidth boundary is the *sloped* part of the roofline. By default, this slope is determined entirely by the memory transfer rate of the GPU but can be customized inside the *SpeedOfLight_RooflineChart.section* file if desired.
- ***Peak Performance Boundary*** - The peak performance boundary is the *flat* part of the roofline By default, this value is determined entirely by the peak performance of the GPU but can be customized inside the *SpeedOfLight_RooflineChart.section* file if desired.
- ***Ridge Point*** - The ridge point is the point at which the memory bandwidth boundary meets the peak performance boundary. This point is a useful reference when analyzing kernel performance.
- ***Achieved Value*** - The achieved value represents the performance of the profiled kernel. If baselines are being used, the roofline chart will also contain an achieved value for each baseline. The outline color of the plotted achieved value point can be used to determine from which baseline the point came.

The arithmetic intensity in the x axis is the ratio of compute in FLOP/s divided by the bytes read from memory. The two peak performance boundaries are for double precision floating point (the low one) and for single precision floating point (the high one).

As we said before, the most important targets when optimizing is memory and occupancy. Nsight Compute offers memory charts and memory tables and also an occupancy calculator to help optimize the kernel.

Documentation: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

First I started by optimizing the black and white kernel. The original kernel I had before optimizing, the one shown up, ran in 129.15 microseconds for the 4k image and 505.31 microseconds for the 8k image (1ms = 1000us // 1s = 1000000us).

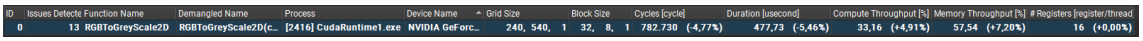
```
__global__ void RGBToGreyScale(const uchar3* const colorPixels, unsigned char* greyPixels, int rowElements, int columnElements)
{
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    if (y < columnElements && x < rowElements)
    {
        int index = y * rowElements + x;
        greyPixels[index] = 0.299f * colorPixels[index].x + 0.587f * colorPixels[index].y + 0.114f * colorPixels[index].z;
    }
}
```

The first thing Nsight compute warns us is that we have uncoalesced global memory accesses and that we don't reach the 100% theoretical occupancy. Additionally, we see that the compute and memory throughputs are at 32% and 54%. Accordingly, we have room for optimization on the kernel. As mentioned before, the first focus for optimization is memory. Therefore, the first thing we have to find out is the possibility of a coalesced global memory access.

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device Name | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/thread] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 14 | RGBToGreyScale2D | RGBToGreyScale2D... | [26760] CudaRuntime1.exe | NVIDIA GeF... | 480, 270, 1 | 16, 16, 1 | 821.939 | 505,31 | 31,61 | 53,68 | 16 |

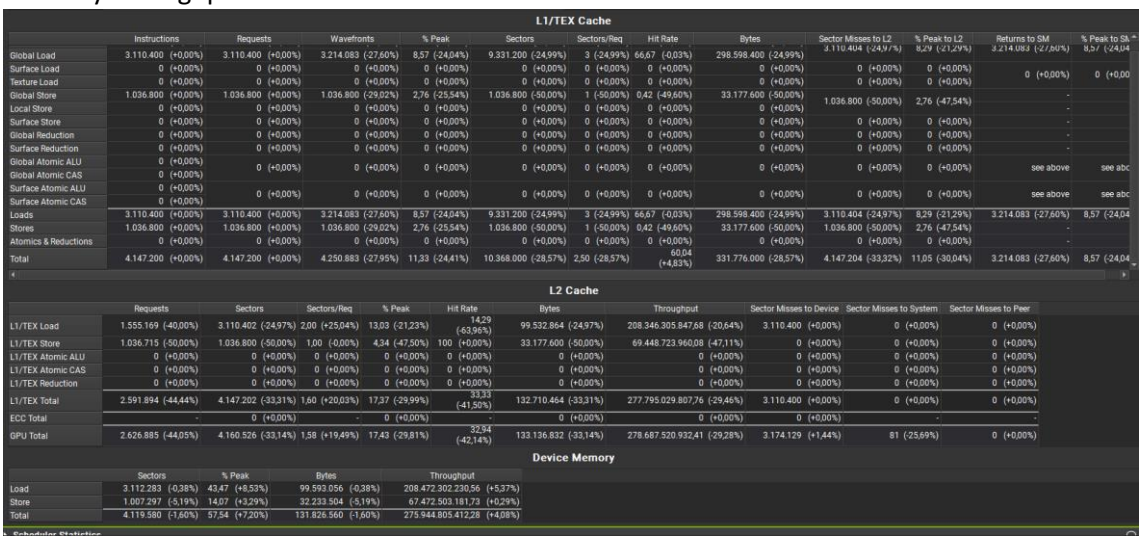
The first mistake I saw on the kernel call is on the grid disposition. The block size of the grid is (16,16,1). The problem with this block size is that the x dimension is not multiple of 32, the warp size. Consequently, it is not possible to make a coalesced memory access because of how the texels are laid out on memory. On device memory, the texture is saved row by row. The

warps of our kernel have to access 16 adjacent bytes from one row and then jump to the next row to get the next 16 texels making it necessary to create 2 memory read requests of two parts of the memory. So the first action I did was correct the block size and put it to (32,8,1).



| ID | Issues Detecte | Function Name | Demangled Name | Process | Device Name | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/thread] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 13 | RGBToGreyScale2D | RGBToGreyScale2D(c... | [2416] CudaRuntime1.exe | NVIDIA GeForc... | 240, 540, 1 | 32, 8, 1 | 782.730 (-4,77%) | 477,73 (-5,46%) | 33,16 (+4,91%) | 57,54 (+7,20%) | 16 (+0,00%) |

After the changes we reduced a 5% the time of the kernel due to an increase of the 7% of memory throughput.



**L1/TEX Cache**

| | Instructions | Requests | Wavefronts | % Peak | Sectors | Sectors/Req | Hit Rate | Bytes | Sector Misses to L2 | % Peak to L2 | Returns to SM | % Peak to SM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Global Load | 3.110.400 (+0,00%) | 3.110.400 (+0,00%) | 3.214.083 (-27,60%) | 8,57 (-24,04%) | 9.331.200 (-24,99%) | 3 (-24,99%) | 66,67 (-0,03%) | 298.598.400 (-24,99%) | 3.110.404 (-24,97%) | 8,29 (-21,29%) | 3.214.083 (-27,60%) | 8,57 (-24,04 |
| Surface Load | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00 |
| Texture Load | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | | |
| Global Store | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 1.036.800 (-29,02%) | 2,76 (-25,54%) | 1.036.800 (-50,00%) | 1 (-50,00%) | 0,42 (-49,60%) | 33.177.600 (-50,00%) | 1.036.800 (-50,00%) | 2,76 (-47,54%) | - | |
| Local Store | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | | | - | |
| Surface Store | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | |
| Global Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | |
| Surface Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | |
| Global Atomic ALU | 0 (+0,00%) | | | | | | | | | | | |
| Global Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | see above | see abc |
| Surface Atomic ALU | 0 (+0,00%) | | | | | | | | | | | |
| Surface Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | see above | see abc |
| Loads | 3.110.400 (+0,00%) | 3.110.400 (+0,00%) | 3.214.083 (-27,60%) | 8,57 (-24,04%) | 9.331.200 (-24,99%) | 3 (-24,99%) | 66,67 (-0,03%) | 298.598.400 (-24,99%) | 3.110.404 (-24,97%) | 8,29 (-21,29%) | 3.214.083 (-27,60%) | 8,57 (-24,04 |
| Stores | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 1.036.800 (-29,02%) | 2,76 (-25,54%) | 1.036.800 (-50,00%) | 1 (-50,00%) | 0,42 (-49,60%) | 33.177.600 (-50,00%) | 1.036.800 (-50,00%) | 2,76 (-47,54%) | - | |
| Atomics & Reductions | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | |
| Total | 4.147.200 (+0,00%) | 4.147.200 (+0,00%) | 4.250.883 (-27,95%) | 11,33 (-24,41%) | 10.368.000 (-28,57%) | 2,50 (-28,57%) | 60,04 (+4,83%) | 331.776.000 (-28,57%) | 4.147.204 (-33,32%) | 11,05 (-30,04%) | 3.214.083 (-27,60%) | 8,57 (-24,04 |

**L2 Cache**

| | Requests | Sectors | Sectors/Req | % Peak | Hit Rate | Bytes | Throughput | Sector Misses to Device | Sector Misses to System | Sector Misses to Peer |
|---|---|---|---|---|---|---|---|---|---|---|
| L1/TEX Load | 1.555.169 (-40,00%) | 3.110.402 (-24,97%) | 2,00 (+25,04%) | 13,03 (-21,23%) | 14,29 (-63,96%) | 99.532.864 (-24,97%) | 208.346.305.847,68 (-20,64%) | 3.110.400 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Store | 1.036.715 (-50,00%) | 1.036.800 (-50,00%) | 1,00 (-0,00%) | 4,34 (-47,50%) | 100 (+0,00%) | 33.177.600 (-50,00%) | 69.448.723.960,08 (-47,11%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Atomic ALU | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Total | 2.591.894 (-44,44%) | 4.147.202 (-33,31%) | 1,60 (+20,03%) | 17,37 (-29,99%) | 33,33 (-41,50%) | 132.710.464 (-33,31%) | 277.795.029.807,76 (-29,46%) | 3.110.400 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| ECC Total | - | 0 (+0,00%) | - | 0 (+0,00%) | | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| GPU Total | 2.626.885 (-44,05%) | 4.160.526 (-33,14%) | 1,58 (+19,49%) | 17,43 (-29,81%) | 32,94 (-42,14%) | 133.136.832 (-33,14%) | 278.687.520.932,41 (-29,28%) | 3.174.129 (+1,44%) | 81 (-25,69%) | 0 (+0,00%) |

**Device Memory**

| | Sectors | % Peak | Bytes | Throughput |
|---|---|---|---|---|
| Load | 3.112.283 (-0,38%) | 43,47 (+8,53%) | 99.593.056 (-0,38%) | 208.472.302.230,56 (+5,37%) |
| Store | 1.007.297 (-5,19%) | 14,07 (+3,29%) | 32.233.504 (-5,19%) | 67.472.503.181,73 (+0,29%) |
| Total | 4.119.580 (-1,60%) | 57,54 (+7,20%) | 131.826.560 (-1,60%) | 275.944.805.412,28 (+4,08%) |

This improve in memory throughput comes because we get 2 sectors per request to L2 cache and we just request 3 sectors per instruction load and 1 sector for instruction store to L1 cache effectively reducing the memory operations required for the kernel.

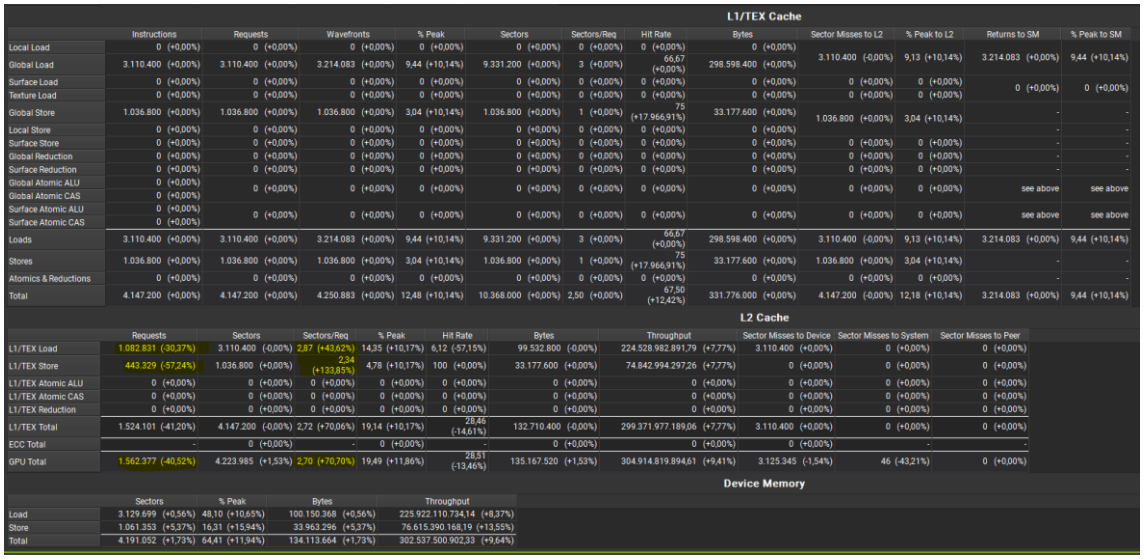
After this optimization I saw that there was no need to access the data in 2 dimensions because we just need to access the pixels one after the other and modify them. This can simplify the kernel optimization so I changed the kernel to be 1D using a 256 threads block on the x dimension (256,1,1).

```
__global__ void RGBToGreyScale(const uchar3* const colorPixels, unsigned char* greyPixels, unsigned int numElements)
{
    const int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < numElements)
        greyPixels[idx] = 0.299f * colorPixels[idx].x + 0.587f * colorPixels[idx].y + 0.114f * colorPixels[idx].z;
}
```

This format changes unexpectedly optimized the kernel quite a lot.

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device N | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/thread] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 13 | RGBToGreyScale | RGBToGreyScale(... | [38664] C... | NVID... | 129600, 1, 1 | 256, 1, 1 | 711.688 (-11,30%, z=-1,32) | 443,30 (-9,81%, z=-1,27) | 30,44 (-6,02%, z=-1,16) | 64,41 (+15,83%, z=+1,32) | 16 (+0,00%, z=+0,00) |

The kernel was an 11% faster than the first kernel and a 9,81% faster than the second. This again was the cause of an increase of memory throughput to the 64%.



**L1/TEX Cache**

| | Instructions | Requests | Wavefronts | % Peak | Sectors | Sectors/Req | Hit Rate | Bytes | Sector Misses to L2 | % Peak to L2 | Returns to SM | % Peak to SM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Local Load | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | | | | |
| Global Load | 3.110.400 (+0,00%) | 3.110.400 (+0,00%) | 3.214.083 (+0,00%) | 9,44 (+10,14%) | 9.331.200 (+0,00%) | 3 (+0,00%) | 66,67 (+0,00%) | 298.598.400 (+0,00%) | 3.110.400 (-0,00%) | 9,13 (+10,14%) | 3.214.083 (+0,00%) | 9,44 (+10,14%) |
| Surface Load | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| Texture Load | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | | |
| Global Store | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 3,04 (+10,14%) | 1.036.800 (+0,00%) | 1 (+0,00%) | 75 (+17.966,91%) | 33.177.600 (+0,00%) | 1.036.800 (+0,00%) | 3,04 (+10,14%) | - | - |
| Local Store | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | | | - | - |
| Surface Store | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| Global Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| Surface Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| Global Atomic ALU | 0 (+0,00%) | | | | | | | | | | | |
| Global Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | see above | see above |
| Surface Atomic ALU | 0 (+0,00%) | | | | | | | | | | | |
| Surface Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | see above | see above |
| Loads | 3.110.400 (+0,00%) | 3.110.400 (+0,00%) | 3.214.083 (+0,00%) | 9,44 (+10,14%) | 9.331.200 (+0,00%) | 3 (+0,00%) | 66,67 (+0,00%) | 298.598.400 (+0,00%) | 3.110.400 (-0,00%) | 9,13 (+10,14%) | 3.214.083 (+0,00%) | 9,44 (+10,14%) |
| Stores | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 1.036.800 (+0,00%) | 3,04 (+10,14%) | 1.036.800 (+0,00%) | 1 (+0,00%) | 75 (+17.966,91%) | 33.177.600 (+0,00%) | 1.036.800 (+0,00%) | 3,04 (+10,14%) | - | - |
| Atomics & Reductions | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| Total | 4.147.200 (+0,00%) | 4.147.200 (+0,00%) | 4.250.883 (+0,00%) | 12,48 (+10,14%) | 10.368.000 (+0,00%) | 2,50 (+0,00%) | 67,50 (+12,42%) | 331.776.000 (+0,00%) | 4.147.200 (-0,00%) | 12,18 (+10,14%) | 3.214.083 (+0,00%) | 9,44 (+10,14%) |

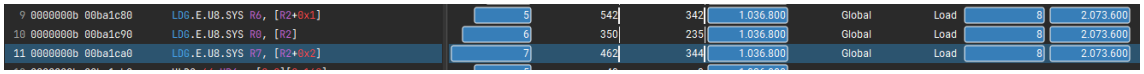
**L2 Cache**

| | Requests | Sectors | Sectors/Req | % Peak | Hit Rate | Bytes | Throughput | Sector Misses to Device | Sector Misses to System | Sector Misses to Peer |
|---|---|---|---|---|---|---|---|---|---|---|
| L1/TEX Load | 1.082.831 (-30,37%) | 3.110.400 (-0,00%) | 2,87 (+43,62%) | 14,35 (+10,17%) | 6,12 (-57,15%) | 99.532.800 (-0,00%) | 224.528.982.891,79 (+7,77%) | 3.110.400 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Store | 443.329 (-57,24%) | 1.036.800 (+0,00%) | 2,34 (+133,85%) | 4,78 (+10,17%) | 100 (+0,00%) | 33.177.600 (+0,00%) | 74.842.994.297,26 (+7,77%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Atomic ALU | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Atomic CAS | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Reduction | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| L1/TEX Total | 1.524.101 (-41,20%) | 4.147.200 (-0,00%) | 2,72 (+70,06%) | 19,14 (+10,17%) | 28,46 (-14,61%) | 132.710.400 (-0,00%) | 299.371.977.189,06 (+7,77%) | 3.110.400 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) |
| ECC Total | - | 0 (+0,00%) | - | 0 (+0,00%) | - | 0 (+0,00%) | 0 (+0,00%) | 0 (+0,00%) | - | - |
| GPU Total | 1.562.377 (-40,52%) | 4.223.985 (+1,53%) | 2,70 (+70,70%) | 19,49 (+11,86%) | 28,51 (-13,46%) | 135.167.520 (+1,53%) | 304.914.819.894,61 (+9,41%) | 3.125.345 (-1,54%) | 46 (-43,21%) | 0 (+0,00%) |

**Device Memory**

| | Sectors | % Peak | Bytes | Throughput |
|---|---|---|---|---|
| Load | 3.129.699 (+0,56%) | 48,10 (+10,65%) | 100.150.368 (+0,56%) | 225.922.110.734,14 (+8,37%) |
| Store | 1.061.353 (+5,37%) | 16,31 (+15,94%) | 33.963.296 (+5,37%) | 76.615.390.168,19 (+13,55%) |
| Total | 4.191.052 (+1,73%) | 64,41 (+11,94%) | 134.113.664 (+1,73%) | 302.537.500.902,33 (+9,64%) |

This increment of memory throughput was mainly caused again by the increase of sectors requested for load and store operations to L2 cache from the L1 which directly translate to less device memory requests increasing the memory throughput a lot. Nevertheless, Nsight Compute still warned us of the uncoalesced global access. When we go see the assembly instructions generated for the warp we notice that we get 3 separate ldg calls to global memory. Why does this happen if the 3 rgb bytes are one after the other in memory?



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 0000000b 00ba1c80 | LDG.E.U8.SYS R6, [R2+0x1] | 5 | 542 | 342 | 1.036.800 | Global | Load | 8 | 2.073.600 |
| 10 0000000b 00ba1c90 | LDG.E.U8.SYS R0, [R2] | 6 | 350 | 235 | 1.036.800 | Global | Load | 8 | 2.073.600 |
| 11 0000000b 00ba1ca0 | LDG.E.U8.SYS R7, [R2+0x2] | 7 | 462 | 344 | 1.036.800 | Global | Load | 8 | 2.073.600 |

This happens because we require 3 bytes per thread which gets misaligned with the sector size of 4 bytes which is the minimum size we can request to memory. That is why the compiler has to request the bytes individually because the 3 rgb bytes together could fall in between 2 memory sector requests. For example, the first thread of the warp would access the 3 first rgb bytes from the first sector. Then the next thread has to access the last byte of memory from the first sector and then 2 more bytes from the next sector. Continuously, the third thread would access 2 bytes from the second sector and 1 byte from the third. Finally, to restart the cycle, the forth thread would have to access the 3 last bytes from the third sector. In order to satisfy the requests like this the compiler could branch this four requests depending on the module of 4 of the threadIdx but this would get why slower because of thread divergence and a lot more ldg instructions with the different branches. That is why it requests each byte individually although they are next to each other. The problem with that is that we end up making 3 ldg calls to access just 8 bytes each. For each of this calls we are accessing 4 bytes with the minimum sector size. As we see above we performed each of the 3 ldg instructions 1.036.800 times ((number of theads in kernel == number of texels(in 8k texture 33.177.600 pixels)) / 32(warp size)). For each of the ldg instructions we are just reading a maximum of 2 bytes from the 4 available on the sector which means we are reading 2.073.600 (2*1.036.800) excessive bytes of data per ldg. In total we are accessing 6.220.600(2.073.600*3) of excessive data making 3.110.400(1.036.800*3) ldg calls that are stalling the pipeline waiting for data.

We could improve the algorithm to take advantage of the fact that the bytes are next to each other. The idea is to first load all the data needed by the blocks in a coalesced manner to cache memory. To achieve that, each thread can load 4 bytes to cache. This way, as the data is in packs of three the last fourth of the threads of a block don't need to load data. The last forth of the block size also is a multiple of the warp size which avoids thread divergence on the warp level. As each thread is loading 4 bytes that means that each warp loads 128 bytes (4*32) that

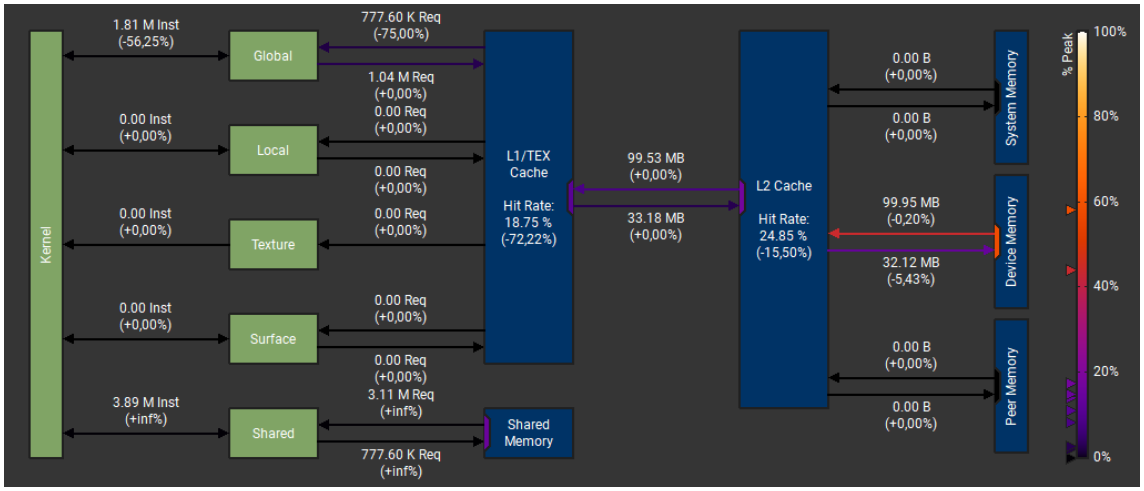
is the entire cache line used on each ldg request of the warps. Once cached, each thread can access their required bytes individually because aside from bank conflicts, there is no penalty for non-sequential accesses by a warp in shared memory.

```
//cacheSize = blockDim.x * 3 * 4
__global__ void CachedRGBToGreyScale(const uchar4* const colorPixels, unsigned char* greyPixels, unsigned int numElements)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < numElements)
    {
        extern __shared__ uchar4 cacheData[];
        const unsigned int uneededBlockCache = (blockDim.x * 1 / 4);
        if (threadIdx.x < blockDim.x - uneededBlockCache)
            cacheData[threadIdx.x] = colorPixels[index - uneededBlockCache * blockIdx.x];
        __syncthreads();
        const uchar3 currentTexel = ((uchar3*)cacheData)[threadIdx.x];
        greyPixels[index] = 0.299f * currentTexel.x + 0.587f * currentTexel.y + 0.114f * currentTexel.z;
    }
}
```

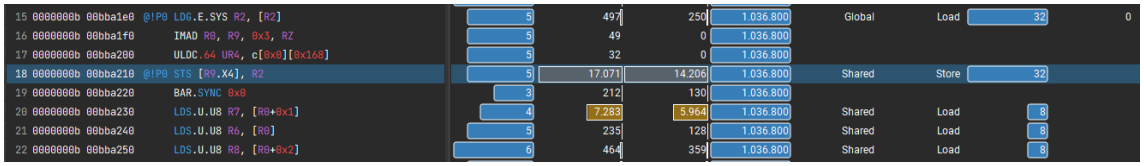
With this new algorithm we achieved the coalesced global access with just one ldg call however unexpectedly our kernel run 6,89% slower than the previous kernel.

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device N | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput | Memory Throughput | # Registers | [register/thread] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | CachedRGBToGrey... | CachedRGBToGreySc... | [28264] C... | NVID... | 129600, 1, 1 | 256, 1, 1 | 775.396 (+8,95%) | 473,86 (+6,89%) | 39,09 (+28,43%) | 58,25 (-9,56%) | 16 | (+0,00%) |

Notice the importance of memory throughput that even though our compute throughput went up by 30% the 10% decrease on memory throughput made the algorithm slower than the previous. This indicates that our kernel is heavily memory bound. In order to improve the kernel we have to find out why the memory throughput decreased despite of the achieved coalesced global memory access.




The answer is shown on this memory chart. As we predicted before, we reduced a 56% the global memory instructions with the coalesced memory access that translated to a 75% less global memory requests. However, all those extra request are now made to shared memory in form of load and store. We also see that we didn't need this shared memory on the first place because although we did the three load requests to global memory, after the first request to fill the red data the other two requests already had the data cached from the excessive data read from the first ldg. This explains the decrease of 72% of L1 cache hit rate that now is moved to shared memory. However, this move to shared memory adds memory instructions to store and then load data to shared memory. This extra instructions where not needed because the data was already cached on L1 as we have just seen and they cause the pipeline to stall with memory operations. Moreover, the cache addition also made the need of using a synchronization barrier (_synchthreads()) that slows things even a little bit more.

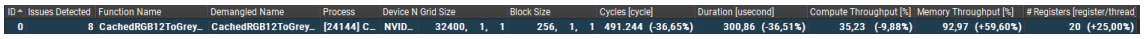



Here we can see that the 3 ldg have converted to just one now but now we have to do the 3 loads from shared memory.

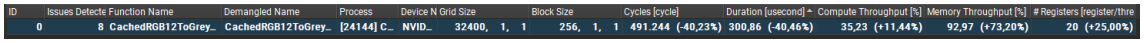
Considering all the above, another way we could optimize the algorithm is by always accessing all the 128 bytes of the cache line when doing a memory operation. Right now we just achieve it on the global load and shared store. We can improve it on the shared load and the global store. As seen before, in order to do so we need to make warp memory accesses where each thread accesses 4 bytes (4*32=128). Consequently, the store to global memory needs to be 4 bytes so we need 4 times less threads (each thread was storing 1 texel). To store 4 bytes of memory we need rgb data of 4 texels per thread (3*4=12 bytes). This 12 bytes can be serviced in 3 sequential packs of 4 bytes per each thread. The fact that each thread needs the next 2 neighbor's packs creates a strided access to global memory. We can correct that using the cache memory. We first load all the block data needed on the cache in a coalesced maner and then we perform the strided access on the cache with no penalty.

```
//cacheSize = blockDim.x*4*3
__global__ void CachedRGB12ToGreyScale(const uchar4* const colorPixels, uchar4* greyPixels, unsigned int numThreads)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < numThreads)
    {
        const unsigned int offset = 2 * blockDim.x * blockIdx.x;
        extern __shared__ uchar4 shared[];
        shared[threadIdx.x] = colorPixels[index + offset];
        shared[threadIdx.x + blockDim.x] = colorPixels[index + offset + blockDim.x];
        shared[threadIdx.x + 2*blockDim.x] = colorPixels[index + offset + 2*blockDim.x];
        __syncthreads();
        const uchar4 threadDataBlock1 = shared[threadIdx.x * 3];
        const uchar4 threadDataBlock2 = shared[threadIdx.x * 3 + 1];
        const uchar4 threadDataBlock3 = shared[threadIdx.x * 3 + 2];
        uchar4 result;
        result.x = 0.299f * threadDataBlock1.x + 0.587f * threadDataBlock1.y + 0.114f * threadDataBlock1.z;
        result.y = 0.299f * threadDataBlock1.w + 0.587f * threadDataBlock2.x + 0.114f * threadDataBlock2.y;
        result.z = 0.299f * threadDataBlock2.z + 0.587f * threadDataBlock2.w + 0.114f * threadDataBlock3.x;
        result.w = 0.299f * threadDataBlock3.y + 0.587f * threadDataBlock3.z + 0.114f * threadDataBlock3.w;
        greyPixels[index] = result;
    }
}
```

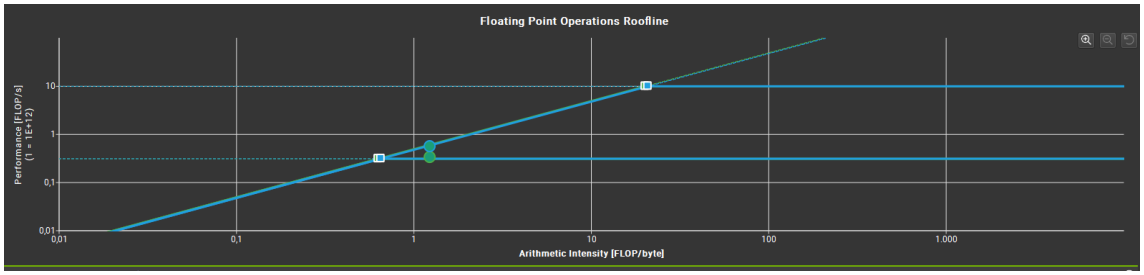
Finally, this time the results where definitive. We achieved a 60% increase on memory throughput compared to the last kernel reaching the 93%! This lead to the 36% decrease on the kernel time, now of just 300 us!




Overall, the final optimization compared to the first starting kernel was of 40% the time thenks to the increase of 60% on memory throughput.

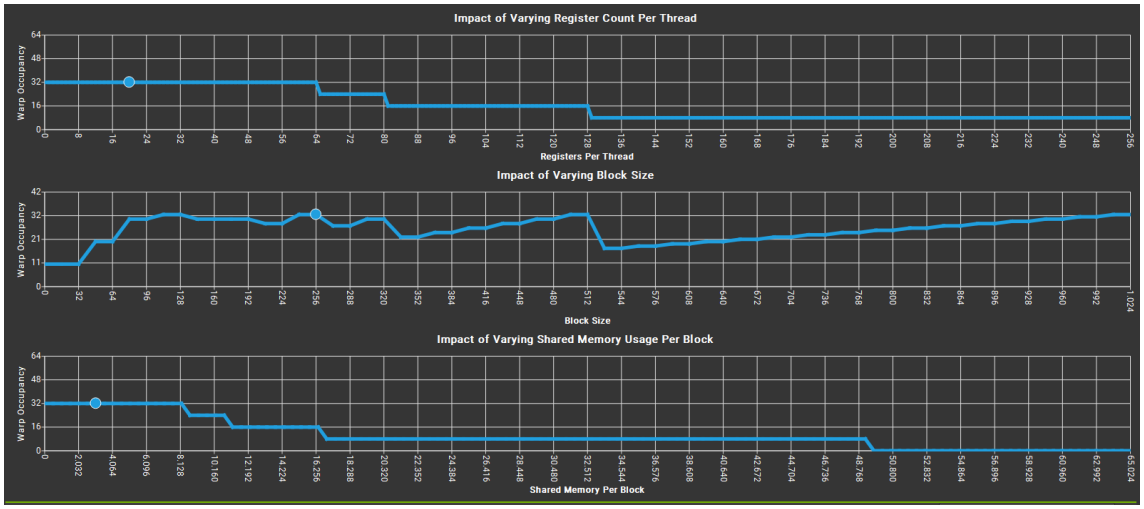



Once Reached this point I have some indicators that further optimizations are not possible or would not be very significant.




First, as we can see on the roofline analysis the current value highlighted in blue has already reached the memory bandwidth boundary. Accordingly, optimizations are only possible increasing the arithmetic intensity because as we have been seeing this kernel has a memory bottleneck. With the roofline we can also see the starting point of the optimizations highlighted in green and we can see that we just did memory optimizations as the point got straight up without changing the arithmetic intensity.

Another hint that it is not very feasible to further optimize is occupancy. The current kernel already has a theoretical occupancy of 100% as it is not limited by threads, registers or shared memory and in practice is measured to reach the 92%.




Also it can't be optimized using concurrency because it is just one kernel.

Finally, we can look at the device specs to see how well the kernel performed. The Nvidia RTX 2080 super has a max theoretical bandwidth of 496,1GB/s. If our kernel took 300 microseconds to execute this means that the max theoretical bytes read with that time is 148.830.000 bytes (496,1*0,0003*100000000). Our kernel requires a minimum of 132.710.400 bytes (7680*4320(8k texels)*4(the kernel reads and writes 4 times per texel)). Although this makes the approximate 10% difference left to optimize there are some cycles where we are stalled on barriers or making computations. This 10% can decrease to 5%, which would make sense with the 93% memory throughput we are getting on the kernel.

The last thing I got curious to try was to do this new algorithm without using the cache. Like before this will get rid of the coalesced memory access as we will be accessing strided global memory. However, I'm courious if it would not matter as before because out of the 3 ldg the next 2 will already be cached like it happened before.

```
__global__ void RGB12ToGreyScale(const uchar4* const colorPixels, unsigned char* greyPixels, unsigned int numThreads)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < numThreads)
    {
        const uchar4 threadDataBlock1 = colorPixels[index * 3];
        const uchar4 threadDataBlock2 = colorPixels[index * 3 + 1];
        const uchar4 threadDataBlock3 = colorPixels[index * 3 + 2];
        uchar4 result;
        result.x = 0.299f * threadDataBlock1.x + 0.587f * threadDataBlock1.y + 0.114f * threadDataBlock1.z;
        result.y = 0.299f * threadDataBlock1.w + 0.587f * threadDataBlock2.x + 0.114f * threadDataBlock2.y;
        result.z = 0.299f * threadDataBlock2.z + 0.587f * threadDataBlock2.w + 0.114f * threadDataBlock3.x;
        result.w = 0.299f * threadDataBlock3.y + 0.587f * threadDataBlock3.z + 0.114f * threadDataBlock3.w;
        ((uchar4*)greyPixels)[index] = result;
    }
}
```

As predicted, it didn't matter. The memory througput and timing gains where very similar because the first uncoalesced global accesses was then cached for the next ldgs.

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device N | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/thread |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 12 | RGB12ToGreyScale | RGB12ToGreyScale(c... | [33964] C... | NVID... | 32400, 1, 1 | 256, 1, 1 | 486.523 (-0,96%) | 300,96 (+0,03%) | 35,58 (+1,01%) | 93,49 (+0,57%) | 21 (+5,00%) |

On the 4k image with cache things where the same as the 8k (40% timing gain and 93% memory throughput)

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device N | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [ | # Registers [register/thread |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 8 | CachedRGB12ToGrey... | CachedRGB12ToGrey... | [23624] C... | NVID... | 8100, 1, 1 | 256, 1, 1 | 123.489 (-40,33%) | 78,62 (-39,12%) | 35,11 (+11,72%) | 91,99 (+60,93%) | 20 (+25,00%) |

On the 4k image without cache the memory throughput reached the 99%!

| ID | Issues Detecte | Function Name | Demangled Name | Process | Device N | Grid Size | Block Size | Cycles [cycle] | Duration [usecond] | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/thread |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 12 | RGB12ToGreyScale | RGB12ToGreyScale(c... | [32140] C... | NVID... | 8100, 1, 1 | 256, 1, 1 | 116.033 (-43,93%) | 79,58 (-38,38%) | 37,35 (+18,84%) | 98,81 (+72,86%) | 21 (+31,25%) |

All the Nsight compute profiling reports are available at the github.

**Platform Layer**

The problem with the CUDA platform is that it only targets Nvidia GPUs. My code will not run on AMD or Intel GPUs and it won't run on integrated GPUs either. This is inconvenient when making a videogame because we can't target a lot of people. Therefore, for videogame development we tend to use graphics APIs like OpenGl, vulkan or directX. This APIs are the interface between the programmers and the hardware manufacturers that write drivers for those APIs. Accordingly, the programmers have to write code using the interface and automatically will be targeting the different hardware through the drivers and the operating system. To see how game developers, interact with GPUs in games I will write a very simple platform layer that could be the base code to start a game engine or game. As most of the games target the windows operating system and my computer also has it I will target this OS. The platform layer will not use high level abstractions libraries like SDL and we will be interacting directly with the operating system API to be able to see the hole process.

Starting with the development we need to first prepare the build system for the little app. For windows development the simplest and recommended way is to use Microsoft tools like visual studio and their compiler msvc. I personally like a lot visual studio debugger. However, for the syntax error tips, code formatting, warnings and good optimizations thanks to the llvm platform I prefer the clang compiler. There is a tool to generate build-systems that enables me to use both called Cmake. Moreover, using Cmake I can also use ninja to compile the source files as fast as possible. I also use two simple batch files (.bat) to compile fast on release or debug using clang without having to repeatedly specify the compile flags.

```
@echo off
pushd .
clang++ -std=c++14 "..\src\WindowsEngine.cpp"  "..\src\Engine.res" -DNDEBUG -DINITGUID -I"..\src\Dependencies\D3D12\include"
 -I"../src/Dependencies/pix/Include" -l"..\src\Dependencies\pix\bin\x64\WinPixEventRuntime.lib" -lkernel32.lib -luser32.lib
 -fno-cxx-exceptions -fno-exceptions -fno-stack-protector -mno-stack-arg-probe -mstack-probe-size=9999999 -fuse-ld=lld -g
-Xlinker -nodefaultlib -Xlinker -subsystem:windows -Xlinker -stack:0x100000,0x100000
popd
```

```
cmake_minimum_required(VERSION "3.13")

project(Engine)

#NOTE: This could be set manually. If you change the path have in mind that the output binary should be on the same directory as the dlls and the D3D12 folder
#set(CMAKE_LIBRARY_OUTPUT_DIRECTORY build/)
#set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY build/)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ../build/ CACHE PATH "Where to output the final executable")
if(MSVC)
#don't add the release and debug folders after the directory
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_RELEASE ${CMAKE_RUNTIME_OUTPUT_DIRECTORY})
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_DEBUG ${CMAKE_RUNTIME_OUTPUT_DIRECTORY})

set_directory_properties(PROPERTIES VS_STARTUP_PROJECT ${PROJECT_NAME})
endif(MSVC)

add_executable(${PROJECT_NAME} WIN32 src/WindowsEngine.cpp) #src/Renderer.cpp)
#Correct way to add .rc files???
#https://discourse.cmake.org/t/how-to-use-resource-files-rc-in-cmake/2628
target_sources(${PROJECT_NAME} PRIVATE src/Engine.rc)
target_include_directories(${PROJECT_NAME} BEFORE PRIVATE src/Dependencies/D3D12/include src/Dependencies/pix/Include) #src/Dependencies/DirectXTex/include)
#if i don't define INITGUID all the GUID from DEFINE_GUID() macro used in wincoded.h evaluate to null and then don't work on release mode
add_compile_definitions(INITGUID)
target_link_directories(${PROJECT_NAME} PRIVATE src/Dependencies/pix/bin/x64/)#src/Dependencies/DirectXTex/lib/Release/)
#TODO: treure el pix profiler al fer les release builds
target_link_libraries(${PROJECT_NAME} PRIVATE kernel32.lib User32.lib windowscodecs.lib WinPixEventRuntime.lib) #DXGI.lib d3d12.lib DirectXTex.lib)  windowscodecs.lib
#not sure if the fno-stack-protecktor is working (check)
#TODO: disable runtime error chacks from debug code (/RTC) $<$<CONFIG:DEBUG>:/RTC>
#TODO: /EHa en release pk si no error al linking de _std_terminate (s'ha de mirar una mica lo de les exceptions')
target_compile_options(${PROJECT_NAME} PRIVATE $<$<AND:$<CXX_COMPILER_ID:MSVC>,$<NOT:$<CONFIG:DEBUG>>>:/EHa /GR-> $<$<AND:$<CXX_COMPILER_ID:Clang>,
$<NOT:$<CONFIG:DEBUG>>>:-fno-cxx-exceptions -fno-exceptions --no-standard-libraries -fno-builtin -DNDEBUG>   $<$<CXX_COMPILER_ID:MSVC>: -GS- -Gs9999999>
$<$<CXX_COMPILER_ID:Clang>:-fuse-ld=lld -fno-stack-protector -mstack-probe-size=9999999 -mno-stack-arg-probe>)
#error clang: cmake omits a repeated flag (=> it just puts the first -Xlinker and omits the rest causing it to crush the compilation)!!!!
target_link_options(${PROJECT_NAME} PRIVATE $<$<AND:$<CXX_COMPILER_ID:MSVC>,$<NOT:$<CONFIG:DEBUG>>>:/NODEFAULTLIB> $<$<CXX_COMPILER_ID:MSVC>:-stack:0x100000,0x100000>
$<$<CXX_COMPILER_ID:Clang>:--for-linker -stack:0x100000,0x100000> $<$<AND:$<CXX_COMPILER_ID:Clang>,$<NOT:$<CONFIG:DEBUG>>>:-Xlinker -nodefaultlib>)
```

One thing to consider for those build scripts is that I won't be using the runtime library on release mode to interact directly with the OS. This needs to be specified to the linker with the /NODEFAULTLIB flag. Other interesting flags I use are the not stack probing and stack fixed allocation of 1MB (little optimization) and the –fuse-ld=lld for the clang compilation to use the llvm linker. Notice also that we don't link a lot of libraries because the majority of them will be dynamically linked at runtime. The last things to comment about the project building are the definition of INITGUID and the setting of the runtime output directory which are necessary for things we will see later.

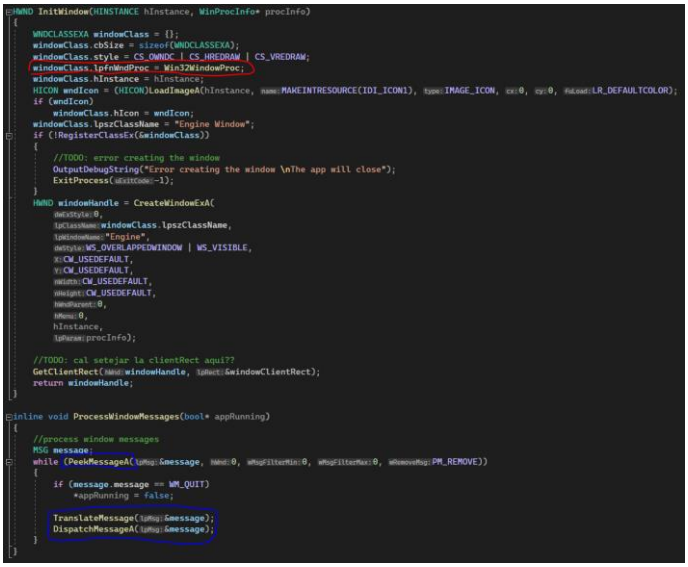
Starting to code we first write the variables, methods and containers we will need from the runtime library we removed for the release mode. The majority of this utility routines and containers are implemented on Utils.cpp. We also define the entry point to start. The default entry point for applications is a call to the *CRTStartup* function that will setup all the variables of the CRT runtime library and then automatically call the *WinMain* entry point. As we don't have the CRT on release we define the startup function ourselves that just calls the *WinMain* function.

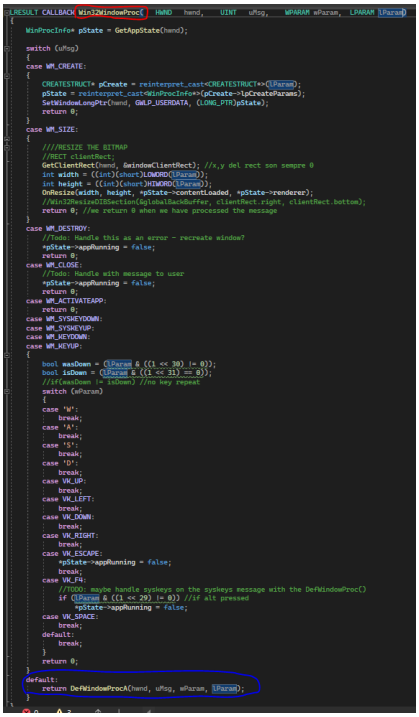
```
//pragma function because the compiler has reserved the memset
#pragma function(memset)
    void* memset(void* dest, int c, size_t count)
    {
        char* bytes = (char*)dest;
        while (count--)
        {
            *bytes++ = (char)c;
        }
        return dest;
    }

#pragma function(memcpy)
    void* memcpy(void* dest, const void* src, size_t count)
    {
        char* dest8 = (char*)dest;
        const char* src8 = (const char*)src;
        while (count--)
        {
            *dest8++ = *src8++;
        }
        return dest;
    }
#pragma function(memcmp)
    int memcmp(const void* ptr1, const void* ptr2, size_t num)
    {
        const unsigned char* one = (const unsigned char*)ptr1;
        const unsigned char* two = (const unsigned char*)ptr2;
        unsigned int i = 0;
        while (num--)
        {
            if (one[i] != two[i])
            {
                if (one[i] < two[i])
                    return 1;
                else
                    return -1;
            }
            ++i;
        }
        return 0;
    }
//#endif
```

```
//WinMainCRTStartup to pass ansi cmd and wWinMainCRTStartup to pass Unicode cmd calling wWinMain or WinMain
#ifndef _DEBUG
void __stdcall WinMainCRTStartup()
{
    int Result = WinMain(GetModuleHandle(0), 0, 0, 0);
    ExitProcess(Result);
}
#endif
```

Once inside the entry point, the first thing to code for a windows application is the window creation and the message loop. Windows creates a message queue for the threads that create one or more windows. When creating the window we pass a pointer to a function

implemented by us that will be called later when dispatching messages. After the window creation we create the main while loop for the application. Inside we do the message loop. We have to periodically check that message queue to pull the messages the operating system has set for the window (*PeekMessage*)and act accordingly (*DispatchMessage*). *TranslateMessage* is for cases when the application must obtain character input from the user. The *DispatchMessage* routine will call the function we passed to the window (Win32WindowProc). This way we can process the operating system messages and react to them. If our app doesn't need to react to a certain messages we can just call the default function *DefWindowProc* to handle them. This is the way to get the mouse and keyboard input for the window.





After having the main window and input we can start including the libraries required for the multimedia of the game. As we will be programing in windows we will use the directX API and libraries. However, before starting we need one special library that is the base of how the majority of windows APIs interact with us, the COM library. Microsoft COM (Component Object Model) is a very complex topic that I am going to briefly introduce to understand our code. The main idea is the separation between the code interface and its implementation. The interface is a group of well-defined functions all accessible by a pointer. In C++ this will map to a virtual table from a class. The client side code normally makes a call to the server to get this interface pointer and then uses the functions defined in the interface (In C++ we get a class pointer and we call the methods of the class through that pointer). On the other side, this unique identified interface can have many different implementations but they have to implement all the interface functions. This way, the code can be changed or improved by the OS or library provider but it will still work for the client side using the interface. When using COM, the first thing we need to do is to load the COM library. To do so I dynamically link the Ole32.dll and get the function pointers of the functions we will be using from the library. When dynamically linking a library we make a call to *LoadLibrary* to map the dll to our process address space and then we make calls to *GetProcAdress* to retrieve the function pointers we will need from the library. This functions and all the library functionallity are declared on the

library header files.

```
bool Win32InitializeCOM()
{
    HMODULE hOle32Library = LoadLibraryA("Ole32.dll");
    if (hOle32Library)
    {
        CoInitializeExPtr = (MyCoInitializeEx*)GetProcAddress(hOle32Library, "CoInitializeEx");
        CoCreateInstancePtr = (MyCoCreateInstance*)GetProcAddress(hOle32Library, "CoCreateInstance");
        PropVariantClearPtr = (MyPropVariantClear*)GetProcAddress(hOle32Library, "PropVariantClear");
        memset(&MY_GUID_NULL, 0, sizeof(IID));
        return SUCCEEDED(CoInitializeExPtr(nullptr, COINIT_MULTITHREADED));
    }
    return false;
}
```

Once we have COM setup we can start with the directX multimedia libraries. The first one we setup is the audio library. However, the audio will need to read files from disk. Therefore, we first prepare some helper path functions and routines to deal with file data reading and writing.

```
HRESULT ReadFileToBuffer(const char* filePath, void** buffer, DWORD* bufferSize, FilePointer fPtr = { 0, NULL, FILE_BEGIN })
{
    //TODO: manage better when error with reading file (NOT ALL PATHS WITH ERRORS CLOSE THE FILE!!!)
    HRESULT hr = S_OK;
    HANDLE hFile = CreateFile(filePath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (INVALID_SET_FILE_POINTER == SetFilePointer(hFile, fPtr.distanceToMove, fPtr.pDistanceToMoveHigh, fPtr.startingPoint)) {
        CloseHandle(hFile);
        return HRESULT_FROM_WIN32(GetLastError());
    }
    LARGE_INTEGER fileSize;
    if (!GetFileSizeEx(hFile, &fileSize)) {
        CloseHandle(hFile);
        return HRESULT_FROM_WIN32(GetLastError());
    }
    //TODO: memory allocation
    *buffer = HeapAlloc(GetProcessHeap(), HEAP_NO_SERIALIZE, fileSize.QuadPart);
    if (0 == ReadFile(hFile, *buffer, fileSize.LowPart, bufferSize, NULL))
        hr = HRESULT_FROM_WIN32(GetLastError());
    CloseHandle(hFile);
    return hr;
}

HRESULT WriteBufferToFile(const char* filePath, void* buffer, DWORD bufferSize, FilePointer fPtr = { 0, NULL, FILE_BEGIN })
{
    //TODO: manage better when error with reading file (NOT ALL PATHS WITH ERRORS CLOSE THE FILE!!!)
    HRESULT hr = S_OK;
    HANDLE hFile = CreateFile(filePath, GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_ALWAYS, 0, NULL);
    if (INVALID_SET_FILE_POINTER == SetFilePointer(hFile, fPtr.distanceToMove, fPtr.pDistanceToMoveHigh, fPtr.startingPoint)) {
        CloseHandle(hFile);
        return HRESULT_FROM_WIN32(GetLastError());
    }
    if (0 == WriteFile(hFile, buffer, bufferSize, NULL, NULL))
        hr = HRESULT_FROM_WIN32(GetLastError());
    CloseHandle(hFile);
    return hr;
}

bool FileExists(const char* path)
{
    DWORD dwAttrib = GetFileAttributes(path);
    return (dwAttrib != INVALID_FILE_ATTRIBUTES && !(dwAttrib & FILE_ATTRIBUTE_DIRECTORY));
}
```

Notice that for memory allocations we do not use the new and delete operators but directly use the win32 API functions *HeapAlloc* and *HeapFree*.

Starting with the audio, the latest directX library is XAudio2. It is a library that uses COM objects so we first need to call the library provided method to get the COM interface object pointer. We dynamically link the library and we just retrieve this method called *XAudio2Create*. After calling *XAudioCreate* we get the IXAudio2 interface pointer. Using this class pointer, we have access to all the library API functions.

```
typedef HRESULT CreateXAudio2(IXAudio2** ppXAudio2, UINT32 Flags, XAUDIO2_PROCESSOR XAudio2Processor);
CreateXAudio2* Win32LoadXAudio2()
{
    HMODULE hXaudio2Library = LoadLibraryA("XAUDIO2_9.DLL");
    if (!hXaudio2Library)
        hXaudio2Library = LoadLibraryA("XAUDIO2_8.DLL");
    if (hXaudio2Library)
        return (CreateXAudio2*)GetProcAddress(hXaudio2Library, "XAudio2Create");
    return nullptr;
}
```

```
inline void InitAudio(AudioStruct& audio, bool COMInitialized)
{

    if (COMInitialized)
    {
        CreateXAudio2* createAudio = Win32LoadXAudio2();
        if (createAudio)
        {
            //IXAudio2* pXAudio2 = nullptr;
            if (SUCCEEDED(createAudio(&audio.pXAudio2, 0, XAUDIO2_DEFAULT_PROCESSOR)))
                //IXAudio2MasteringVoice* pMasterVoice = nullptr;
                if (FAILED(audio.pXAudio2->CreateMasteringVoice(&audio.pMasterVoice)))
                    OutputDebugStringA("Failed creating mastering voice");
        }
    }
    if (audio.pXAudio2 && audio.pMasterVoice)
    {
```

After setting up the audio I proceed with the controller input with the XInput library. The XInput library does not use COM objects because it just uses mainly two function calls. *XInputGetState* is to retrieve the current state of the controller and *XInputSetState* is used to set the vibration of the controller. When dynamically linking the XInput library I leave the option to have the library uninitialized by making default functions that just return the gamepad not connected enumeration code. Accordingly, the game does not have to crash if the user does not have the XInput library as he can still play with mouse and keyboard.

```
#include <Xinput.h>

//xInput
//needed functions from xinput (XInputGetState // XInputSetState)
//linked at runtime to avoid problems with windows versions that don't support the xinput library
#define X_INPUT_GET_STATE(name) DWORD WINAPI name(DWORD dwUserIndex, XINPUT_STATE* pState)
typedef X_INPUT_GET_STATE(ControllerGetState);
X_INPUT_GET_STATE(UnlinkedXInputGetState)
{
    return ERROR_DEVICE_NOT_CONNECTED;
}
static ControllerGetState* DyXInputGetState = UnlinkedXInputGetState;

#define X_INPUT_SET_STATE(name) DWORD WINAPI name(DWORD dwUserIndex,XINPUT_VIBRATION* pVibration)
typedef X_INPUT_SET_STATE(ControllerSetState);
X_INPUT_SET_STATE(UnlinkedXInputSetState)
{
    return ERROR_DEVICE_NOT_CONNECTED;
}
static ControllerSetState* DyXInputSetState = UnlinkedXInputSetState;

#define XInputGetState DyXInputGetState
#define XInputSetState DyXInputSetState

inline void InitInput()
{
    HMODULE hXInputLibrary = LoadLibraryA("XInput1_4.dll");
    if (!hXInputLibrary)
        hXInputLibrary = LoadLibraryA("XInput9_1_0.dll");
    else if (!hXInputLibrary)
        hXInputLibrary = LoadLibraryA("xinput1_3.dll");

    if (hXInputLibrary)
    {
        DyXInputGetState = (ControllerGetState*)GetProcAddress(hXInputLibrary, "XInputGetState");
        DyXInputSetState = (ControllerSetState*)GetProcAddress(hXInputLibrary, "XInputSetState");
    }
}
```

```
//poll controller states
for (DWORD controllerIndex = 0; controllerIndex < XUSER_MAX_COUNT; ++controllerIndex)
{
    XINPUT_STATE controllerState;
    if (XInputGetState(controllerIndex, &controllerState) == ERROR_SUCCESS /*ERROR_DEV
    {
        //This controller is plugged in
        //TODO: some inputs missing
        XINPUT_GAMEPAD* pad = &controllerState.Gamepad;

        bool up = (pad->wButtons & XINPUT_GAMEPAD_DPAD_UP);
        bool down = (pad->wButtons & XINPUT_GAMEPAD_DPAD_DOWN);
        bool left = (pad->wButtons & XINPUT_GAMEPAD_DPAD_LEFT);
        bool right = (pad->wButtons & XINPUT_GAMEPAD_DPAD_RIGHT);
        bool start = (pad->wButtons & XINPUT_GAMEPAD_START);
        bool back = (pad->wButtons & XINPUT_GAMEPAD_BACK);
        bool leftShoulder = (pad->wButtons & XINPUT_GAMEPAD_LEFT_SHOULDER);
        bool rightShoulder = (pad->wButtons & XINPUT_GAMEPAD_RIGHT_SHOULDER);
        bool aButton = (pad->wButtons & XINPUT_GAMEPAD_A);
        bool bButton = (pad->wButtons & XINPUT_GAMEPAD_B);
        bool xButton = (pad->wButtons & XINPUT_GAMEPAD_X);
        bool yButton = (pad->wButtons & XINPUT_GAMEPAD_Y);

        short stickX = pad->sThumbLX;
        short stickY = pad->sThumbLY;

        //TODO: handle the deadzone apropietly
        //#define XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE  7849
        //#define XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE 8689
        //xOffset += stickX / 4096; //4096 = 2^12
        //yOffset += stickY / 4096;
    }
    //else
    //{
    //  //controller not aviable
    //}
}

XINPUT_VIBRATION vibration;
vibration.wLeftMotorSpeed = 6000;
vibration.wRightMotorSpeed = 6000;
XInputSetState(0, &vibration);
```

After the input we finally can start with the DirectX12 API that will be responsible for interacting with our rendering hardware, the GPU if we have one or the integrated graphics unit of the processor. The first step is to set up the API. In order to get the latest version of the API we need to use the Agility SDK. The Agility SDK is the way Microsoft found to update DirectX graphics features without having to rely onto an operating system update. The implementation code for DirectX12 was previously into d3d12.dll. When Microsoft wanted to

add new exciting features to the DirectX12 API it had to lunch a Windows update that updated the D3D12.dll file including the new features. The problem with this setup is that developers that want to use this new launched features do not know if the users have updated to the latest version of the Windows operating system. If they have not, the new features cannot be used. To troubleshoot this problem, Microsoft converted the runtime D3D12.dll into a simple loader of D3D12Core.dll, the new runtime library that implements the functionality. All the clients now have the D3D12.dll loader in the system regardless of the version. When they update the OS they get the new version of the D3D12Core.dll library with the new features without changing the D3D12.dll loader. The thing is that now the developer has to deploy the version of the D3D12Core.dll with the features he uses to the project. When initializing the application, the D3D12.dll library of the clients will compare the version of the D3D12Core.dll libraries from the developer and the one the OS has, and will load the highest version one. To do this loading I have to first download the D3D12Core.dll dynamic library with the features I want to use (I just use the latest available with version 600) and put it on the output directory. Then I specify the directory and version in two exported variables that the client D3D12.dll will use. The D3D12SDKVersion to compare my runtime library version to the OS version and the D3D12SDKPath to find my D3D12Core.dll relative to the executable if my library has to be loaded. Finally, I dynamically link the D3D12.dll library and get pointers to the functions I will be using.

```
//exports to set the agility sdk parameters to let the d3d12.dll load the correct D3D12Core.dll (system or apps)
extern "C"
{
    __declspec(dllexport) extern const UINT D3D12SDKVersion = 600;
    __declspec(dllexport) extern const char* D3D12SDKPath = u8".\\D3D12\\";
}
```

The last setup things we need to do is to include the D3Dx12.h and link the DXGI library. D3Dx12.h is a header only library that provides helper structures and routines to facilitate the interaction with DirectX12 API. The DXGI (DirectX Graphics Infrastructure) library is a complementary library to the DirectX graphics runtime. As some parts of graphics programing evolve slower, those slow changing parts are included on the DXGI library. We will mainly use it to enumerate hardware devices (adapters) and to create the swap chain.

Following the setup, I can start using the DirectX12 API. As the audio library, it also uses COM interfaces to interact with the library. The first step is to create the ID3D12Device interface. This interface represents our virtual adapter (GPU) and will be used to create the majority of interfaces needed on the API. To create it we first need to find the hardware device we will be linking the interface to that will be responsible for our rendering and GPUGP. To find it we use the enumerating adapters process of the DXGI library.

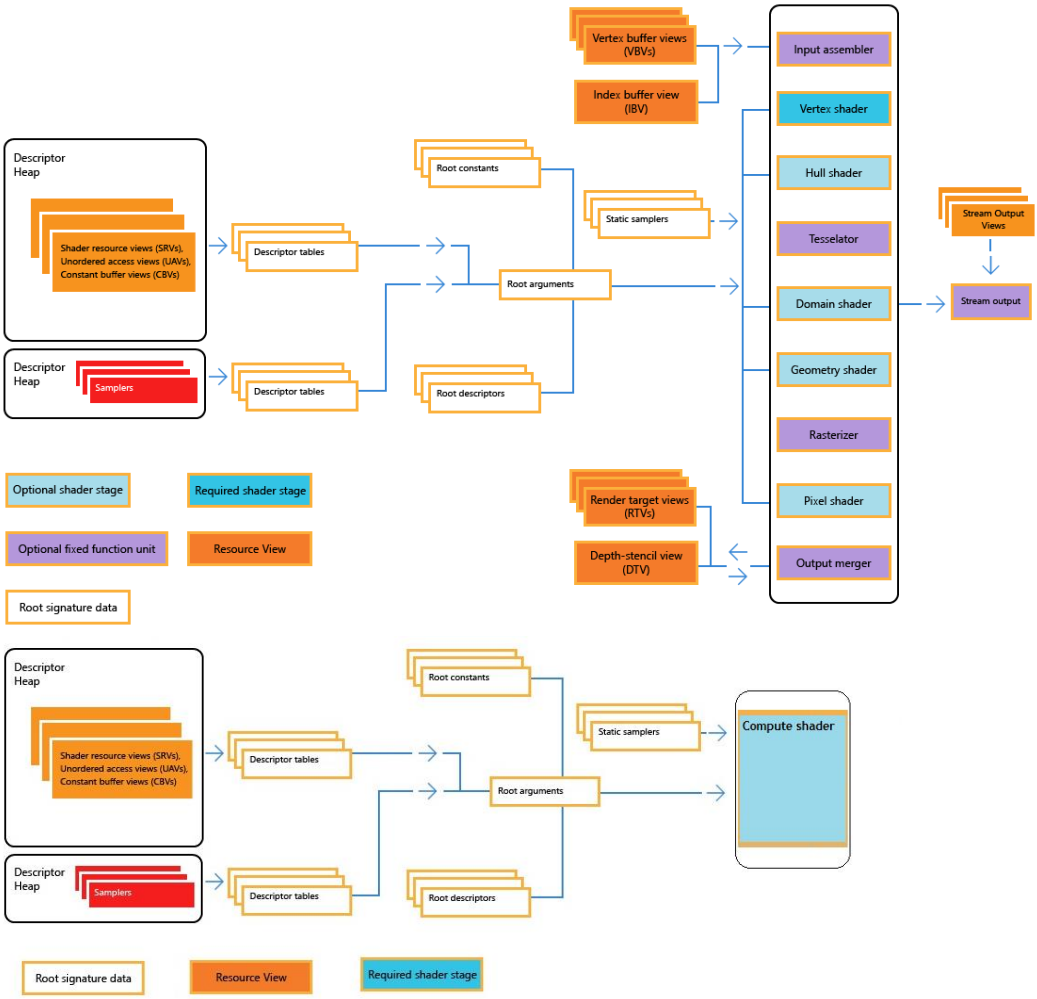
```
IDXGIAdapter4* GetAdapter(bool useWarp)
{
    IDXGIFactory4* dxgiFactory;
    unsigned int dxgiFactoryFlags = 0;
#ifdef _DEBUG
    dxgiFactoryFlags |= DXGI_CREATE_FACTORY_DEBUG;
#endif
    CreateDXGIFactory2Ptr(dxgiFactoryFlags, IID_PPV_ARGS(&dxgiFactory));

    IDXGIAdapter1* dxgiAdapter1 = nullptr;
    IDXGIAdapter4* dxgiAdapter4 = nullptr;

    if (useWarp)
    {
        dxgiFactory->EnumWarpAdapter(IID_PPV_ARGS(&dxgiAdapter1));
        dxgiAdapter1->QueryInterface(IID_PPV_ARGS(&dxgiAdapter4));
        dxgiAdapter1->Release();
    }
    else
    {
        SIZE_T maxDedicatedVideoMemory = 0;
        for(unsigned int i = 0; dxgiFactory->EnumAdapters1(i,&dxgiAdapter1) != DXGI_ERROR_NOT_FOUND; ++i)
        {
            DXGI_ADAPTER_DESC1 dxgiAdapterDesc1;
            dxgiAdapter1->GetDesc1(&dxgiAdapterDesc1);

            // Check to see if the adapter can create a D3D12 device without actually
            // creating it. The adapter with the largest dedicated video memory
            // is favored.
            if ((dxgiAdapterDesc1.Flags & DXGI_ADAPTER_FLAG_SOFTWARE) == 0 &&
                SUCCEEDED(D3D12CreateDevicePtr(dxgiAdapter1,D3D_FEATURE_LEVEL_12_1, __uuidof(ID3D12Device), nullptr)) &&
                dxgiAdapterDesc1.DedicatedVideoMemory > maxDedicatedVideoMemory)
            {
                maxDedicatedVideoMemory = dxgiAdapterDesc1.DedicatedVideoMemory;
                //TODO://ThrowIfFailed(dxgiAdapter1.As(&dxgiAdapter4));
                if(dxgiAdapter4!=nullptr)
                    dxgiAdapter4->Release();
                dxgiAdapter1->QueryInterface(IID_PPV_ARGS(&dxgiAdapter4));
            }
            dxgiAdapter1->Release();
            dxgiAdapter1 = nullptr;
        }
    }

    dxgiFactory->Release();
    //dxgiAdapter4->Release();
    return dxgiAdapter4;
}
```

In our case, to find our GPU we search for the device with the highest memory that will probably be the best GPU on the system. The DirectX 12 API allows us to create and use more than 1 device however I just have 1. Notice that we do the call to CreateDevice here but we pass a nullptr. This is just to check if the device supports the specified feature level. Feature levels are introduced by Microsoft to let each individual video card implement certain part of DirectX. The feature level just imply functionality implementation and not performance of the GPUs. The minimum feature level a DirectX12 driver will support is D3D_FEATURE_LEVEL_11_0. Once we retrieve the adapter we use it to create our ID3D12Device interface. Immediately after device creation, if we are on debug mode we create the debug layer using the ID3D12InfoQueue interface we can get from the Device.

```
    //Creating a device
    ID3D12Device2* device2 = nullptr;
    //if the adapter was null we would be using the first adapter returned by IDXGIFactory1::EnumAdapters
    ///*if (SUCCEEDED(*/D3D12CreateDevicePtr(adapter, D3D_FEATURE_LEVEL_12_2, IID_PPV_ARGS(&device2));
    /*if (SUCCEEDED(*/D3D12CreateDevicePtr(adapter, D3D_FEATURE_LEVEL_12_1, IID_PPV_ARGS(&device2));

#ifdef _DEBUG
    // Enable debug messages in debug mode.
    ID3D12InfoQueue* infoQueue;
    if(device2 != nullptr && device2->QueryInterface(IID_PPV_ARGS(&infoQueue)) != E_NOINTERFACE)
    {
        infoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_CORRUPTION, true);
        infoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_ERROR, true);
        infoQueue->SetBreakOnSeverity(D3D12_MESSAGE_SEVERITY_WARNING, true);

        //To ignore or supress some warnings or messages
        // Suppress whole categories of messages
        //D3D12_MESSAGE_CATEGORY Categories[] = {};

        // Suppress messages based on their severity level
        D3D12_MESSAGE_SEVERITY Severities[] =
        {
            D3D12_MESSAGE_SEVERITY_INFO
        };

        // Suppress individual messages by their ID
        D3D12_MESSAGE_ID DenyIds[] = {
            D3D12_MESSAGE_ID_CLEARRENDERTARGETVIEW_MISMATCHINGCLEARVALUE,   // I'm really not sure how to a
            D3D12_MESSAGE_ID_MAP_INVALID_NULLRANGE,                        // This warning occurs when usi
            D3D12_MESSAGE_ID_UNMAP_INVALID_NULLRANGE,                      // This warning occurs when usi
            //D3D12_MESSAGE_ID_CREATERESOURCE_STATE_IGNORED,                 // This warning occurs when c
        };
        D3D12_INFO_QUEUE_FILTER NewFilter = {};
        //NewFilter.DenyList.NumCategories = _countof(Categories);
        //NewFilter.DenyList.pCategoryList = Categories;
        NewFilter.DenyList.NumSeverities = _countof(Severities);
        NewFilter.DenyList.pSeverityList = Severities;
        NewFilter.DenyList.NumIDs = _countof(DenyIds);
        NewFilter.DenyList.pIDList = DenyIds;
        //TODO: ThrowIfFailed(infoQueue->PushStorageFilter(&NewFilter));
        infoQueue->PushStorageFilter(&NewFilter);

        infoQueue->Release();
    }
#endif

    //TODO: adapter->Release();
    return device2;
```

We set the debug layer to break an exception when errors, warnings and corruption happen during runtime. This is very useful to find runtime errors when debugging your code. If we consider that some messages are not of concern we can also deactivate them so the debug layer will not issue them anymore.

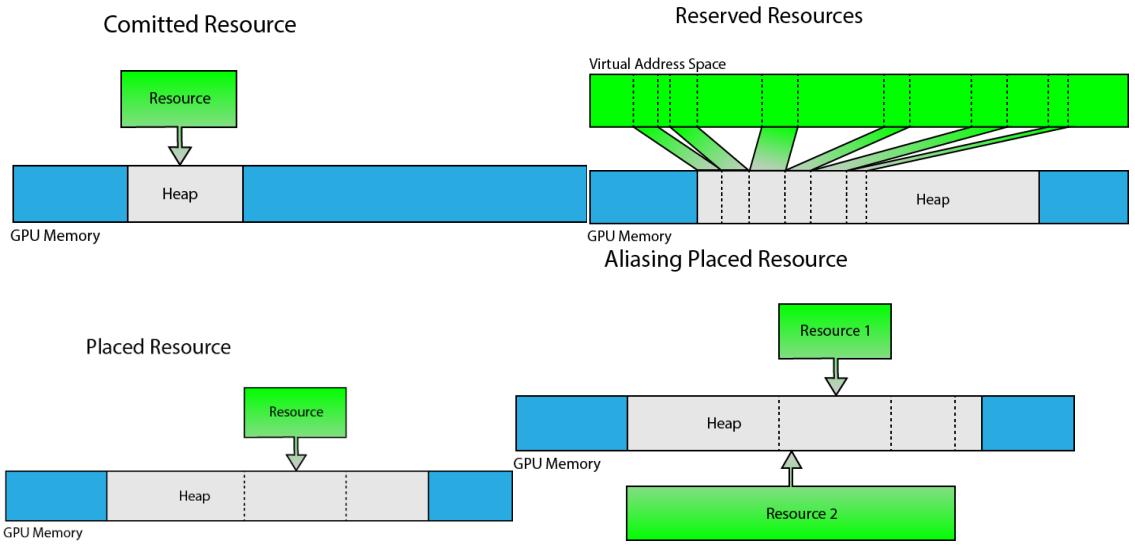
Once the device and debugger are setup we can start using the API to submit work to the GPU. As mentioned before, we will use the ID3D12Device interface to create the interfaces needed for this process. The work the GPU will execute is a pipeline of different stages executed one after the other. There are 2 kind of pipelines we can execute: the graphics or the compute pipeline. The graphics pipeline needs no presentation. Quickly introduced, it is the sequence of steps to transform the input data to the rendered frames to present to the screen. The other pipeline just has one step and can be used for all other purposes that could require GPU computing.




Some of this pipeline steps are programable by shaders that are GPU programs (the kernels in CUDA language). The other steps are fixed steps that the drivers implement.This different steps can have different configurations that define the state of the pipeline. The majority of this state (input assembler, rasterizer, pixel shader, output merger…) can be configurated with pipeline state objects (PSO/ID3D12PipelineStateObject).

**Resources**

The first thing we require to use this pipelines are the resources. The resources are the actual data that will be used by the pipelines. They can represent buffer data like vertex or index buffers or they can also represent textures. This resources are represented by the ID3D12Resource interface. All the resources data is contained on some heap (ID3D12Heap). The heap is an abstraction of a memory allocation. There are different ways that the GPU allocates resources within a memory heap.




The first are committed resources. They are created with an implicit heap with a call to the device routine *CreateCommittedResource*.

We also have the placed resources. For them we first have the heap already created and then we just reserve a portion of that heap memory for our resource. This way, the heap can contain more than one resource and we can specify the layout of the different resources on memory. Despite this resources being able to be aliased in the heap just one of them can be active at a time. This is useful to save memory if the resources are needed not needed at the same time when executing a shader.

The last resource type are the reserved resources. The reserved resources are like the placed resources but they can be not bound to a heap or they can bound just different parts of them. This means that they can be bigger in size than their containing heap.

We also have different type of heaps depending on its use. We have heaps to upload data to the GPU (CPU write once, GPU read once), heaps only accessible for the GPU or heaps to read data back from the GPU to the CPU.

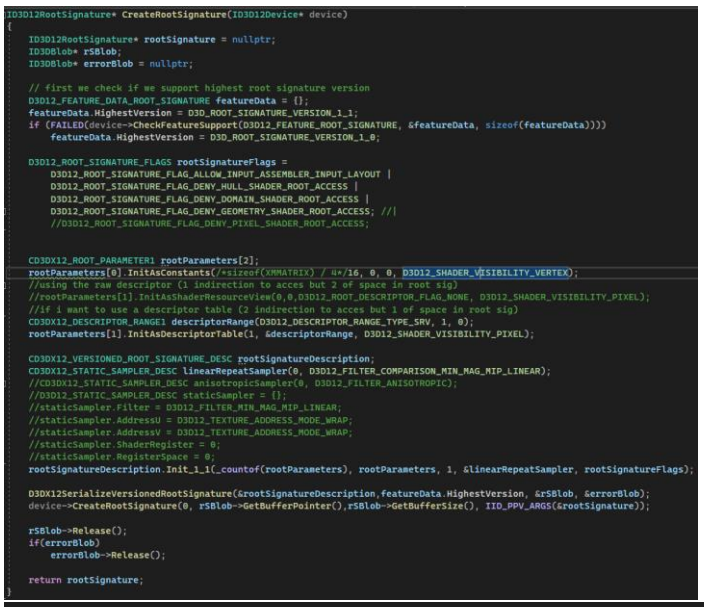
All this flexibility we have in DirectX12 with the different types of heaps and resources is possible because DirectX 12 requires the application to manage memory residency. A memory chunk is resident when the GPU has access to it. We are responsible to make sure the needed resources are resident while the GPU work is executing. This requires the use of CPU-GPU synchronization mechanisms (we will explain later).

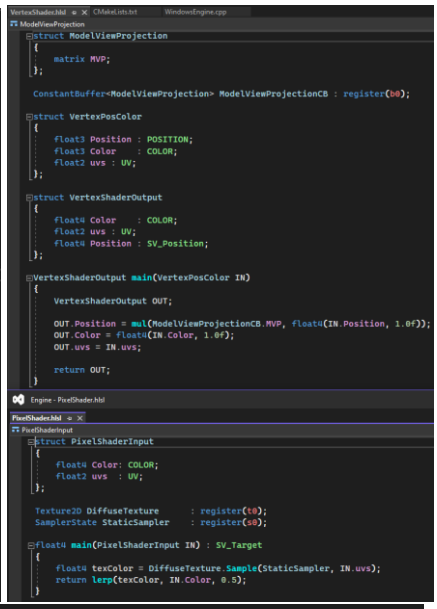
The resources can be on different states depending on how it is being accessed. For example, a texture can be accessed through a Shader Resource View or a Render Target View (Views are explained later). Previous to DirectX12, this state had to be tracked by the drivers on the background causing some overhead. With DirectX12 we have to track the state of resources using resource barriers. This barriers will transition the state of the resources or alias the different resources on the placed or reserved heaps. The resources must be on the correct state determined by the view descriptors bound to the pipeline while executing on the GPU, which again will require CPU-GPU synchronization (explained later).

**Resource Binding**

Binding is the process of linking the resource objects to the graphics or compute pipeline. It is the way to tell the pipeline where to find the resources and objects the shader will use and

information the driver needs about them at runtime. A descriptor or view is an opaque block of data that fully describes an object to the GPU (what it is and where to find it on memory). As resources, descriptors are also stored in a special kind of heaps called descriptor heaps (ID3D12DescriptorHeap). The descriptor heaps abstract the memory that stores these descriptors. As resources again, this memory can be accessible to the GPU or not. Therefore, it also requires careful monitoring and synchronization CPU-GPU (explained later) to make sure they are resident when the GPU needs them. Normally, we have CPU descriptor heaps with a lot of descriptors that get copied to small GPU resident descriptor heaps when they need to be used by a shader. The last descriptor related concept is the descriptor table. The descriptor table is an array of descriptors. Having explained the descriptors, the important object that informs what resources are bound to the pipeline is the root signature (ID3D12RootSignature). The root signature is like a function signature but for the shaders of a pipeline. It has root parameters that can be constants, descriptors or descriptor tables. The actual values we set to these root parameters are called root arguments and define the concrete resources or objects we are referring to when we will execute the pipeline. Root signatures can be defined with C++ or on the shader that will use them. Once serialized they are passed to the pipeline state object.



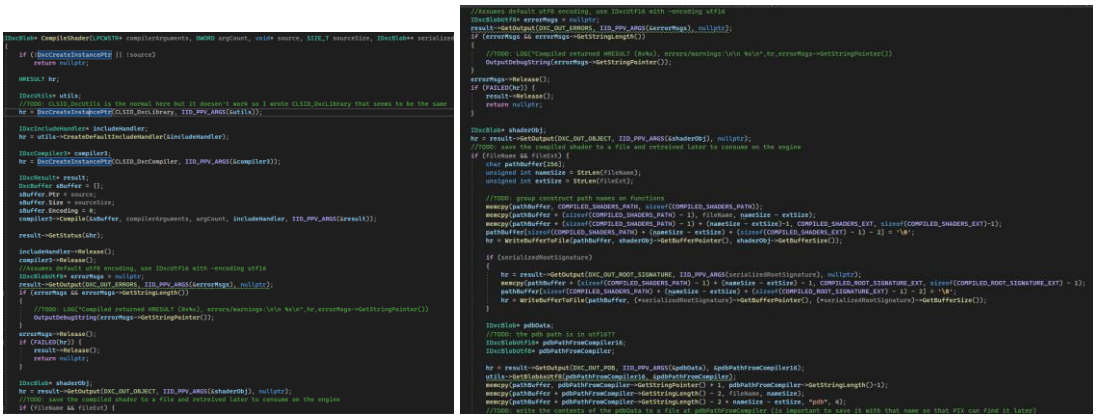


```
#define GenerateMips_RootSignature \
    "RootFlags(0), " \
    "RootConstants(b0, num32BitConstants = 6), " \
    "DescriptorTable( SRV(t0, numDescriptors = 1) )," \
    "DescriptorTable( UAV(u0, numDescriptors = 4) )," \
    "StaticSampler(s0," \
        "addressU = TEXTURE_ADDRESS_CLAMP," \
        "addressV = TEXTURE_ADDRESS_CLAMP," \
        "addressW = TEXTURE_ADDRESS_CLAMP," \
        "filter = FILTER_MIN_MAG_MIP_LINEAR)"
```

In those up images we can see the vertex and pixel shader I used and its corresponding C++ root signature and on the down image the root signature I used for my compute shader.

**Shaders**

We have been showing shaders and talking about them as the GPU source code for programs. Now I'll explain more accurately how they end up as GPU instructions following my implementation. DirectX shaders are written in High Level Shader Language (HLSL). These shaders source code is compiled by the DirectXCompiler(DXC). We can compile shader code in three ways. The first one is by using the executable of dxc on the command line with its respective arguments. Another way is to include the shader files in visual studio and they will automatically get compiled and the binaries will be put onto header files as variables we can use in C++. The last way is to use the API and dxcompiler.dll runtime library to make the application able to compile HLSL. I used the third option. The compiler library is again a COM style library.



The way it works is by first getting an instance of the compiler and making a call to compile. After that we just make repetive calls to GetOutput specifiyng what output we want (like the output compiled code DXC_OUT_OBJECT or the root signature DXC_OUT_ROOT_SIGNATURE).

After compilation, the source code is transformed to an intermediate language representation called DirectX Intermediate Language (DXIL) that all D3D12 drivers understand and can translate to their hardware specific instructions. This binary compiled code we just pass it to the pipeline state object.

**Work Submission to the GPU**

By now we have the resources and their bindings completed. The last thing to achieve is to submit the work to the GPU. To improve CPU efficiency Direct3D 12 does no longer support an immediate context with a device. Instead, the application records and submits command lists (ID3D12CommandList) which contain drawing and resource management calls. This command lists can be then submitted from multiple threads to one or more command queues (ID3D12CommandQueue). These changes objectives are the increase of single thread efficiency by allowing our app to pre-compute rendering work for later reuse and the ability to take advantage of multi-core CPUs as it can spread rendering work between different threads. To achieve these objectives again the app has to explicitly control when work is submitted to the GPU with CPU-GPU synchronization (we will very shortly explain this).

We can exploit even further the pre-computing of rendering code using bundles. Bundles are like a second level command lists but they cannot be submitted directly to a command queue (they have to go through a command list that calls *ExecuteBundle* to execute them). When a

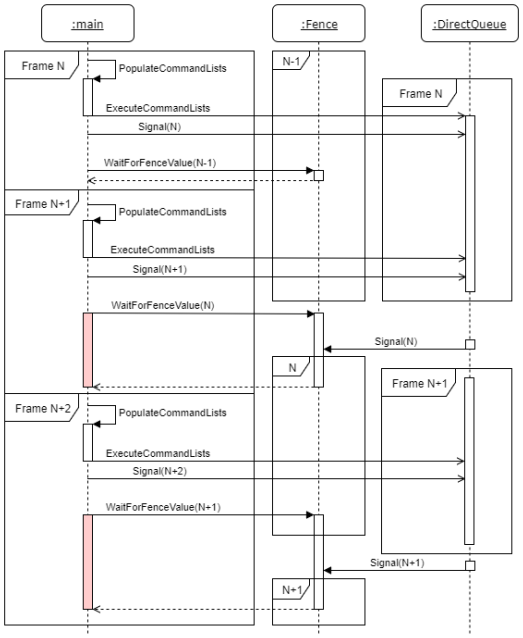
bundle is created the driver will perform as much pre-processing as possible to make it cheap to execute later. Bundles are designed to be re-used any number of times while command lists are typically executing just once because the application has to ensure that its previous execution has finished before being able to submit it again.

```
static bool first = true;
static ID3D12GraphicsCommandList2* bList = nullptr;

mvpMatrix = DirectX::XMMatrixMultiply(modelMatrix, viewMatrix);
mvpMatrix = XMMatrixMultiply(mvpMatrix, projectionMatrix);
if (first)
{
    first = false;
    ID3D12CommandAllocator* bAllocator = CreateCommandAllocator(renderer.device, D3D12_COMMAND_LIST_TYPE_BUNDLE);
    renderer.device->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_BUNDLE, bAllocator, nullptr, IID_PPV_ARGS(&bList));
    bList->SetPipelineState(loadedAssets.pipelineState);
    //bList->SetGraphicsRootSignature(loadedAssets.rootSignature);
    ////bind the matrix to the GPU
    //bList->SetGraphicsRoot32BitConstants(0, sizeof(DirectX::XMMATRIX) / 4, &mvpMatrix, 0);
    bList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    bList->IASetVertexBuffers(0, loadedAssets.numVertexViews, loadedAssets.vertexBufferViews);
    bList->IASetIndexBuffer(&loadedAssets.indexBufferView);
    //Draw Command
    bList->DrawIndexedInstanced(_countof(indicies), 1, 0, 0, 0);
    bList->Close();
}
```

Example of bundle recording. It is the same as the command list recording but it has less API avieble that are just aviable using command lists. We call *Close* when we finished recording the command list or bundle. After closing it we can submit the command list to a command queue or the bundle to a command list.

**CPU-GPU synchronization**

As we have seen through all the development process using DirectX 12 a lot of times the driver has eluded responsibilities to allow optimizations relaying on the application to correctly synchronize the CPU or multiple CPU threads and the GPU. To understand the situation, first we must have in mind that when we submit work to execute on the GPU using a command queue the CPU thread does not stop executing instructions waiting for the GPU to finish the work. The CPU just continues executing the next instruction after the command queue work submission. That is why we need this synchronization between the two hardware pieces. The way to synchronize them is to let them both continue executing until one of them requires something from the other to be finished. Accordingly, the hardware that requires for the other to finish some execution has to stop executing and wait until the other has finished the work of interest. After this happens, the one that just finished has to communicate to the other one that it already finished and that it can continue. The way for the CPU to wait for the GPU is to put the thread to sleep waiting for an event. This event will be signaled by the command queue when the GPU reaches the interested section. This coordination is achieved using a number that gets incremented continiously.



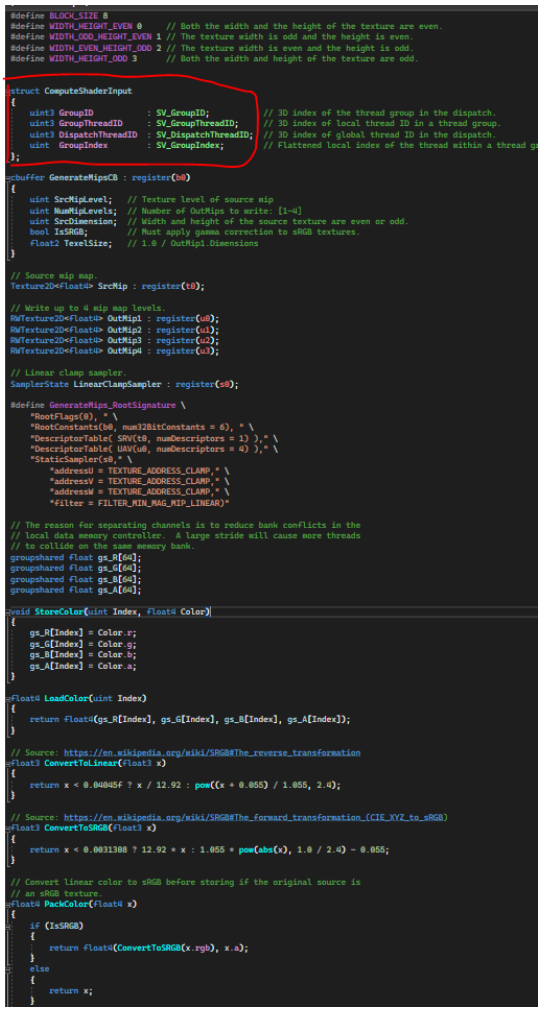

```
cList->Close();
ID3D12CommandList* const ppCommandLists[] = {cList};
renderer.cQueue.iCQueue->ExecuteCommandLists(1, ppCommandLists);
renderer.rtvFenceValues[currentBackBufferIndex] = ++cQueue.fence.value;
cQueue.iCQueue->Signal(cQueue.fence.iFence, cQueue.fence.value);
cQueue.cAllocators.Push({ cAllocator, cQueue.fence.value });
cQueue.cLists.Push(cList);
//present the frame
unsigned int sync = renderer.vSync ? 1 : 0;
unsigned int presentFlags = (renderer.tearingSupport && !renderer.vSync) ? DXGI_PRESENT_ALLOW_TEARING : 0;
renderer.sChain->Present(sync, presentFlags);
currentBackBufferIndex = renderer.sChain->GetCurrentBackBufferIndex();
unsigned long long fValue = renderer.rtvFenceValues[currentBackBufferIndex];
//wait for fence value
if (cQueue.fence.iFence->GetCompletedValue() < fValue)
{
    cQueue.fence.iFence->SetEventOnCompletion(fValue, cQueue.fence.event);
    WaitForSingleObject(cQueue.fence.event, INFINITE);
    PIXNotifyWakeFromFenceSignal(cQueue.fence.event);
}
```

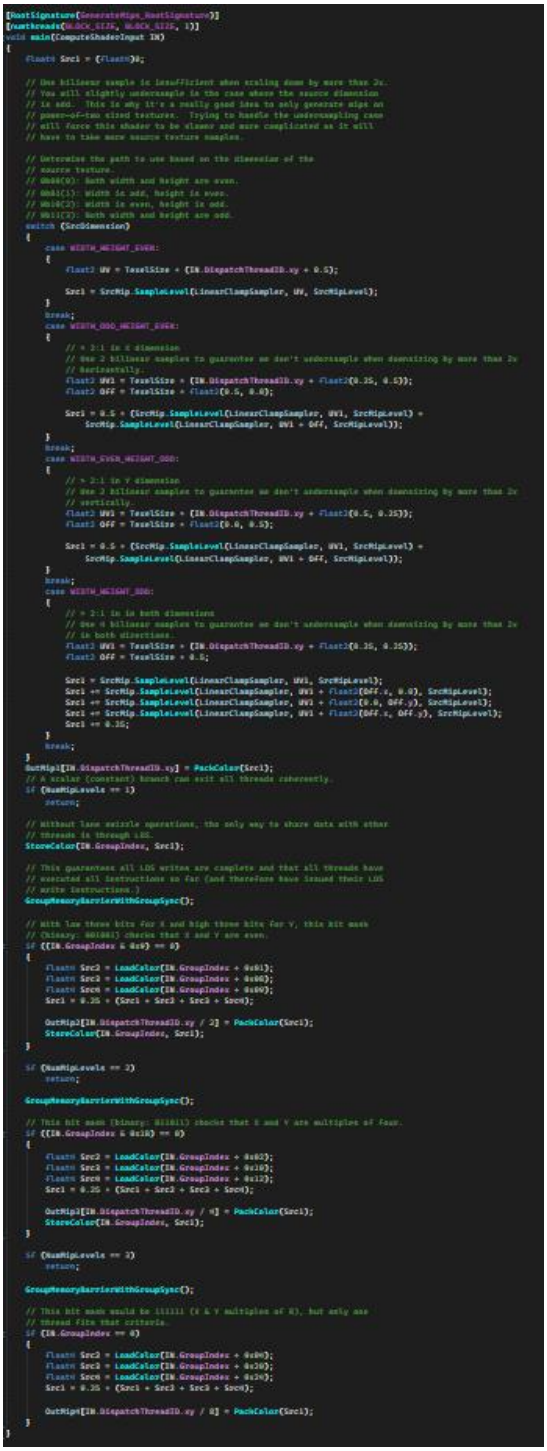
This is an example of my code synchronization between CPU and GPU when rendering.

**Compute pipeline**

Another task I performed was trying to use the compute pipeline. For that I decided to create a compute shader to generate texture mipmapping. To do this compute shader I first added a subset of the DirectXTex library with just the functionality I needed to load textures from files.

Mipmapping is the process of generating a series of images from a texture where each image in the series is half the size of the previous image. It is used in games to correctly filter textures in situations where a lot of texels (maybe the entire texture) only maps to one or few pixels. The filter on these kind of situations will not be good but if we have the mipmapps, the lowest size mippmaps can be filtered giving a perfect result very quickly.





Notice that as CUDA we alsor get the threadIdx and blockIdx variables but in other forms that we see on the ComputeShaderInput struct.

Final result.

**DirectXMath**

DirectXMath is an only header library that provides SIMD-friendly C++ types and functions for common linear algebra and graphics math operations. I used it to create the MVP matrix and to rotate the rendered cube. It is very comfortable to use because you just use the types and functions of the library without worring about alignment or SIMD and it will correctly use the SIMD instructions aviable on the platform.

```
DirectX::XMMATRIX modelMatrix;
DirectX::XMMATRIX viewMatrix;
DirectX::XMMATRIX projectionMatrix;
float angle = 0;
void OnUpdate()
{
    PIXScopedEvent(PIX_COLOR_INDEX(0), "Update");
    //TODO: chck for cpu avx support
    bool CPUSupportsInstructionSet = DirectX::XMVerifyCPUSupport();


    // Update the model matrix.
    //float angle = static_cast<float>(e.TotalTime * 90.0);
    ++angle;
    const DirectX::XMVECTOR rotationAxis = DirectX::XMVectorSet(1, 0, 0, 0);
    modelMatrix = DirectX::XMMatrixRotationAxis(rotationAxis, DirectX::XMConvertToRadians(angle));

    // Update the view matrix.
    const DirectX::XMVECTOR eyePosition = DirectX::XMVectorSet(0, 0, -10, 1);
    const DirectX::XMVECTOR focusPoint =  DirectX::XMVectorSet(0, 0, 0, 1);
    const DirectX::XMVECTOR upDirection = DirectX::XMVectorSet(0, 1, 0, 0);
    viewMatrix = DirectX::XMMatrixLookAtLH(eyePosition, focusPoint, upDirection);

    // Update the projection matrix.
    float aspectRatio = windowClientRect.right / static_cast<float>(windowClientRect.bottom);
    float fov = 60;
    projectionMatrix = DirectX::XMMatrixPerspectiveFovLH(DirectX::XMConvertToRadians(fov), aspectRatio, 0.1f, 100.0f);

}
```

As we can see on the header implementation of the library, it uses the SIMD instrinsics including the correct header depending on what is the maximum defined supported version for the compilation.

**Profile**

The last thing I did for the platform layer was to incorporate the PIX profiler. I included it because it is the one created by Microsoft but I did not use it much because I preferred the Nvidia Nsight tools. I just incorporated it to be able to profile the code without an Nvidia GPU. It is a code insturmentalization profiler.

# 6. Conclusion

This TFG had two main leading lines to reach conclusions.

Starting with the CUDA project we could see from a firsthand example the huge difference there is in time between computing a parallel algorithm on the CPU against computing it on the GPU. We also saw that the main point on optimizing a GPU kernel is the memory bandwidth. Accordingly, I can conclude that this is one of the main reasons why the GPU outperforms the CPU on parallel tasks as GPUs memory bandwidth is much higher than system RAM. Other important aspects to have in mind when optimizing any GPU kernel or shader is also the occupancy and the concurrency. With the occupancy and concurrency we also see another reason the GPU outperforms the CPU. The GPU can use the huge amount of cores at the same time with the millions of threads it spawns to hide instruction latencies and beat the CPU inspite of CPU lower instruction latency and higher clock frequency.

The platform layer didn't reach very far however we were able to get the required conclusions to the objectives we had. All the parallel techniques mentioned in the introduction were used creating the platform layer. For the GPUGP we got to see its detailed implementation on DirectX12 with the form of the compute pipeline. For the SIMD instructions we got to see them on the DirectXMath library. The only problem I found with their implementation on the library is that they do not use function dispatching at runtime. They are used when the compiler compiles the code enabling this instruction. As mentioned on the introduction, without function dispatching the program cannot adapt to use the best SIMD instructions because it gets build just targeting one of this instruction sets without having the implementation for the others. For multithreading, we find out that it is one of the main focuses of optimization for the DirectX12 API compared to previous versions. The rendering work can now be submitted from multiple threads. Therefore, you can use multithreading to optimize CPU code. We also discovered that most of the optimizations in DirectX12 related to the previous versions come from reducing CPU overhead at the cost of managing the memory residency and having to synchronize the CPU and the GPU.

Finally, I also discovered a new parallel technique in the form of DirectX12 letting us use 2 GPU at the same time.

## 6.1 Future lines

After this TFG, the most exciting expansion we can make is transforming the platform layer into a full game engine using multithreading for rendering. Another interesting thing we can do is optimize more kernels like the blur kernel I started. It still did not reach the times needed to be used in real-time at 8k. Finally, we can also try to explore how we could make runtime function dispatching with the DirectXMath library functions.

# 7. Bibliography

- Win32 Windows and messages: https://docs.microsoft.com/en-us/windows/win32/winmsg/using-messages-and-message-queues
- XAudio2: https://docs.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-introduction
- XInput: https://docs.microsoft.com/en-us/windows/win32/xinput/getting-started-with-xinput
- DirectXMath: https://docs.microsoft.com/en-us/windows/win32/dxmath/directxmath-portal
- COM: https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal
- DirectXTex: https://github.com/microsoft/DirectXTex
- Direct3D 12: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics
- DXGI: https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/dx-graphics-dxgi
- Agility SDK: https://devblogs.microsoft.com/directx/gettingstarted-dx12agility/
- HLSL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl
- DXC: https://github.com/microsoft/DirectXShaderCompiler
- Learning DirectX12: https://www.3dgep.com/learning-directx-12-1/
- CUDA: https://docs.nvidia.com/cuda/
- Nsight Compute: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html
- PIX: https://devblogs.microsoft.com/pix/introduction/