1st diapo (TODO)

TODO: Presentation: Ok, Let's start. The topic of this end of degree presentation is parallel game programing. So we can start by defining what is parallel programing.

2nd diapo (TODO)

What?

Parallel programming, in simple terms, is the process of decomposing a problem into smaller tasks that can be executed at the SAME TIME using multiple compute resources. For example, if we have to do 100 additions to do and we have 2 CPUs we could give 50 additions to each CPU instead of making one CPU do all the additions while the other one stays idle.

Why? TODO: answer all questions at the conclusions // is it called proportional growth the example of parallelism?

The main reason to explore parallel programing is for the promising huge performance bust we could get. As seen in the previous example, with two CPUs we could do the 100 additions two times faster than with just one. Adding a third CPU can make this addition program 3 times faster. And we can keep going. Is it really this powerful of an optimization on real hardware? Does it have any drawbacks?

Another reason is that until now I have programed a few games without knowing about parallelism. Is it really necessary? If I have programed x games without parallelization is it really needed? (Veurem k si amb el exe)

The last reason to dig into parallel programming is that today's hardware is improving with parallelism and not instruction processing speed. Not too long ago, a new CPU main improvement specification was the raise of its clock frequency. Higher clock frequency meant that it executed more instructions with less time. The problem with this approach recently is that **CPU clock frequencies stop getting as faster because of heat limit problems.** Meanwhile, following moores law, transistors in a die keep more or less doubling every 2 years. Those transistors can be used on a CPU to build more cores or parallel resources on the CPU.

On the previous addition example, we parallelized by using more than one CPU. Nowadays gaming PCs normally don't have more than one CPU. What really happens is that the CPU has more than one physical core to be able to execute different program instructions at the same time, in parallel. This physical cores also have hyperthreading technology which allows each physical core to act as two logical cores. Even more, nowadays CPUs are superscalar with more than one execution pipelines to be able to execute more than on instruction per clock cycle. This CPUs also have instructions that

work on more than one data element at a time. How on the software level can we use this parallel hardware?

Finally, to top all of this parallel hardware we have a coprocessor that is an entire minicomputer designed for parallelism, the GPU. How does it work? How can we program it? Is it different than traditional CPU programming? Is it only used for graphics? How much performance does it have compared to CPUs to justify its existence/need? And why this difference on performance?

How?

On this project I tried to find answers to the previous questions looking for information and documentation about different parallel techniques while at the same time I was developing a basic game or game engine windows platform layer. I developed the platform layer to see if I could use the techniques that I was documenting. Also, I developed the platform layer myself instead of using an open source library that abstracts that complexity like SDL or GLFW to be able to interact directly with the system low level APIs. This way I could find the tools the OS provides to use the parallel programing techniques I was documenting. I also wanted to use the directX12 API for rendering to compare it to OpenGL. I heard DirectX12 and Vulkan are much lower level API so I was curious if they allowed any parallelism not possible in OpenGl.

At the end, I documented three techniques that are multithreading, SIMD instructions and the GPU that I now will proceed to explain.

Multithreading

Multithreading is the software way to use the multiple CPU logical cores mentioned before. Each logical core executes a sequence of program instructions independently from the other cores. This sequence of instructions is called thread and each program has at least 1 of them.

The **scheduler** is the OS part that is in charge of managing all the different running processes threads. Its job is to distribute all the available threads to run onto the different CPU cores and also to make sure all the ready threads will eventually run according to their priority. Most personal computers operating systems like Windows or Linux are preemptive multitasking operating systems. Multitasking in the sense that they allow more than one process to run concurrently and preemptive because they assign time slices to the threads running on the different logical cores. Once the time slice for a thread finishes, the thread gets interrupted and the scheduler gets invoked again. The windows scheduler works using Round Robin algorithm in different queues of threads ordered by thread priority. The thread that has been interrupted gets preempted to the last of the queue of its priority and the first thread on the queue with highest priority gets scheduled to the core during the next time slice. The other model of multitasking would be cooperative multitasking. In this model there will not

be time slices and each thread would be responsible to stop themselves to allow other threads to run without stalling the cores.

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 some accesses 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 or trying to have more than one thread write to the same location. This can produce undefined results when performing the reading or writing operations that most probably will alter the program final result. 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. For example, if we start a thread that gets blocked waiting for another thread that it is also blocked waiting for the first one the program will never progress.

The windows API provides a set of wait functions and synchronization objects to **synchronize thread execution**. The synchronization objects have 2 states, signaled and non-signaled. This states are set an unset in different ways depending on the synchronization object. One example of synch object could be a mutex that can only be acquired by one thread at a time and when the thread doesn't need it anymore it calls the ReleaseMutex function making it signaled so that another thread can acquire it. Other examples of common synch objects are events and semaphores. The wait functions allow a thread to block its execution until a specified synchronization object is set to the signaled state. Apart from the synch objects, depending on the wait function we can also wait on different handles of system resources like other threads, file handles … To get familiar with different synchronization objects and wait functions I tested them in little programs like this one.

```
SYSTEM_INFO info;
DWORD threadId;
HANDLE hThread;
GetSystemInfo(&info);
//HANDLE hEve = CreateEvent(NULL, TRUE, FALSE, NULL);
HANDLE hMut = CreateMutex(NULL, FALSE, "Hi");
const char* message = "main thread!!\n";
HANDLE handles[5];
handles[0] = CreateThread(NULL, info.dwPageSize, DoWork, hMut, 0, &threadId);
handles[1] = CreateThread(NULL, info.dwPageSize, DoWork, hMut, 0, &threadId);
handles[2] = CreateThread(NULL, info.dwPageSize, DoWork, hMut, 0, &threadId);
handles[3] = CreateThread(NULL, info.dwPageSize, DoWork, hMut, 0, &threadId);
handles[4] = CreateThread(NULL, info.dwPageSize, DoWork, hMut, 0, &threadId);
WaitForMultipleObjects(5, handles, true, INFINITE);
WriteConsoleA(GetStdHandle(STD_OUTPUT_HANDLE), message, strlen(message), NULL, NULL);
//ReleaseMutex(hMut);
//SetEvent(hEve);
//CloseHandle(hEve);
CloseHandle(hMut);
return 0;
```

Multithreaded program can be slower than it originally would be with a single thread because of this different reasons:

overhead on thread code generation and destruction. The extra code to create a thread and then destroy it when exited is not required when processing the algorithm sequentially in one thread. If this overhead is higher than the actual performance improvement, we want to gain with the multithreading we are actually making the multithreaded option slower than the single threaded. One way to address this problem is with thread pools. **When we create a thread it gets automatically destroyed when the function it has been created with returns**. Creating and destroying threads every time we want to execute a function in not a good idea because it generates a lot of overhead. A way around to this problem is a thread pool**. A thread pool** creates the threads once and keeps them alive. The threads keep polling tasks from a pool of tasks to execute until there are no more tasks left to execute. Then, the threads get blocked until more tasks are posted to the queue. Windows provides an API for implementing thread pools. (also the standard library or you can implement one yourself).

Another overhead we have with multithreading is **context switching**. A context switch happens when the scheduler changes one thread for another because its time slice has finished or another thread with higher priority has awakened. This overhead is quite important and expensive in terms of performance. We need to save **all the context** the thread was running **in the stack?** and load the new context of the new thread. Also if the new scheduled thread is not from the same process we get some penalties with the caches and virtual memory because we need to flush the TLB.

Blocked threads with a lot of dependencies or bad synchronization. If an algorithm has a dependency on a memory resource like access to a data structure and we can't guarantee that it won't access the same elements (that actually works as sequentially but worse due to the need of synch)(for example an algorithm that both threads need to access the same vector constantly and we use a mutex. While one thread is working the other one just has to wait to access the vector so we are not really working on parallel. This way the algorithm will be slower than sequentially because we will have all the overheads mentioned previously.)

a lot of threads at the same time means we don't do significant progress on them individually…

This extra complexity generated when troubleshooting these problems makes the program development slower and more difficult to program, debug and profile. Making it unnecessary to use unless we have performance bottlenecks that can be specifically solved with multithreading and you properly profile the improvement.

SIMD

Moreover, **threads are not the only way to parallelize a program on the CPU**. 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.

You have **3 options to use these instruction extensions**. The easiest one is to specify to the compiler by flag the extension supported. Then the compiler will use the instructions if it finds it adequate. However, this way you don't have control of the instructions and maybe what the compiler generates is not what you wanted. In those cases, what you need is to use are or compiler intrinsics or directly assembly language. Intrinsics are special function already known by the compiler that will be directly replaced by concrete assembly instructions when compiled. To use them with C++ you just include the immintrin.h header and you also set the compiler flag to the instruction set you are targeting. Show the intel intrinsics page(https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#). For the assembly language compile by an assembler to object file and link normally. Define the assembly function we want to use in C++ as ("extern C") to don't have problems with name mangling.

Important quick note when developing with them, be careful with the data alignment. It is quite important to not have to read more than one cache line to complete the instruction.

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 still don't support them. Quick note(For example), AVX-512 was supported by 11 generation intel CPUs and not by AMD until the new ryzen 7000 generation released a few days ago (27 setembre 2022).

One way to address this problem when targeting x86-64 ISA is to use just up to the **SSE4 extension**. All CPUs that use this ISA support at least up to this level of vector instructions. In fact, if compiled for this target the compiler will automatically use them if required even though no extension has been specified in compilation flags. Additionally, on x64 windows when calling a function, the 128-bit xmm registers are used to push the floating point parameters of a function into the stack. However, this way we don't have access to the most modern vector extensions, the AVX.

Consequently, another option is 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. (example with directX input)

In the platform layer I developed, the DirectXMath used the SIMD instructions. (The library is a header library that uses compiler intrinsics.

GPU

The **GPU is** a coprocessor alongside the CPU to help it with the parallel code.

CPUs were primarily designed to execute serial code - and extract maximum parallelism out of them to improve performance. GPUs on the other hand are purpose built parallel computers which are fed parallel workloads.

The GPU is designed to maximize instruction throughput (number of instructions completed in a time period) instead of focusing on instruction latency (the time it takes to complete 1 instruction).

Its **main differences with the CPU** is the number of cores to maximize this instruction throughput, a slower clock frequency and a much higher memory bandwidth to be able to feed the data to all those cores. GDDR memories are SDRAM memories like CPU system DDR RAM but they are faster mainly because they have wider interfaces, they are closer to the main chip and they run at higher clock speed. While CPU memories deliver mid tens of GB/s, with the newest reaching maximums of 100GB/s, GPU memory bandwidths are much higher. For example, my home GPU, Nvidia RTX 2080 super, reaches 500 GB/s and the newest 4090 ones that will come out to market in two days reach the TB/s bandwidth.

To really understand the gap between CPU and GPU at computing parallel workloads and learn how to program the GPU I decided to **create and benchmark two programs** that do exactly the same parallel task but one with the CPU and the other with the GPU. The program takes an RGB image and outputs a grayscale image.

```
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;
    }
}
```

For the GPU program I used CUDA. It is a C++ language extension that allows you to create and call special functions named Kernels that will run on the GPU. To mark that a function is a GPU kernel we simply define the __global__ before the function. The GPU will execute this Kernels following the SIMT (single instruction multiple thread) model.

In this algorithm, each iteration of the CPU for loop is independent from the other ones. Each iteration does the same operations but on different data. Therefore they could all be performed at the same time. Simply explained, what the GPU will do is to spawn one thread for each one of the for loop iterations and make each threads run

on one of the thousands of cores it has at the same time. So, the more cores the GPU has, the more threads we can execute at the same time and the faster we will finish.

This threads are a little bit different than CPU threads. They all run the same program and they don't have a stack because all the needed variables stay on registers during the entire execution lifetime of the thread. So we don't even need context switching. This threads also get executed on groups of 32 called warps in the way that a scheduler selects one instruction to run for the 32 threads if they haven't diverged. At the same time, warps are further grouped into different blocks of size decided by the programmer. Recommended size of blocks is multiples of 32 to be able to divide the block into exact warps. This blocks of threads will get scheduled onto the different streaming multiprocessors and will run until completion. As the threads of the same block are garanted to run on the same SM they can share data between them through the cache of the SM and they can also synchronize execution. Threads of different blocks can't share data nor synchronize in any way. This sharing of data is important because differently to CPU programming, with the GPU we can program a portion of the cache called shared memory and we can use it within threads of a block. Finally, all the blocks of a kernel are grouped into what is call a grid.

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).

LETS TRY IT HERE (si la GPU es nvidia)

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.

After that I did a **blur algorithm** to test programing using the **shared memory**. Once I dominated the shared memory I did my **last CUDA test that was optimizing the first**

Grayscale algorithm to grasp even more the GPU hardware. We achieved optimizing the kernel a 40% related to the first one. The process of optimization is documented in detail on the project. It requires a long explanation as it goes deep into my concrete GPU hardware so I'm not going to explain it now. We can maybe go back to it if we have time latter or if you have any question related to it at the end of the presentation. What I can explain is the conclusions extracted of what are the key parts on optimizing a GPU kernel. The most important aspect is memory. 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. What that means is that the max bandwidth of the device is not enough to support all the requests to fulfill the GPU specs. Consequently, our objective is to access adjacent memory locations as much as possible to get the peak memory bandwidth. This access pattern on memory is called a coalesced access to memory. The second most important aspect is occupancy. Occupancy is the measure of how many blocks I can fit into a SM. The number of blocks that fit is limited by the number of registers needed by each block, the threads of each block and the amount of shared memory it requires. It is important that the SM has the highest possible number of blocks to be able to hide instruction latencies. Finally, the last thing to look out for is concurrency. Concurrency between kernels allows us to fit blocks from other grids to the SMs when blocks from the same kernel don't fit anymore. Notice than blocks from the same kernel always have the same requirements so maybe a block from the same kernel doesn't fit but one with different requirements from another grid maybe fits.

DirectX12

The problem we have with CUDA is that it is not portable to systems that don't have a Nvidia graphics card. Also CUDA is used for GPUGP but doesn't provide the graphics pipeline or any support for rendering. That is why in our platform layer we don't use CUDA to program the GPU but we use DirectX12. DirectX12 allows us to interact with the GPU using shaders with the HLSL programing language. The shaders are translated to an intermediate language DXIL (DirectX intermediate language) when compiled by DXC (DirectX compiler) and embaded to the executable. This intermediate language that is understood by all directX12 drivers, is then translated at runtime by each GPU dirver to the concrete instructions needed by the specific GPU. This shaders are programmable steps of the graphics or compute pipeline. The graphics pipeline needs no introduction. We input geometry through the input assembler and it outputs the render on a target buffer. The compute pipeline just has one stage and it allows us to use the GPU for other tasks not related to rendering called GPUGP. What makes DirectX12 so different from OpenGl and previous versions of DirectX is managing memory residency, resource binding and the work submission to the GPU.

Memory residency is managing when the resources are accessible by the GPU. Resource binding is telling the GPU what are the resources needed for the shaders and where they are. This is done using descriptors that are opaque blocks of data that

describe an object to the GPU. We use what is call a root signature that is like a function signature that specifies which descriptors the pipeline are using on that particular call. Work submission on the GPU is done through command lists and command queues. Command lists record rendering commands by calls to the API. This lists are then submitted to the command queues that will submit the work to the GPU. This structure allows to precompute render work and also allows doing rendering work on multiple threads as the command lists can be recorded from multiple threads.

All of this characteristics different from OpenGL come thanks to the fact that directX12 eludes the responsibilities of synchronizing the CPU and the GPU relying on the application to do it. This synchronization is done through the command queue using event synchronization objects and the wait functions we used on the multithreading. (also use fences)

The last thing I did for the project was using the compute pipeline. We did a mipmapping algorithm.

And after all the work we get… the spinning textured cube. (show the program)