TDT4200 Parallel Programming

Bart van Blokland

Lecture 8

&group

Please join the reference group!

Last week

```
if(
```

```
) {
```

```
();

}
```

Last week, we saw how race conditions can lead to problems, and that they can be difficult to find and debug

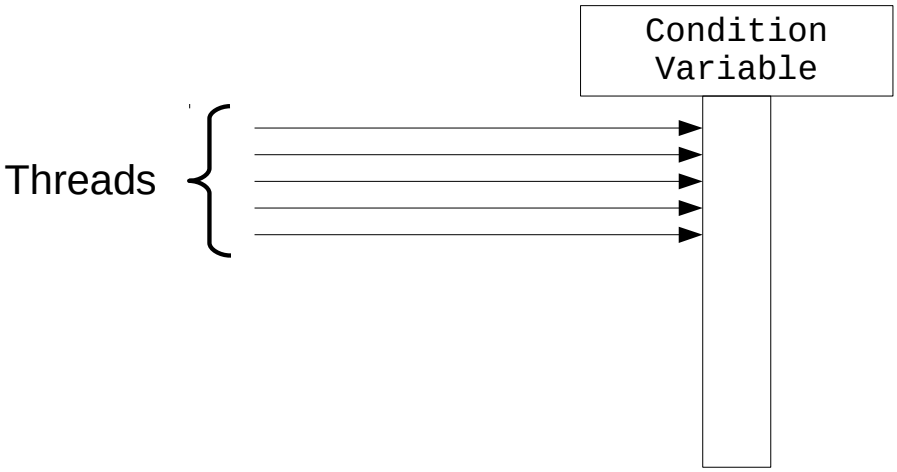
Condition Variables

One thing I didn’t fully explain was condition variables.

Three operations:

Wait
Notify one
Notify all

Threads sleep while waiting.
No busy waiting!



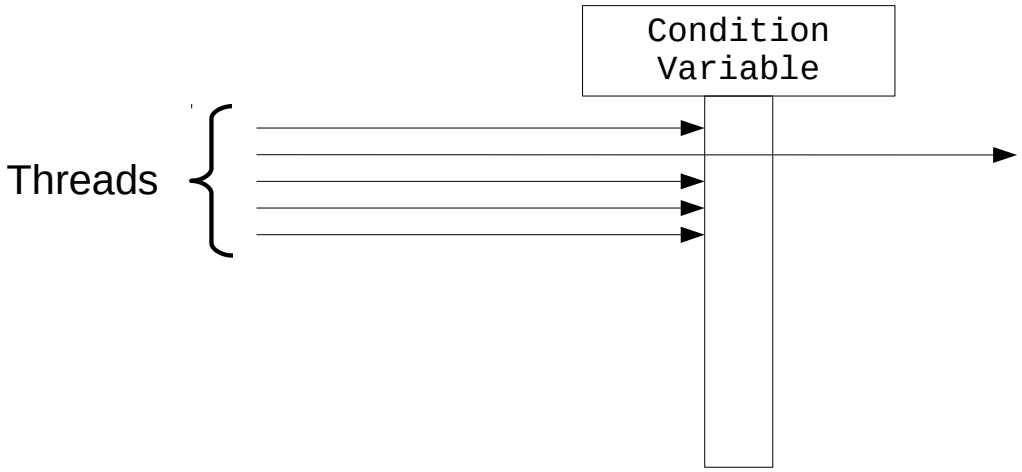
To repeat:

Condition variables have three operations: wait, notify one, and notify all.

Waiting on a condition variable causes the thread to go to sleep. The operating system handles waking it up from here on out. This avoid having the thread do busy waiting, consuming CPU resources unnecessarily.

Three operations:

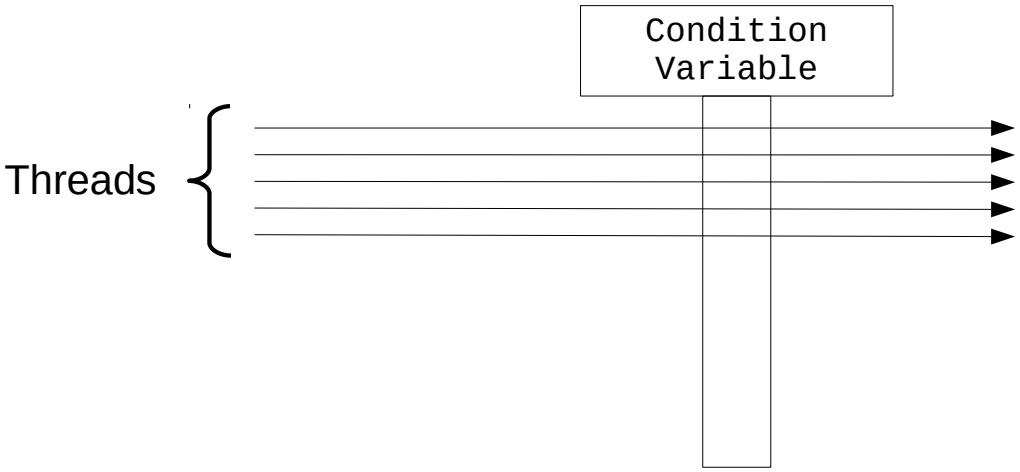
Wait
Notify one
Notify all



Notify one lets one thread through. Usually this is first come first serve to make sure all threads can go through eventually.

Three operations:

Wait
Notify one
Notify all



Notify all opens the floodgates and lets all threads loose.

```
int main() {
    std::condition_variable cv;
    std::mutex mut;
    std::thread threads[thread_count + 1];
    for(int i = 0; i < thread_count; i++) {
        threads[i] = std::thread(await, &cv, &mut, i);
        threads[i].detach();
    }
    threads[thread_count - 1] = std::thread(gatekeeper, &cv);
    threads[thread_count - 1].detach();
    std::this_thread::sleep_for(std::chrono::seconds(thread_count));
}
```

This bit of code sets up the demonstration of the condition variable. Note that I used detach() here, rather than join(). Normally you always want to use join(), but because in this case one of the threads was completing so quickly I had to use detach() to make sure the program did not crash due to one of the threads finishing early before the main thread could join it.

```
int main() {
    std::condition_variable cv;
    std::mutex mut;
    std::thread threads[thread_count + 1];
    for(int i = 0; i < thread_count; i++) {
        threads[i] = std::thread(await, &cv, &mut, i);
        threads[i].detach();
    }
    threads[thread_count - 1] = std::thread(gatekeeper, &cv);
    threads[thread_count - 1].detach();
    std::this_thread::sleep_for(std::chrono::seconds(thread_count));
}
```

I use detach() here because threads were exiting too quickly. You should normally always use join()!

```
#include<thread>
#include<mutex>
#include<condition_variable>
#include<chrono>
#include<iostream>

const int thread_count = 5;

void await(std::condition_variable* cv,
          std::mutex* mut, int id) {
    std::unique_lock<std::mutex> conditionVariableLock(*mut);
    (*cv).wait(conditionVariableLock);
    std::cout << "Thread " << id <<
                 " was allowed through!" << std::endl;
}

void gatekeeper(std::condition_variable* cv) {
    for(int i = 0; i < thread_count; i++) {
        (*cv).notify_one();
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}
```

The await() function shows how to force a thread to wait on a condition variable. First, we need to lock a mutex (created in the main() function). This mutex is meant to protect the condition variable itself against race conditions. Creating a unique_lock does just that. We subsequently call the condition variable's wait() function. This causes the thread to go to sleep until the condition variable's notify_all or notify_one() function is called.

The bottom function shows how to notify one thread at a time. The sleep_for function causes the thread to sleep for one second.

Lock state: wait()

```
std::unique_lock<std::mutex> conditionVariableLock(*mut);

(*cv).wait(conditionVariableLock);
```

unique_lock creation locks the mutex

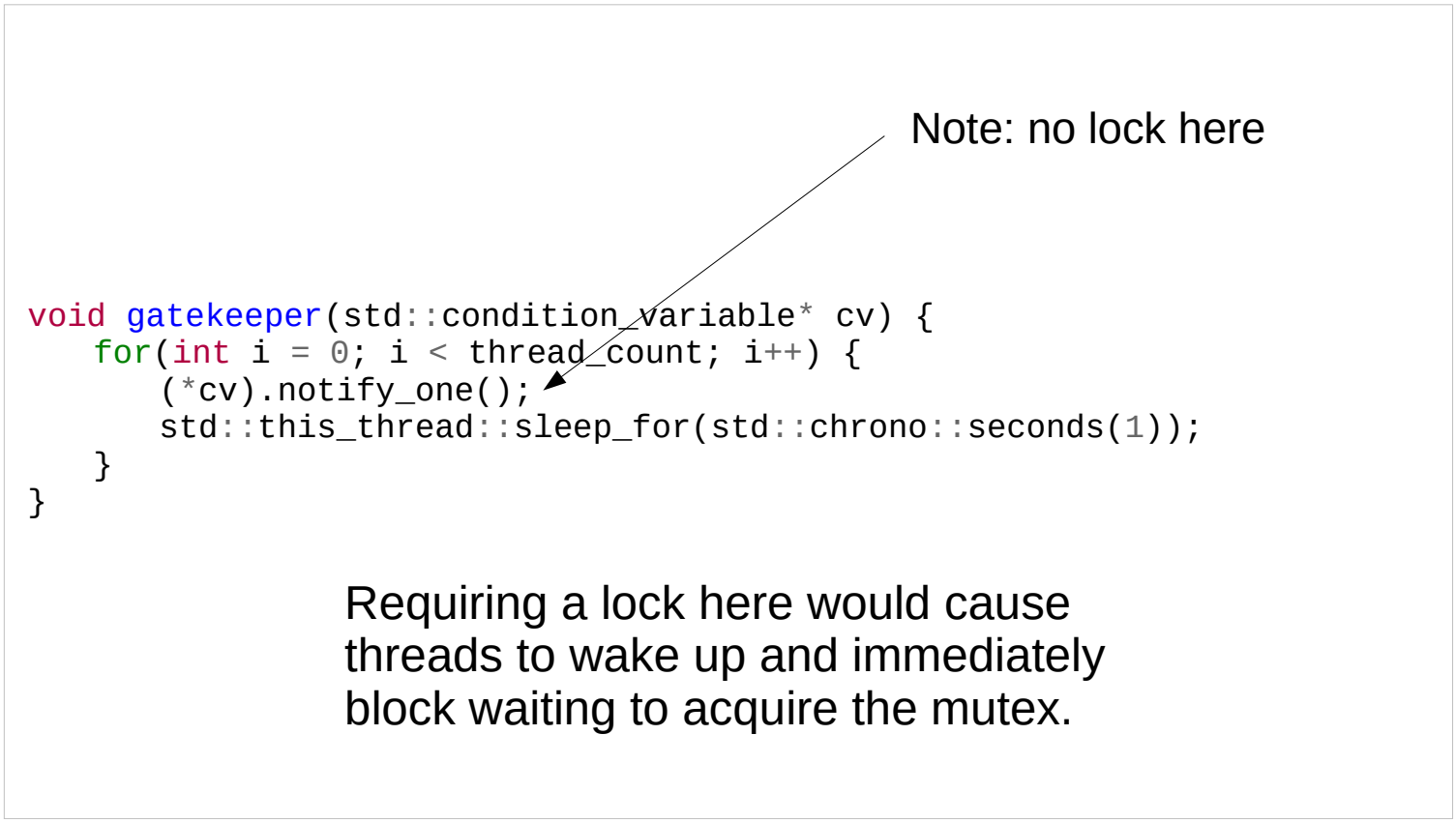
wait() puts the thread to sleep, then unlocks the mutex

After the thread is woken up, it reacquires the lock

It’s also worth noting what the state of the mutex is. The condition variable’s mutex is locked when the std::unique_lock is created (per its specification).

However, once the thread has finished initiating going to sleep, the mutex is unlocked again. This means another thread can acquire the mutex and, if it wants to, wait on the condition variable.

Once threads are notified and wake up, they will try to reacquire the mutex.



From what I can tell there is no need to lock the condition variable when notifying threads. If you do, the threads will wake up and have to go to sleep again as the mutex is not available. This can make a performance difference in some cases.

OpenMP

Next up, OpenMP!

why OpenMP?

```
int main() {
   const unsigned long count = 10000000000l;

   unsigned long* squares = new unsigned long[count];

   for(unsigned long i = 0; i < count; i++) {
      squares[i] = i * i;
   }

   return 0;
}
```

~ 3.8s

Let’s look at why we would want to use OpenMP. Here’s a piece of code that takes about 3.8s to execute on my machine.

```cpp
#include <thread>

const unsigned long count = 10000000000l;
const unsigned int threadCount = 10;
const unsigned long countPerThread = count / threadCount;

void computeSquares(unsigned long* squares, int start, int end) {
    for(unsigned long i = start; i < end; i++) {
        squares[i] = i * i;
    }
}

int main() {
    unsigned long* squares = new unsigned long[count];
    std::thread threads[threadCount];

    for(int i = 0; i < threadCount; i++) {
        threads[i] = std::thread(computeSquares, squares, i * countPerThread,
                                                          (i + 1) * countPerThread);
    }

    for(int i = 0; i < threadCount; i++) {
        threads[i].join();
    }

    return 0;
}
```

~ 0.7s (5x faster)

We can run this code in parallel, as we have done before. This causes it to run in 0.7s instead.

```
#include <omp.h>

int main() {
   const unsigned long count = 10000000000l;

   unsigned long* squares = new unsigned long[count];

   #pragma omp parallel for
   for(unsigned long i = 0; i < count; i++) {
      squares[i] = i * i;
   }

   return 0;
}
```

~ 0.7s (5x faster)

Here’s what parallelising this snippet looks like using OpenMP. Just a single line of code!

```
g++ ompsample.cpp -o omp -fopenmp
```

If you want to compile that snippet, you need to enable OpenMP with a special command line parameter

- Not necessarily faster

+ Easier to implement multithreading

+ Easier to read

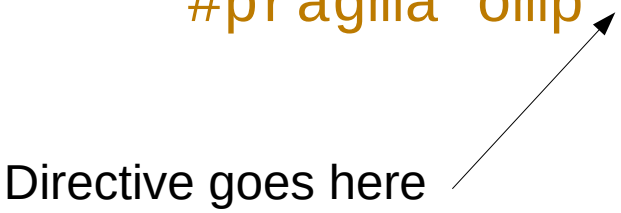
+ Portable

So OpenMP is not necessarily faster, but it makes implementing parallel code a lot easier, which improves readability. It also improves portability of code between platforms.

How do we use OpenMP?

So how do we use it?

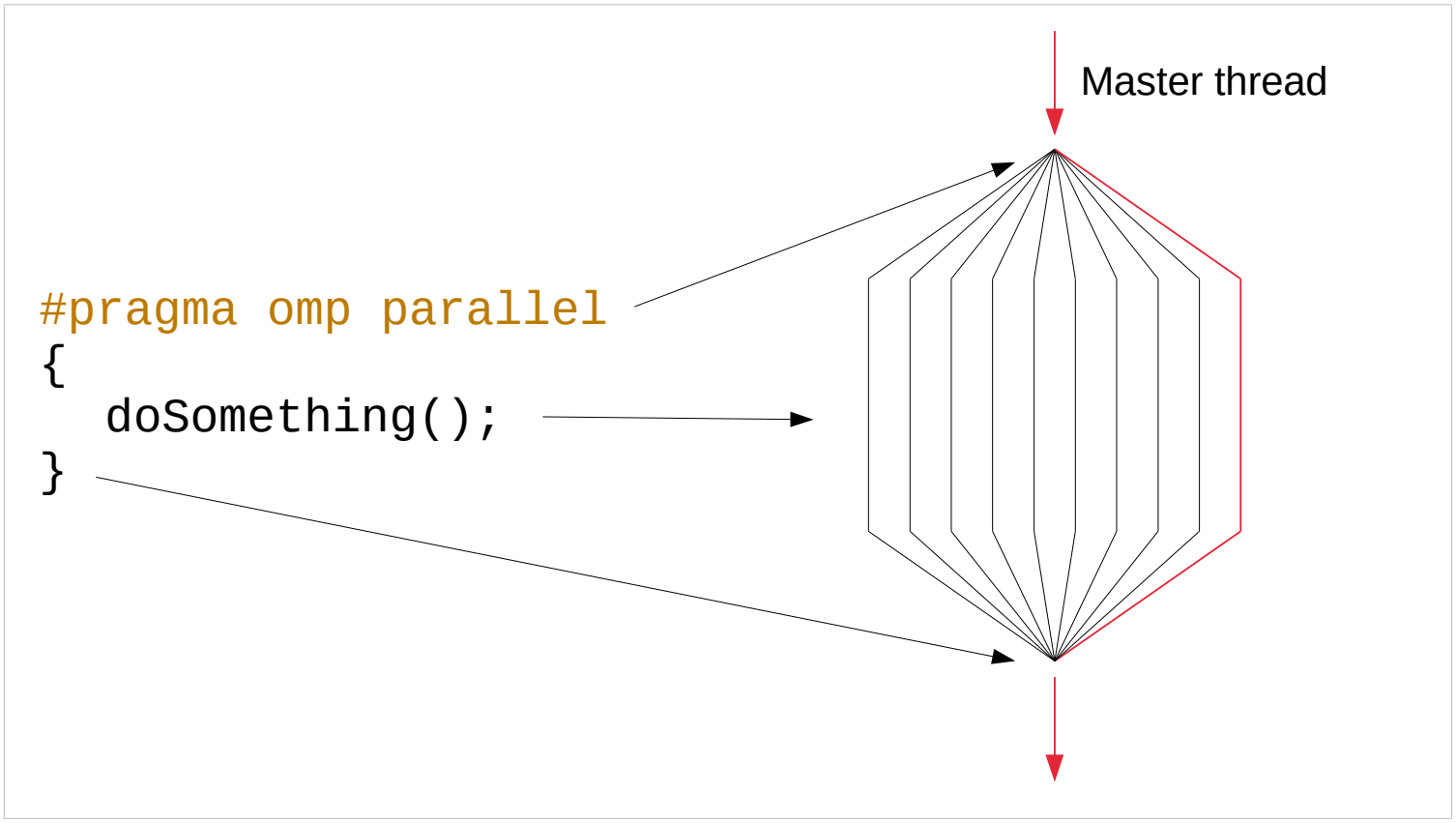
```
#pragma omp
```



Pragmas are compiler directives. They don’t do anything, unless the compiler supports the particular feature they are using. In this case, only if the compiler supports OpenMP (and has it enabled), will the #pragma omp be interpreted as an OpenMP directive. Otherwise, the directive has no effect.

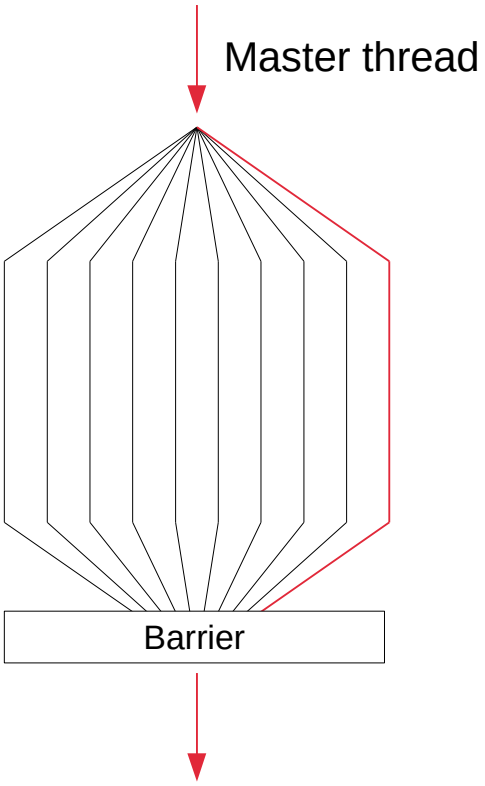
```
// Usage examples

#pragma omp parallel
{
   doSomething();
}


#pragma omp parallel
doSomethingElse();
```

Here’s how to use pragmas to refer either to multiple lines of code (top example), or just a single one (bottom example). You can also use the bottom variant on loops.

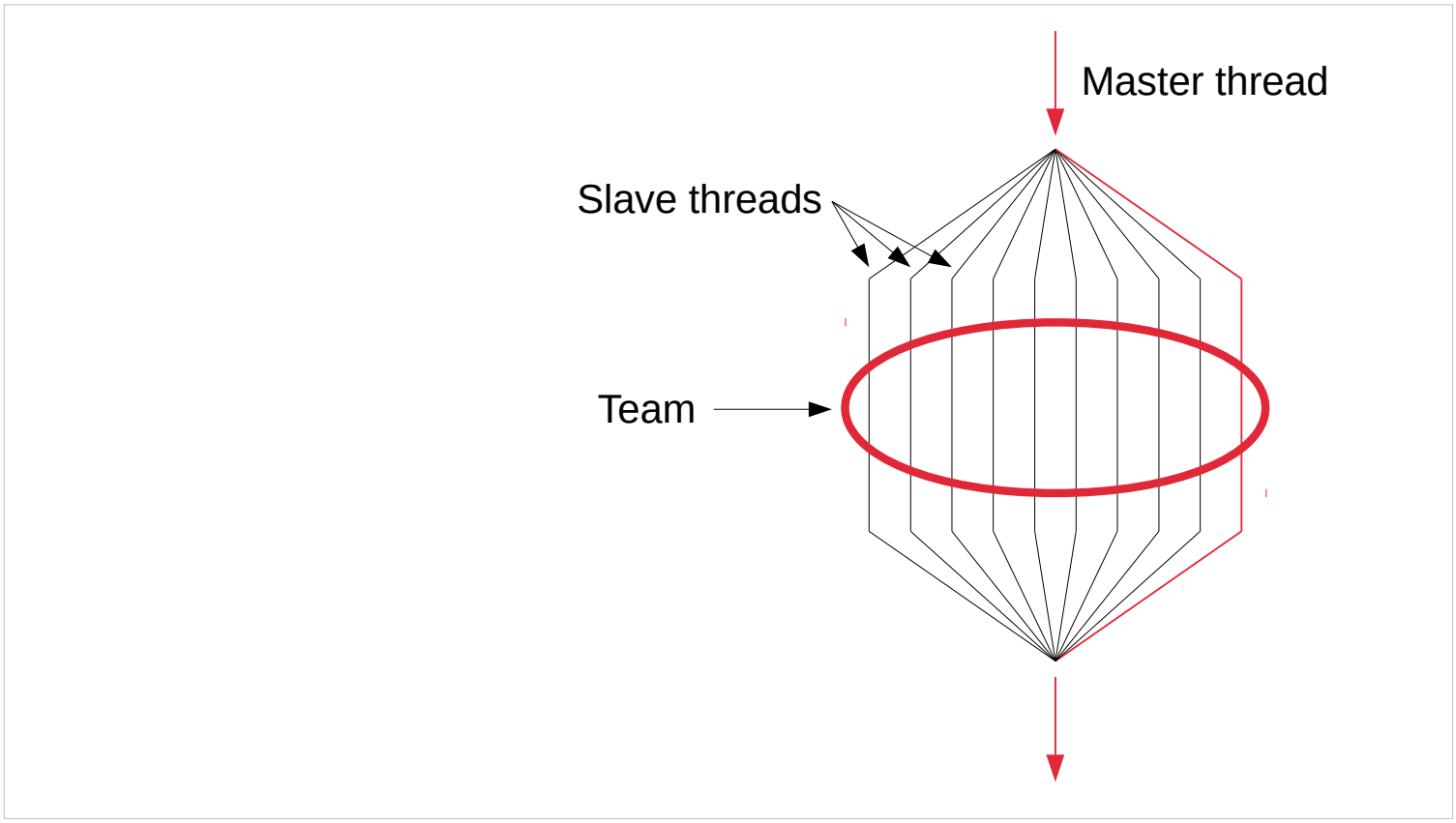


In a parallel block, a number of threads are spawned at the start. The “master thread” spawns a number of ~~thralls~~ ~~slaves~~ associates.

```
#pragma omp parallel
{
   doSomething();
}
```



After each parallel section there is an implicit barrier.



Some terminology. A master thread spawns some slaves, which together with the master thread are called a team.

So much for “there’s always a master and an apprentice”.

```
#include <omp.h>
#include <iostream>

int main() {

    #pragma omp parallel
    {
        std::cout
            << "Thread " << omp_get_thread_num()
            << " out of " << omp_get_num_threads() << std::endl;
    }

    return 0;
}
```

You can figure out which thread is which using the omp_get_thread_num() function. You can also figure out the size of the team using omp_get_num_threads()

```
#include <omp.h>
#include <iostream>

int main() {

    #pragma omp parallel num_threads(5)
    {
        std::cout
            << "Thread " << omp_get_thread_num()
            << " out of " << omp_get_num_threads() << std::endl;
    }

    return 0;
}
```

Normally, the number of threads spawned is equal to the number of cores you have. However, you can force a different number if you want.

Scope

Let us take a *closer look* at how scope interacts with OpenMP threaded code.

Scope

```
#include <thread>

void doSomething(int* outside) {
   int inside = 0;
}

int main() {
   int outside = 0;
   std::thread(doSomething,
               &outside);
   return 0;
}
```

```
#include <omp.h>

int main() {
   int outside = 0;
   #pragma omp parallel
   {
      int inside = 0;
   }

   return 0;
}
```

These examples are conceptually equivalent.

Scope

Variables declared outside the parallel section are shared between all threads

Variables declared inside the parallel section are local to each thread

```
#include <omp.h>

int main() {
   int outside = 0;
   #pragma omp parallel
   {
      int inside = 0;
   }

   return 0;
}
```

Scope

```c
#include <omp.h>

int main() {
    int count = 0;
    #pragma omp parallel
    {
        for(int i = 0;
                i < 100000; i++) {
            count++;
        }
    }

    return 0;
}
```

```c
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int count = 0;
        for(int i = 0;
                i < 100000; i++) {
            count++;
        }
    }

    return 0;
}
```

If you want to count with a common variable, make sure to declare it outside the parallel section. Otherwise, each thread will work on its own copy, and the result is not accumulated.

Although for this particular case you'd ideally want to use a reduction (more on those later).

Scope

```
#include <omp.h>
#include <iostream>

int main() {

    int count = 0;

    #pragma omp parallel default(none) private(count)
    {
        for(int i = 0; i < 100000; i++) {
            #pragma omp atomic
            count++;
        }
    }

    std::cout << count << std::endl;

    return 0;
}
```

Count = 0

You can explicitly control whether a variable defined outside a parallel section is private to the thread, or shared between all threads.

Scope

```
#include <omp.h>
#include <iostream>

int main() {

    int count = 0;

    #pragma omp parallel default(none) shared(count)
    {
        for(int i = 0; i < 100000; i++) {
            #pragma omp atomic
            count++;
        }
    }

    std::cout << count << std::endl;

    return 0;
}
```

Count = 100000

Scope

```
#include <omp.h>
#include <iostream>

int main() {

    int count = 0;

    #pragma omp parallel default(none) shared(count)
    {
        for(int i = 0; i < 100000; i++) {
            #pragma omp atomic
            count++;
        }
    }

    std::cout << count << std::endl;

    return 0;
}
```

Variables defined inside a parallel section are **always** local to the thread.

```
#pragma omp parallel default(var1, var2, var3)

 #pragma omp parallel shared(var1, var2, var3)

#pragma omp parallel private(var1, var2, var3)
```

So what about race conditions?

```cpp
#include <omp.h>
#include <iostream>

int main() {

   int count = 0;

   #pragma omp parallel
   {
      for(int i = 0; i < 100000; i++) {
         count++;
      }
   }

   std::cout << count << std::endl;

   return 0;
}
```

```
#include <omp.h>
#include <iostream>

int main() {

   int count = 0;

   #pragma omp parallel
   {
      for(int i = 0; i < 100000; i++) {
         count++;
      }
   }

   std::cout << count << std::endl;

   return 0;
}
```

149140
101024
158051
142048
141030
144466
101397

Nope!

OpenMP does not protect you against race conditions. It does make them easier to deal with, however.

```cpp
#include <omp.h>
#include <iostream>

int main() {

    int count = 0;

    #pragma omp parallel
    {
        for(int i = 0; i < 100000; i++) {
            #pragma omp critical
            {
                count++;
            }
        }
    }

    std::cout << count << std::endl;

    return 0;
}
```

A critical section uses a mutex to limit access to an area.

```
#include <omp.h>
#include <iostream>

int main() {

   int count = 0;

   #pragma omp parallel
   {
      for(int i = 0; i < 100000; i++) {
         #pragma omp atomic
         count++;
      }
   }

   std::cout << count << std::endl;

   return 0;
}
```

You can also use an atomic directive to perform arithmetic atomically.

Most common directive: parallel for

The most common one you'll be using is the parallel for directive

```cpp
#include <omp.h>
#include <iostream>

int main() {

   int count = 0;

   #pragma omp parallel for
   for(int i = 0; i < 100000; i++) {
      count++;
   }

   std::cout << count << std::endl;

   return 0;
}
```

Using it is extremely simple

```
#include <omp.h>
#include <iostream>

int main() {

   int count = 0;

   #pragma omp parallel for
   for(int i = 0; i < 100000; i++) {
      count++;
   }

   std::cout << count << std::endl;

   return 0;
}
```

25000	25000	25000	25000

Divides the loop into equal pieces by default (4 threads in the case shown above).

It chops the loop into equal pieces. Each thread executes one part.

Fine print

Loop must be of the following form:

```
for(int i = 0; i < 100000; i++) {
   count++;
}
```

Breaking early is not permitted:

```
for(int i = 0; i < 100000; i++) {
   if(i == 9) { return; }
}
```

The loop does need to adhere to some conditions though.

Fine print

No dependencies between iterations

```
for(int i = 0; i < 100000; i++) {
   list[i + 1] += list[i] – list[i + 1];
}
```

Alternate scheduling

```
#pragma parallel for schedule(dynamic)
for(int i = 0; i < 100000; i++) {
   list[i + 1] += list[i] – list[i + 1];
}
```

Sometimes chopping a loop into equal pieces is not ideal for performance, in particular when each loop iteration does not take the same amount of time.

Alternate scheduling

Static: Equal size chunks divided between threads

Good if each iteration takes about the same time

Alternate scheduling

Static: Equal size chunks divided between threads

Dynamic: Chunks are handed out dynamically

Better for more non-uniform workloads, more overhead

Dynamic scheduling means chunks are handed out dynamically. Chunks are several iterations of the loop. However, this also implies the need to dynamically divide work between threads.

Alternate scheduling

Static: Equal size chunks divided between threads

Dynamic: Chunks are handed out dynamically

Guided: Dynamic scheduling with decreasing chunk size

Can further improve performance in some situations

Guided is a more extreme form of dynamic, which is aimed at more specific situations where the loop contains both extremely short and extremely long iterations. It hands out tasks dynamically, but decreases the chunk size over time.

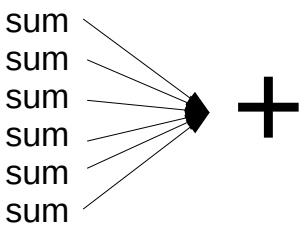
Reductions

```
int sum = 0;
#pragma parallel for reduction(+:sum)
for(int i = 0; i < 100000; i++) {
   sum += i;
}
```

You can also do reductions. Each reduction requires you to specify a variable and an operation to perform between them.

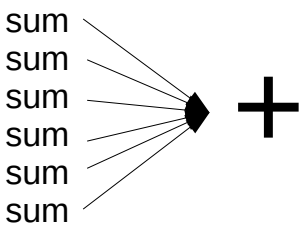
Reductions

```
int sum = 0;
#pragma parallel for reduction(+:sum)
for(int i = 0; i < 100000; i++) {
   sum += i;
}
```



Reductions

```
int sum = 0;
#pragma parallel for reduction(+:sum)
for(int i = 0; i < 100000; i++) {
   sum += i;
}
```



Note:
Integer only!

Because floating point operations are not commutative, these reductions are integer only.

Reductions

Supported operations:

+, -, *, /, max, min

&, &&, |, ||, ^

These are the supported operations for a reduction.

Trick: reuse threads in repeating for loop

```
#pragma omp parallel
for(int i = 0; i < 1000000; i++)
{
   #pragma omp for
   for(int j = 0; j < 10; j++) {
      doSomething();
   }
}
```

std::thread or OpenMP?

A small note on when to use std::thread and OpenMP

OpenMP ⊂ std::thread

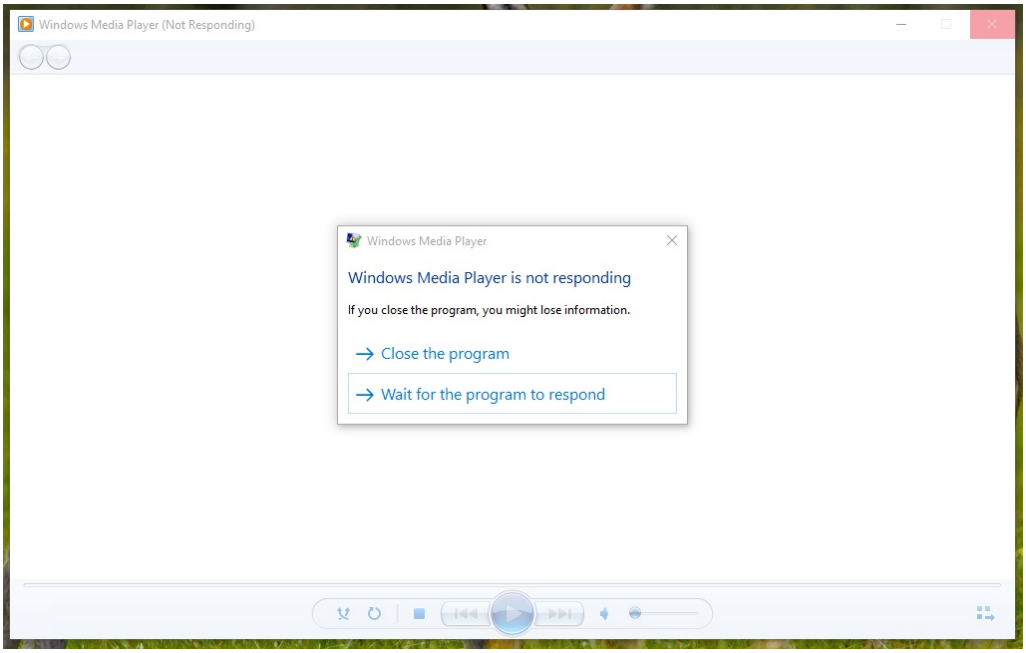
First off, everything you can do in OpenMP you can also implement manually in std::thread.

OpenMP:

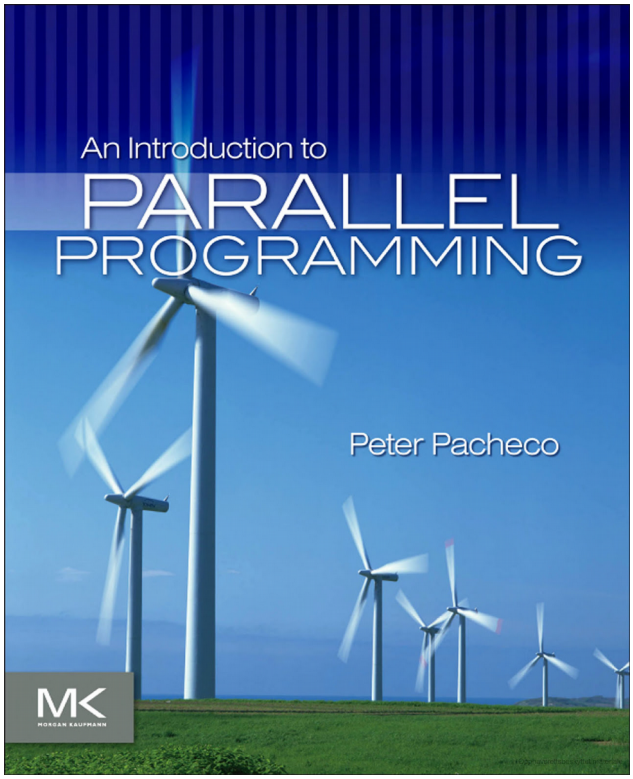
```
for(int i = 0; i < veryLargeNumber; i++) {
    expensiveComputation();
}
```

In practice, OpenMP is really useful for quick parallelisation. It can really save you a lot of time!

User Interfaces

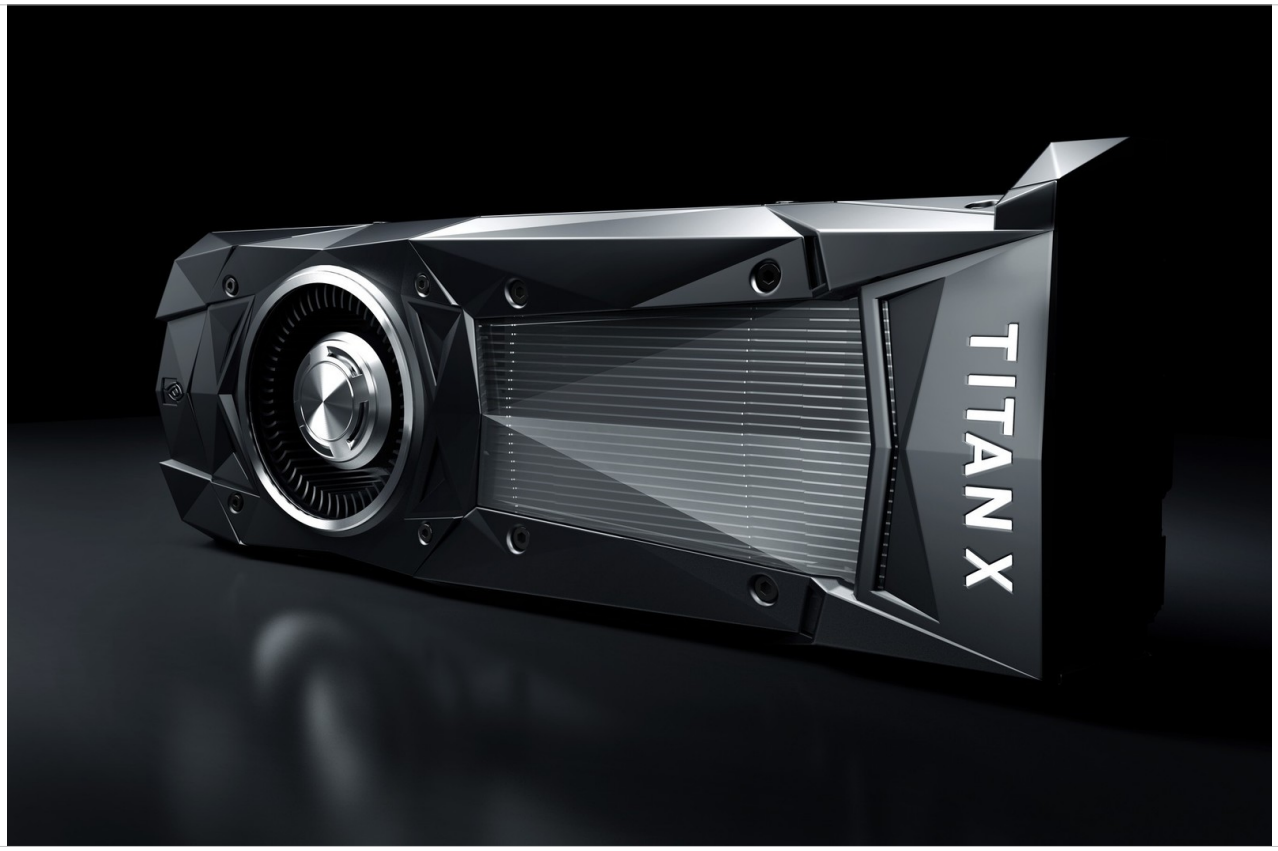


std::thread is great for offloading more “unique” work to background threads. Things like doing some processing in the background (video or image decoding, loading of files, etc).

</ 

>

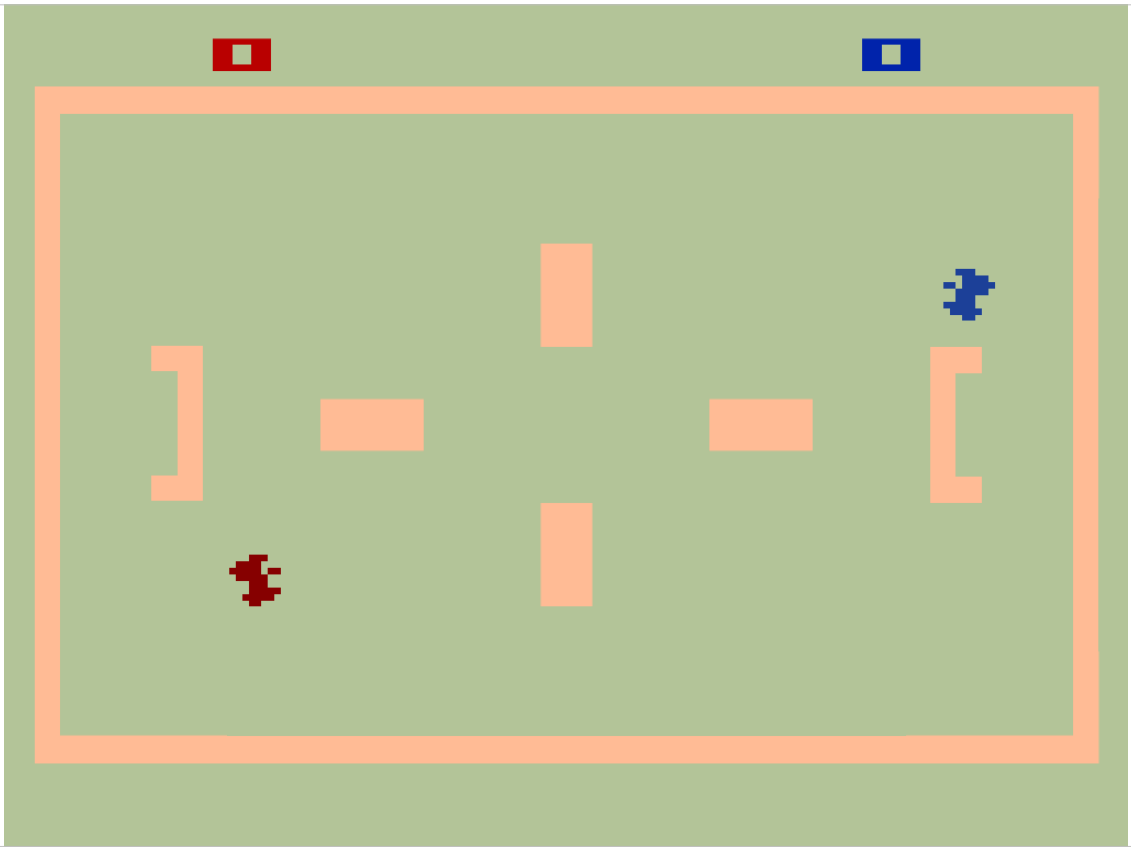
And that concludes the main book of the course!



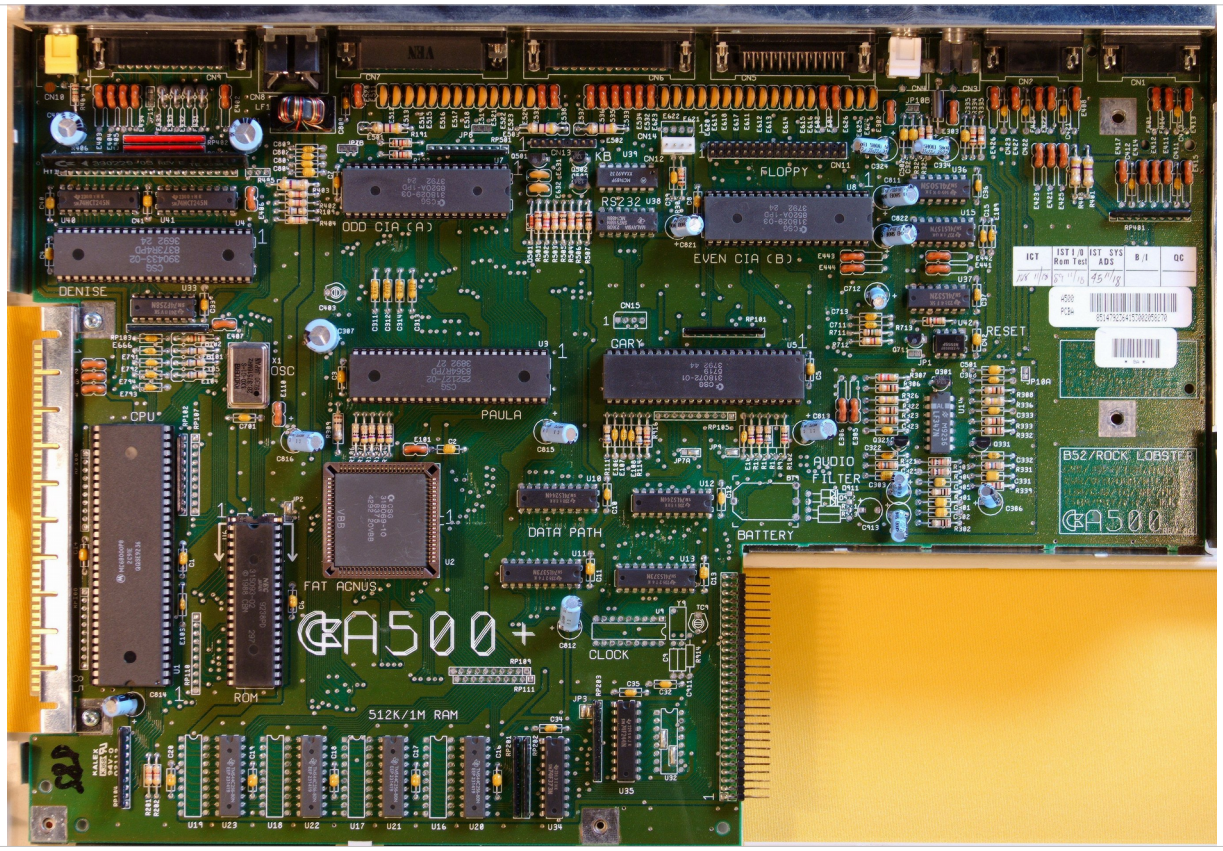
On to GPUs!

GPU: Graphics Processing Unit

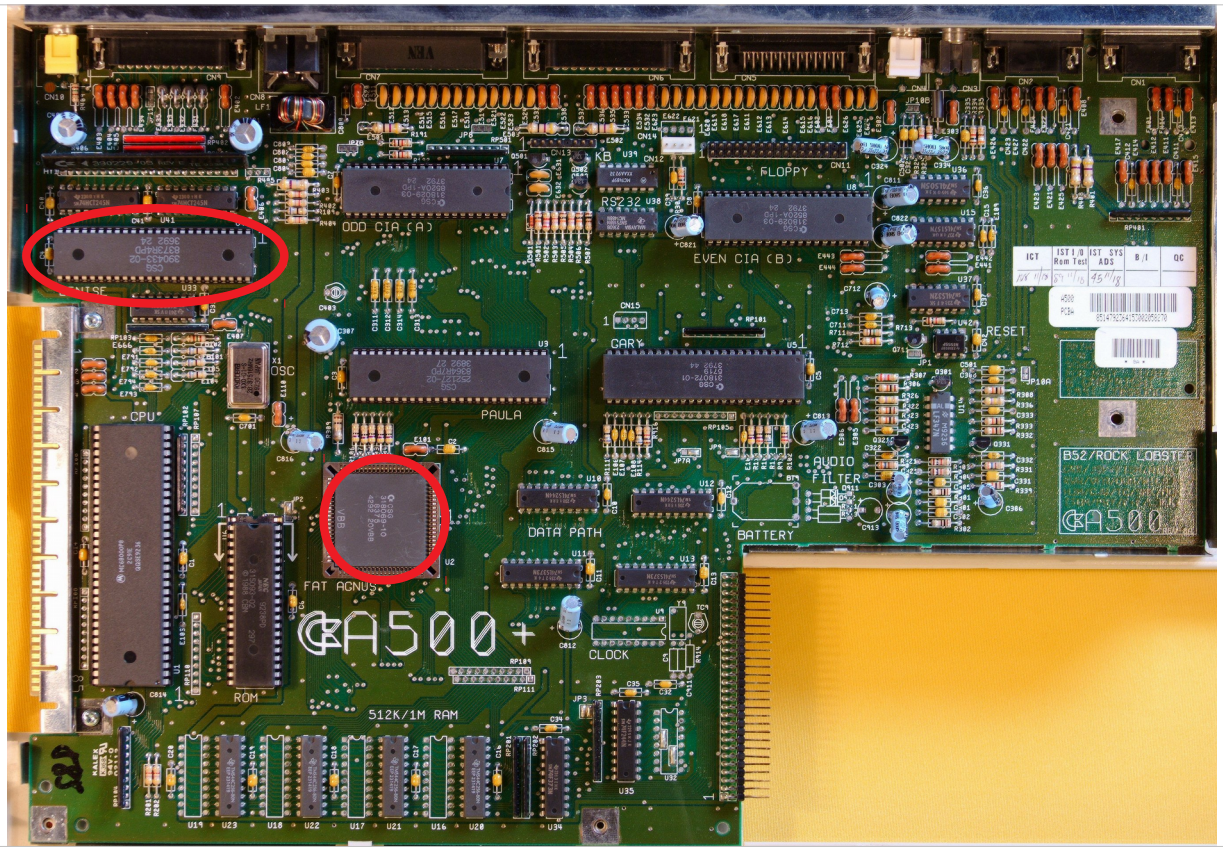
GPU: **Graphics?** Processing Unit



The first games were not very noteworthy. But once they got some traction, people wanted more.

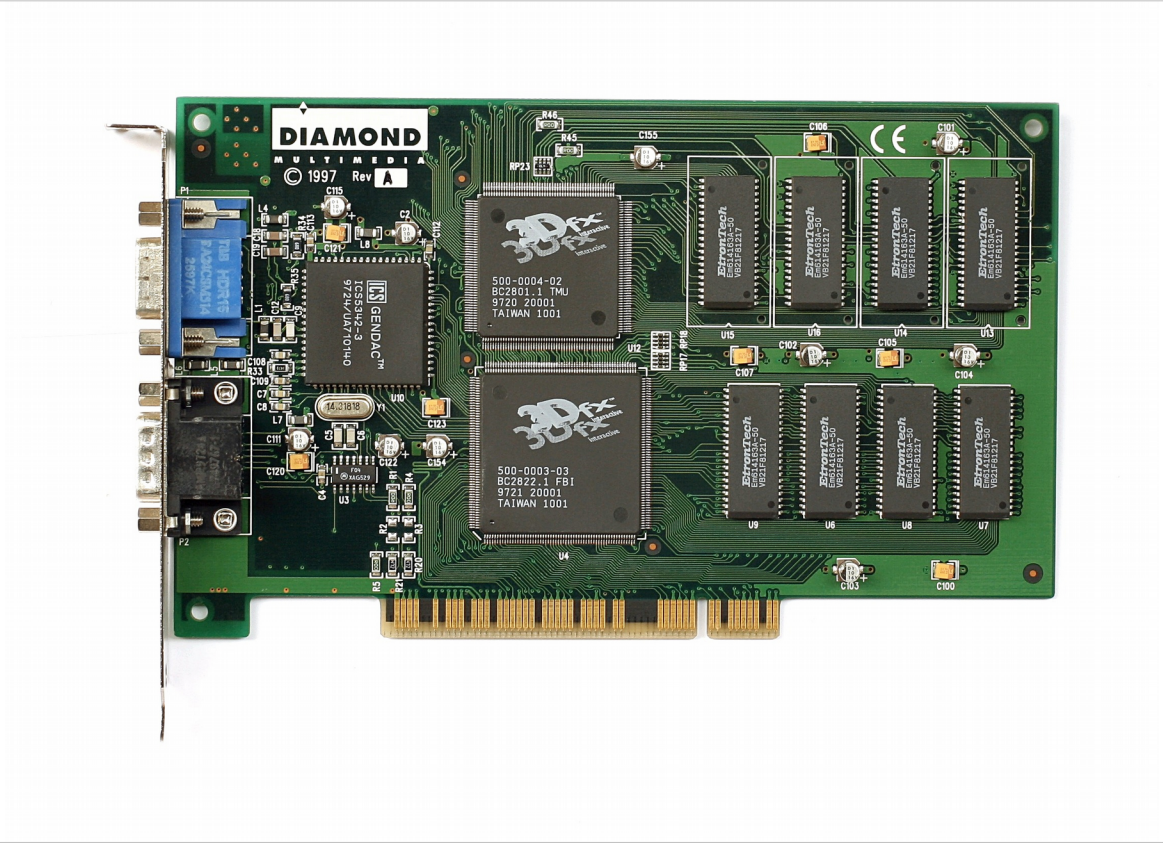


This is the main board of the commodore Amiga. It’s one of the early examples where specialised hardware was used for performing graphics-related operations.



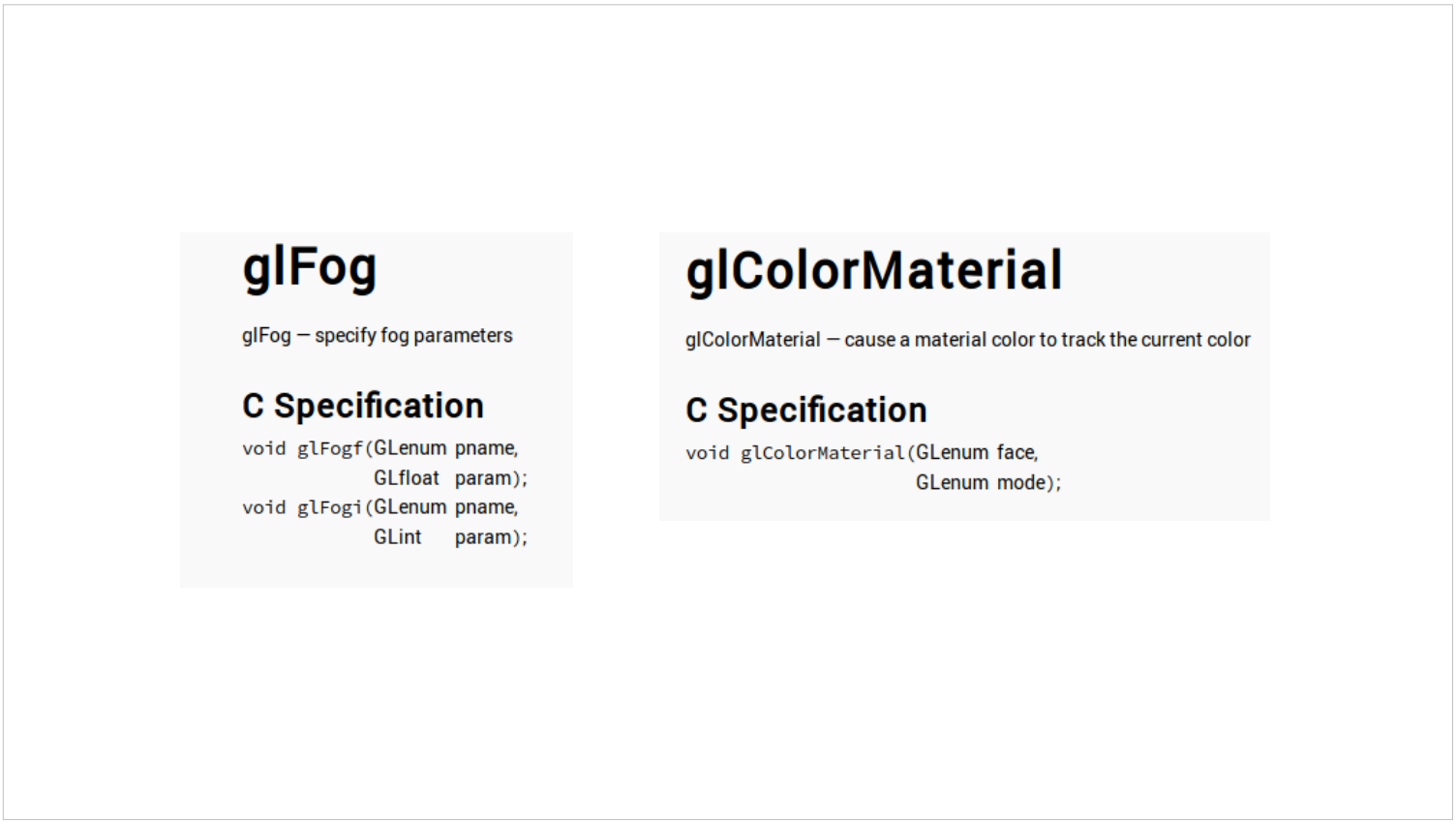
These are the graphics “coprocessors” of the amiga 500.

These early VGA cards allowed the creation of games such as Wolfenstein 3D. They were still made for 2D rendering however.

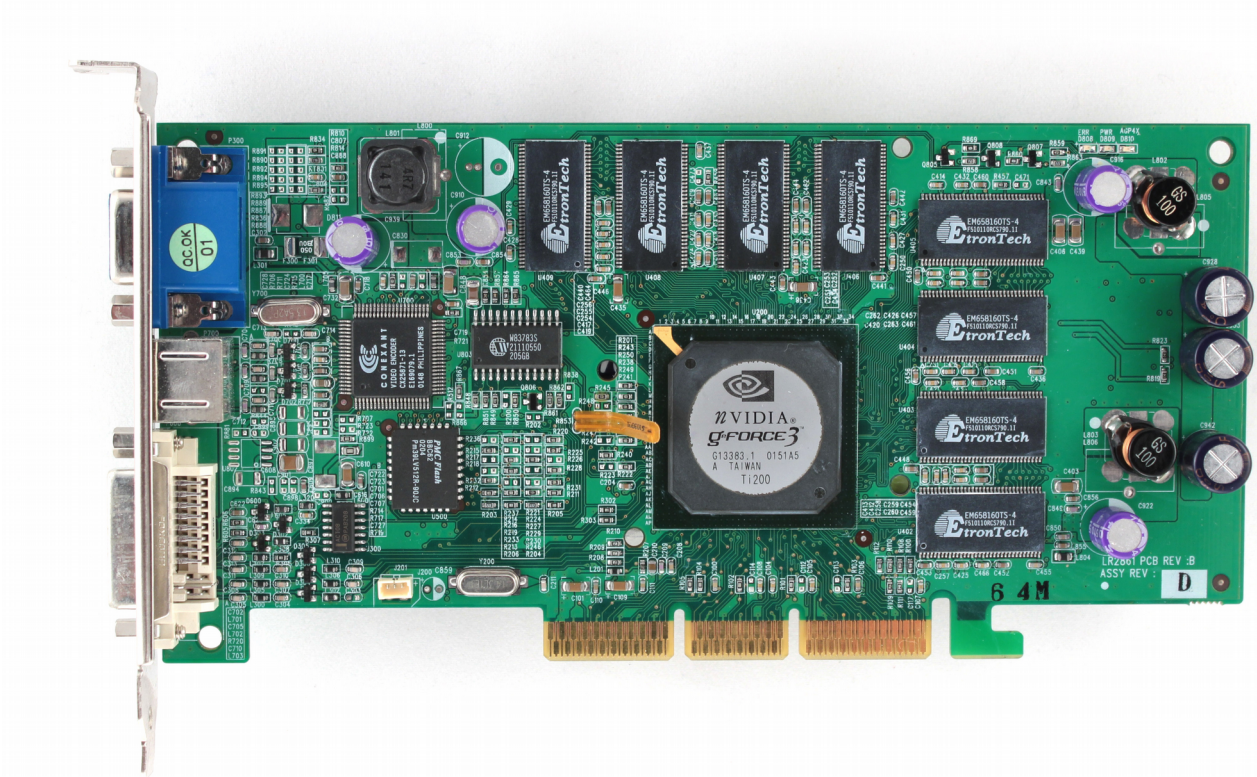


Fun fact: VGA cards such as this one often had multiples of 4 memory banks (right side), because memory chips at the time were not fast enough to keep up with the data rate the display required to draw its picture. By drawing data from multiple banks simultaneously, the card could keep up.

Improved graphics cards started implementing rasterisation, but these were primarily “fixed functionality”. This meant the cards could render for what was at the time incredible scenes at a high framerate, but was still limiting for developers.



Developers required more features, but rendering API’s responded by adding more specialised features.



At some point, the idea arose to make a part of the rendering pipeline more configurable by making them somewhat programmable. There were a lot of hoops to jump through, but this allowed an incredible level of configurability.

Shaders

The programs that ran in the programmable portions of this pipeline were called “shaders”.

Shaders

A notable example of one kind of shader computes the colour of each pixel. This means the same computation was done over and over again for each pixel separately.

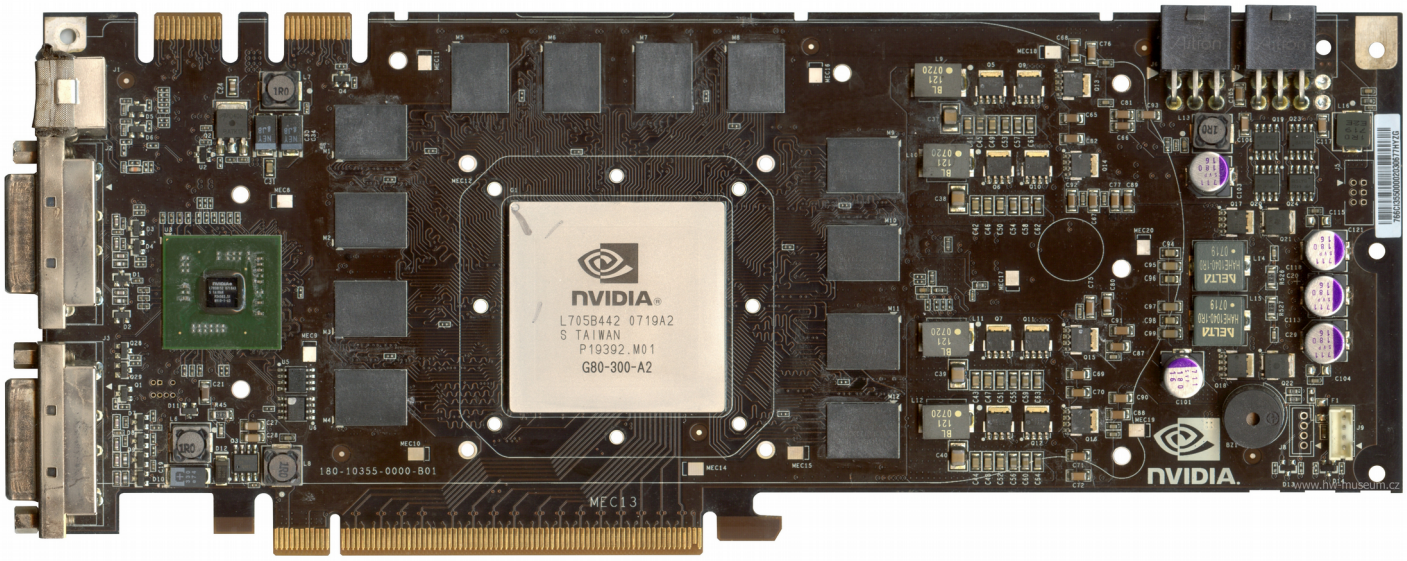
Observation:

Pixel colours are independent

→ Inherently parallel!

Now here’s the kicker: each pixel is independent of one another. This made the design of the processor inherently parallel.

8800 GTX: First GPGPU



Future GPU's became more and more capable, included more parallel processors, while at the same time ditching the old "fixed" components of the hardware.

At some point, the idea arose that such parallel hardware can be used for computations that are not purely graphical. There exist a lot of extremely parallel problems out there, and this hardware allowed them to run significantly faster compared to CPU's.

GPU Hardware Overview

See the lecture video where I show the main parts of a GPU.

Let’s look at the GPU processors themselves!

How do they stack up against the CPU?

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
				

Best and worst performer?

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
Q3‘07	Q1‘09	Q3’09	Q3’17	Q3’18

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
Q3‘07	Q1‘09	Q3’09	Q3’17	Q3’18
16 cores	128 cores	4 cores	18 cores	4352 cores

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
Q3‘07	Q1‘09	Q3’09	Q3’17	Q3’18
16 cores	128 cores	4 cores	18 cores	4352 cores
0.7 GHz	1.8 GHz	3.2GHz	3.4 GHz	1.5 GHz

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
Q3‘07	Q1‘09	Q3’09	Q3’17	Q3’18
16 cores	128 cores	4 cores	18 cores	4352 cores
0.7 GHz	1.8 GHz	3.2GHz	3.4 GHz	1.5 GHz
23 GFLOPS	480 GFLOPS	32 GFLOPS	1000 GFLOPS	13 400 GFLOPS

Nvidia Quadro FX 370	Nvidia (EVGA) GTS 250	Intel Core i5-750	Intel Core i9-7980XE	Nvidia RTX 2080 Ti
Q3‘07	Q1‘09	Q3’09	Q3’17	Q3’18
16 cores	128 cores	4 cores	18 cores	4352 cores
0.7 GHz	1.8 GHz	3.2GHz	3.4 GHz	1.5 GHz
23 GFLOPS	480 GFLOPS	32 GFLOPS	1000 GFLOPS	13 400 GFLOPS

15x

13.4x

What’s the catch?

GPU Architecture

Turing Architecture

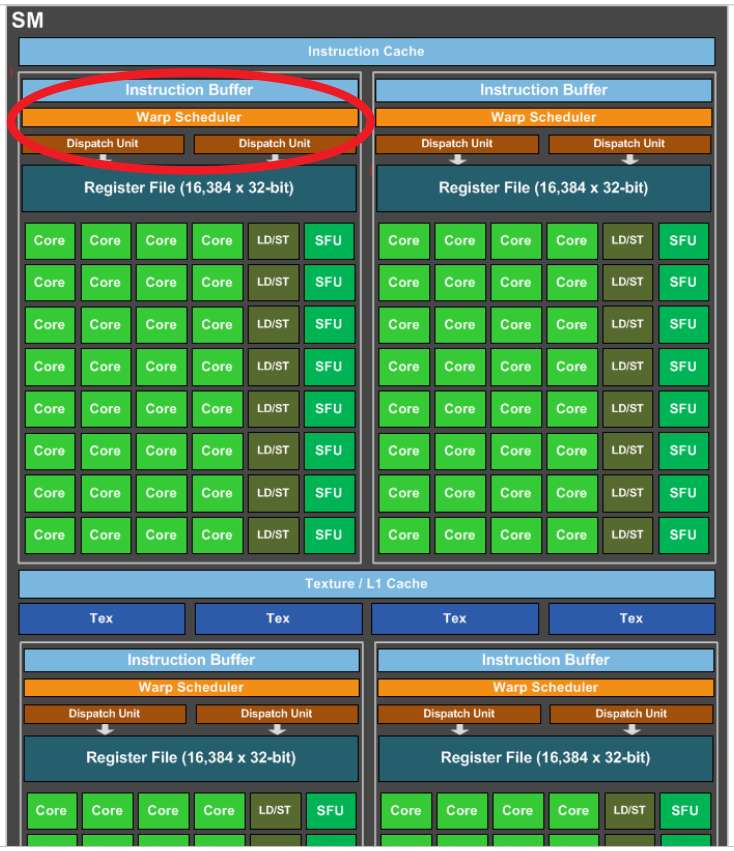
The latest architecture is very complicated. I'll therefore (try to) explain one that's I think simpler to understand: Pascal

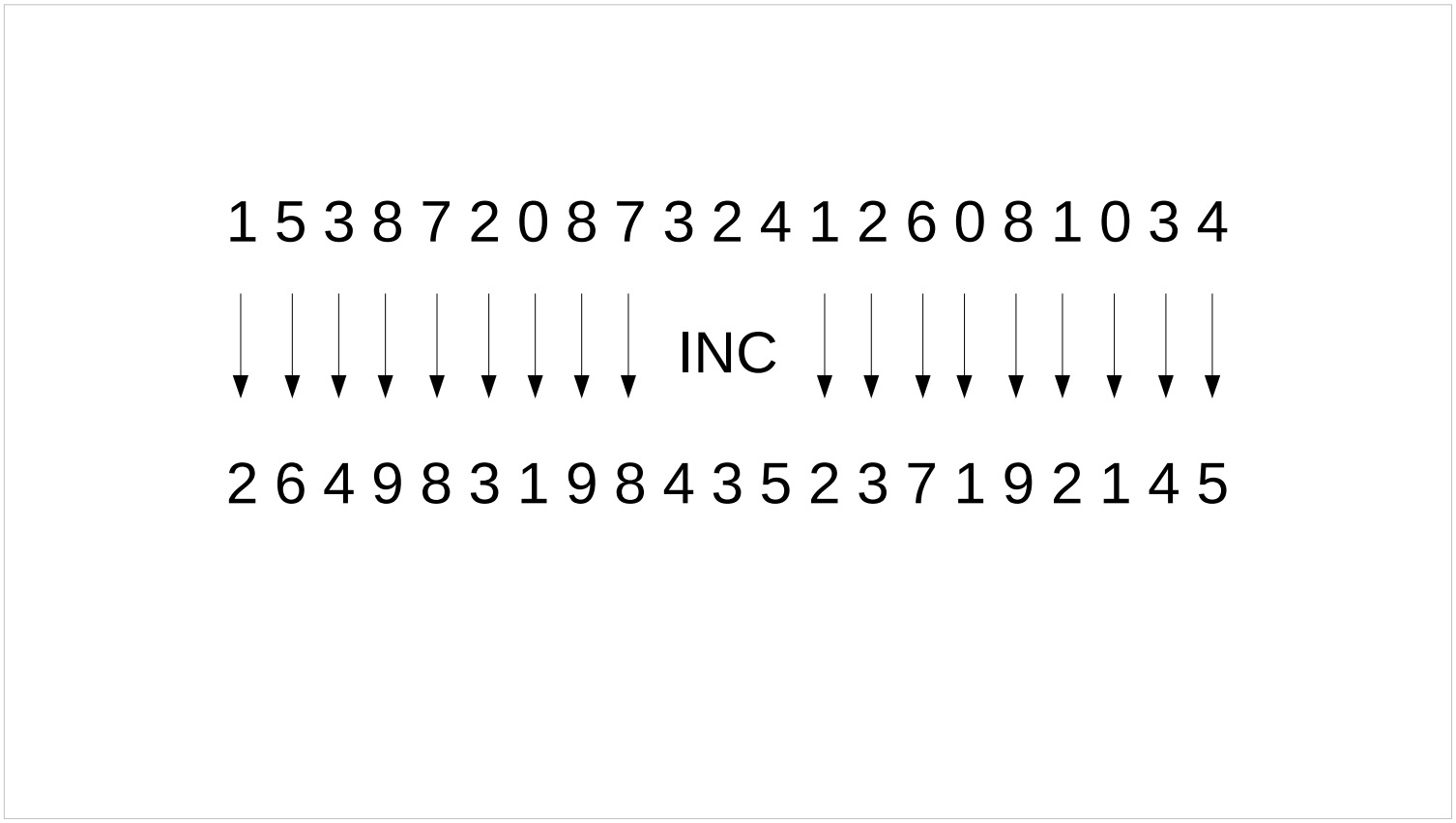
Here’s the layout of the GTX 1080 Ti. For all of these GPU slide, keep in mind that AMD has a very similar structure.

The GPU is divided into a number of so-called “Streaming multiprocessors”. Each of these contains sets of “cores”. Each core contains an instruction decoder, a large number of registers, and a bunch of ALU’s / Floating Point Units. The idea here is that all of these share the same instruction decoder, but execute that instruction on 32 pieces of data at once. This allows a LOT of throughput, at the cost of flexibility of individual cores.


SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex

Control and decoding logic is shared between streaming processors!





This means a single instruction is applied 32 times in one cycle!

1 5 3 8 7 2 0 8 7 3 2 4 1 2 6 0 8 1 0 3 4

2 6 4 9 8 3 1 9 8 4 3 5 2 3 7 1 9 2 1 4 5

One instruction performs 32 operations!

Variant of SIMD:

Single Instruction Multiple Thread (SIMT)

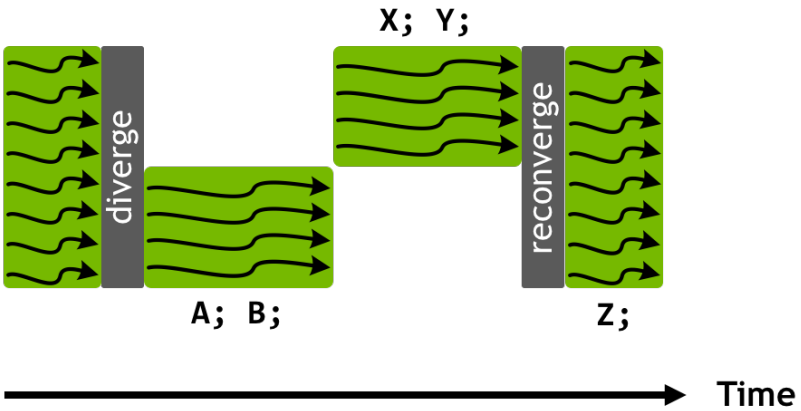
This is a variation of SIMD, where rather than an instruction being executed on a single piece of data, the operation is applied once for a number of threads simultaneously. The result is somewhat the same though.

MAJOR DOWNSIDE:

Thread divergence

The problem lies with branches.

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

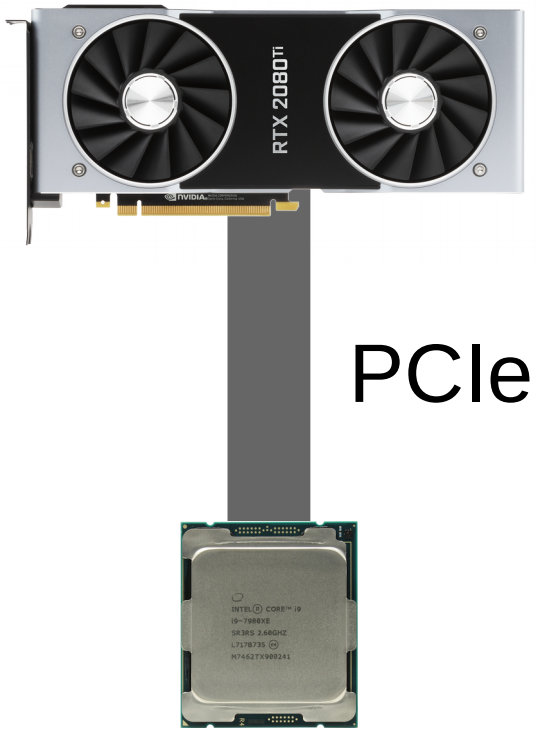


Since all stream processors (“CUDA cores”) need to execute the same instruction, branches cause some threads to wait around until others are done.

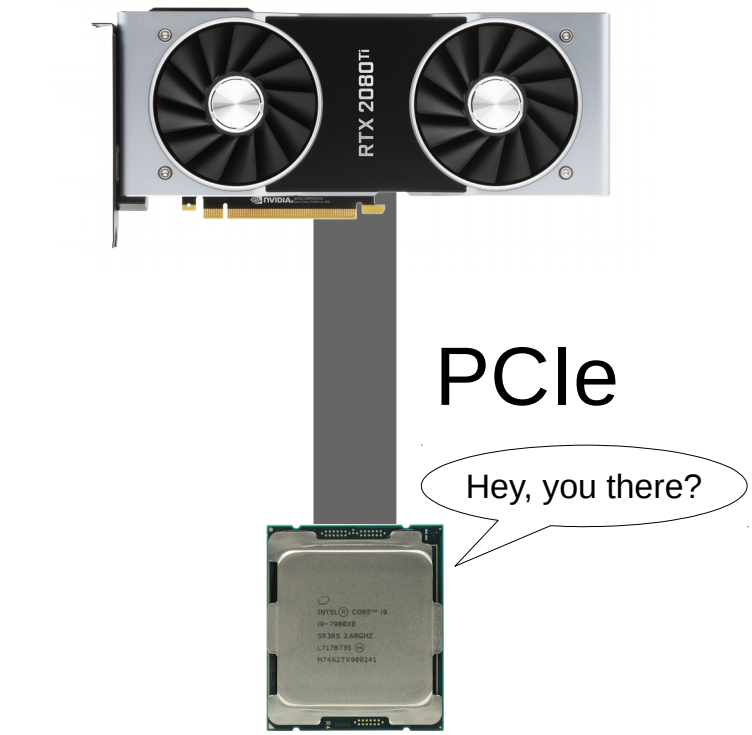
GPU’s work GREAT on similar operations being run an incredibly large number of times

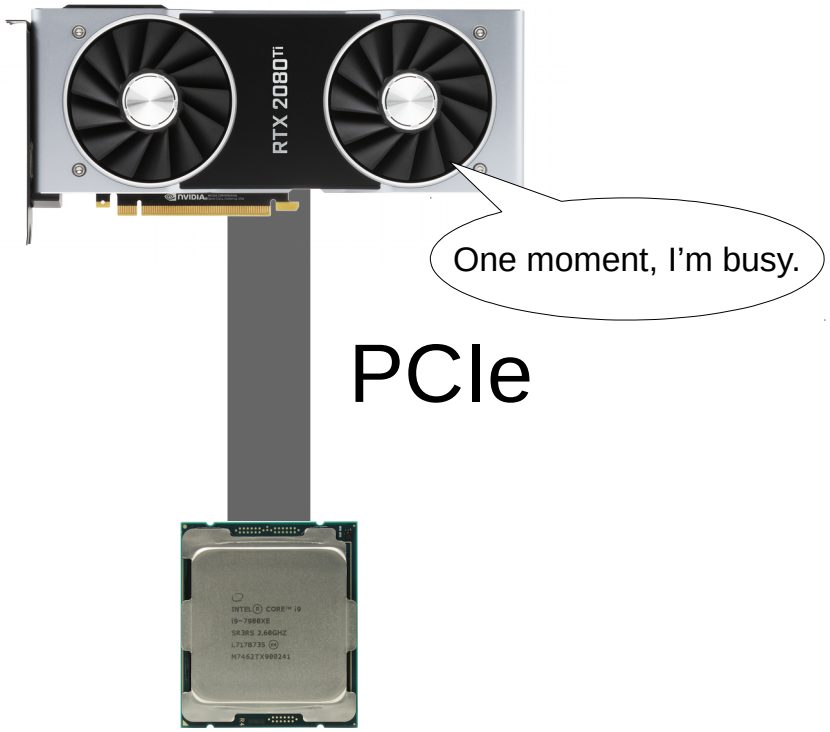
(shaders require exactly that)

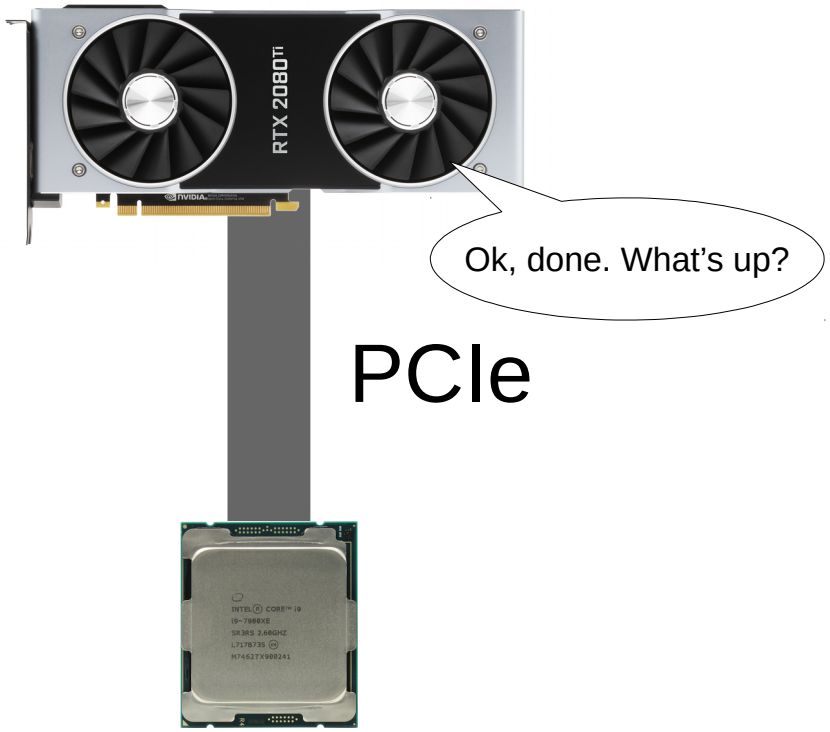
Interacting with the GPU

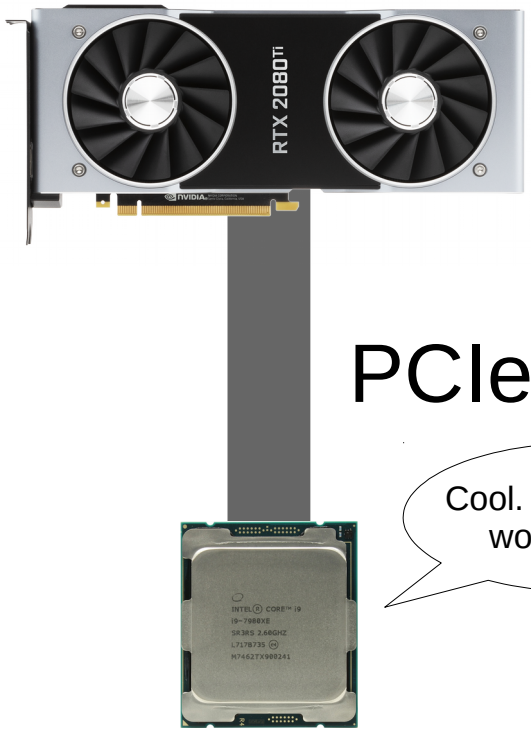


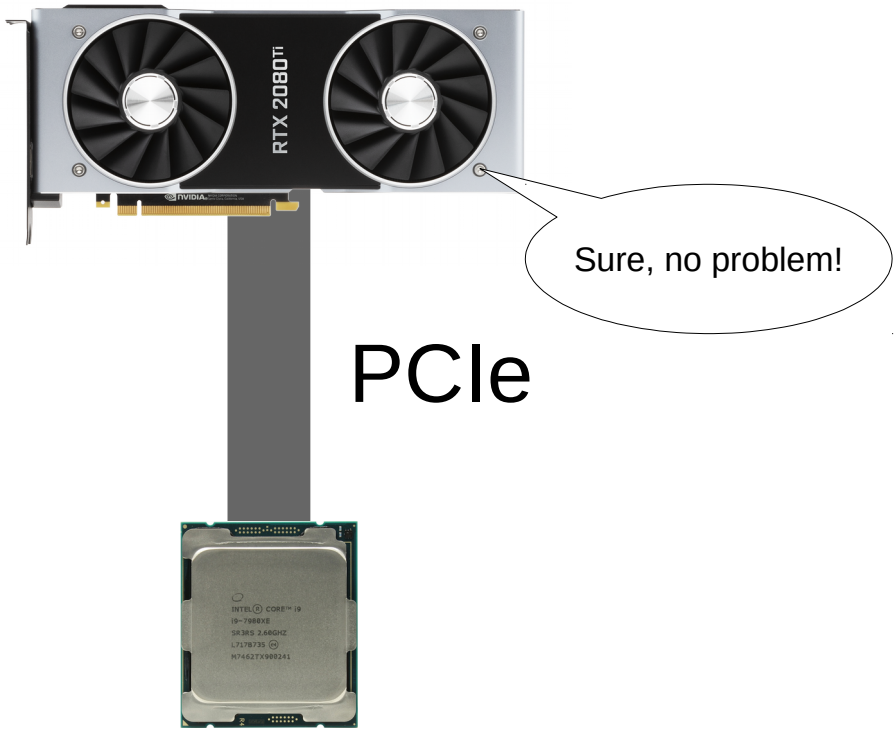
Interacting with the GPU sort of works like talking to a webserver. Fortunately, the GPU driver takes care of most of the heavy lifting for you.

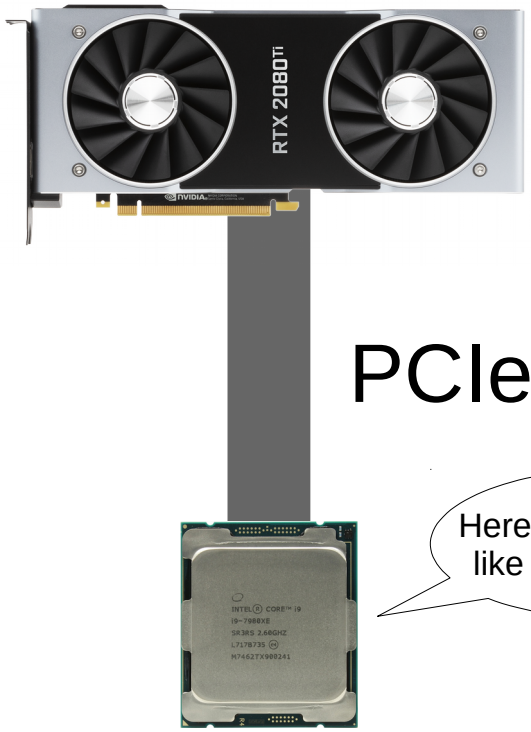

RTX 2080Ti
PCIe
Hey, you there?

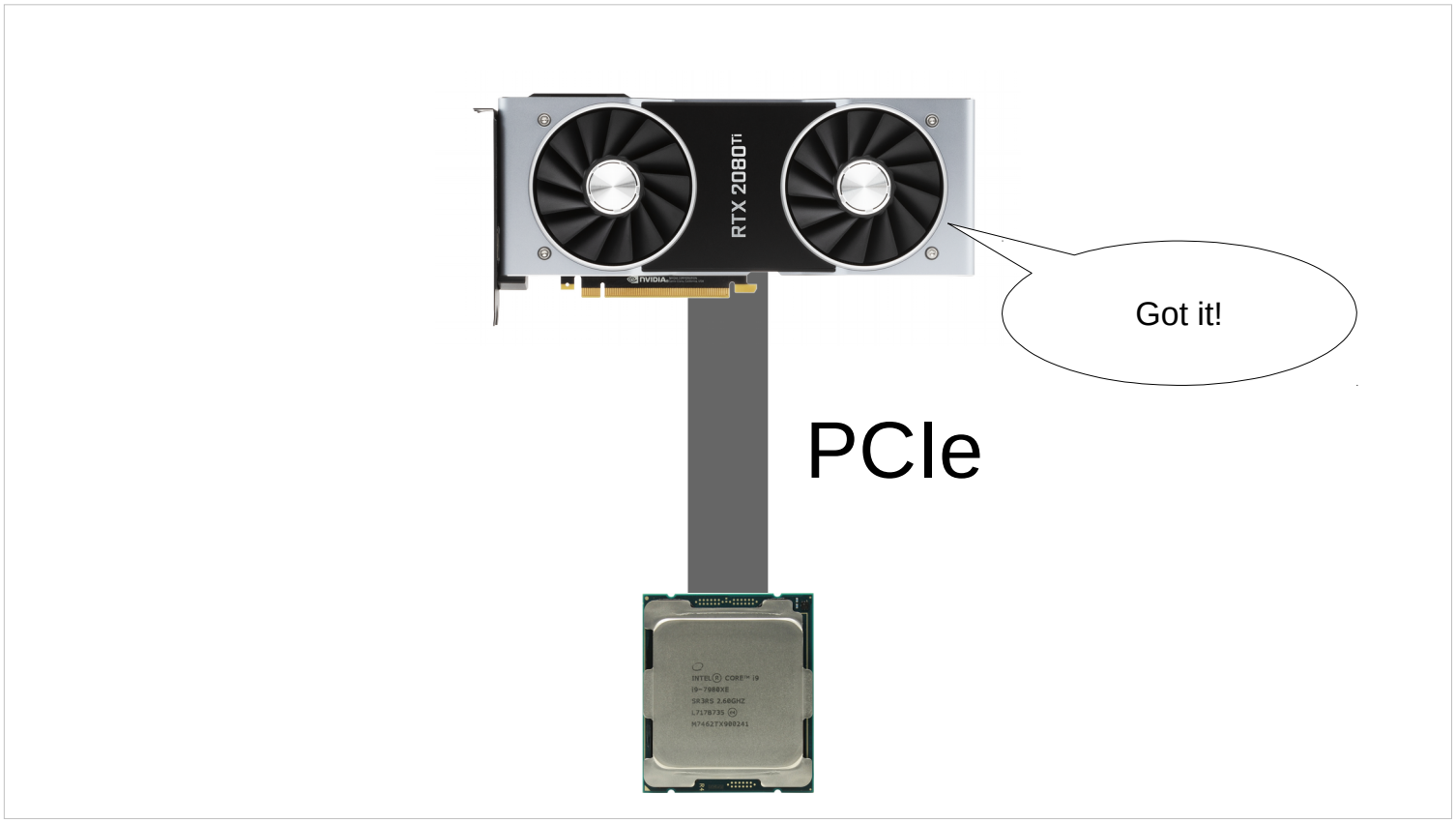

RTX 2080Ti
One moment, I’m busy.
PCIe

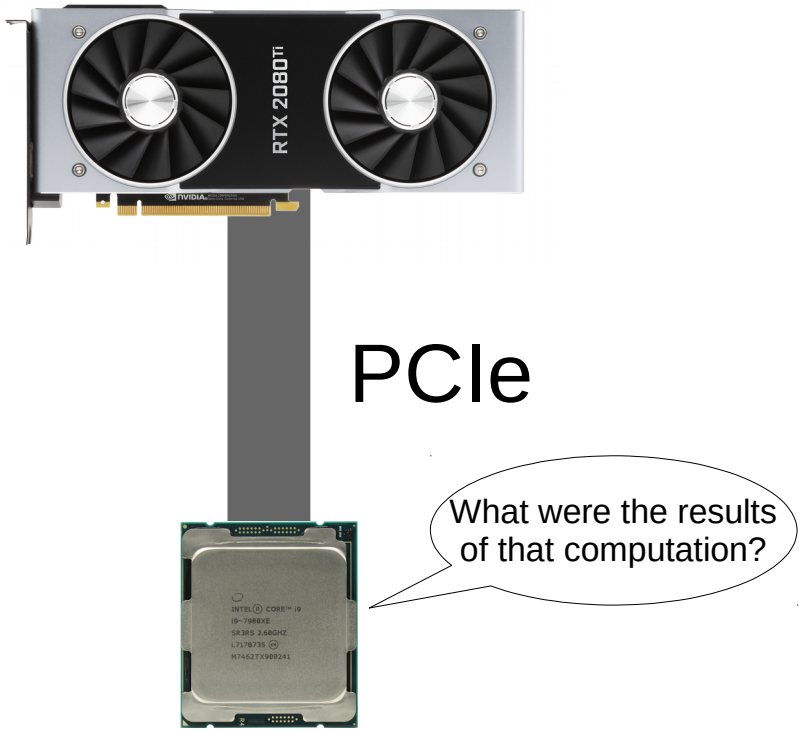

RTX 2080Ti
Ok, done. What's up?
PCIe

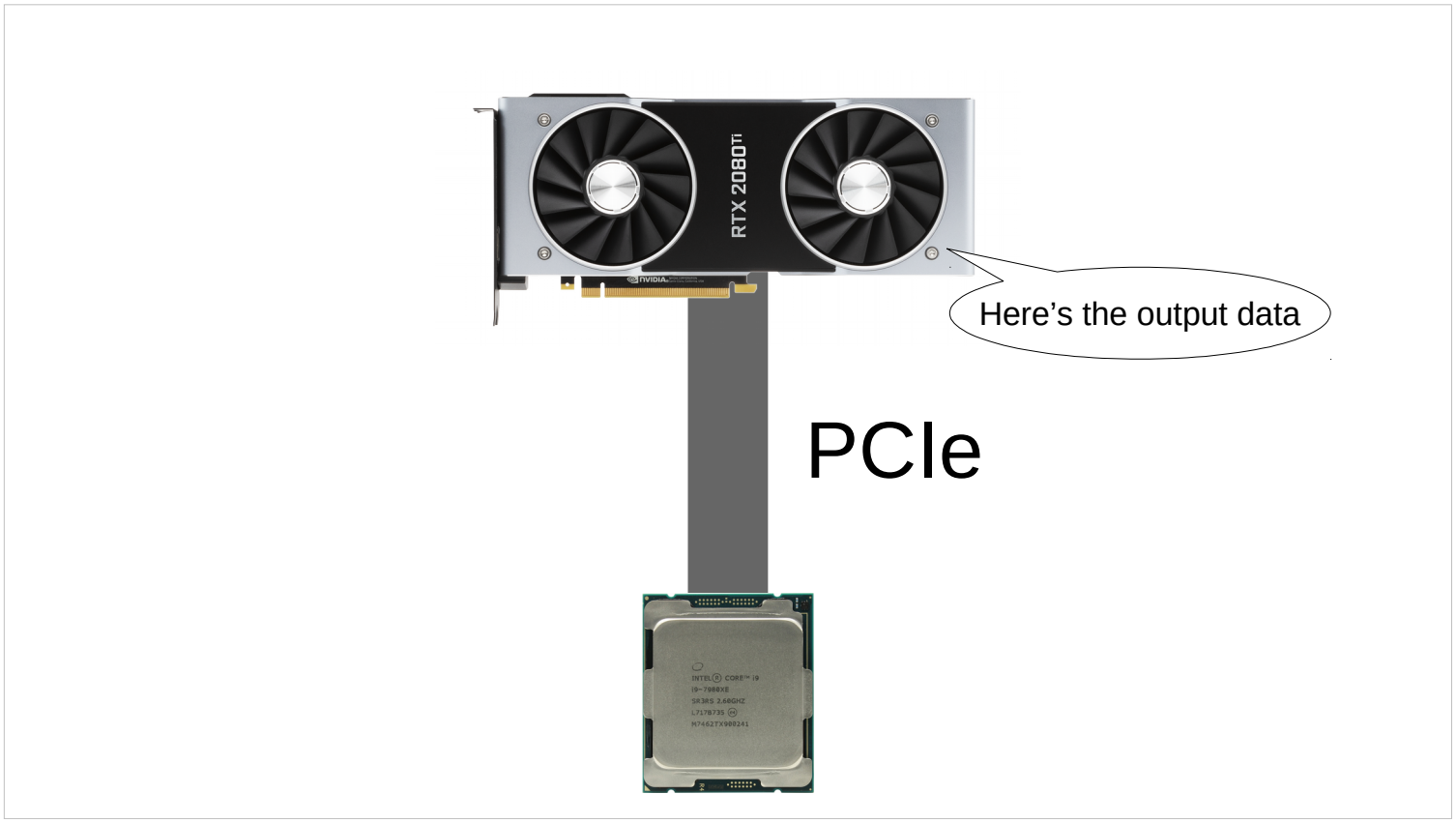

RTX 2080Ti
PCIe
Cool. I have some work for you.


RTX 2080Ti
Sure, no problem!
PCIe

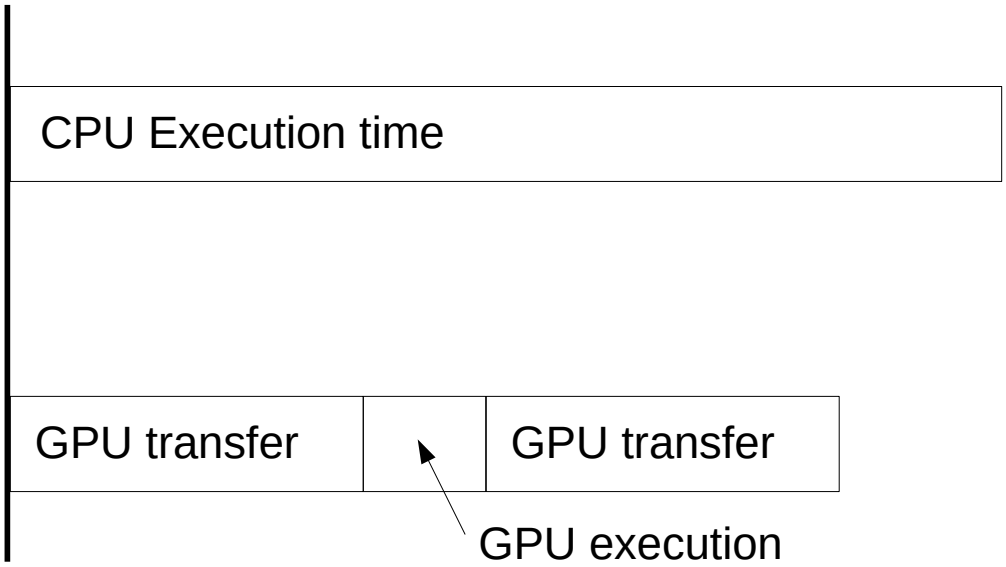

RTX 2080Ti
PCIe
Here’s some data I’d like you to process


RTX 2080Ti
Got it!
PCIe


RTX 2080Ti
PCIe
What were the results of that computation?


RTX 2080Ti
Here’s the output data
PCIe

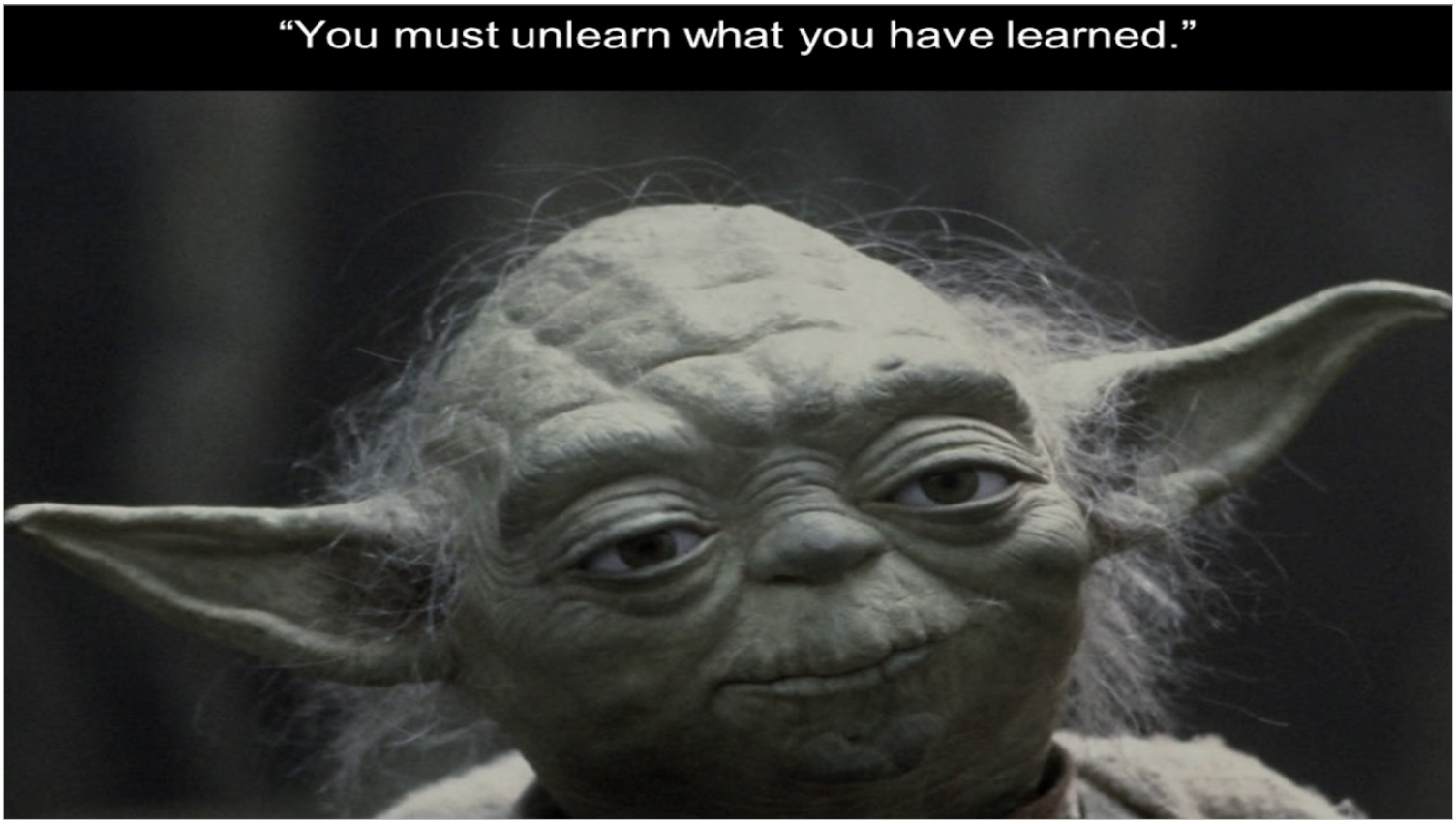
Tradeoff:



Because data needs to be transferred explicitly, deciding to run something on the GPU also needs to take transfer times into account.

Programming the GPU

Programming the GPU can be quite a bit different from the CPU.


“You must unlearn what you have learned.”

On the CPU:

Using lots of threads is expensive

Using lots of threads can limit performance

Using the cache helps a lot

On the GPU:

Using lots of threads is cheap

Using lots of threads is needed
for optimal performance

The cache cannot help you

On the GPU:

Using lots of threads is cheap

Using lots of threads is needed
for optimal performance

The cache cannot help you

Since GPU’s tend to work through extremely large amounts of data, its cache is not as vital as it is on the CPU. There are a few cases where it helps you, but its primary use is to cache instructions and stack variables.

GTX 1080 Ti Memory Bandwidth:

484 GB / s

L2 cache:

2.1 MB

GTX 1080 Ti Memory Bandwidth:

484 GB / s

L2 cache:

2.1 MB

Typical workload does not reuse data. Cache is therefore *mostly* useless

GPU Architecture (revisited)

I will be going over this again in a more structured manner next week.


SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory


SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex

Next week:

Diving into CUDA