TDT4200 Parallel Programming

Bart van Blokland

Lecture 10

Going to show small text, please come closer!

&group

Looking Ahead: Schedule

Today: CUDA Performance
November 8th: OpenCL Intro
November 15th: Parallel Programming A to Z

Digital Exam: 28.11.2018 at 09:00

More info on digital exams:
https://innsida.ntnu.no/wiki/-/wiki/English/Digital+exam+for+students

For November 15th:

Send me topics you want me to explain once more!

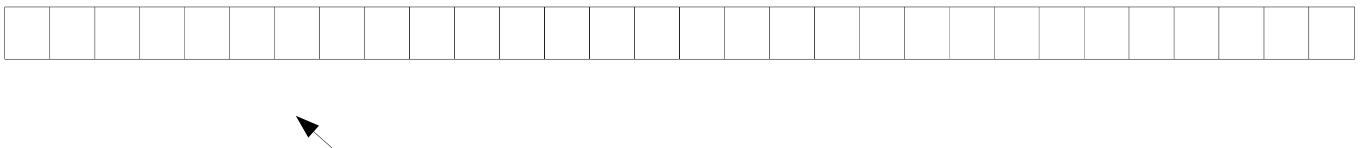
Link:

https://docs.google.com/forms/d/e/1FAIpQLSfEOE4N6FD_q0ujHKvXEP3SupPcHHoYWf21J0WIcfzorer-5A/viewform

Last time:

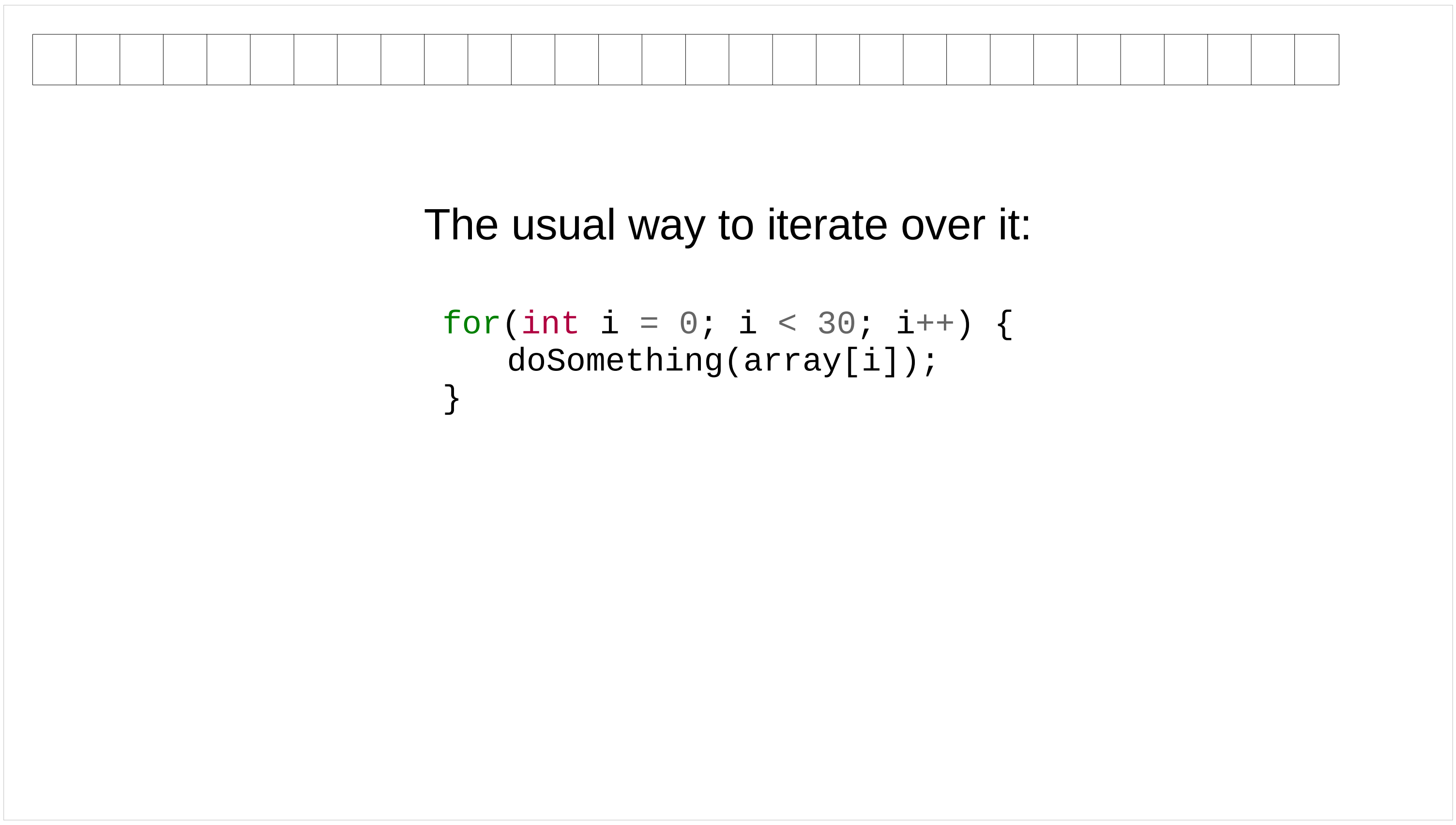
Thread Hierarchy

Streaming Multiprocessors (SM’s)



```
unsigned int* array = new unsigned int[30];
```

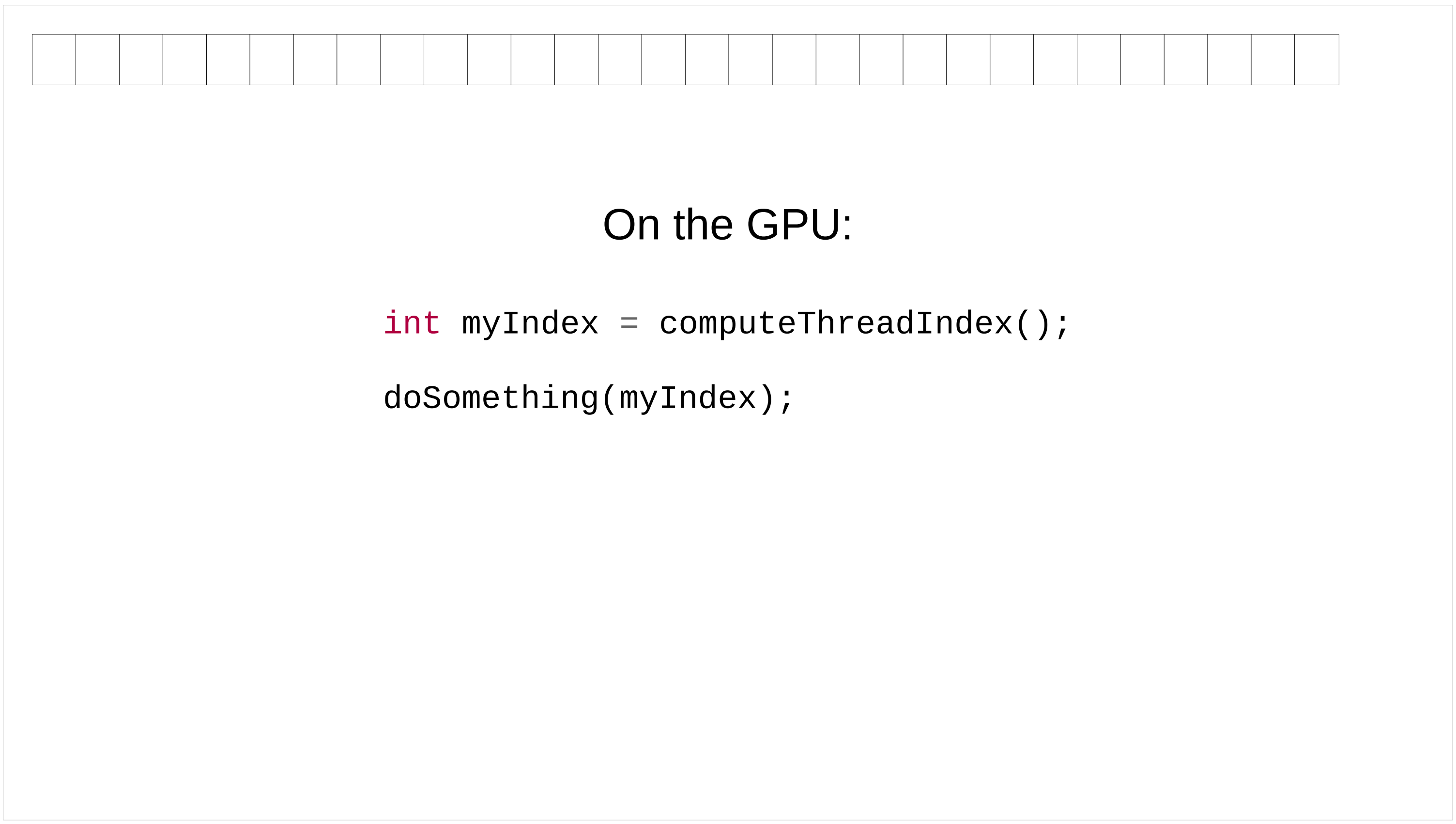
Let’s start with a typical job that we can run in parallel: an array on which we like to apply some operation.



The usual way to iterate over it:

```
for(int i = 0; i < 30; i++) {
    doSomething(array[i]);
}
```

Normally, for code running on the CPU, the way to do that is by using a loop to iterate over each item.



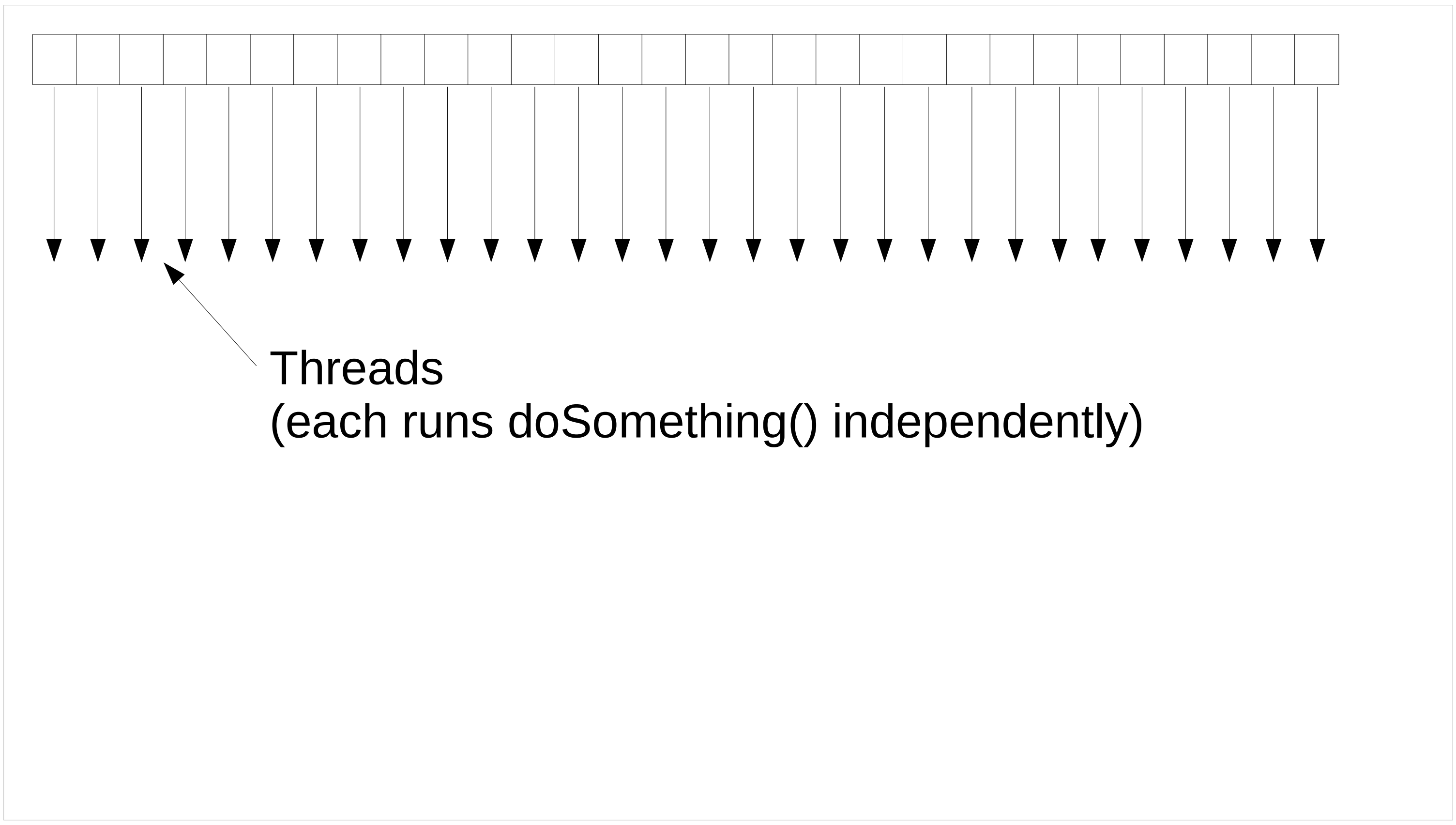
On the GPU:

```
int myIndex = computeThreadIndex();

doSomething(myIndex);
```

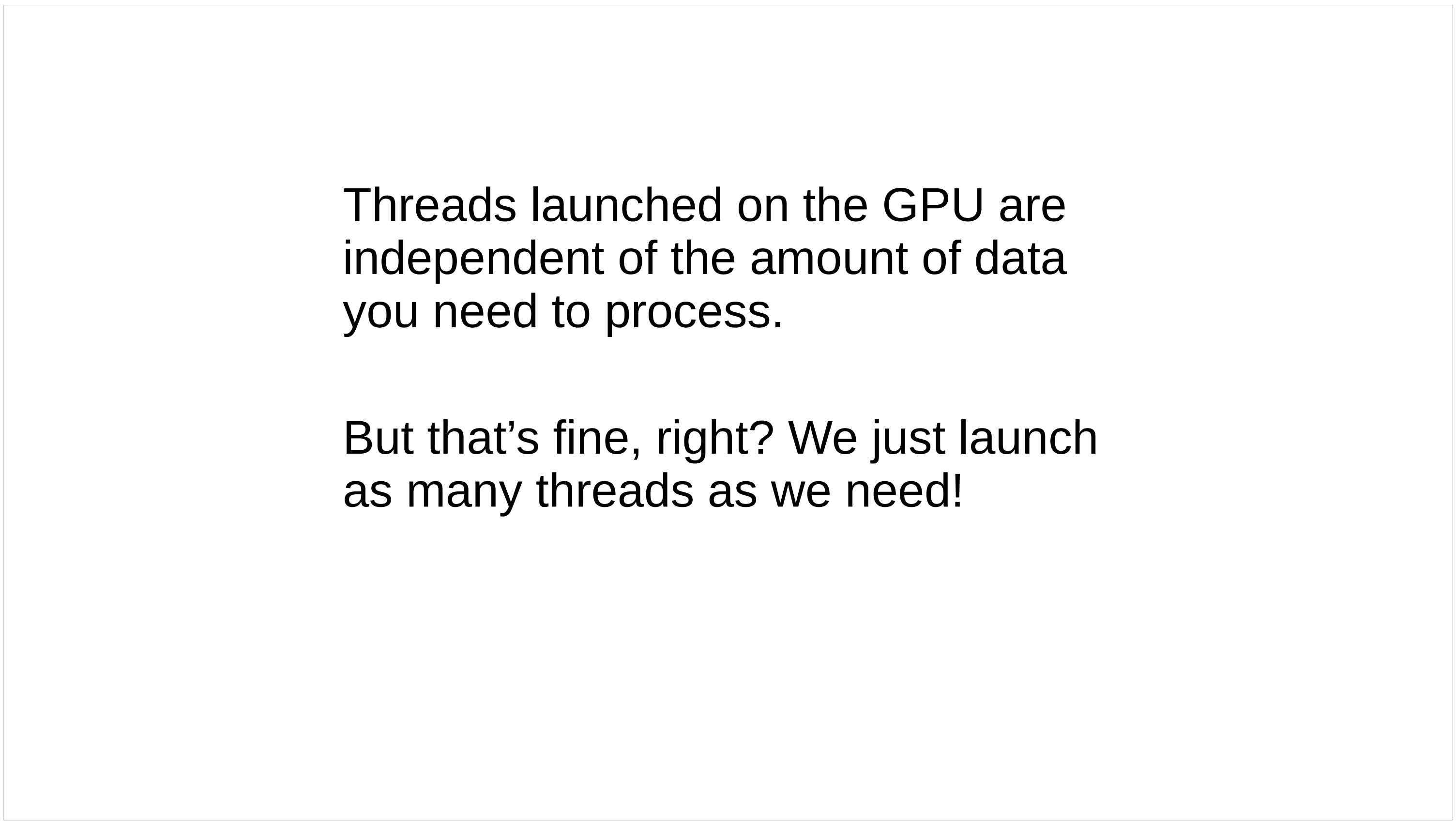
However, on the GPU the general approach is to use parallelism to solve all of your problems (ok, maybe not ALL of them).

So instead we launch a single thread per iteration of the loop, and run each iteration in parallel.

While threads on the CPU tend to be somewhat expensive (mostly so if they are rapidly created and destroyed in large numbers), the GPU is made to handle lots of them. I’m talking about millions there.



So for processing our array, we just launch one thread per index.



Threads launched on the GPU are independent of the amount of data you need to process.

But that’s fine, right? We just launch as many threads as we need!

The GPU doesn’t know (or care) that you have a buffer of a particular length that you’re processing. You yourself have to make sure you are spawning the number of threads you need.

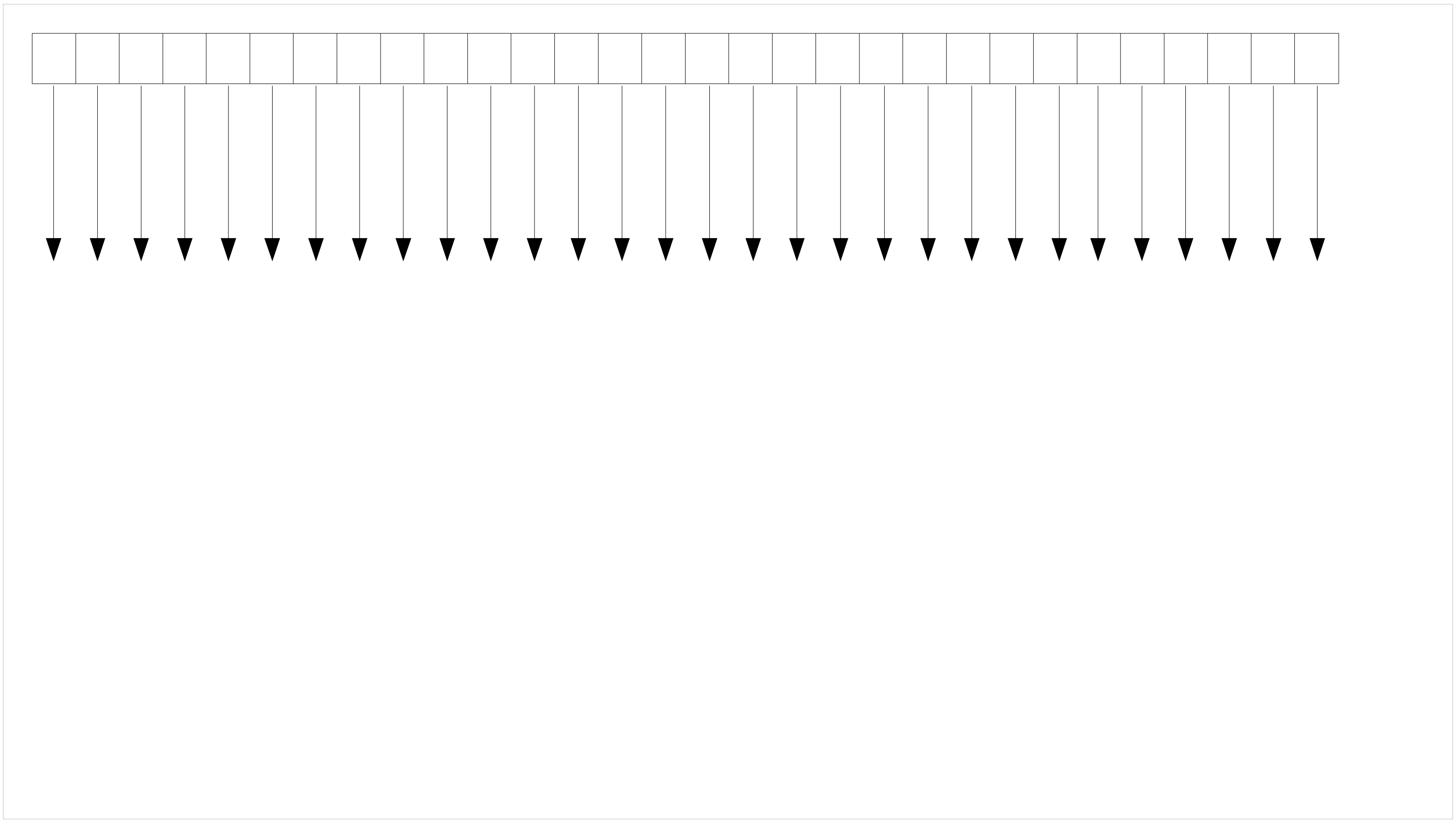
But that’s fine, right? We can just launch the number we need, and that’s the end of it.

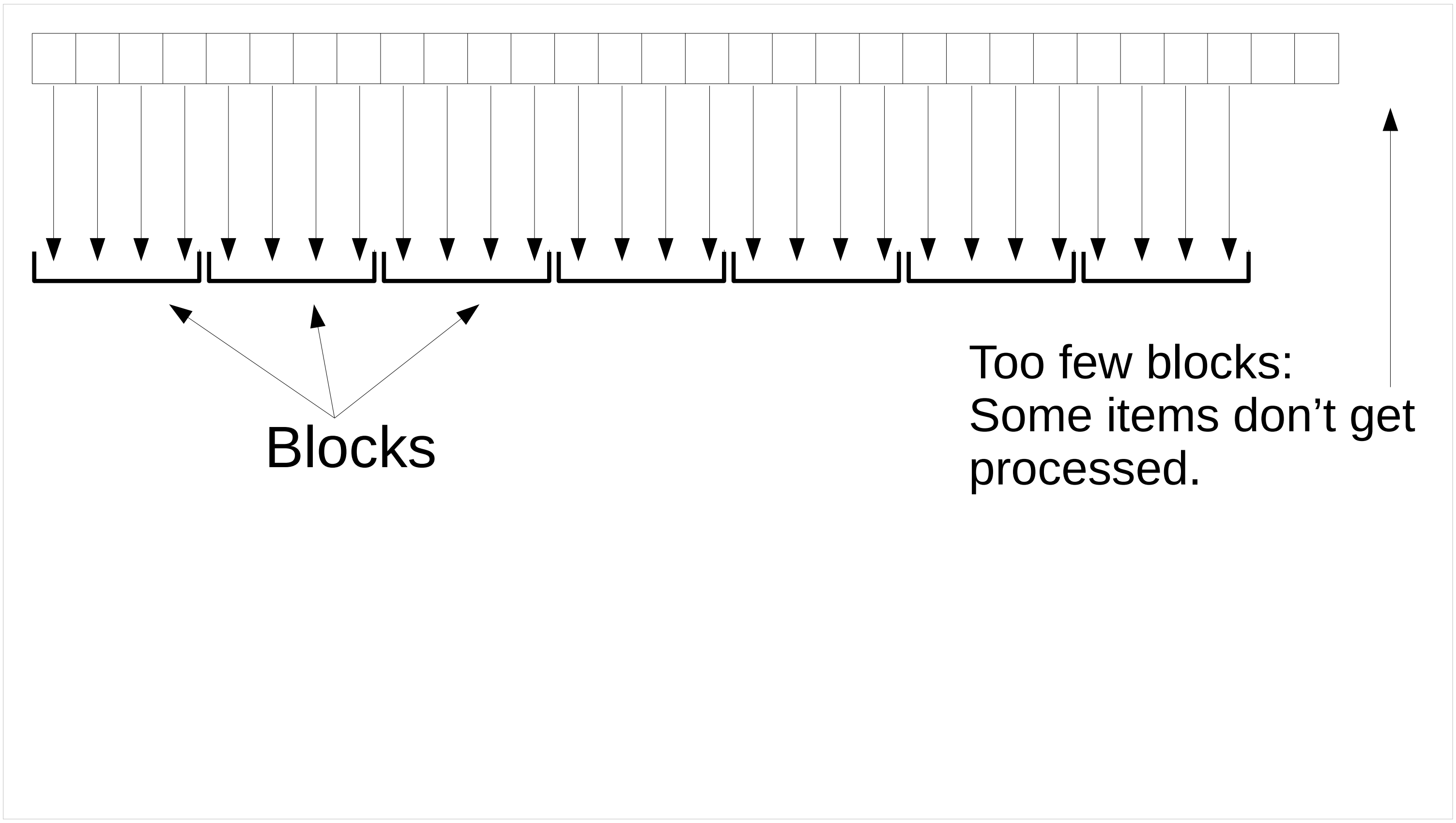
Unfortunately, it’s a bit more complicated:

Threads are grouped into Blocks.

Reason: Thread Cooperation

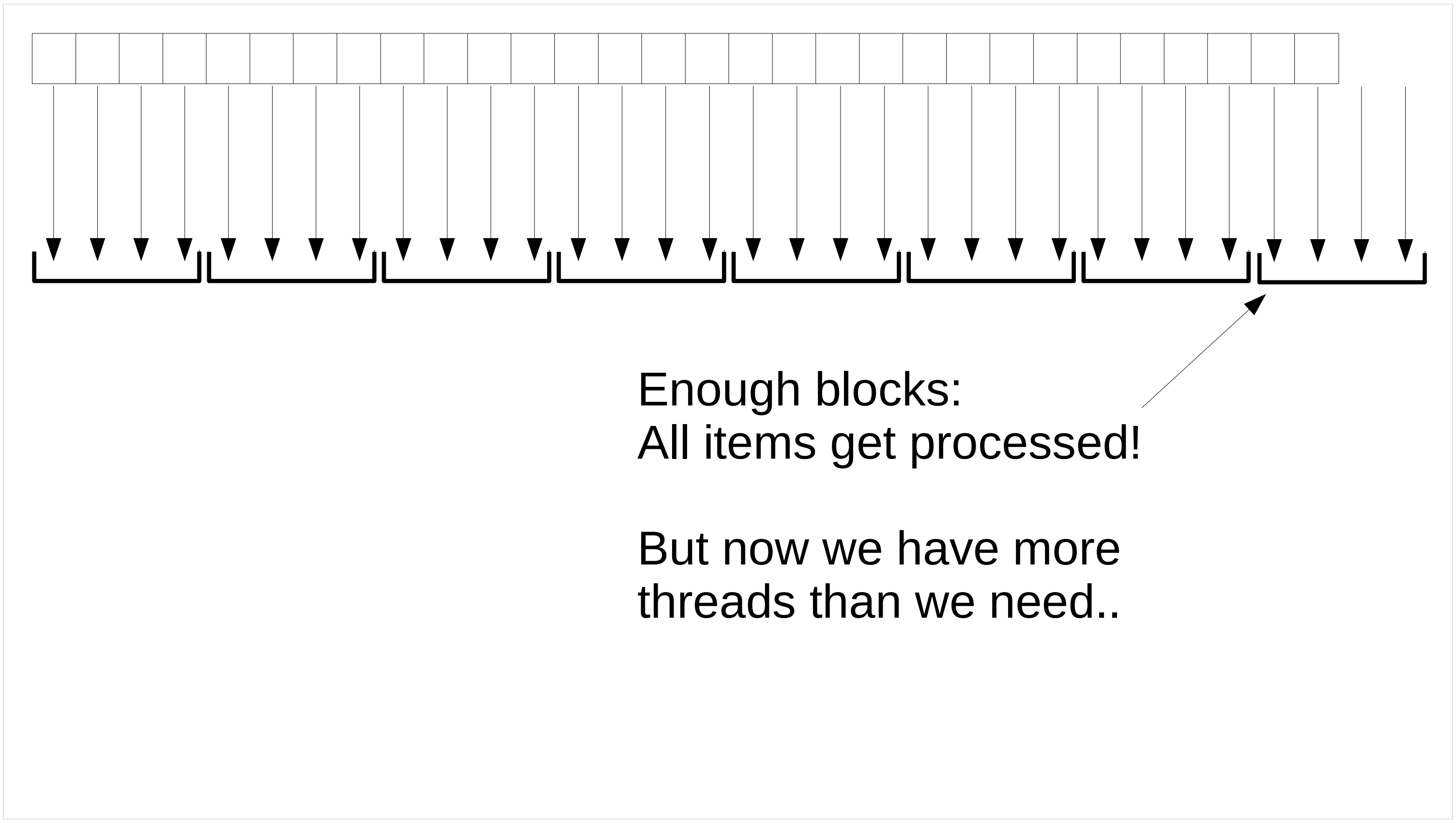
Actually.. It’s a bit more complicated than that. Remember blocks from last time? The GPU requires that we group our threads in so-called blocks.





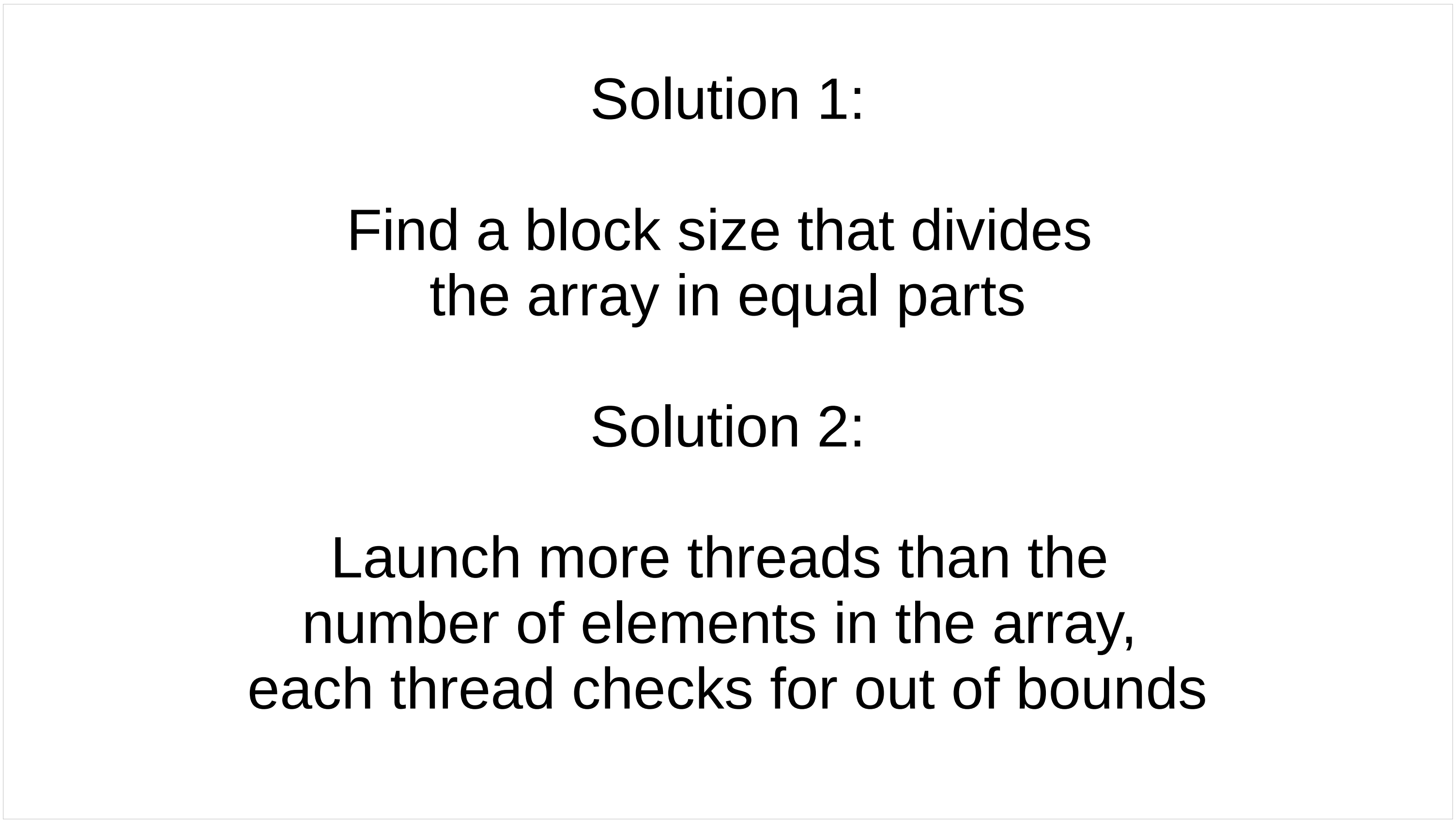
The only way you can spawn threads is to choose a certain number of threads you want each block to have, and subsequently how many blocks of threads you want to execute.

Unfortunately, there's a little problem now: the buffer you're trying to process doesn't always match the number of threads.



But not to worry! We can just spawn an additional thread block, and all our worries are gone.

Until we resolve the segmentation fault that occurs due to the last threads requesting indices that are out of bounds, anyway.



Solution 1:

Find a block size that divides
the array in equal parts

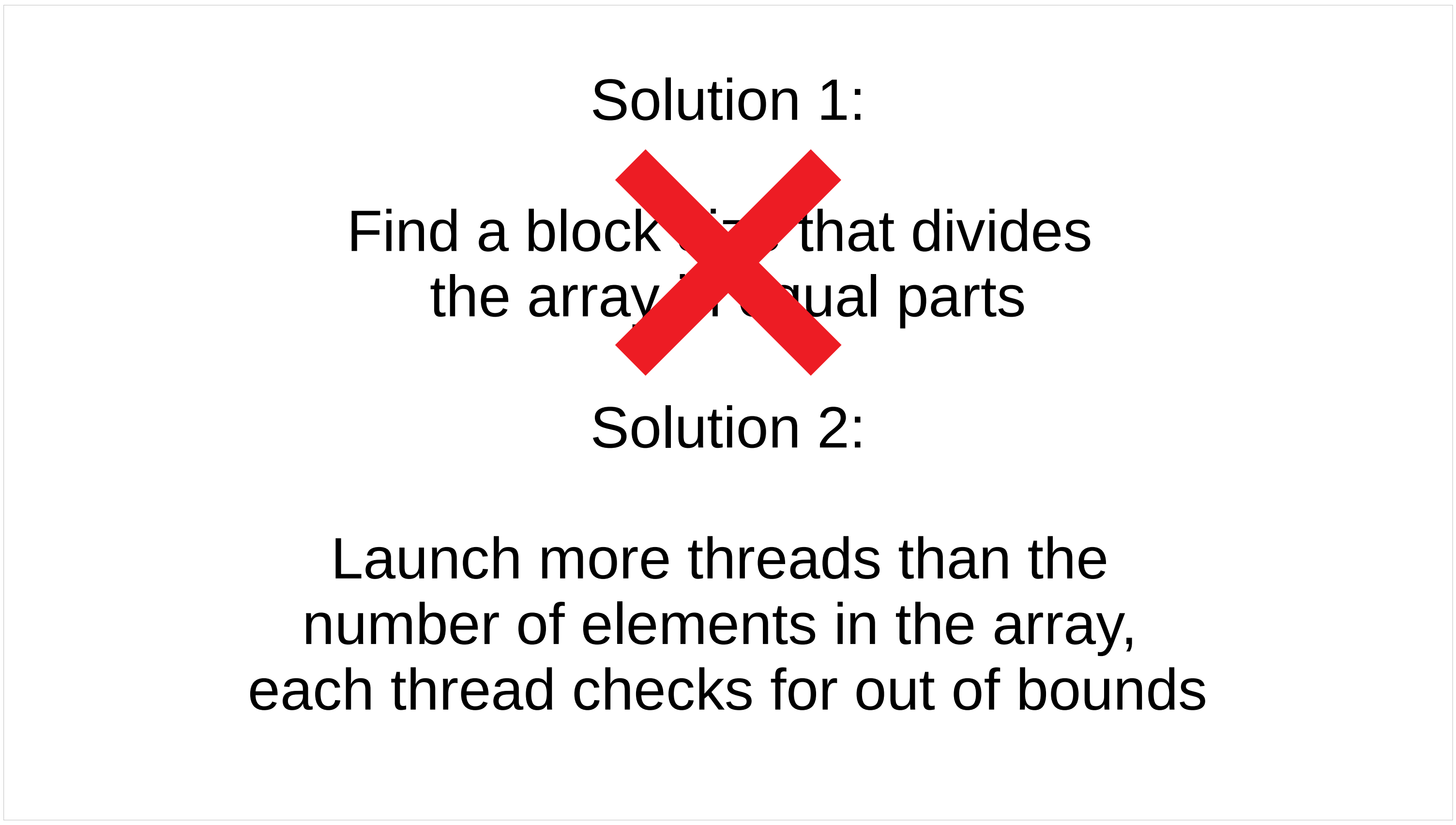
Solution 2:

Launch more threads than the
number of elements in the array,
each thread checks for out of bounds

Of course you can divide the number of threads into equally sized blocks if you can find a number of threads that divides the original number you actually need.

However, that doesn’t work if your array happens to be a prime number of elements long.

So instead, the solution is always to launch some extra threads, and ensure the extras immediately return before they can read any memory that’s out of bounds.



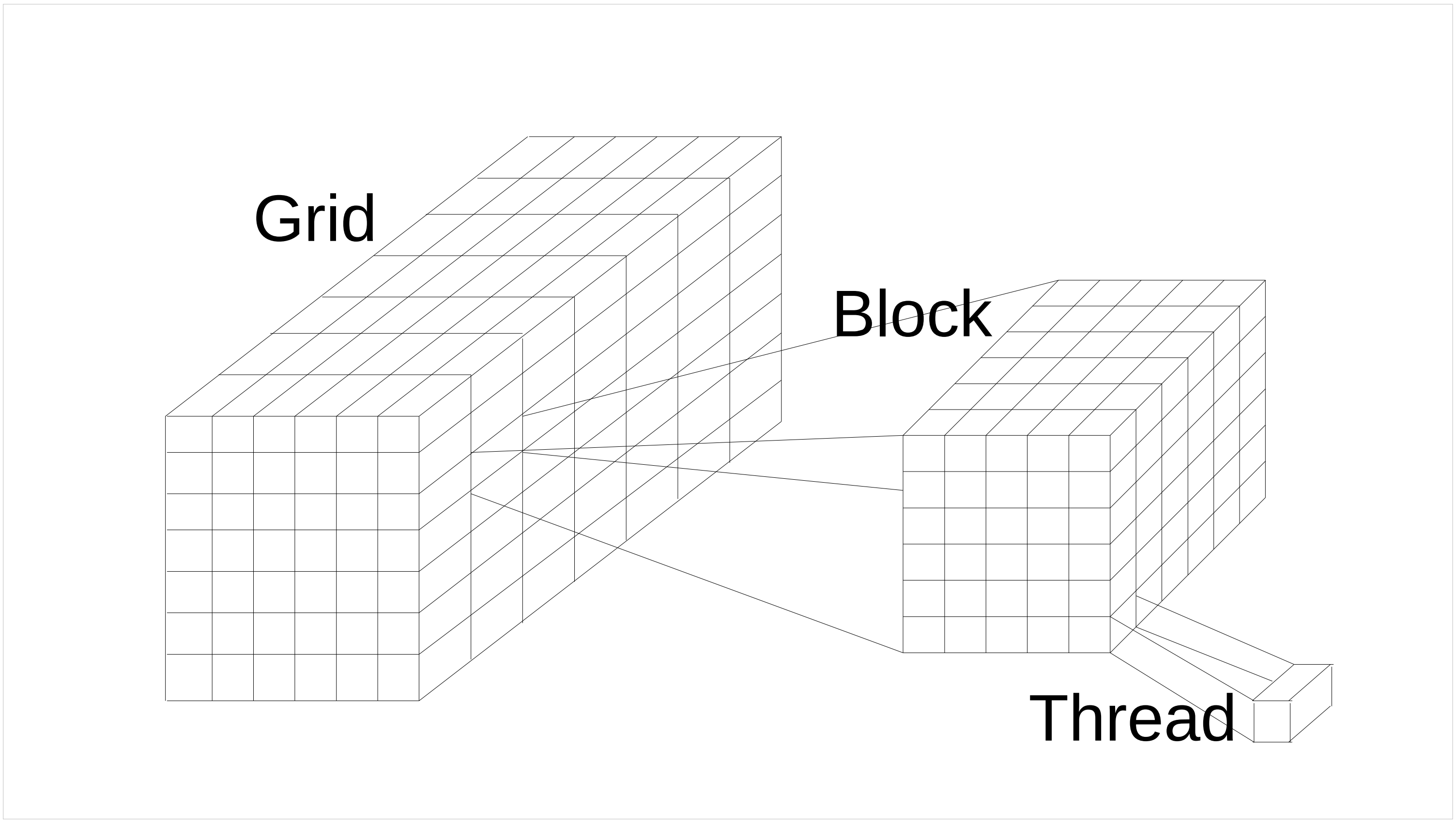
Solution 1:

~~Find a block size that divides the array in equal parts~~

Solution 2:

Launch more threads than the number of elements in the array, each thread checks for out of bounds

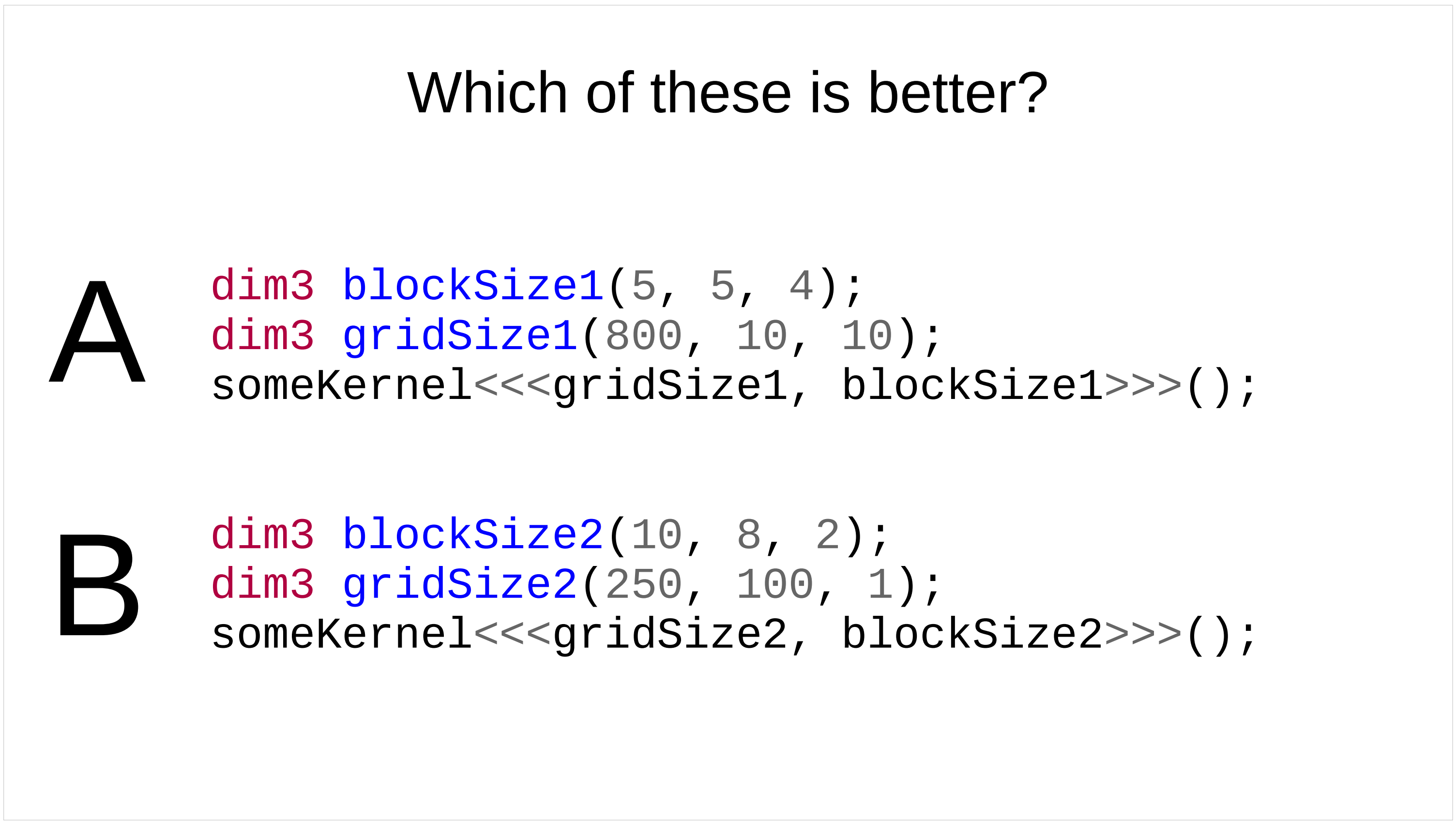
Launching additional threads may seem wasteful, but don’t worry, it really isn’t.



CUDA uses this thread hierarchy:

A grid is a three-dimensional structure of blocks.

Blocks are a three-dimensional structure of threads.



Which of these is better?

A

```
dim3 blockSize1(5, 5, 4);
dim3 gridSize1(800, 10, 10);
someKernel<<<gridSize1, blockSize1>>>();
```

B

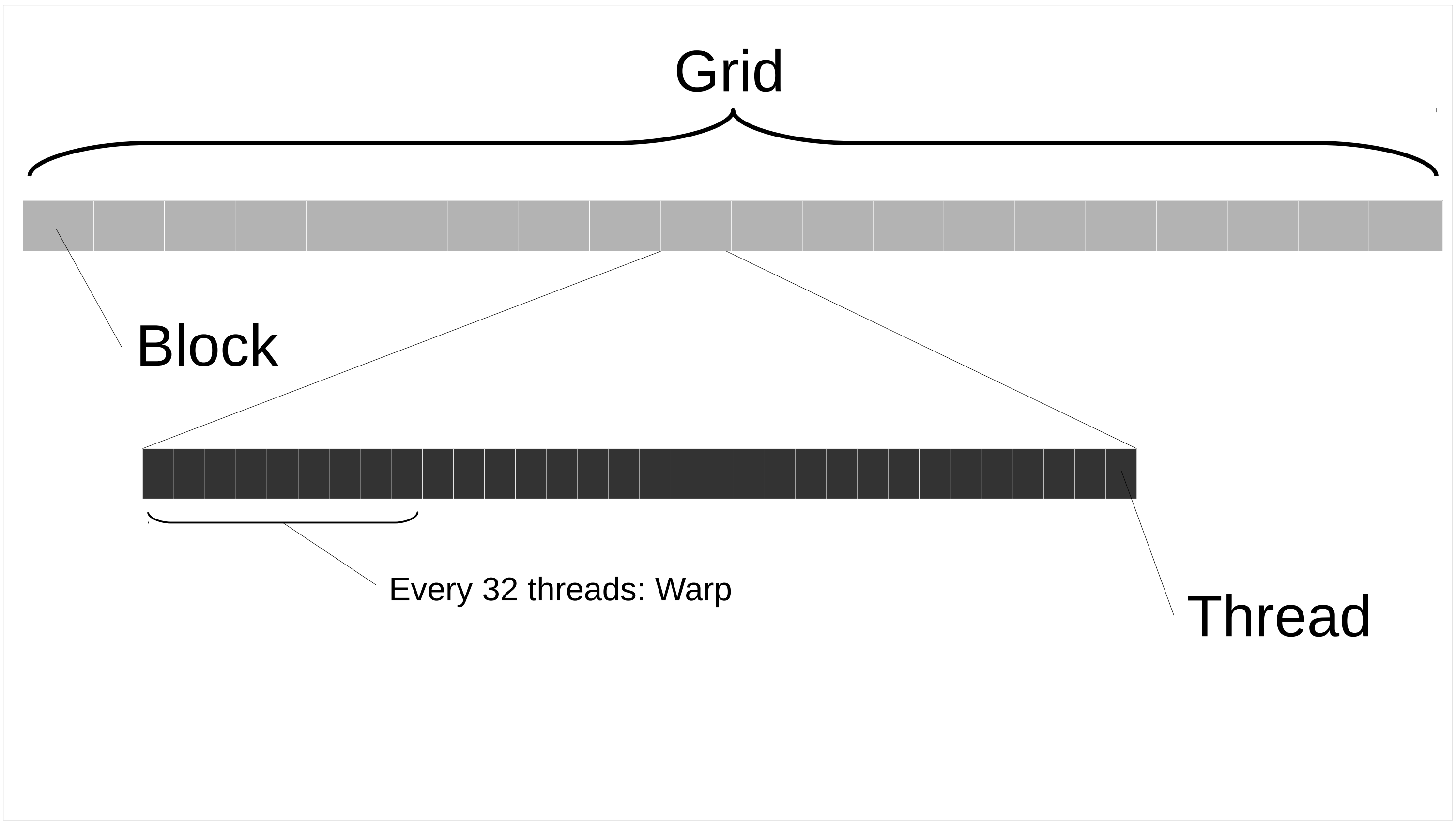
```
dim3 blockSize2(10, 8, 2);
dim3 gridSize2(250, 100, 1);
someKernel<<<gridSize2, blockSize2>>>();
```

In theory, there are many ways of launching threads. For instance, I can launch 6 threads as two blocks of 3 threads, or 3 blocks of 2 threads each.

One of the key topics of today is that not all block sizes are made equal. And I already mentioned one of them last week.

In this case, A launches 100 threads per block, while B has 160. Since the GPU can only execute warps of threads, it rounds up the number of threads in a block to the nearest multiple of 32. In the case of A, that will mean one of the warps only has 4 active threads in it, while 28 are doing nothing.

Meanwhile, B’s 160 perfectly divide 32, and thus causes no threads to idle.

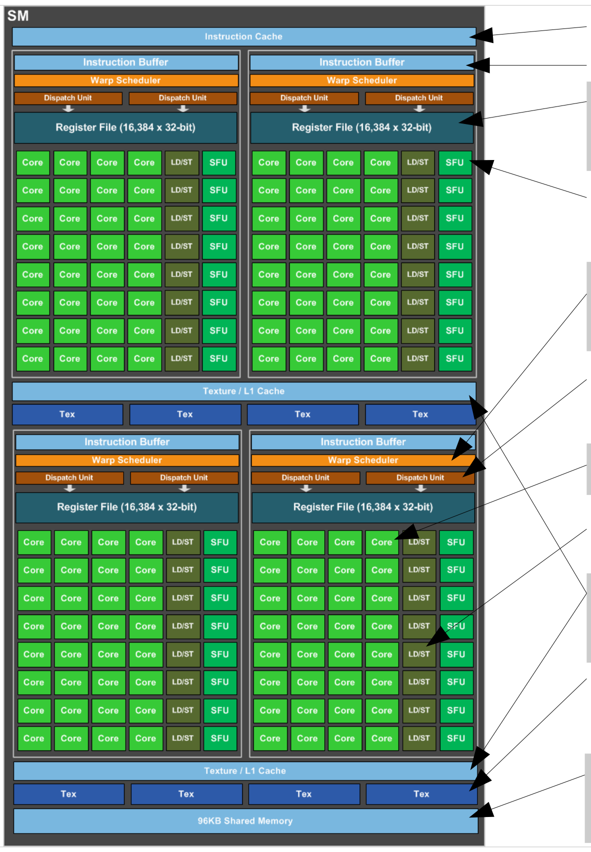


So while you can specify arbitrary numbers of threads in a block, you generally want to make sure the product of the X, Y, and Z dimensions of your blocks are multiples of 32 (multiple dimensions are “flattened” and cut into warps).

Last time:

Thread Hierarchy

Streaming Multiprocessors (SM’s)



Instruction Cache

Instruction Buffer: Small local instruction cache

Register File: Stores all register values of all threads

Special Function Units: Compute special instructions, operations for rendering

Warp Scheduler: Determines which warp gets to execute next.

Dispatch Units: dispatch commands to functional units (FP/DP cores, LD/ST, SFU, Tex, ….)

Stream Processor: Essentially an FPU + ALU

Load / Store unit: Handles memory requests

Texture / L1 cache: stack memory for threads, 2D data used by texture units

Texture units: hardware implementations for handling 2D data

Shared memory: A programmable L1 cache Used for cooperating between threads

The streaming process is the GPU's main execution. I'm going to talk about these components in detail during this lecture, so I'm listing them here.

Today:

GPU Performance

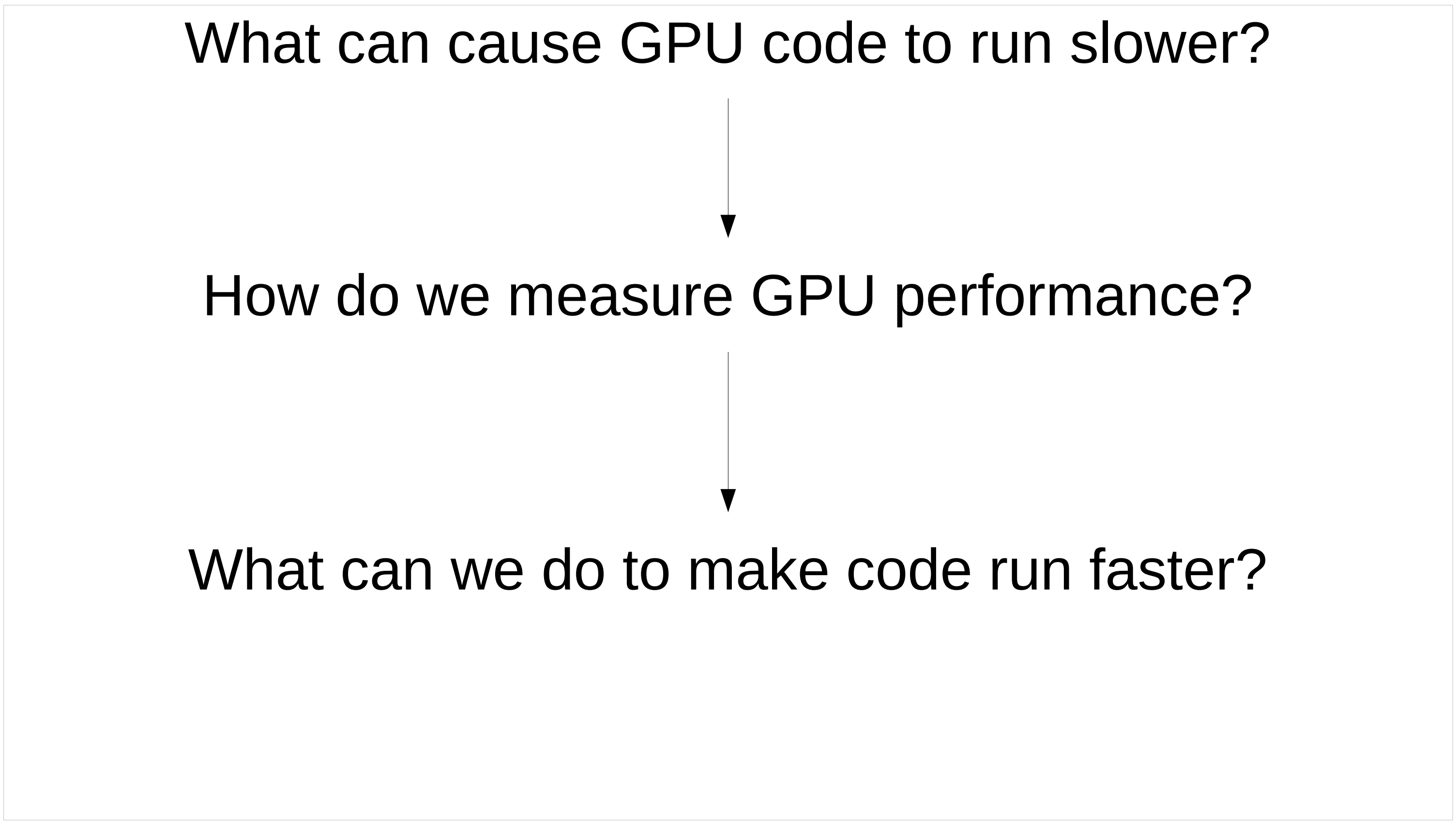
TL;DSTTL:

Use the GPU

Measure

The main takeaways of this lecture are that a GPU by its nature will be slower than a CPU if you don’t properly write code for it. Your job is therefore to ensure the available hardware is *utilised*, rather than idling.

And of course, the most important thing when trying to write code that is fast, is to keep measuring. Always. It’s the only means by which you can be sure whether your changes have a positive effect.



$$\begin{tikzcd}
\text{What can cause GPU code to run slower?} \arrow[d] \\
\text{How do we measure GPU performance?} \arrow[d] \\
\text{What can we do to make code run faster?}
\end{tikzcd}$$

The general structure of today are three questions.

Part 1/3:

What can cause GPU code to run slower?

Non-Coalesced Accesses
Low Occupancy
Thread Divergence
Register Spilling
Resource Contention
Synchronization

And many more..

These are the main things that can cause your code to run slower. Of course there are reasons that count for the CPU too; you may for instance be using a very poor algorithm to solve your problem.

But these factors here are those specific to a GPU, and which I believe tend to have the greatest impact on performance.

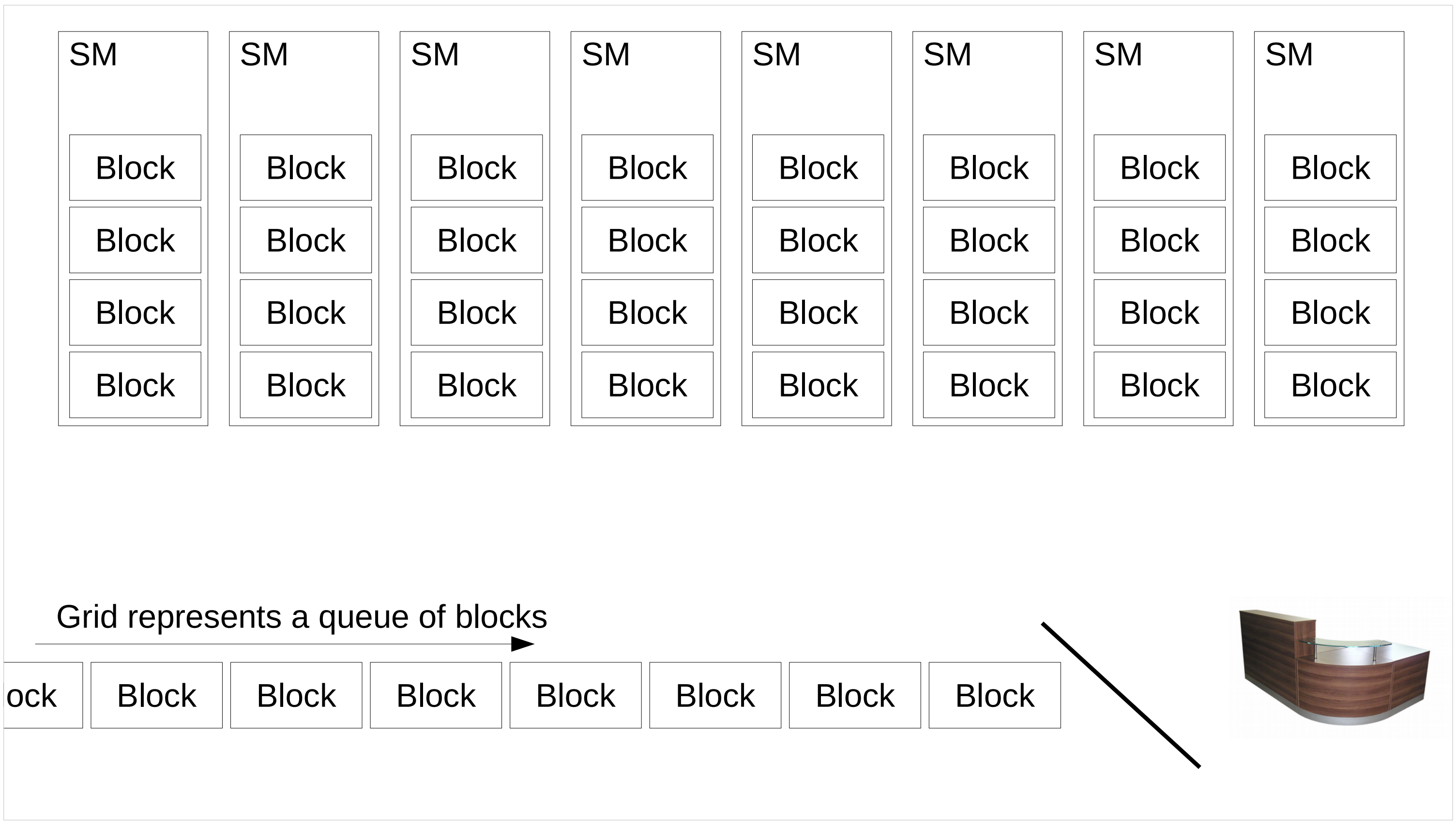
But first:

In order to understand these,
we need to understand the
hardware a little better.

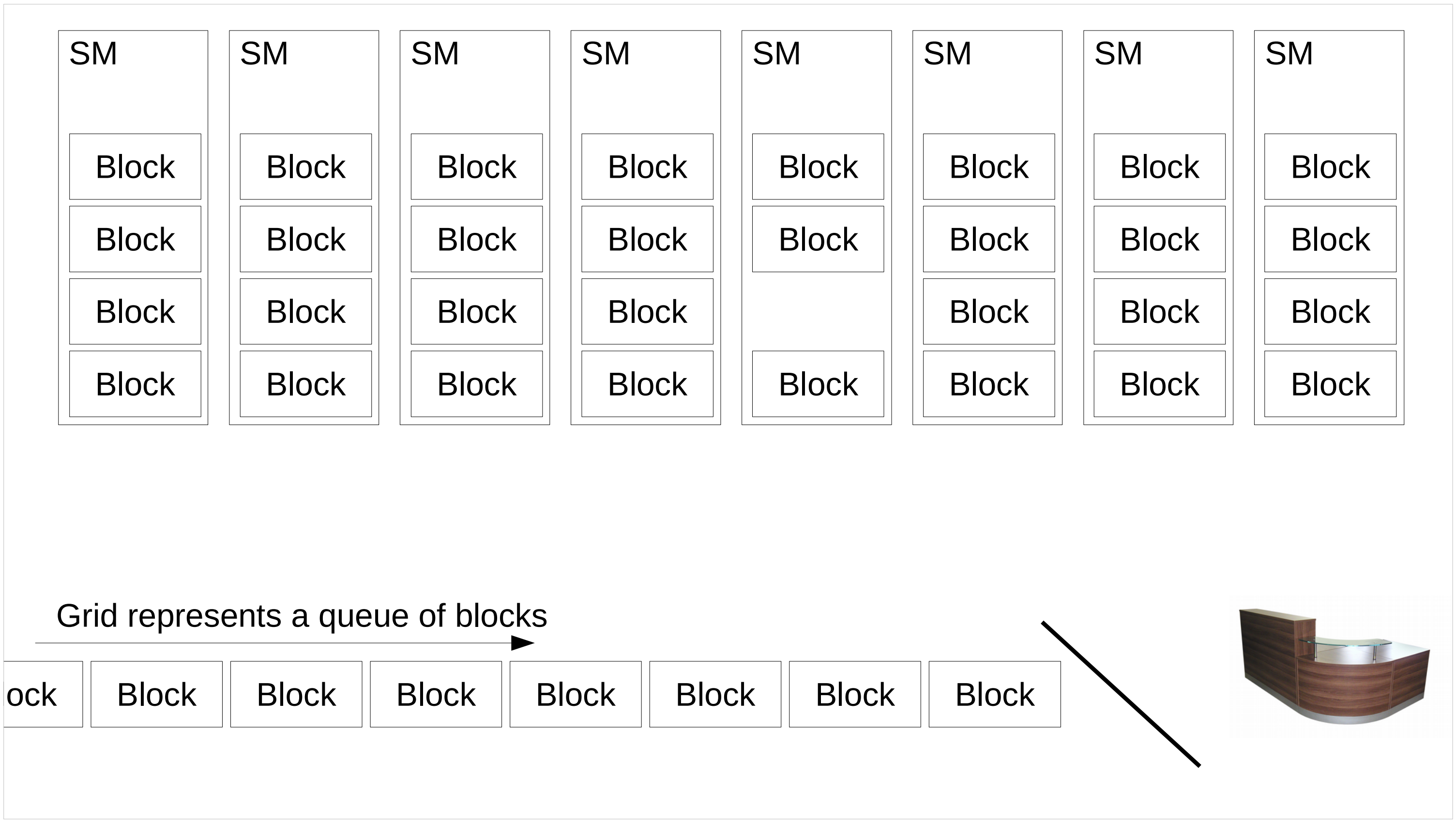
But talking about those requires understanding *why* they would cause your code to run slow.

That in turn requires understanding how the hardware executing them works. These issues really are caused by improperly using the hardware (or perhaps the lack thereof).

How do Streaming Multiprocessors execute kernels?

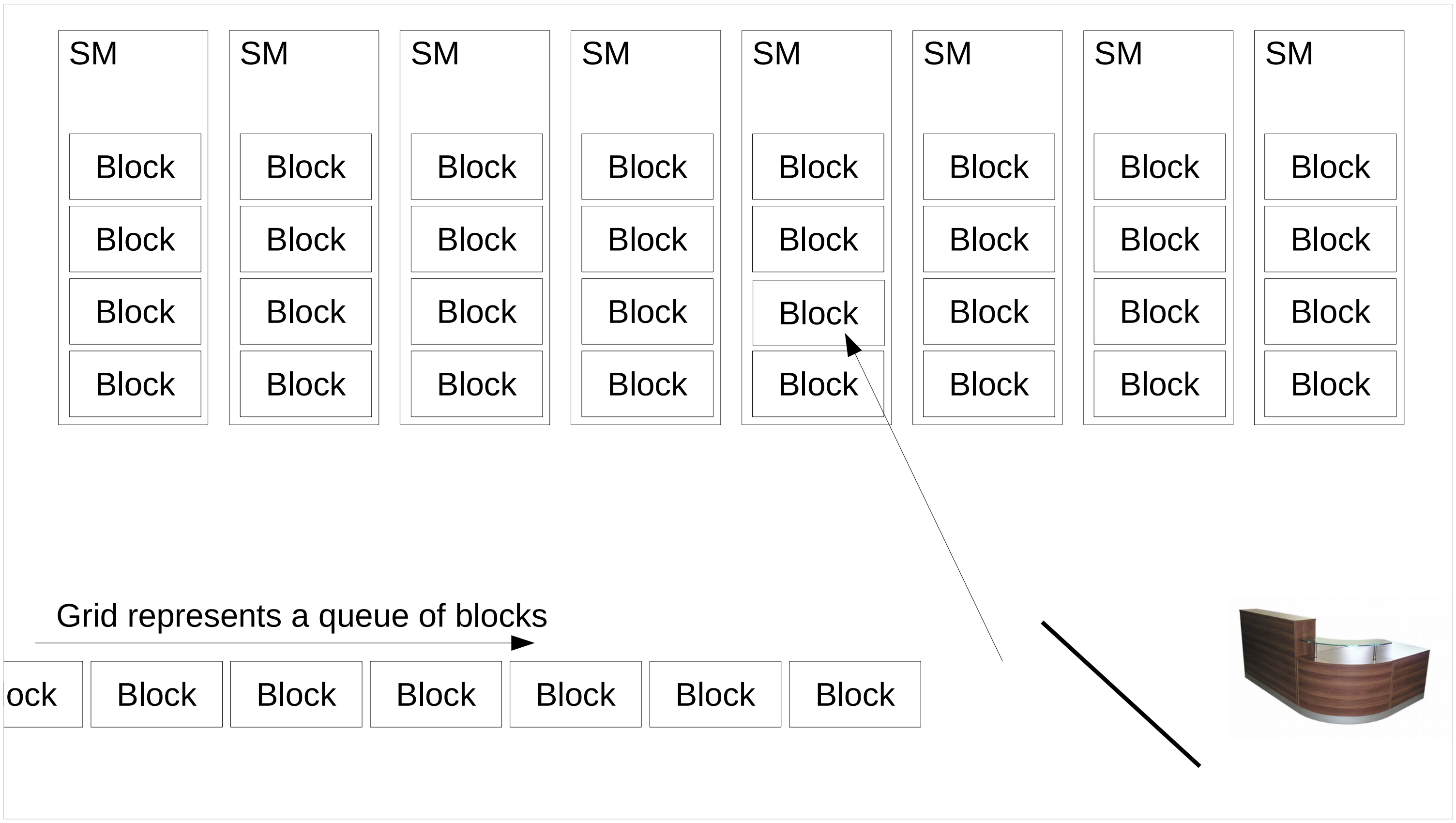


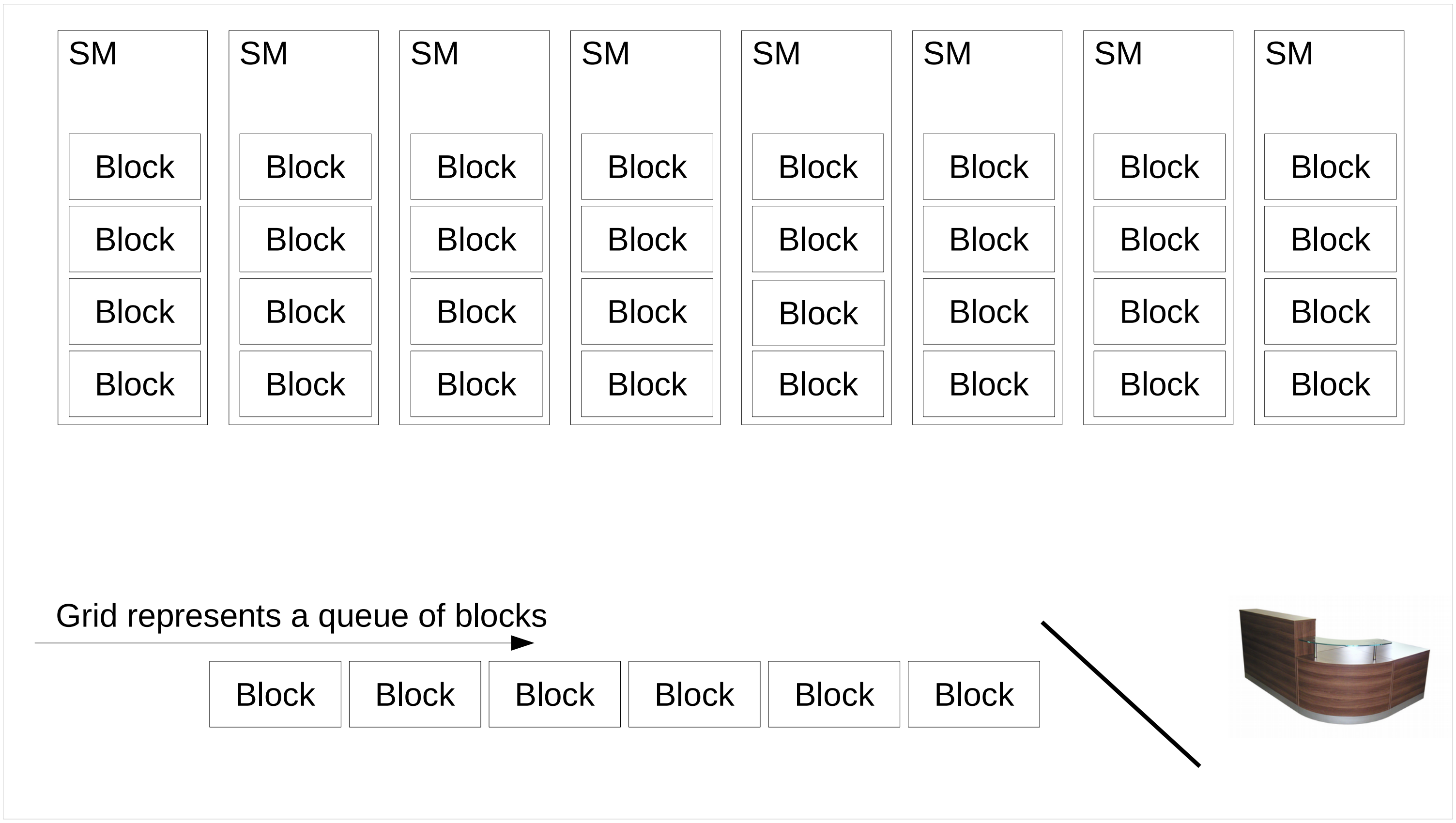
Streaming multiprocessors consume blocks. They can process a certain number of blocks at the same time, and process those blocks in full before starting on another one.



The grid is essentially a large queue of such blocks.

The fact that blocks are executed in full is really important here, and we will see why later.


SM
SM
SM
SM
SM
SM
SM
SM
Block
Grid represents a queue of blocks
ock
Block
Block
Block
Block
Block
Block


SM
SM
SM
SM
SM
SM
SM
SM
Block
Block
Block
Block
Grid represents a queue of blocks
Block
Block
Block
Block
Block
Block

How many blocks can be active on an SM?

How many blocks can be active on an SM?

Device Limit: 32 (since Maxwell)

The Streaming Multiprocessor schedules warps in hardware. As a direct result, the internal scheduler is only able to handle a certain number of threads simultaneously.

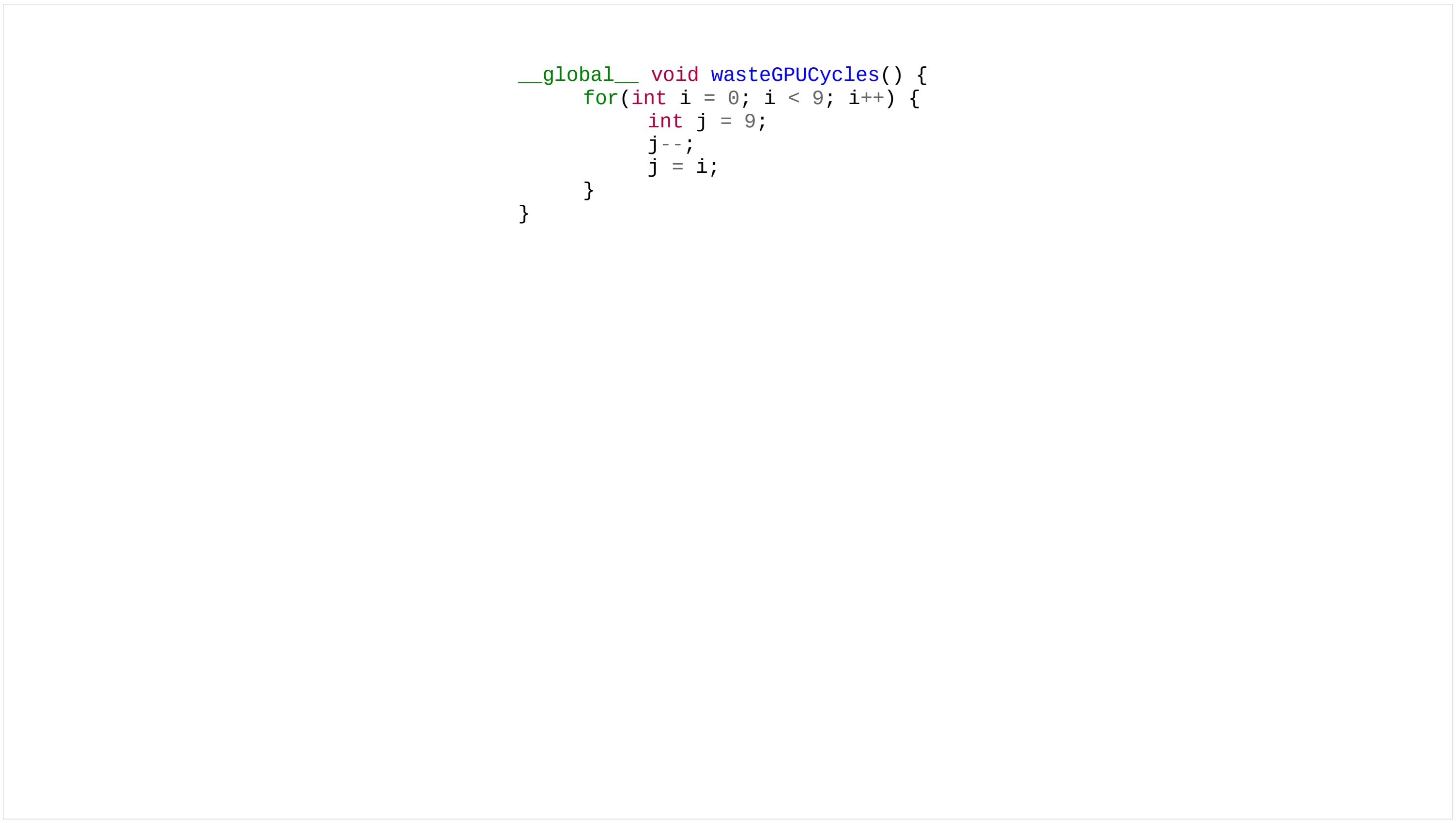
However, that hard limit is not the only thing limiting the number of blocks that can execute on a single SM at the same time.

So the answer to how many blocks can be active at any given time is a bit more nuanced than what I let on last time.

How many blocks can be active on an SM?

Device Limit: 32 (since Maxwell)

In practice: more nuanced..

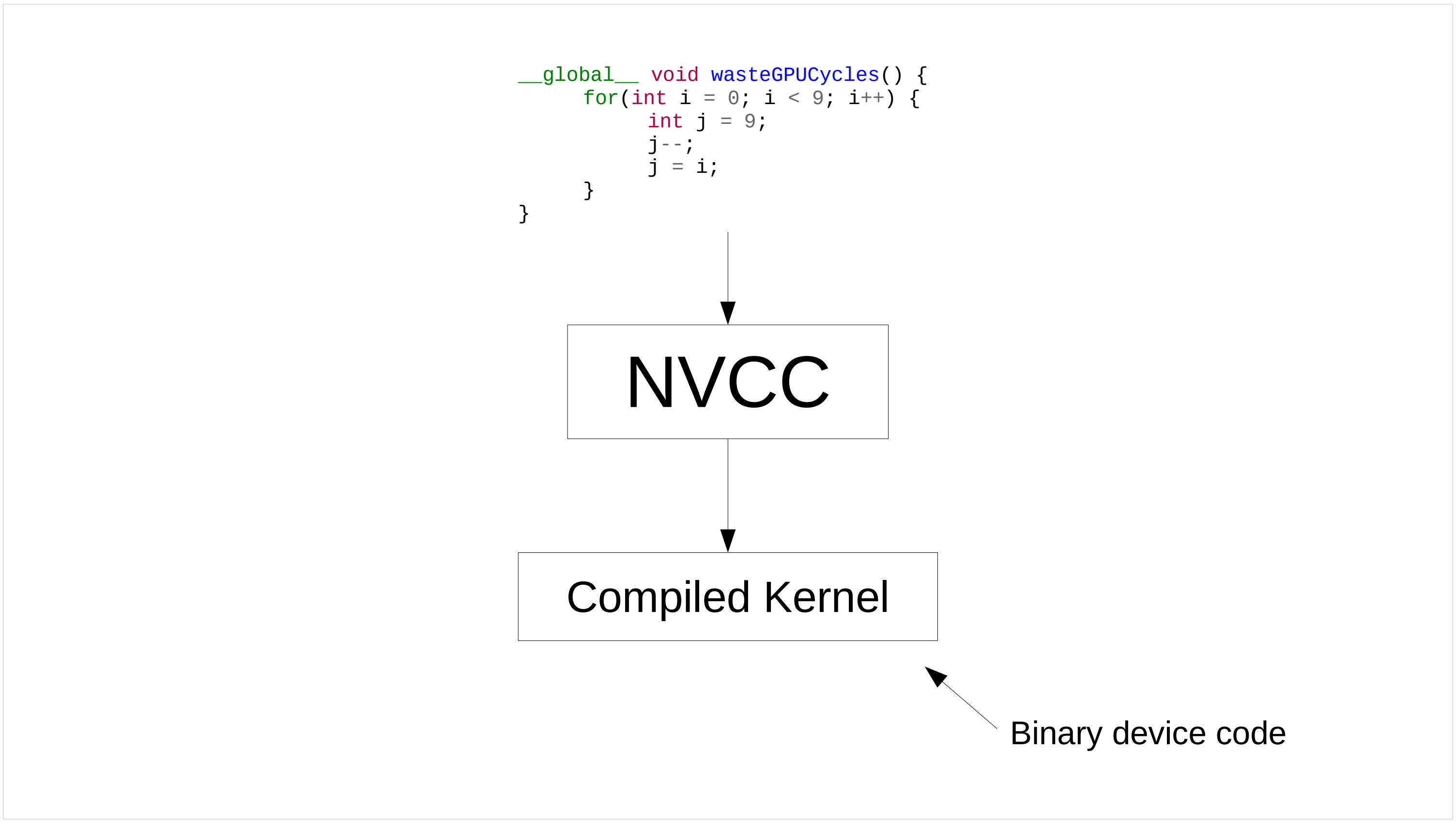


```
__global__ void wasteGPUCycles() {
     for(int i = 0; i < 9; i++) {
          int j = 9;
          j--;
          j = i;
     }
}
```

You see, if I take a kernel, and compile it into device code, I end up with compiled binary code.

An important difference between the CPU and GPU is that the GPU allows threads to have varying numbers of registers.

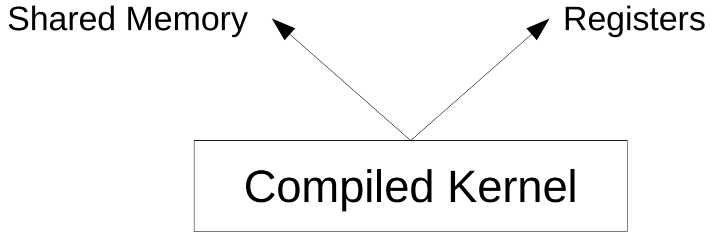
The CPU tends to have a certain fixed number of general purpose registers, but the GPU, due to its design, allows you to use more than that.

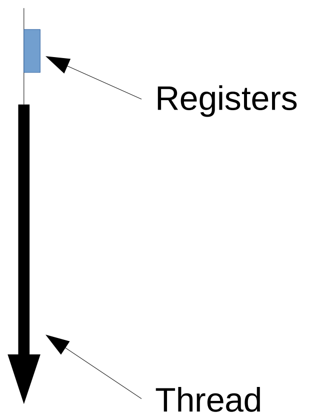


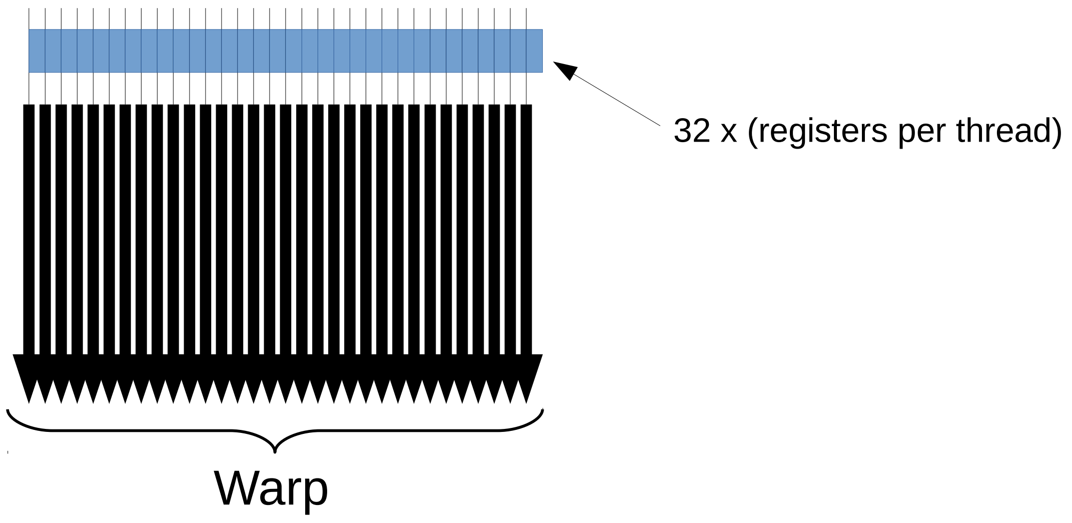
```
__global__ void wasteGPUCycles() {
      for(int i = 0; i < 9; i++) {
            int j = 9;
            j--;
            j = i;
      }
}
```

The compiler will compile the piece of code, and need a certain number of registers to execute it. In general, the more values you have to hold on to for an extended period of time (for example an iterator variable in a for loop), the more registers your code uses.

That number also tends to increase with code complexity.

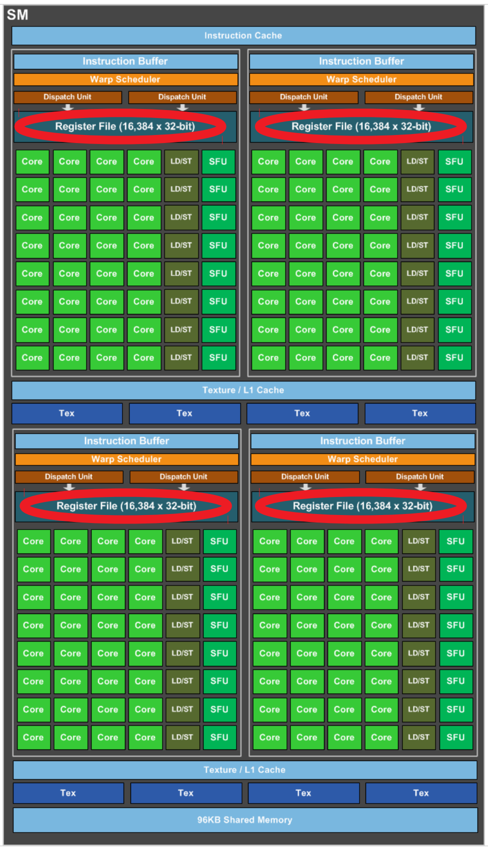

Shared Memory
Registers
Compiled Kernel


Registers
Thread



Because each thread requires a certain number of registers to execute, but we can only execute threads in warps, that means we need 32x the number of registers for a single thread to execute a warp.

I’m just going to simplify that Ultra® High™ Quality®™ Drawing to something that’s a bit smaller. The blue rectangle indicates the number of registers needed by the warp.



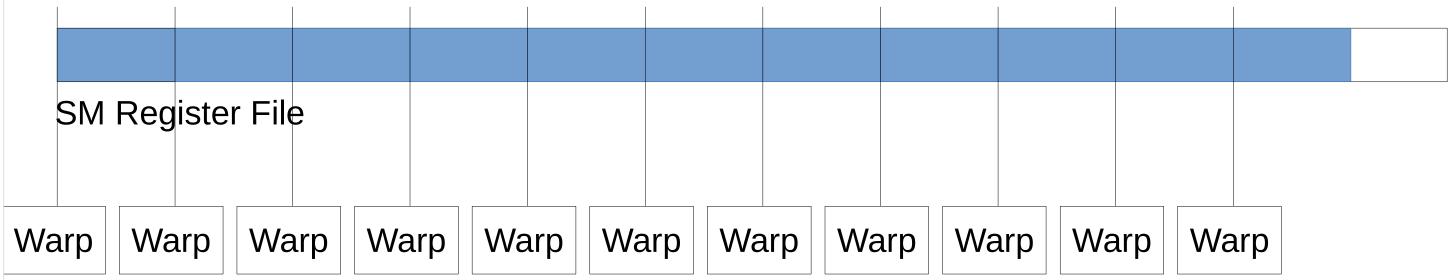
Now remember that the SM has a register file. This register file is the primary reason threads can have varying numbers of threads on the GPU, rather than being limited to a fixed quantity like on the CPU.

SM Register File
Warp

This register file allows the SM to keep hold of all register values of all threads that are currently executing on it.

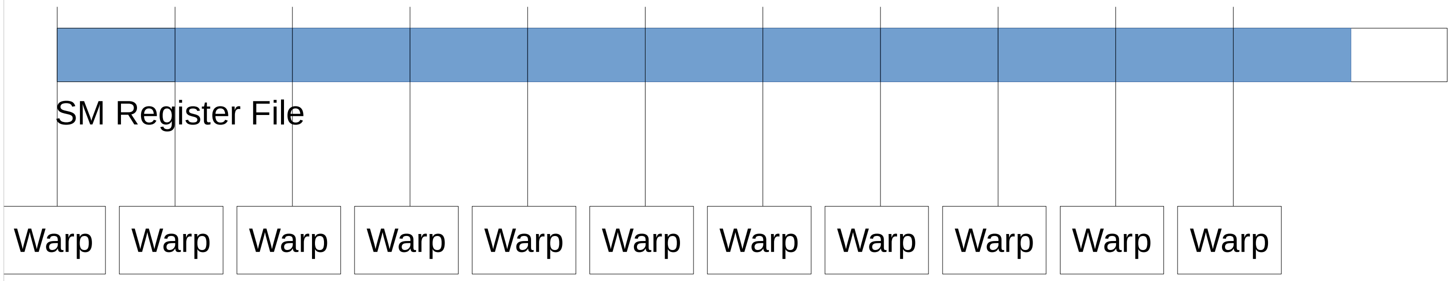
And this is a big deal; whenever the CPU needs to switch to a different thread, it needs to dump all register values to memory, and load in the ones from the other thread, so it can continue where it left off.

The GPU keeps all registers of all threads (1024 of them if it so desires) around until they have finished executing.



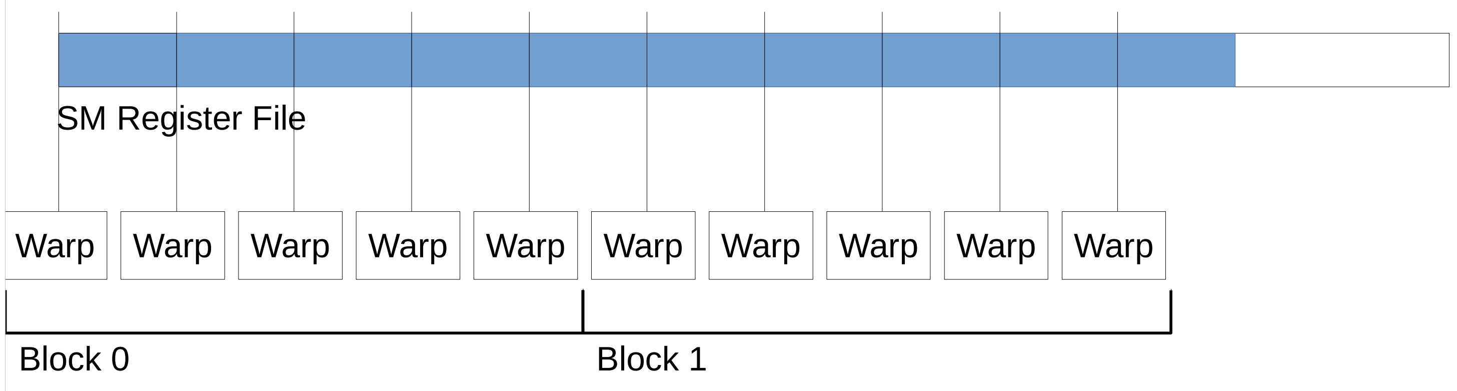
Number of warps that can execute on the SM is dependent on the number of registers needed per thread

Now this register count also determines how many warps can be active at the same time. Since the register file has a certain number of registers, and a warp uses a certain number of them, we can only fit a certain number of them inside the register file.



Number of warps that can execute on the SM is dependent on the number of registers needed per thread

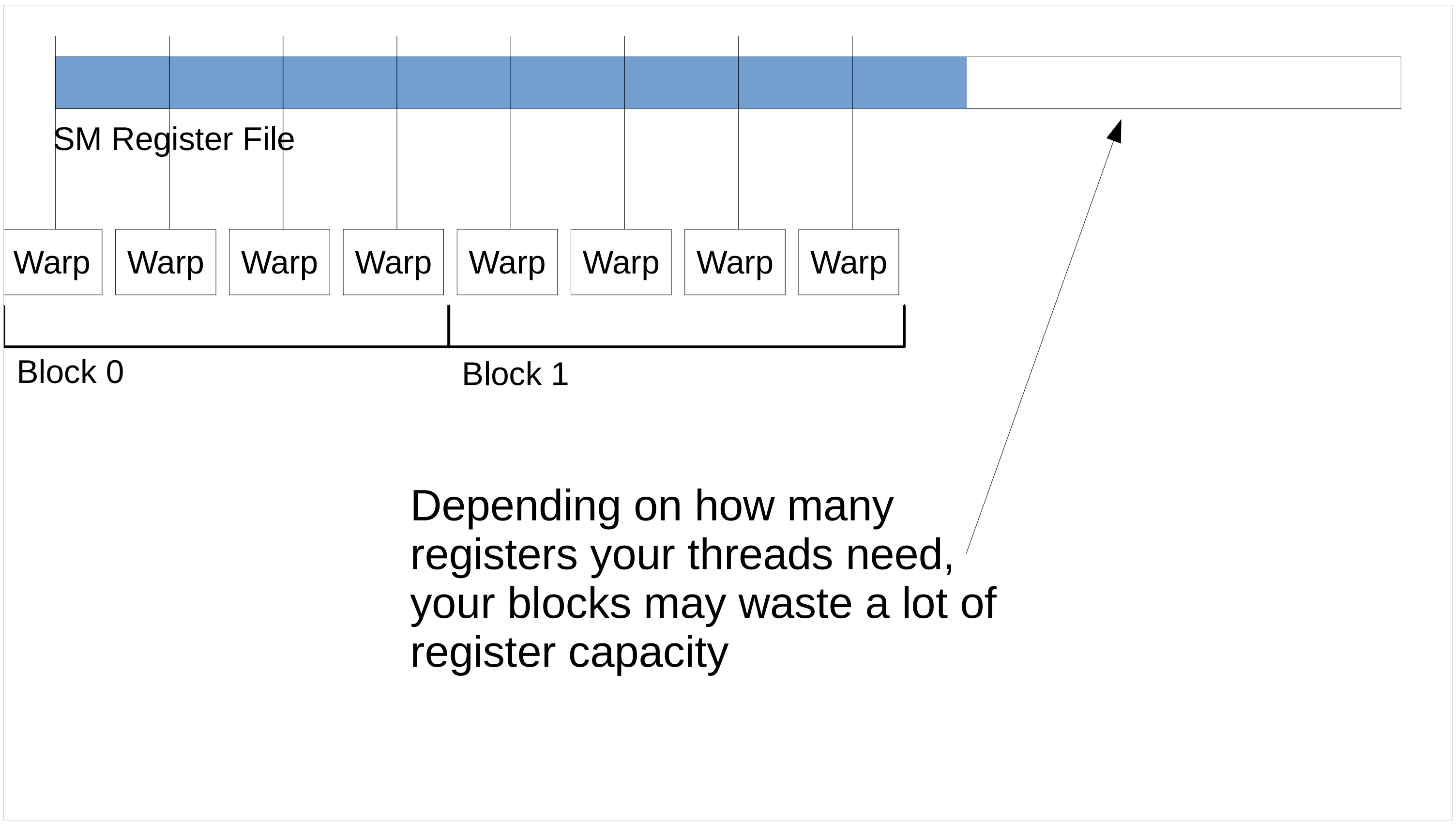
But what about blocks?



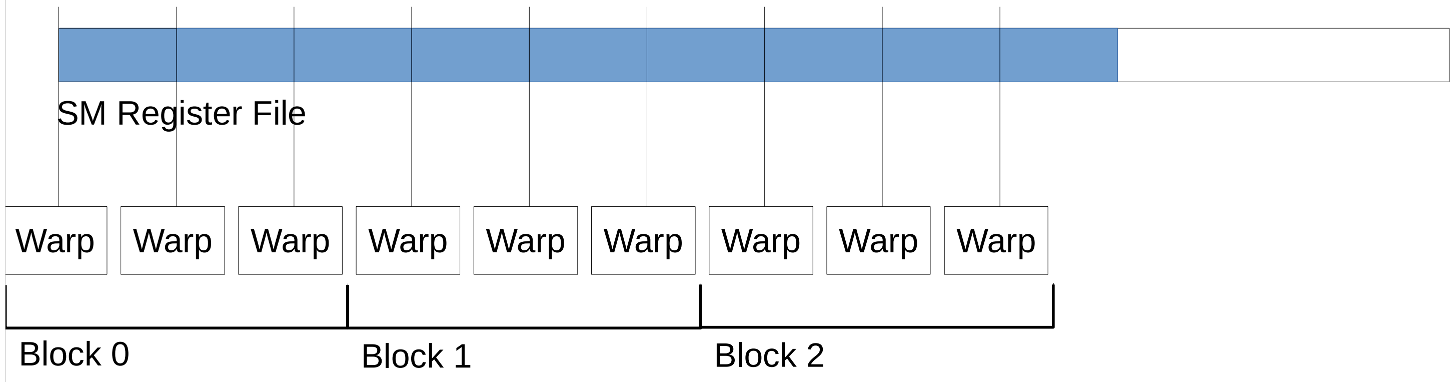
Blocks have a constant number of warps

But must be assigned to an SM in full!

Blocks are an additional issue there. They are allocated *in full* to the SM. The number of threads in your blocks therefore affect how many warps can be executing simultaneously inside the SM.

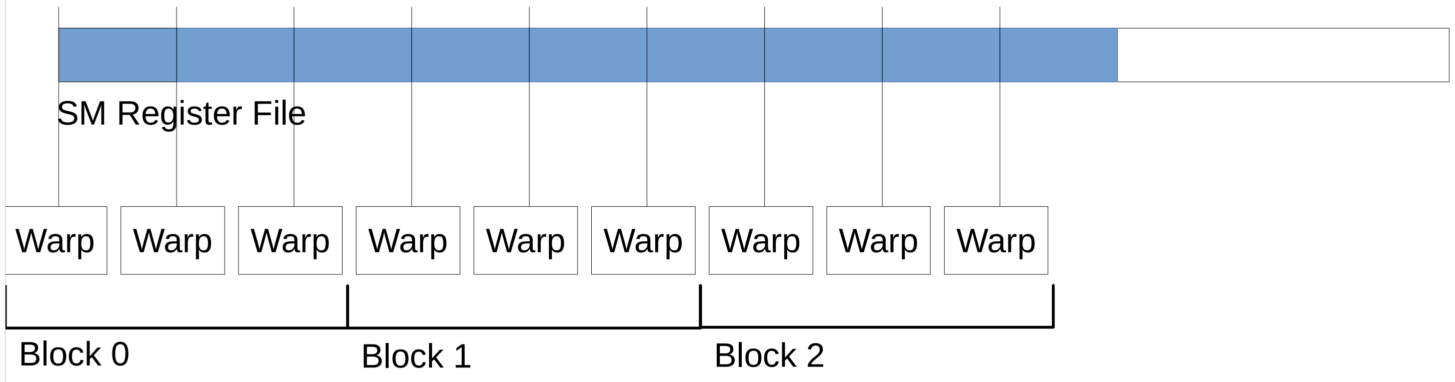

SM Register File
Warp
Warp
Warp
Warp
Warp
Warp
Warp
Warp
Block 0
Block 1
Depending on how many registers your threads need, your blocks may waste a lot of register capacity

For instance, if we vary the number of warps inside a block, depending on the register count per thread, we can sometimes be left with pretty big gaps of registers that are not used by anything. The number of warps per block have a direct effect on this.



Smaller block sizes may allow more blocks to be executing on an SM simultaneously.

Smaller block sizes generally allow more granularity in warp allocations, and better fill up the available space in the register file.

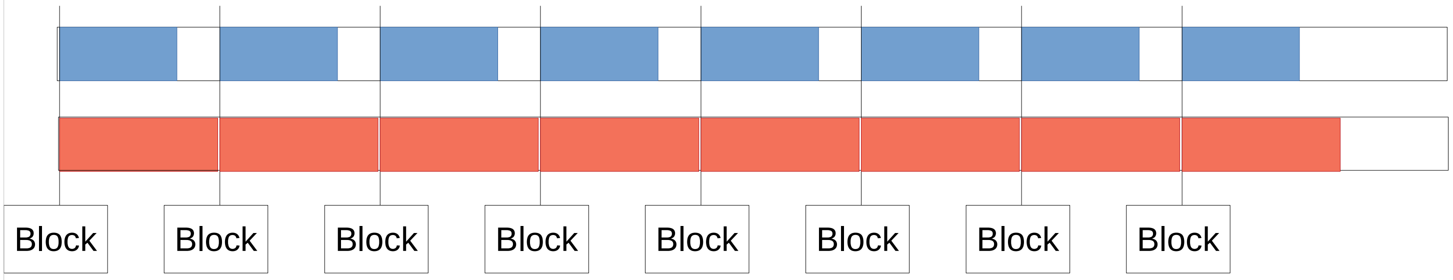


Also, blocks are not ejected from an SM until all warps have completed.

→ Big blocks with long threads can leave the SM underutilised.

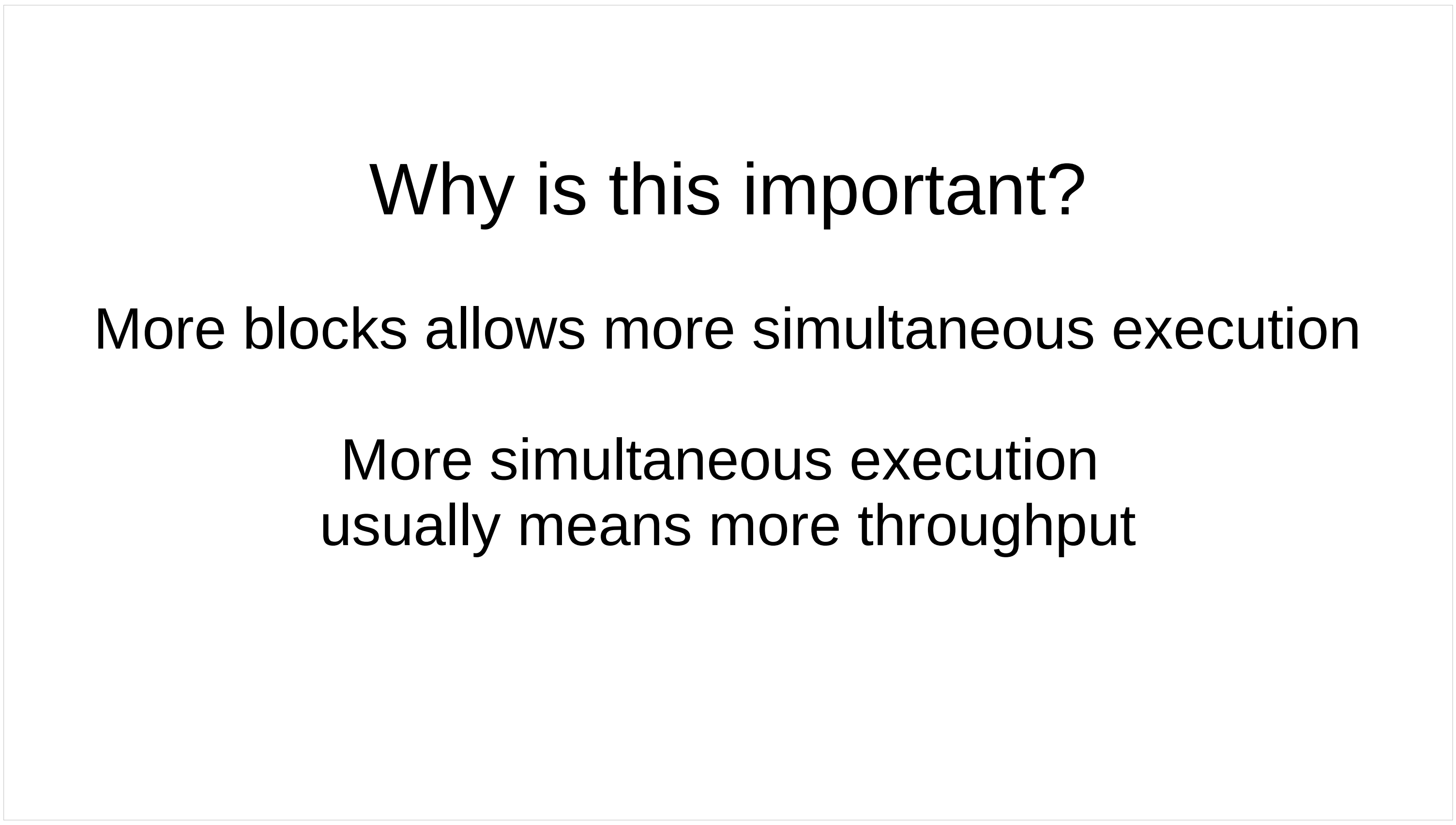
An additional issue here is that blocks execute in full until all their warps are complete. An additional problem of large block sizes is that they need to wait for all warps contained within them to finish before they can be ejected from the SM.

So from the SM’s perspective, in many cases smaller block sizes are better.



Shared memory can also limit the number of blocks that can execute on an SM simultaneously.

Shared memory is also a potentially limiting factor here. Each SM has a certain amount of shared memory available, and you can allocate chunks of it at a block level. If you use a large amount of it, you can also limit the number of blocks which your SM can execute simultaneously.



Why is this important?

More blocks allows more simultaneous execution

More simultaneous execution
usually means more throughput

I have now been spending quite a bit of time talking about what limits the number of blocks which can be executed at the same time, but haven’t really mentioned why that is important in the first place.

In general, the answer is that more active blocks means better parallelism and better utilisation of the available hardware.

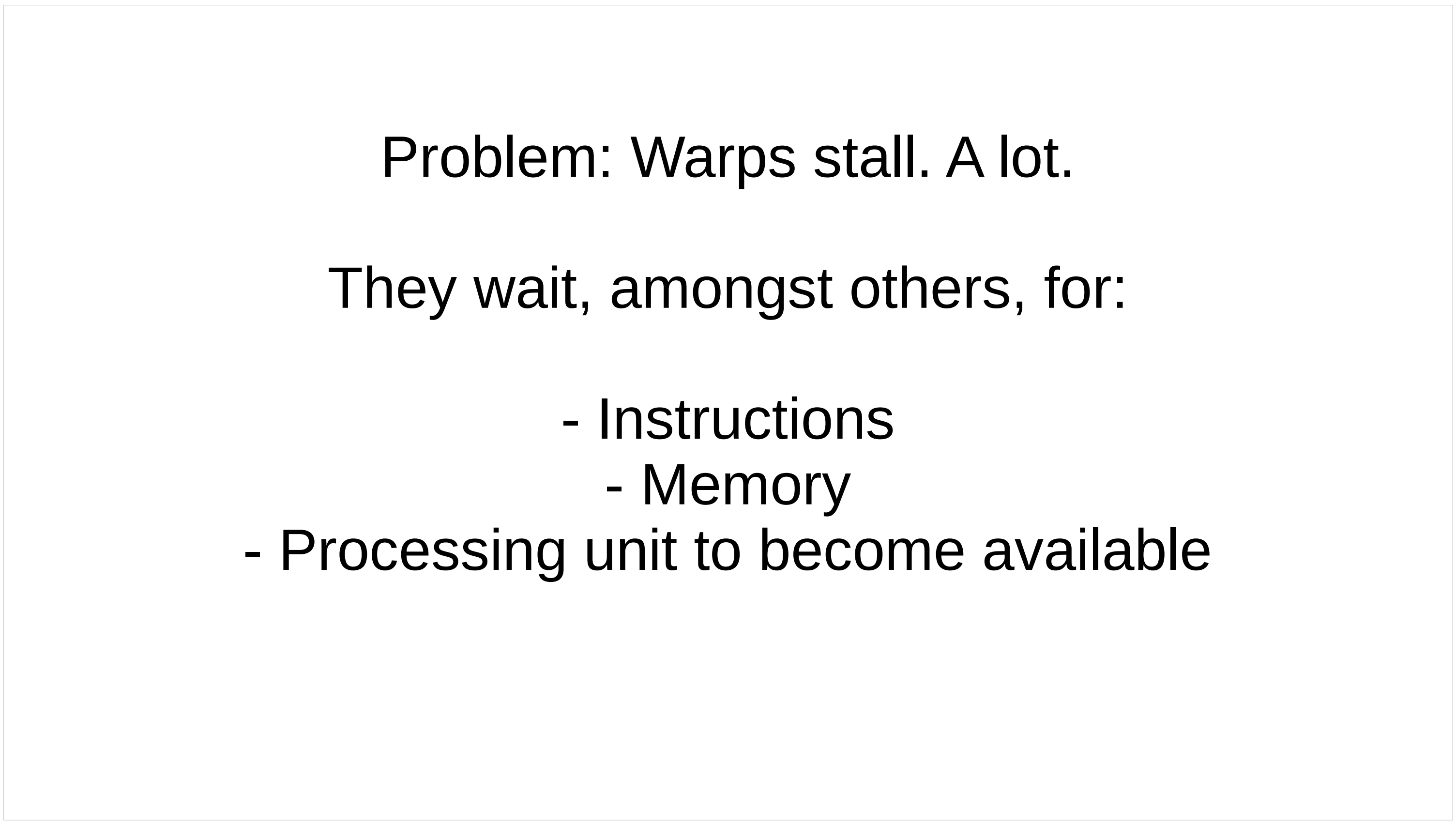
But let me explain why in detail.

Why is this important?

More blocks allows more simultaneous execution

More simultaneous execution
usually means more throughput

But that still doesn’t explain why we need a register file..



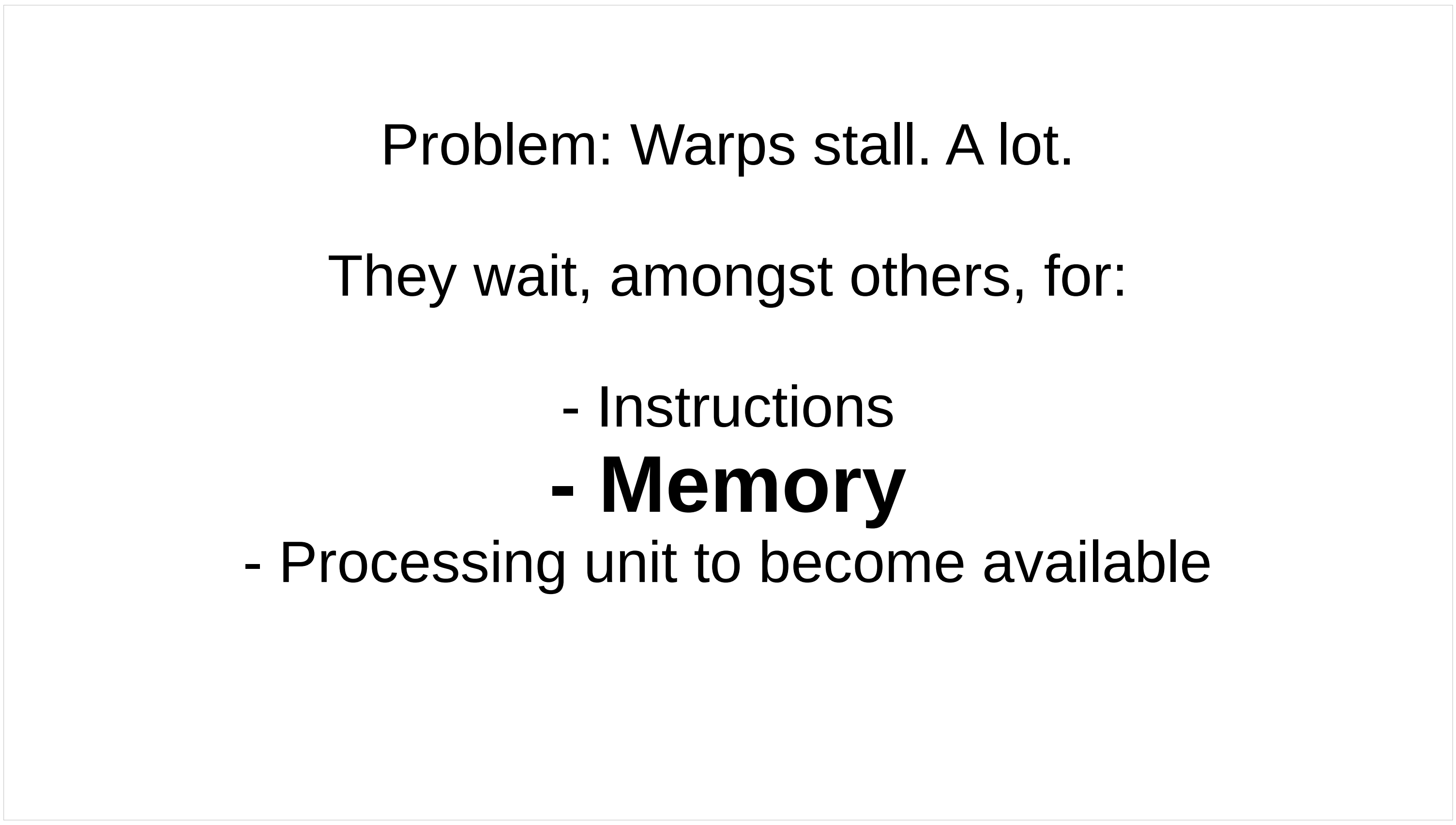
Problem: Warps stall. A lot.

They wait, amongst others, for:

- Instructions
- Memory
- Processing unit to become available

The primary problem this design solves is that in an extremely parallel environment, such as the GPU, contested resources everywhere cause warp stalls. A lot of them.

And they can be waiting for things such as instructions they need to execute, operands that need to come from memory, or shared hardware resources (texture units, special function units, etc) to become available (instructions on some of those tend to require multiple clock cycles to execute).



Problem: Warps stall. A lot.

They wait, amongst others, for:

- Instructions
- **Memory**
- Processing unit to become available

And from this list, memory is by far the most important one.

Problem: Warps stall. A lot.

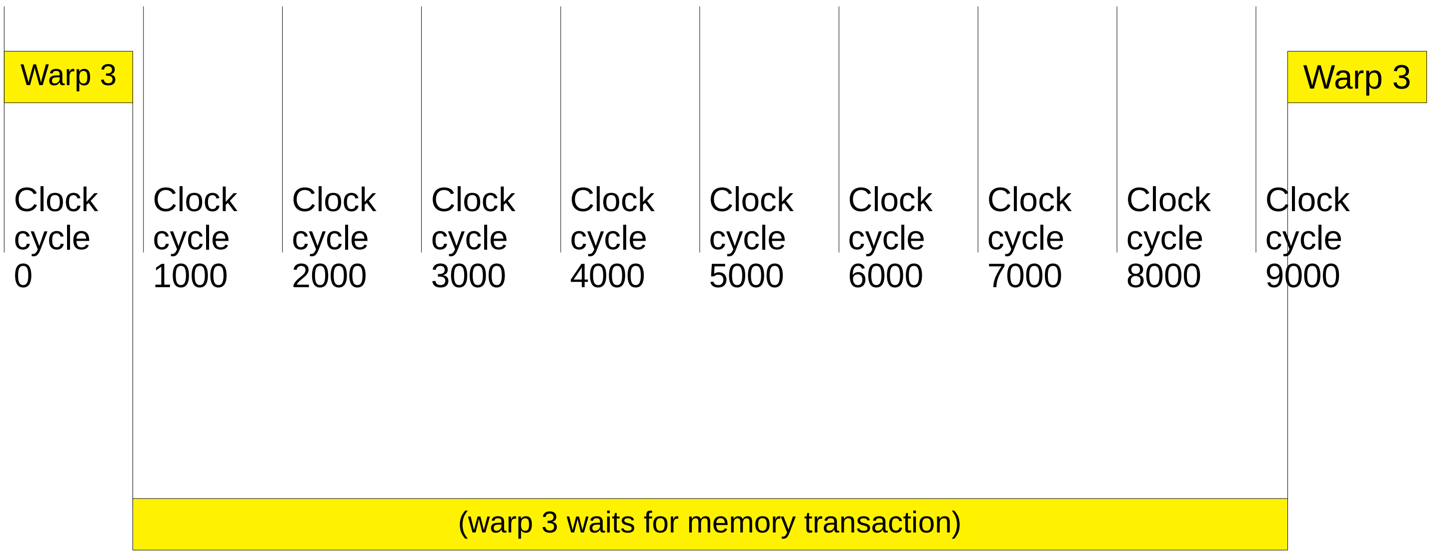
And they can stall a long time...

With 1000’s of cores on a die
constantly requesting memory,
memory transactions have
Extremely high latency to
ensure high bandwidth.

The memory system on the GPU has been designed to have a massive amount of bandwidth, rather than minimising latency (regular RAM tends to foxus on the latter).

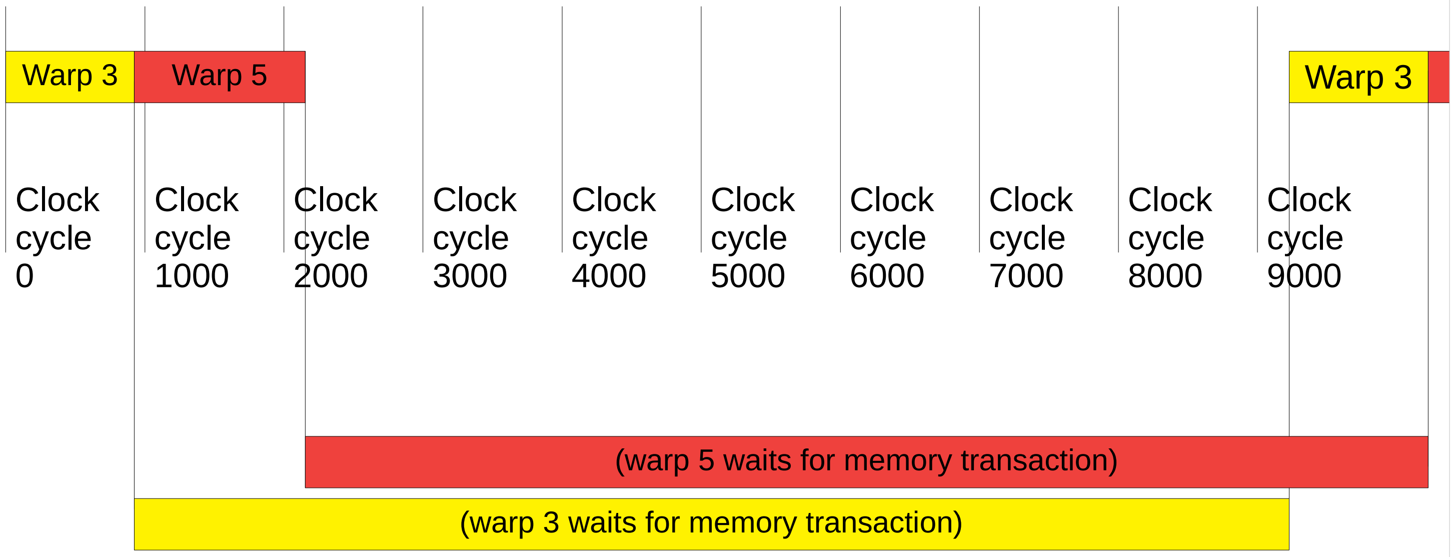
This causes individual requests to have a much longer latency.

Problem: Warps stall. A lot.
If we were to execute threads like on a CPU:



If we were to only execute a single thread at a time, like we would on a CPU, the frequent stalls would cause the processor to idle for extended periods of time, and therefore make poor use of the available hardware.

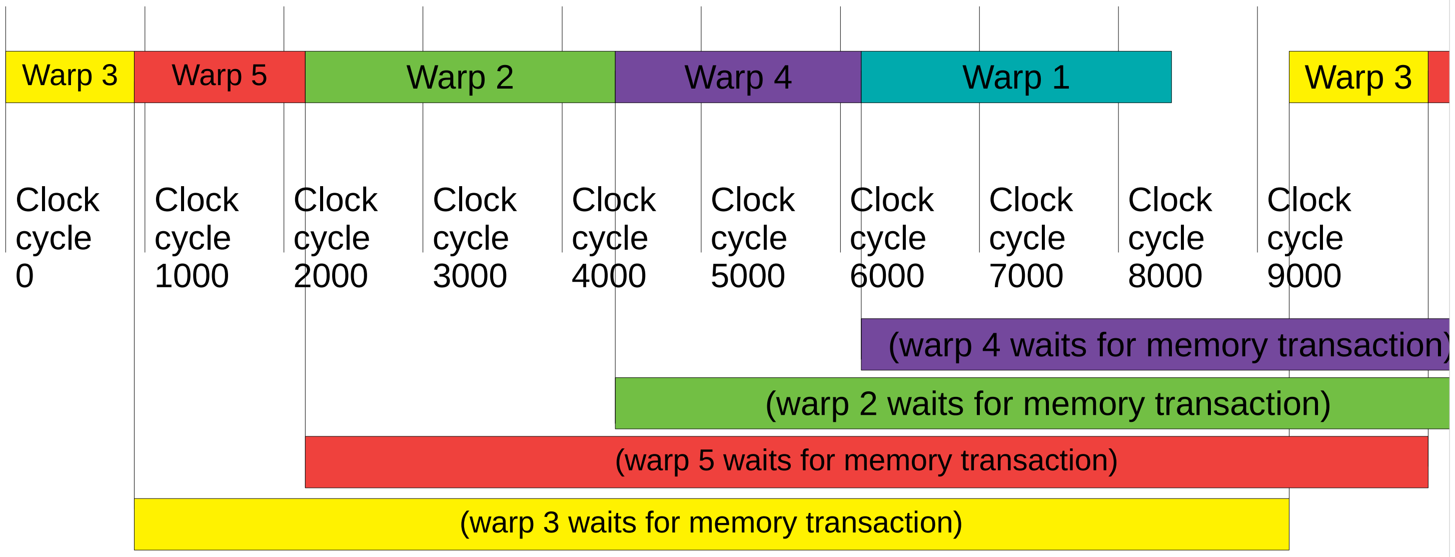
Solution: Quickly switch to other warps



So the main idea is to, instead, use the time in which one warp is stalling to execute other ones.

.. Small note about the listed “clock cycle” values here: I just needed a time value to show longer periods of time. They are utter bollocks when it comes to the GPU because it tends to switch warps every. single. clock cycle.

Solution: Quickly switch to other warps

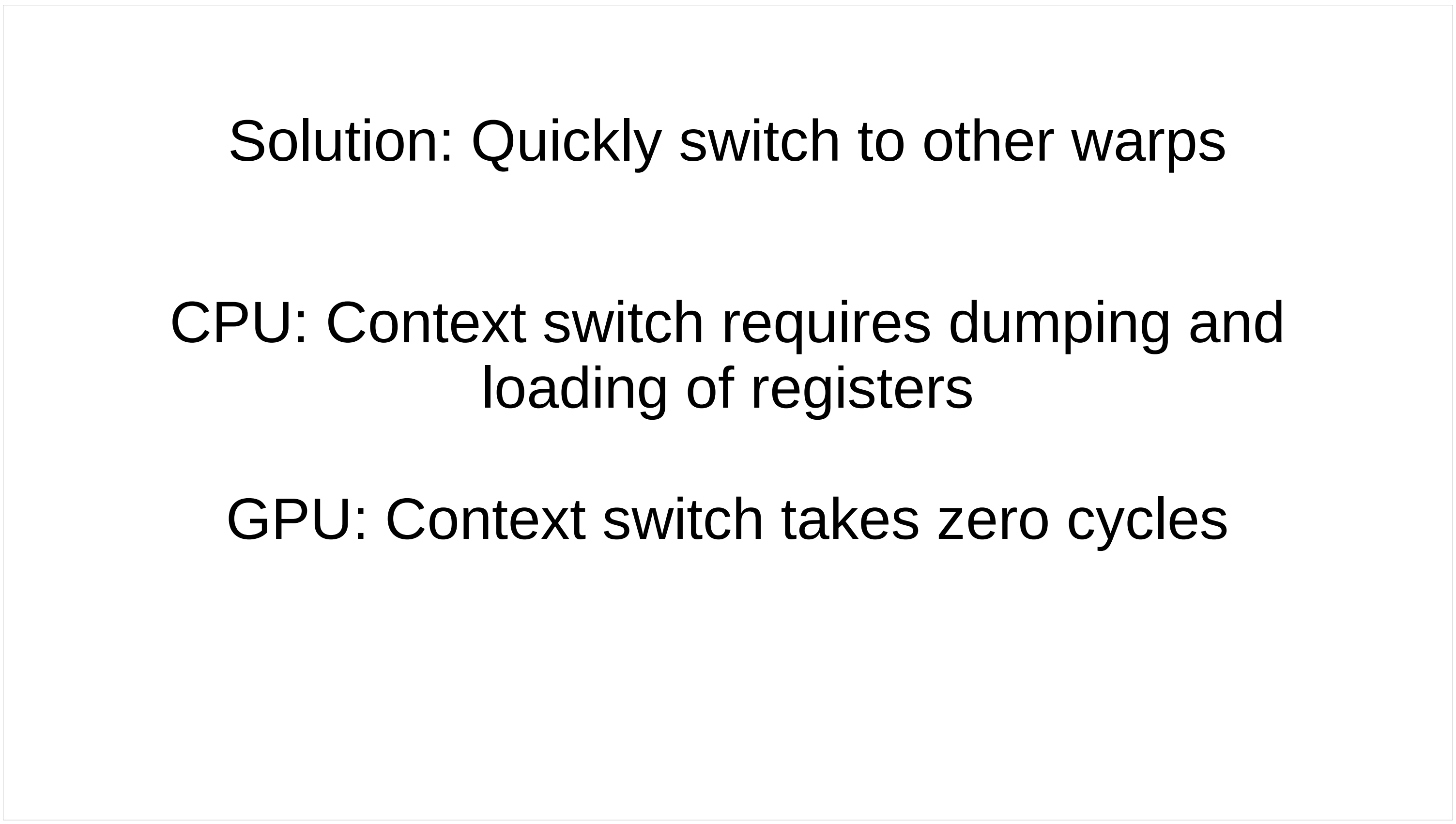


And the more warps we can switch to, the more of the time in between we can fill up with meaningful execution.

Let’s for a moment assume we managed to fill our entire timeline with warps. We can then reorder the timeline, such that each time a warp was executing is grouped together.

Here’s the thing; since the processor was active all the time, it in a way has executed all warps start to finish WITHOUT HAVING TO WAIT FOR THEIR MEMORY REQUESTS.

This is called “latency hiding”, and is the primary reason why the register file is a thing, and why it is desirable to run many blocks on an SM at the same time.



Solution: Quickly switch to other warps

CPU: Context switch requires dumping and loading of registers

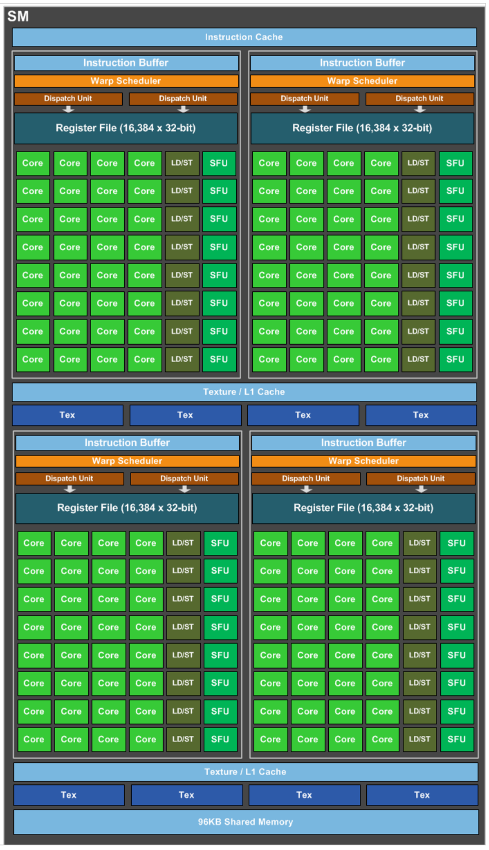
GPU: Context switch takes zero cycles

The fact that the register values of all threads active on an SM are available at any given time also allows for something else: you can switch between them *instantly*.

Now the CPU scrubs pay good money for this “simultaneous multithreading” feature, or “hyperthreading” as intel likes to call it. This allows you to execute two threads simultaneously, more or less in the same way as what I showed on the previous slide.

Simultaneous Multithreading on Steroids!

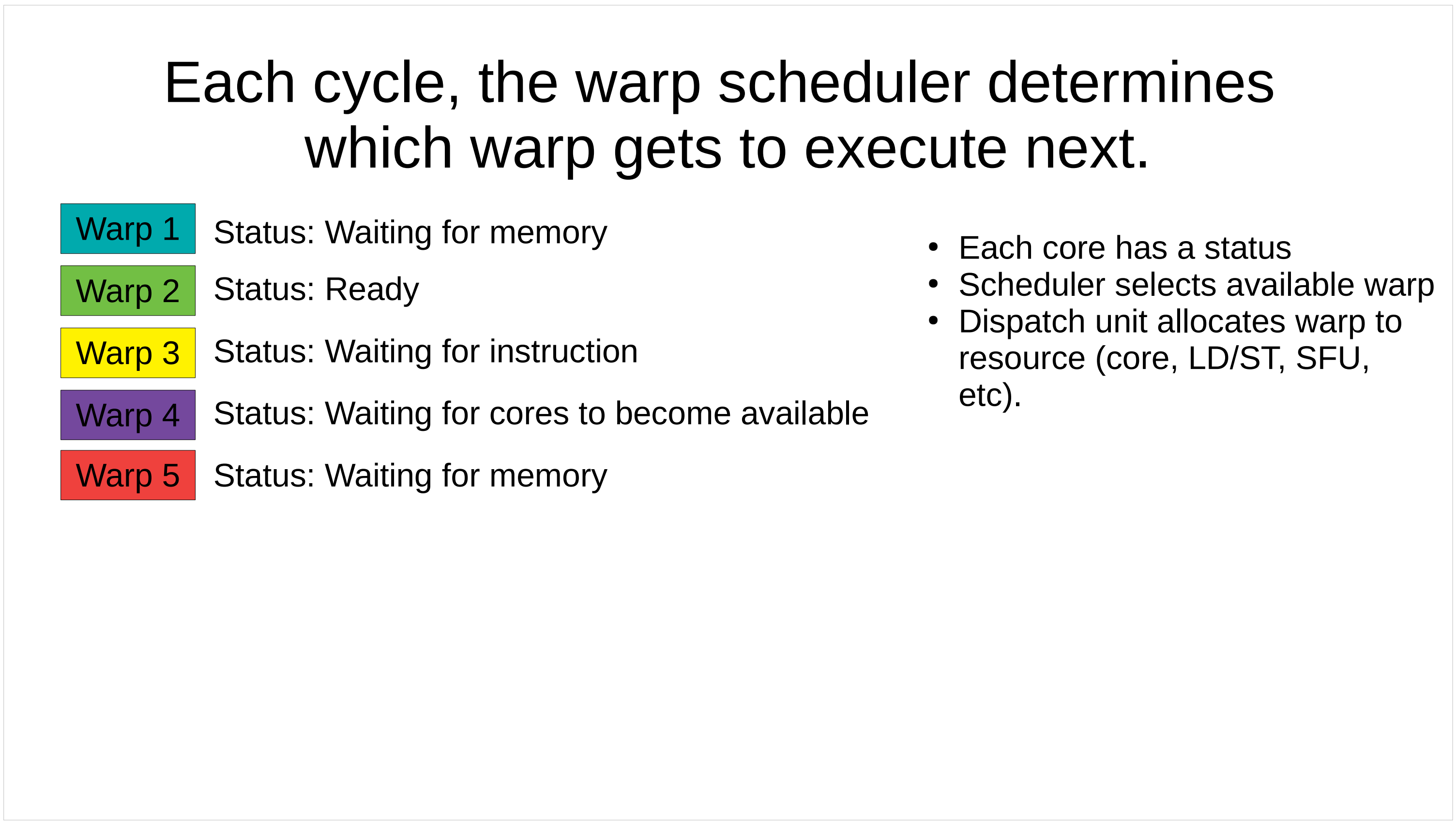
	Dispatch Unit	Dispatch Unit
Clock cycle 0	Warp 3	Warp 1
Clock cycle 1	Warp 5	Warp 3
Clock cycle 2	Warp 2	Warp 3
Clock cycle 3	Warp 5	Warp 4
Clock cycle 4	Warp 3	Warp 1
Clock cycle 5	Warp 5	Warp 4
Clock cycle 6	Warp 4	Warp 2
Clock cycle 7	Warp 2	Warp 1



Modern GPU’s can switch between 64 of them in the same way. It really is SMT on steroids.

Each of the dispatch units in the SM allocates a warp that was selected by the warp scheduler, and assigns it to a particular compute resource (cores, load/store units, texture units, etc).

It does so every single clock cycle.



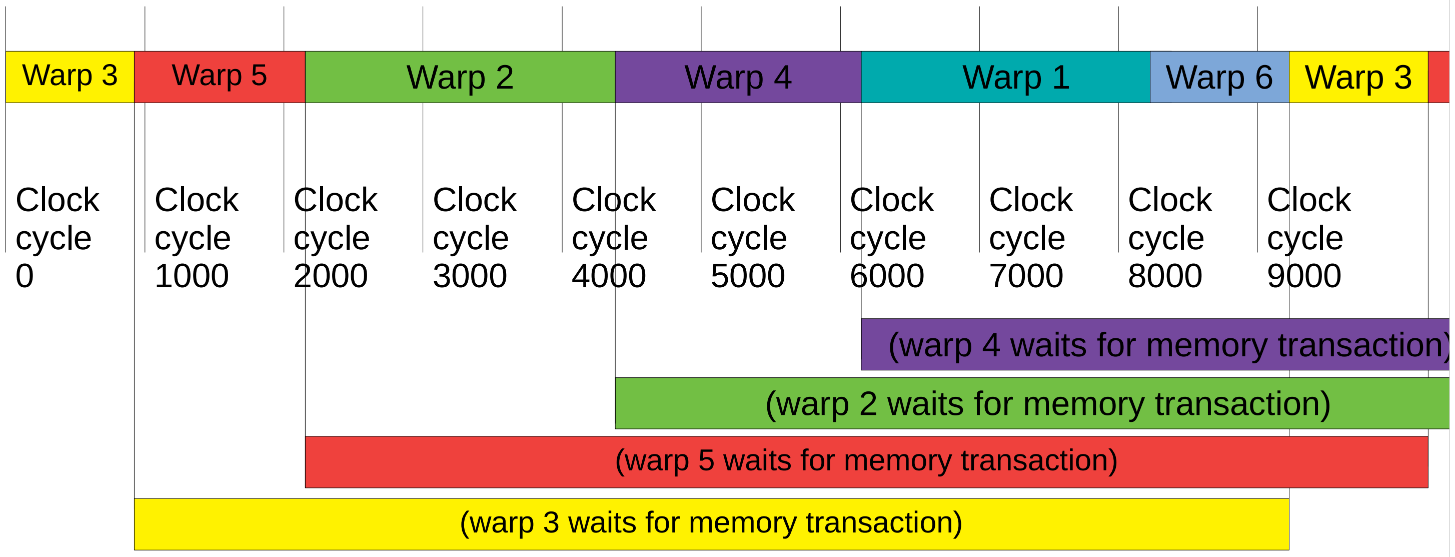
Each cycle, the warp scheduler determines which warp gets to execute next.

Warp 1	Status: Waiting for memory
Warp 2	Status: Ready
Warp 3	Status: Waiting for instruction
Warp 4	Status: Waiting for cores to become available
Warp 5	Status: Waiting for memory

- Each core has a status
- Scheduler selects available warp
- Dispatch unit allocates warp to resource (core, LD/ST, SFU, etc).

Warps can be in various states, but the warp scheduler can only schedule ones that are marked as active (they have their instruction, memory operands, and can be allocated to a specific compute resource in the SM).

Result: Since an SM is ideally always executing code, *memory latency is hidden!*



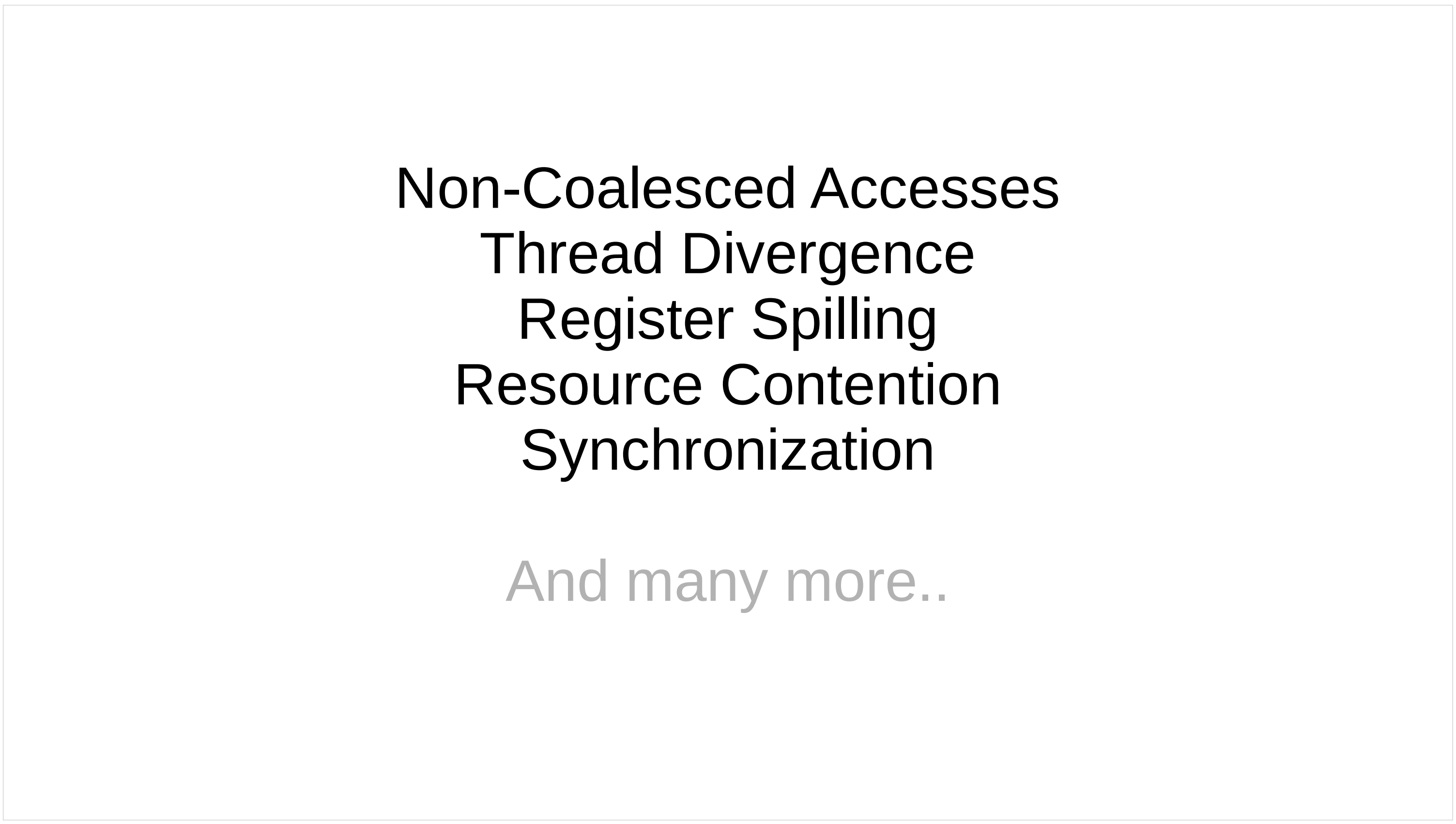
I'm using the illustration for the "CPU approach" here, but the principle is the same. The SM has a bunch of warps that execute instructions, and each cycle, the SM can pick a different warp to use compute resources (cores, LD/ST, SFU, etc) if it so desires. Since ideally each cycle is being used on something useful, each thread in a way effectively is executed without memory latency (if you were to rearrange the timeline to group all times a particular warp was executing together)

But for this to work, you NEED to have a lot of threads "in flight" to keep the SM busy.

Back to our original objective:

What can make warps run slow?

Anyway. Where was I.

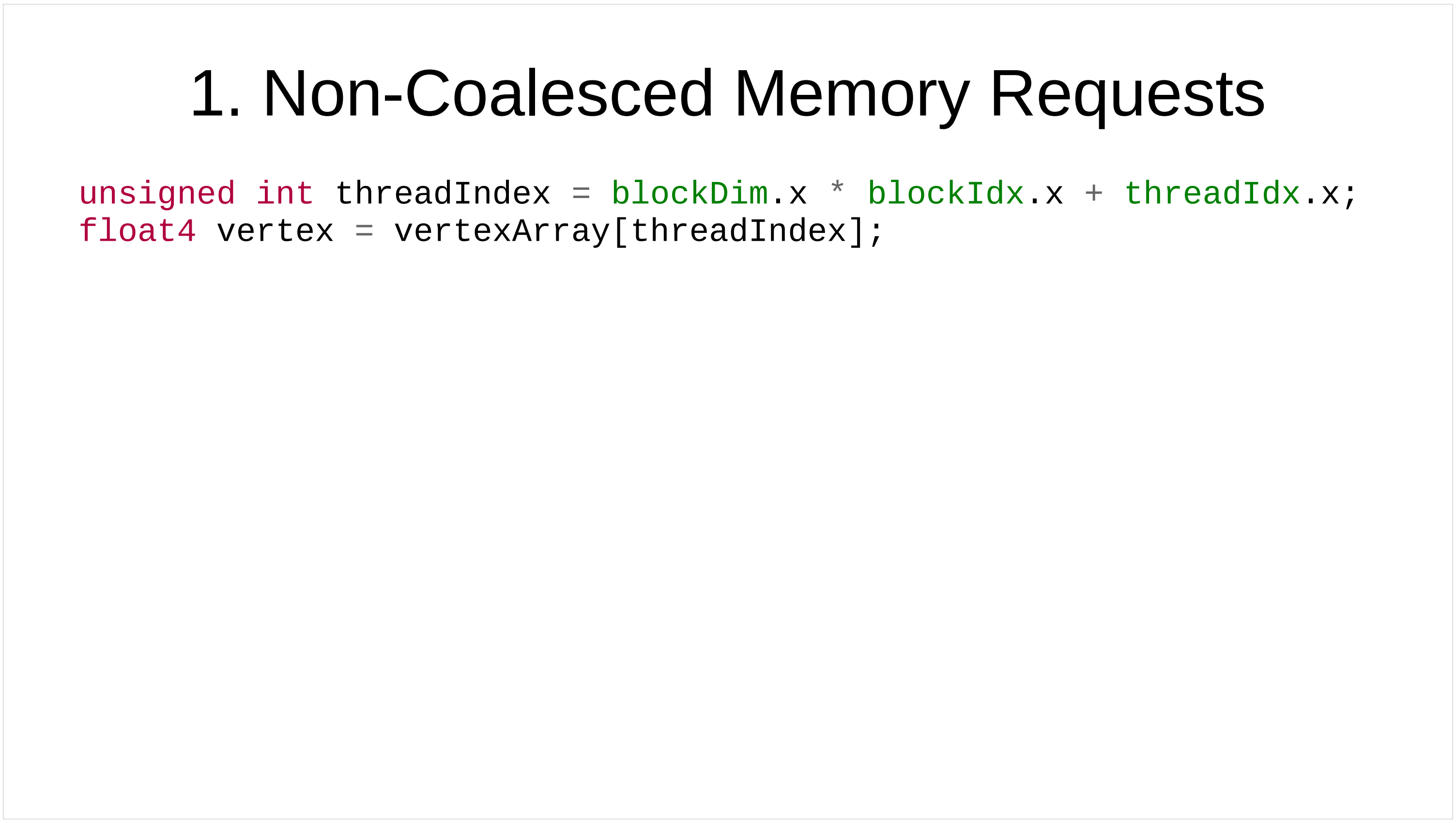


Non-Coalesced Accesses
Thread Divergence
Register Spilling
Resource Contention
Synchronization

And many more..

Ah yeah.. Things that can make your code run slow.

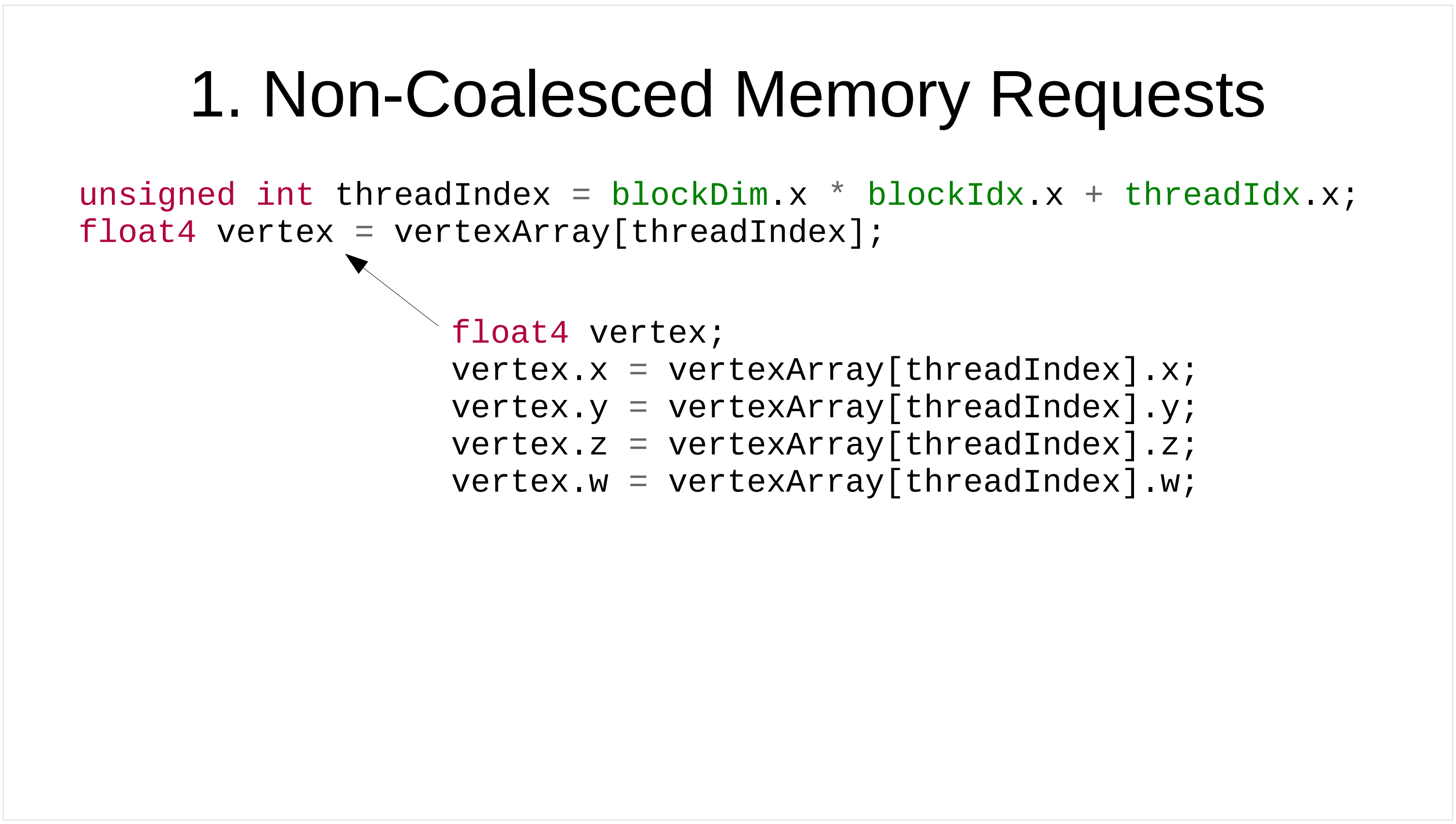
Here’s a list of what I think the most common suspects are.



1. Non-Coalesced Memory Requests

```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```

Consider a typical scenario: reading out a 3 or 4-dimensional coordinate from an array. You’ve already done this in multiple of the assignments.

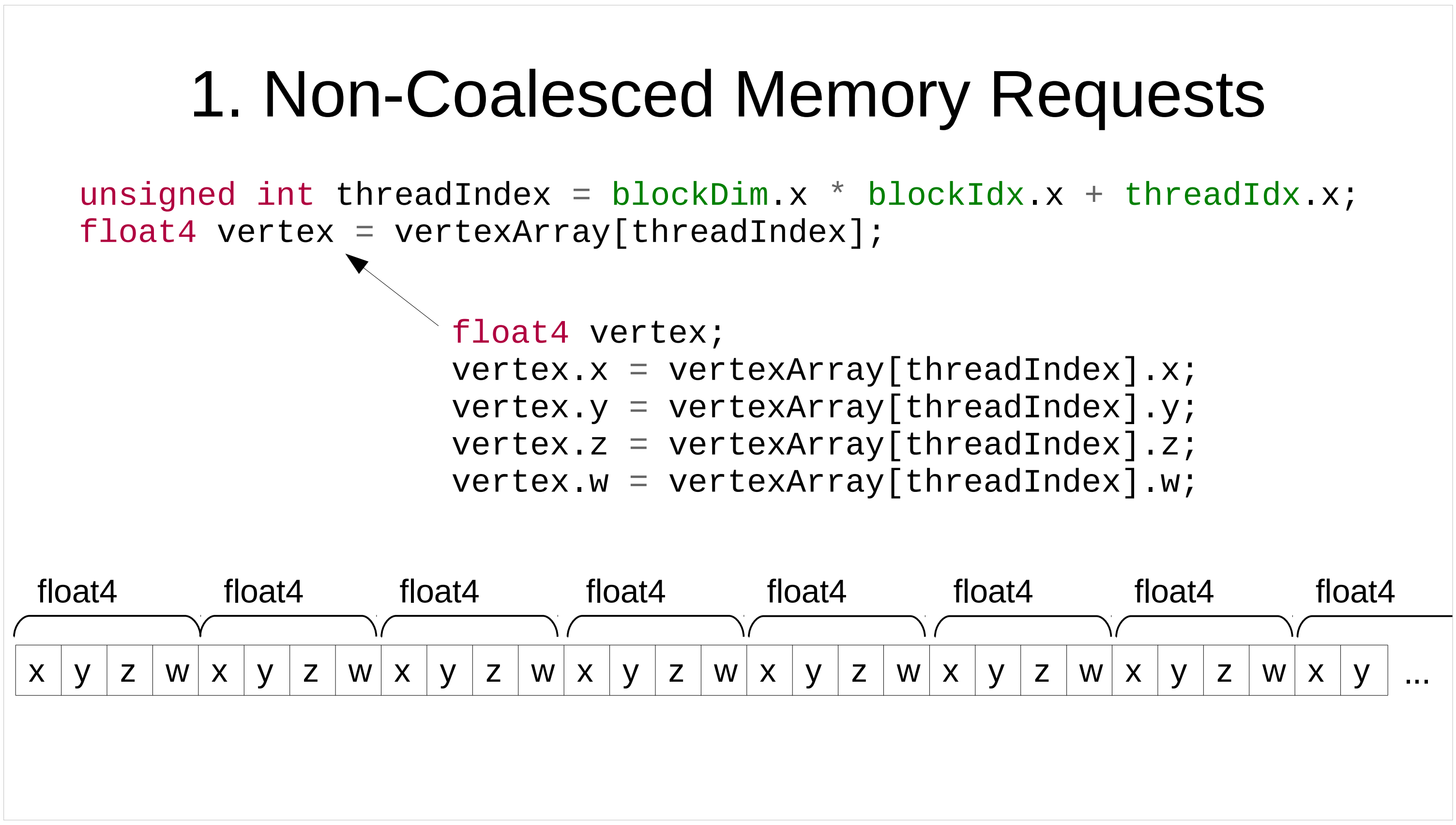


1. Non-Coalesced Memory Requests

```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

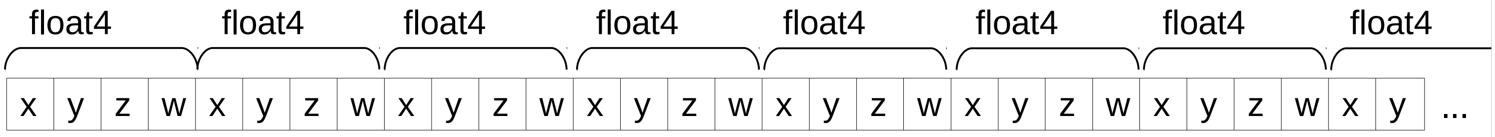
Now when you read a structure into memory, a separate memory request is needed for each of its fields. That means loading a structure such as float4 really consists of four separate memory transactions.



1. Non-Coalesced Memory Requests

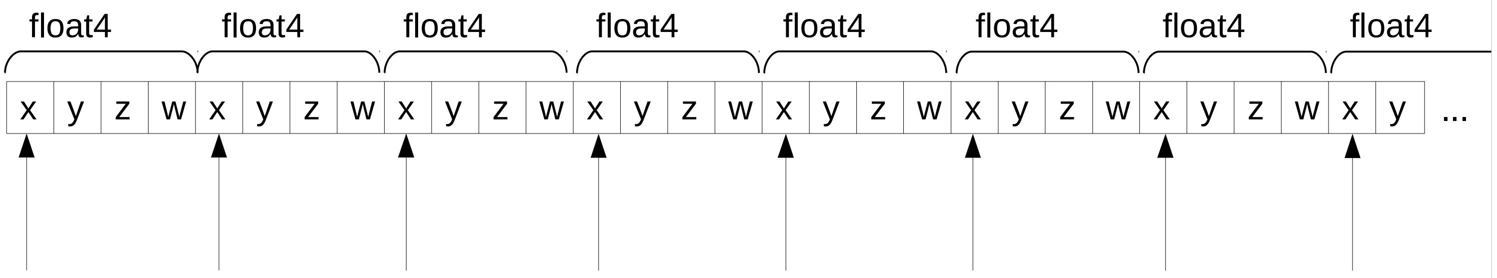
```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```


float4
float4
float4
float4
float4
float4
float4
float4
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y ...

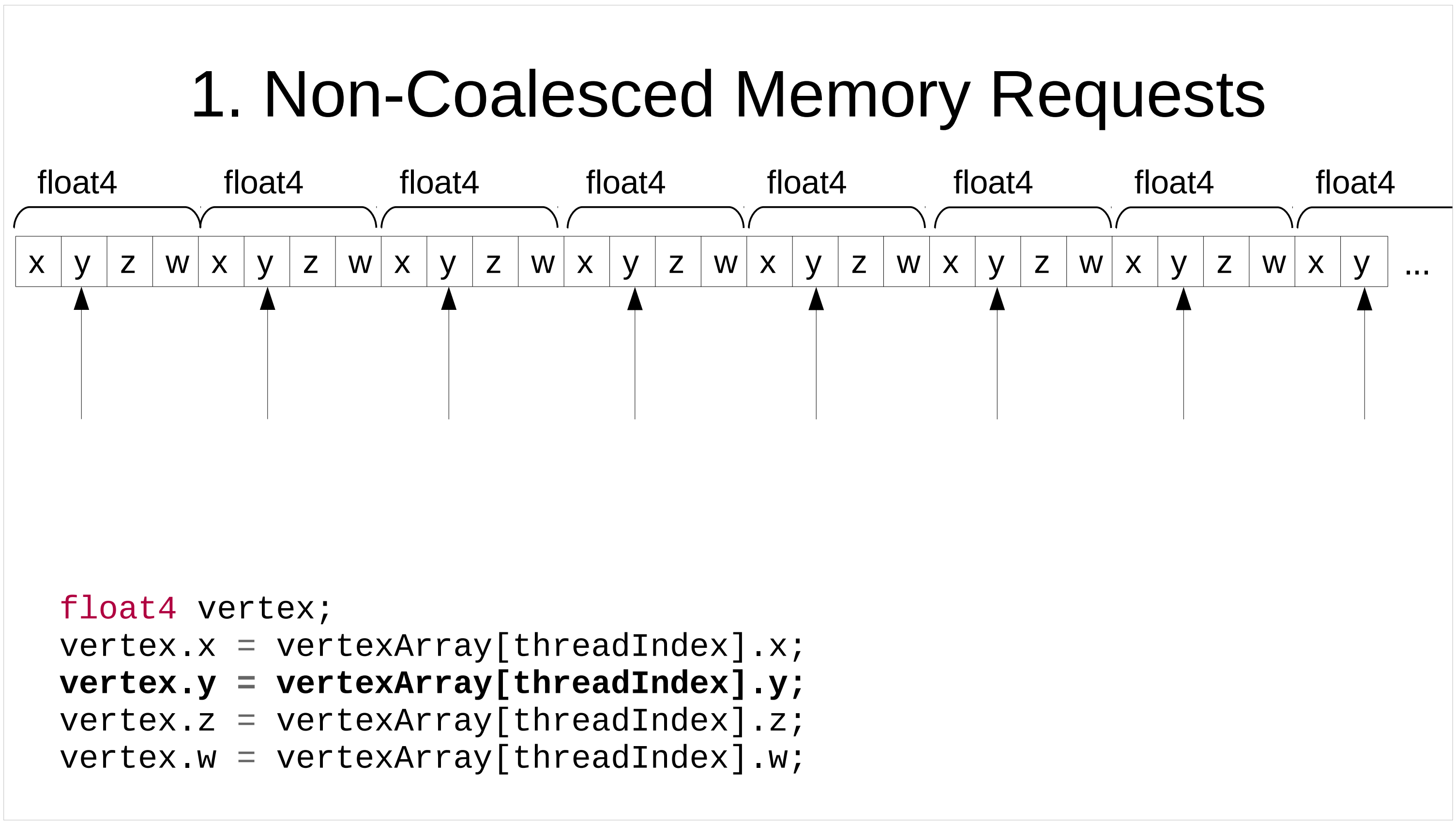
But that is not really the main issue of “non-coalesced” requests. That has to do with how these requests are laid out in memory. When you create an array of structs, their contents are laid out consecutively.

1. Non-Coalesced Memory Requests

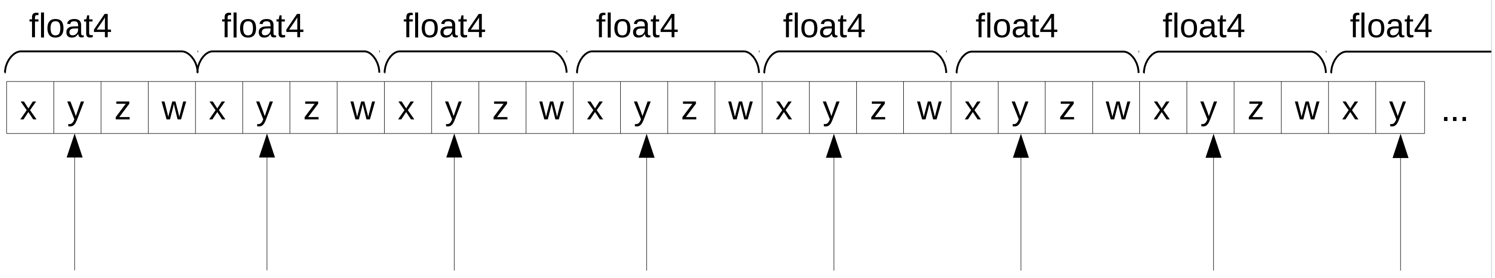

float4
float4
float4
float4
float4
float4
float4
float4
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y ...

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

And when you request an individual field from these structs, there are gaps in between the requests of individual threads.



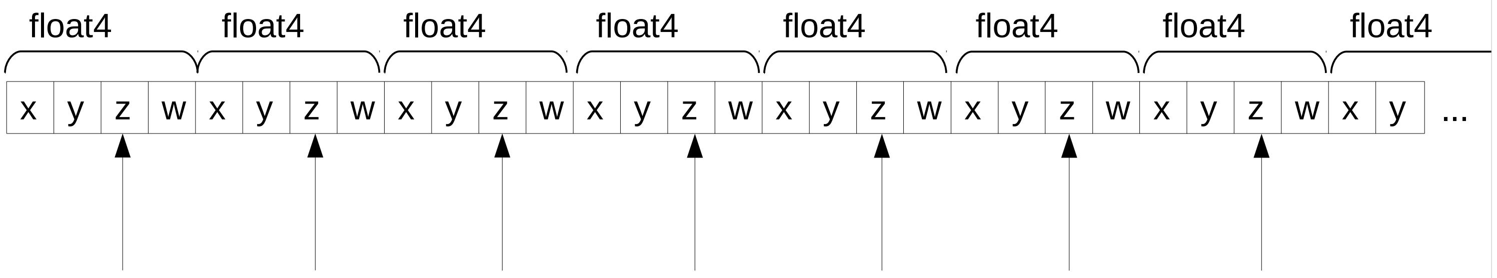
1. Non-Coalesced Memory Requests


float4
float4
float4
float4
float4
float4
float4
float4
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y ...

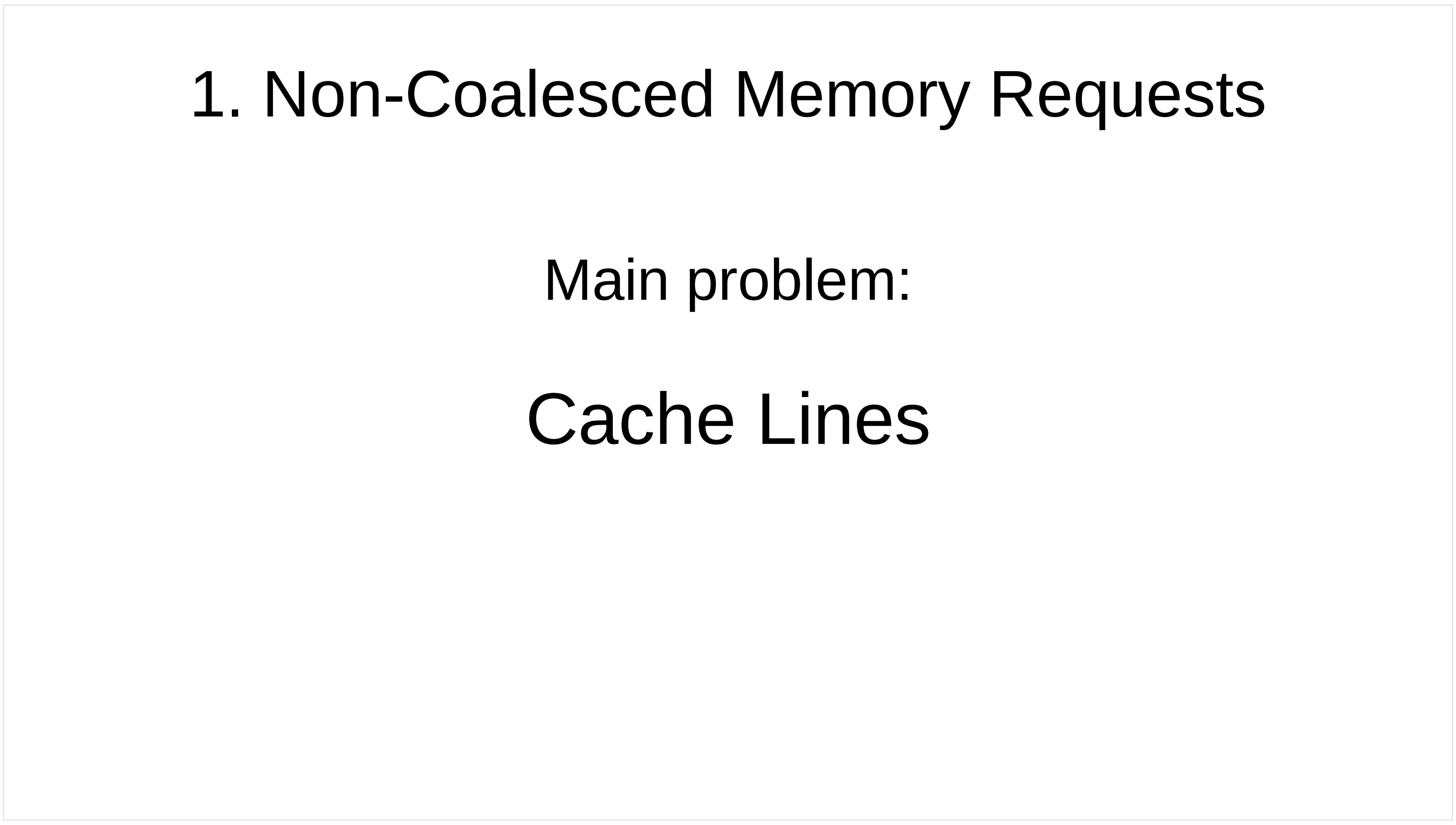
```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

And these gaps are also present in the subsequent requests we need to do to load in the other fields of the struct.

1. Non-Coalesced Memory Requests


float4
float4
float4
float4
float4
float4
float4
float4
x y z w x y z w x y z w x y z w x y z w x y z w x y z w x y ...

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```



1. Non-Coalesced Memory Requests

Main problem:

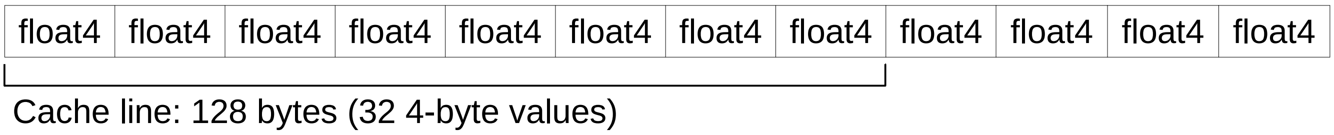
Cache Lines

But here’s the issue: all memory transactions happen in terms of cache lines: chunks of sequential memory.

1. Non-Coalesced Memory Requests

float4	float4	float4	float4	float4	float4	float4	float4	float4	float4	float4	float4

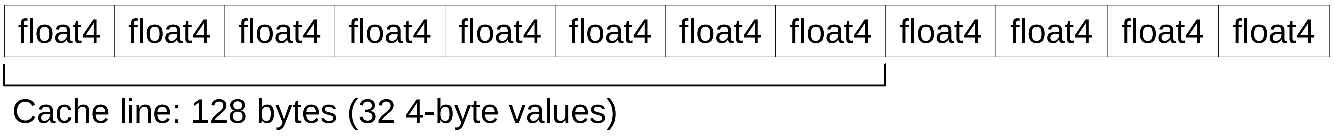
1. Non-Coalesced Memory Requests



A cache line on the GPU is large enough such that each thread in a warp can access a single 4-byte value (a single precision float or unsigned integer).

The memory system can only retrieve an entire cache lines from memory. That means that as long as even a single thread requests a single byte from a particular cache line, the entire 128 bytes need to be fetched from memory and brought into the core.

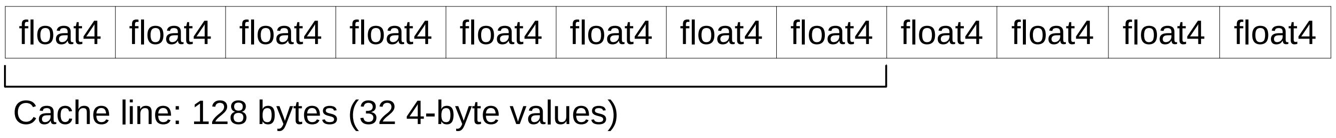
1. Non-Coalesced Memory Requests



Here’s the real kicker:

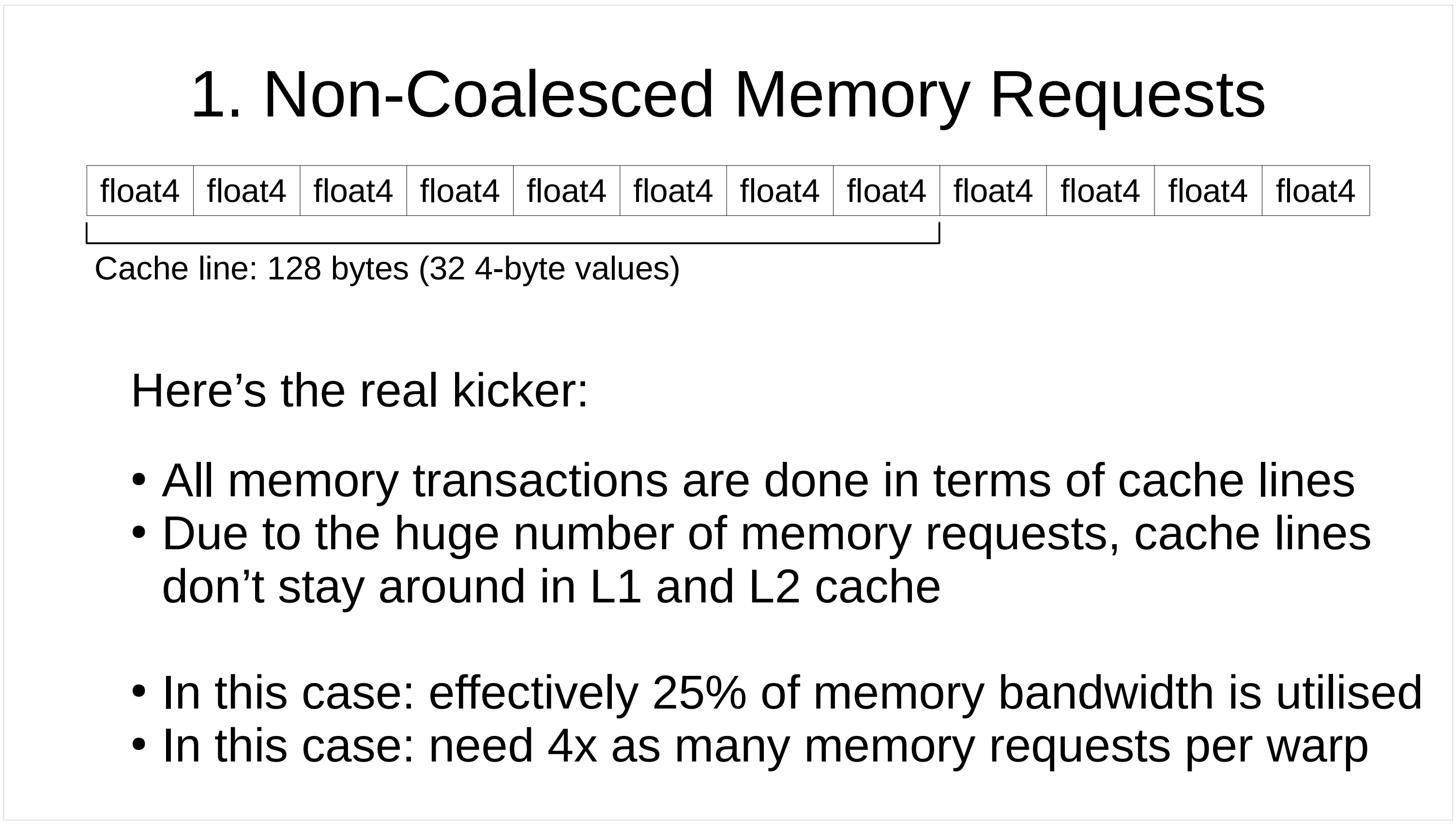
- All memory transactions are done in terms of cache lines

1. Non-Coalesced Memory Requests



Here’s the real kicker:

- All memory transactions are done in terms of cache lines
- Due to the huge number of memory requests, cache lines don’t stay around in L1 and L2 cache



1. Non-Coalesced Memory Requests

float4	float4	float4	float4	float4	float4	float4	float4	float4	float4	float4	float4

Cache line: 128 bytes (32 4-byte values)

Here's the real kicker:

- All memory transactions are done in terms of cache lines
- Due to the huge number of memory requests, cache lines don't stay around in L1 and L2 cache

- In this case: effectively 25% of memory bandwidth is utilised
- In this case: need 4x as many memory requests per warp

But the problem in that is that since we only request one field at a time from the struct, and its fields are laid out sequentially, we need four times as many memory requests to load in our struct.

And worse, every time we do so, we only use 1/4th of the values inside of this cache line.

On the CPU this would not be a problem, because whatever cache line you have fetched from memory would stay around in L1 cache, which the next requests would hit.

On the GPU, due to the incredibly large quantities of memory being processed, values don't really stay around that long for that to work, so generally each memory requests has to go through main memory regardless.

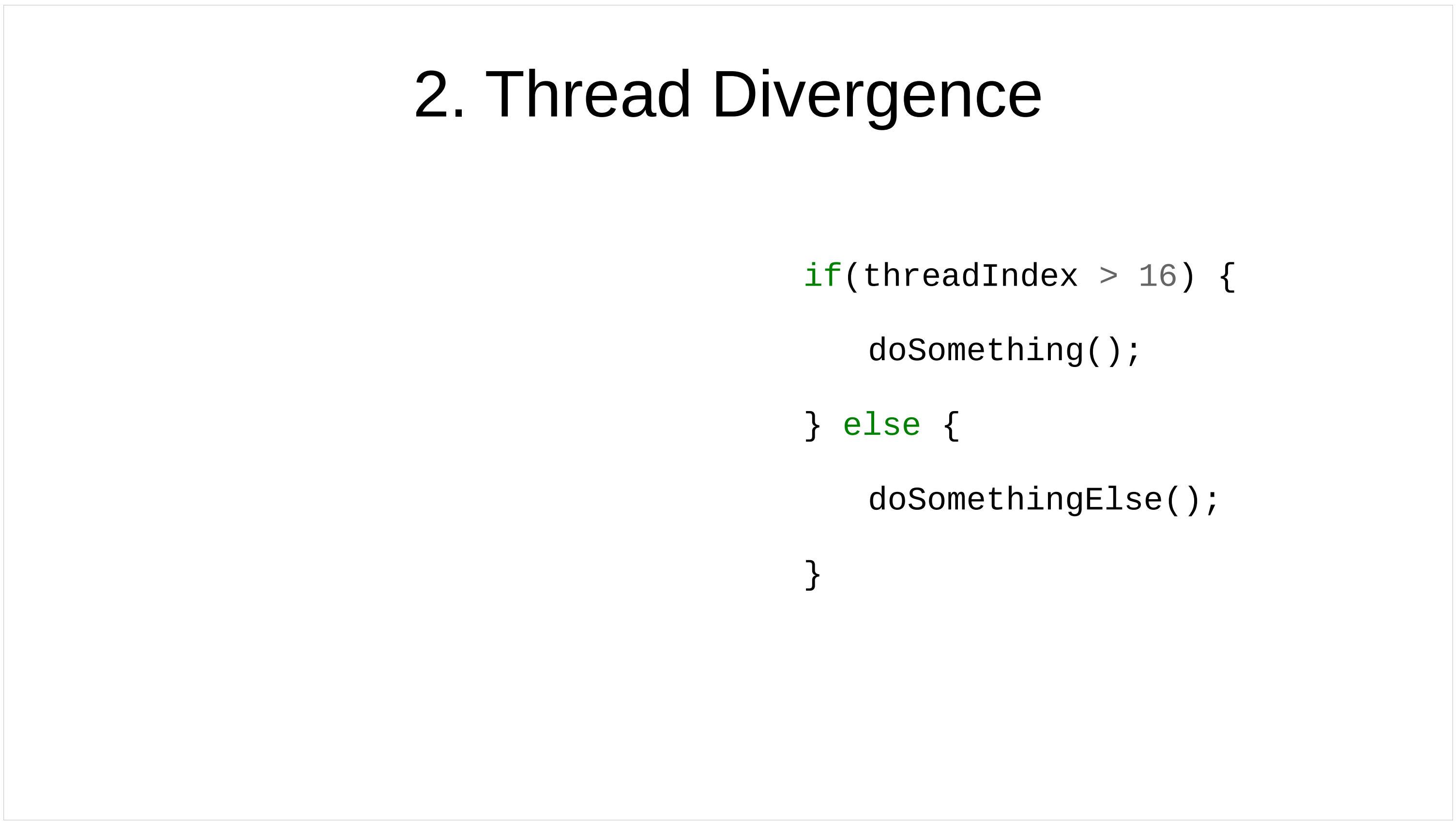
1. Non-Coalesced Memory Requests

Mitigation: always ensure memory requests are within the same cache line

Memory requests are really expensive.
Can easily yield speedups of 1.3.

The way to solve this problem is to rearrange your data (for example using a struct of arrays rather than an array of structs in this scenario), such that each thread in a warp always requests addresses *from the same cache line*.

Note that these do not necessarily need to be sequential, but at least each thread should generally try to write to addresses in the same cache line.

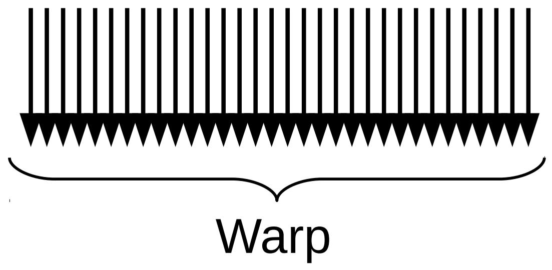


2. Thread Divergence

```
if(threadIndex > 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

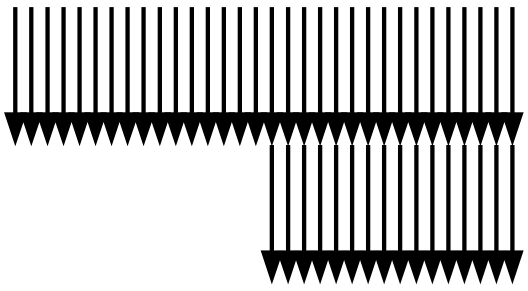
The second problem is thread divergence, which can occur anytime a branch instruction is issued (note that loops also use branches)

2. Thread Divergence



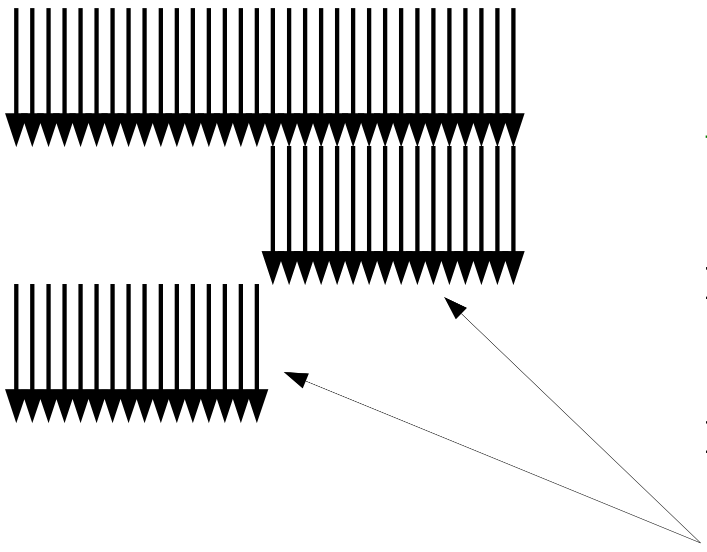
```
if(threadIndex >= 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

2. Thread Divergence



```
if(threadIndex >= 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

2. Thread Divergence



```
if(threadIndex >= 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

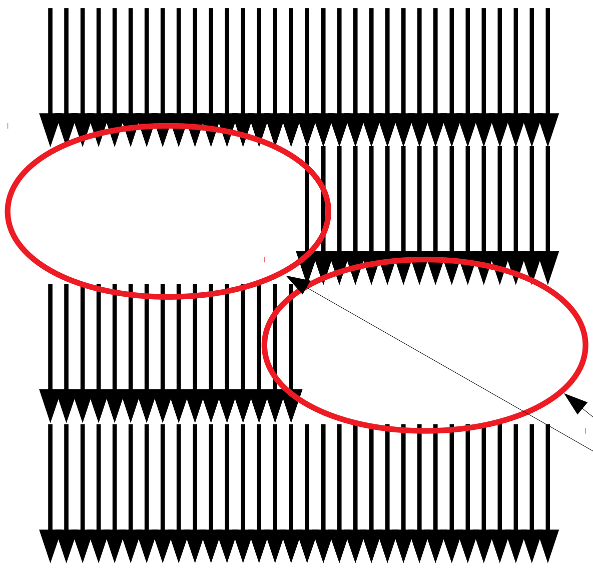
BOTH clauses need to be executed in full

Threads on the GPU can only execute the exact same instruction at any given time. If it encounters a branch, and even if only a single thread has to execute one path of a branch, ALL of those instructions for that path need to be executed.

While the threads which have taken that branch path work their way through that clause, the remaining threads are just idling.

If your code is very heavy on branches, this can mean you effectively only use a very small part of the computing power available to you.

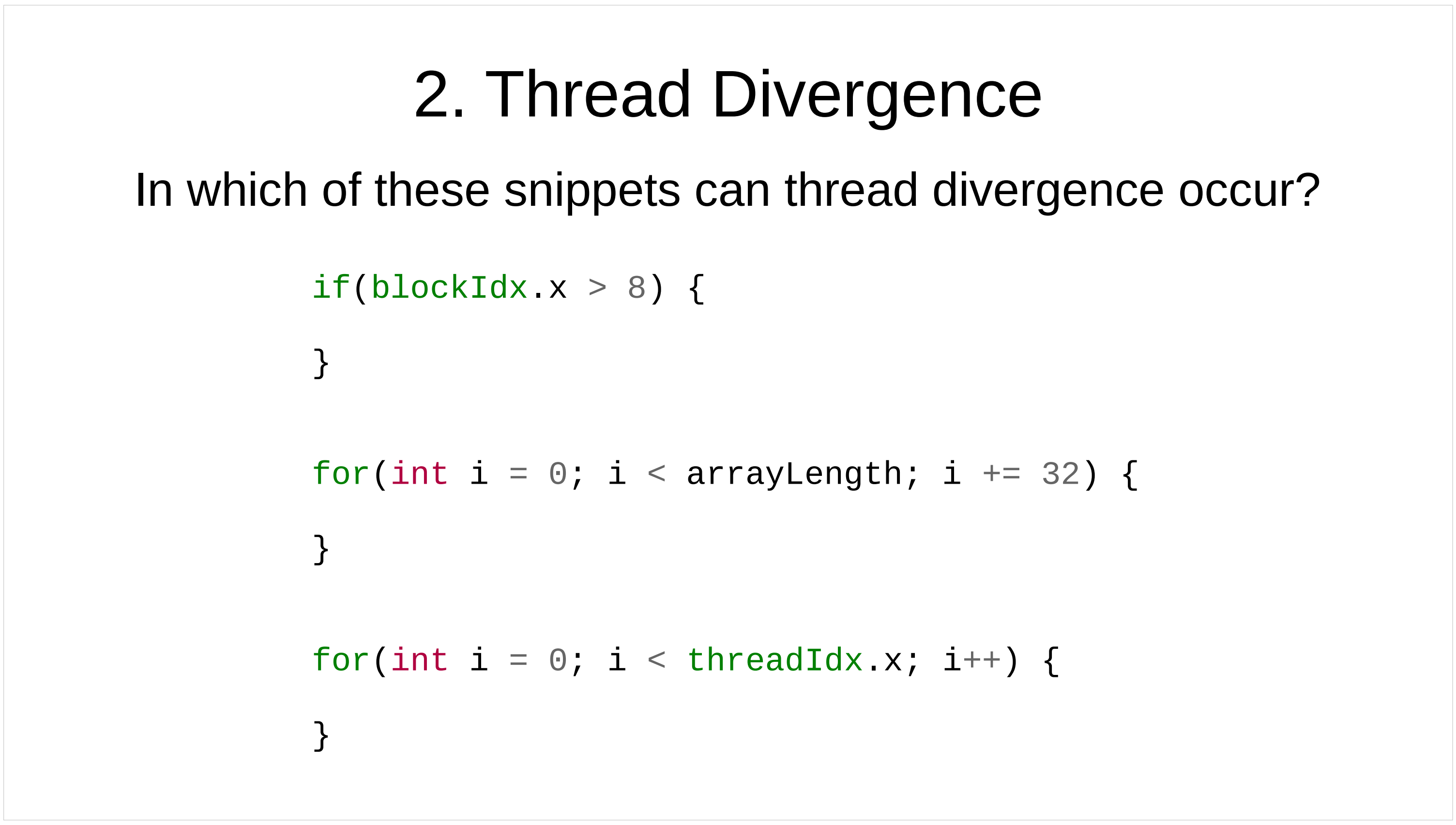
2. Thread Divergence



```
if(threadIndex >= 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

While half of the threads are doing nothing!

One thing you can do is to replace branches as much as possible with non-branching variants. For instance, if the then and else clause of an if statement are mostly the same, you can “cancel out” parts of the computation using arithmetic operations instead. If that avoids having to execute similar instructions twice, that can be a lot faster.



2. Thread Divergence

In which of these snippets can thread divergence occur?

```
if(blockIdx.x > 8) {

}

for(int i = 0; i < arrayLength; i += 32) {

}

for(int i = 0; i < threadIdx.x; i++) {

}
```

The first snippet has no divergence, because all threads in a warp have the same block index.

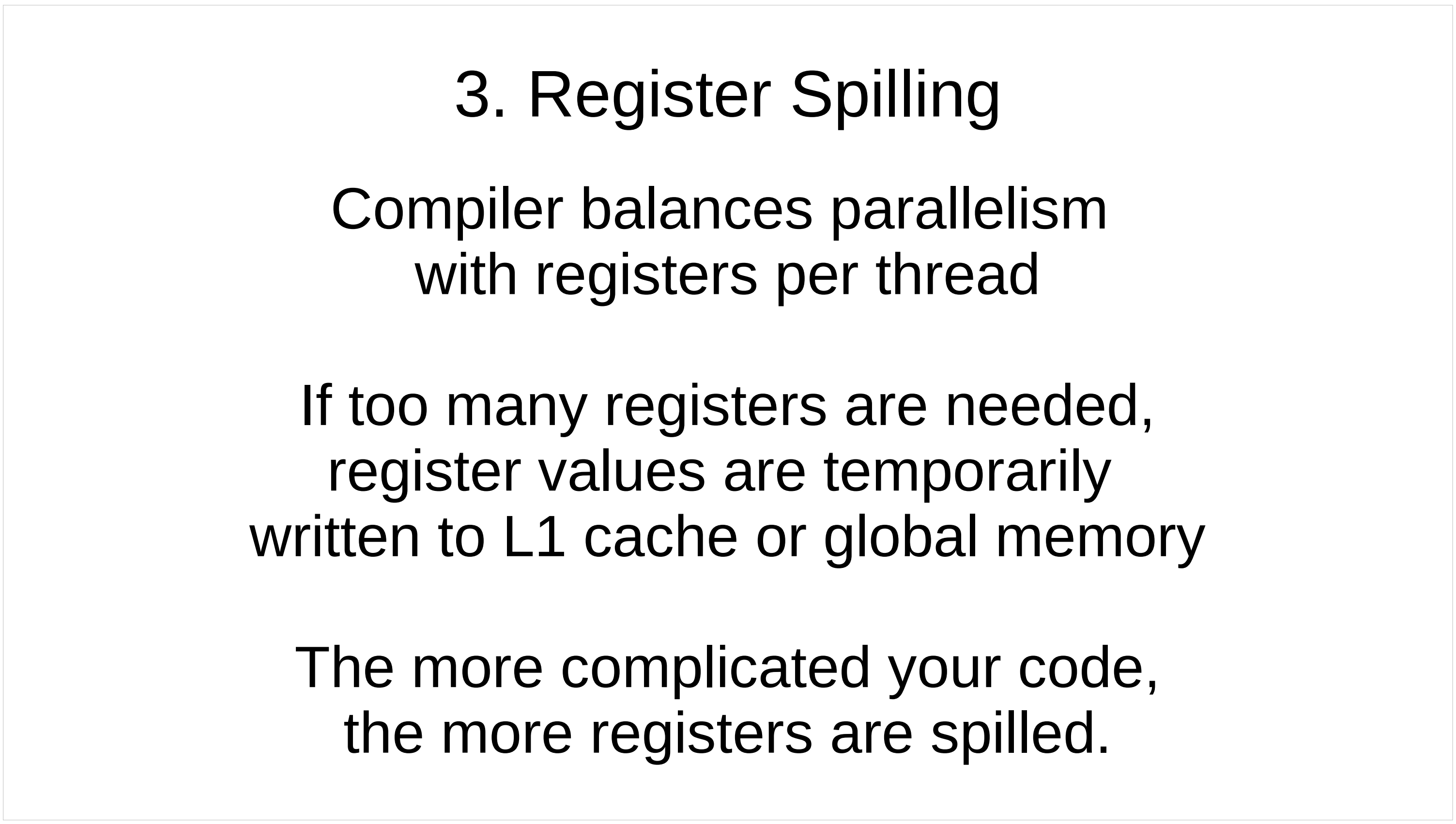
The second one does not diverge, as each thread will iterate over the loop the same number of times.

The third snippet diverges, because each thread will execute the loop a different number of times.

2. Thread Divergence

Note: Min(), Max(), and Abs() have been implemented in hardware due to their regular occurrence.

Another note on removing branches: use the built-in min, max, and abs wherever applicable, because they have hardware implementations due to their frequent occurrence.



3. Register Spilling

Compiler balances parallelism
with registers per thread

If too many registers are needed,
register values are temporarily
written to L1 cache or global memory

The more complicated your code,
the more registers are spilled.

The problem of register spillling happens when you implement an algorithm that is overly complex, and the CUDA compiler decides that using fewer registers (thus allowing more blocks to execute) is preferable over allowing your warps to execute at full speed.

Register spilling will cause the contents of registers to be written to and from L1 cache, to free up registers for other tasks.

It’s generally quite complex to reduce the register count, but in general, the more values you need to hold on to for a longer period of time (such as an iterator variable in a for loop), the more registers your threads will need.

4. Resource Contention

Due to the large number of threads, atomics can cause significant overhead.

```
int nextItemToProcess = atomicAdd(itemsProcessed, 1);
```

32 threads incrementing the same value causes 32 atomic operations in serial.

Resource contention can happen when using atomics on the same place in memory.

Because there are so extremely many threads on a GPU active at any given time, threads can potentially spend a lot of their time waiting.

Even when performing a single atomic operation with a warp of threads means 32 atomic instructions need to be executed, which will be *executed in serial*.

5. Synchronisation

Warps in a block can be synchronised (basically a barrier):

```
__global__ void doSomething(int* array) {
    array[threadIdx.x] = 0;
    // synchronise all threads in this block
    __syncthreads();
    array[threadIdx.x] += array[threadIdx.x + 1];
}
```

Excessive synchronisation limits parallelism

Warps within a block are basically executing like threads on a CPU: they can be scheduled in any order. If your warps need to do an operation in a particular order, you can use __syncthreads() as a barrier warps will wait on before continuing.

Doing this excessively can hurt performance.

Do note however. these barriers only sync threads WITHIN a block. Threads between blocks can execute at their own whim. Only from Pascal + CUDA 9 and up a hardware way of synchronising blocks was supported, although that has a number of caveats.

.. And others

Double precision instructions take 32x times longer than single precision ones.

Not enough blocks to have all SM’s execute code.

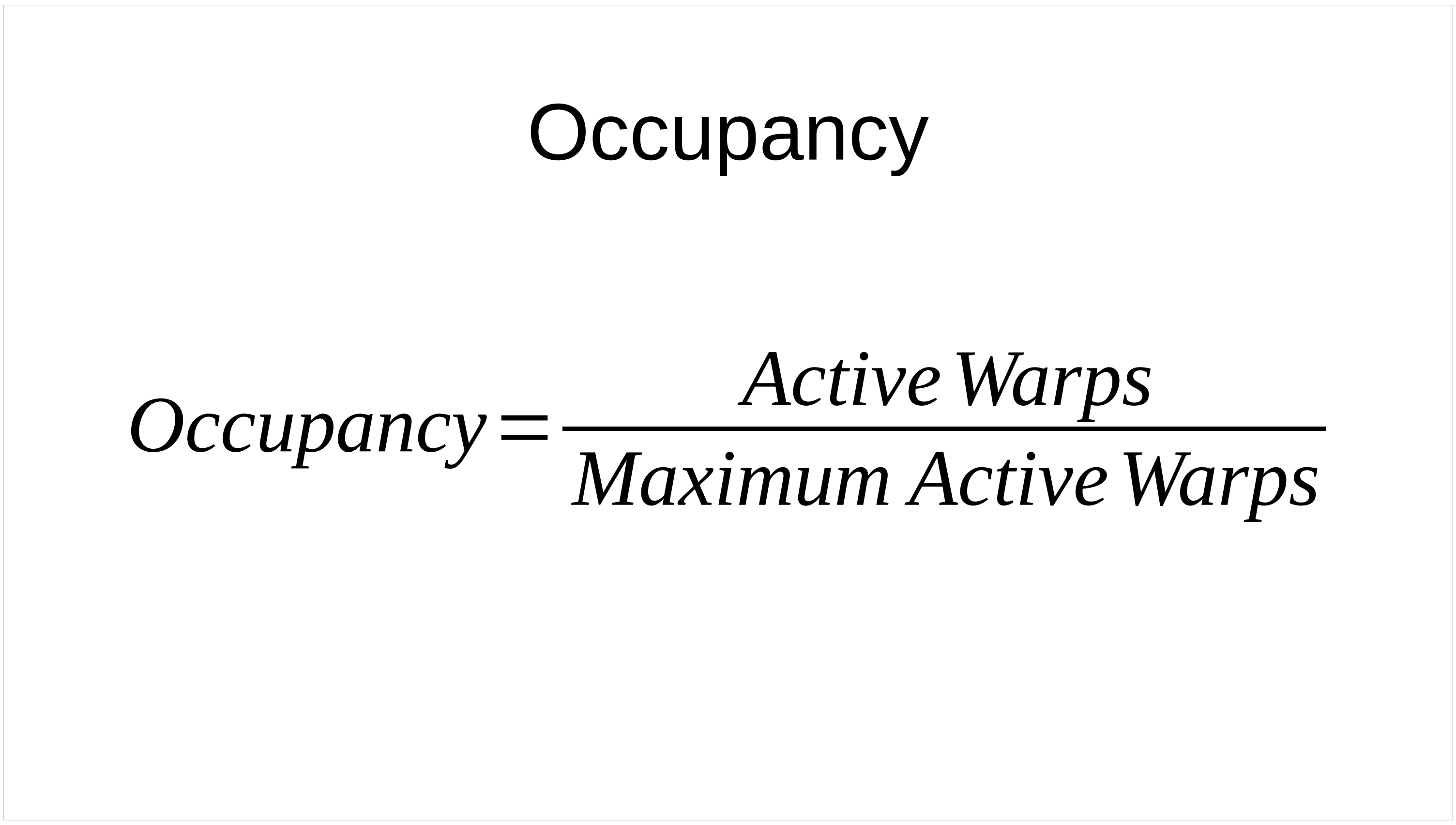
GPUs really are floating point coprocessors at their core (no pun intended). If you need double precision, you either need to empty your wallet (an nvidia V100 goes for over 100 000 norwegian kroner at the time of writing)

Or make some tough decisions and/or rewrites to fit your data into a single precision value.

Also, make sure you launch enough threads. You need a lot of blocks to saturate many GPUs, and only that will allow you to utilise them in full.

Part 2/3:

How do we measure GPU performance?



Occupancy

$$Occupancy = \frac{Active\ Warps}{Maximum\ Active\ Warps}$$

Occupancy is the number of cycles the SM’s on the GPU are executing a warp, as a fraction of the total number of cycles.

Occupancy

$$Occupancy = \frac{Active\,Warps}{Maximum\,Active\,Warps}$$

In other words: the percentage of cycles an SM can schedule a warp

Occupancy

Depends on launch parameters and your code

$$Occupancy = \frac{Active\ Warps}{Maximum\ Active\ Warps}$$

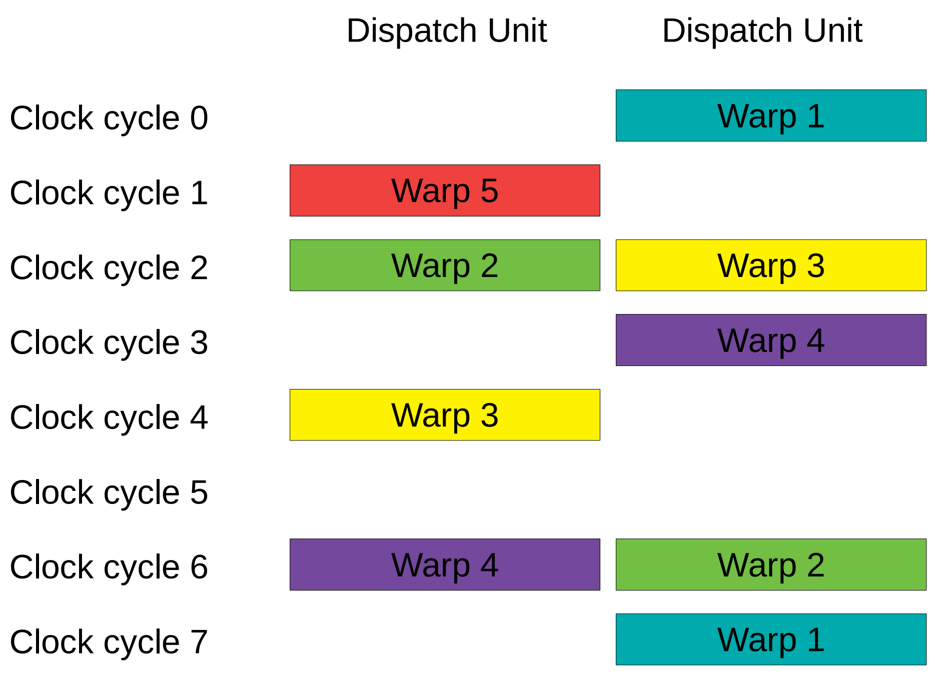
Determined by the hardware limit

Occupancy

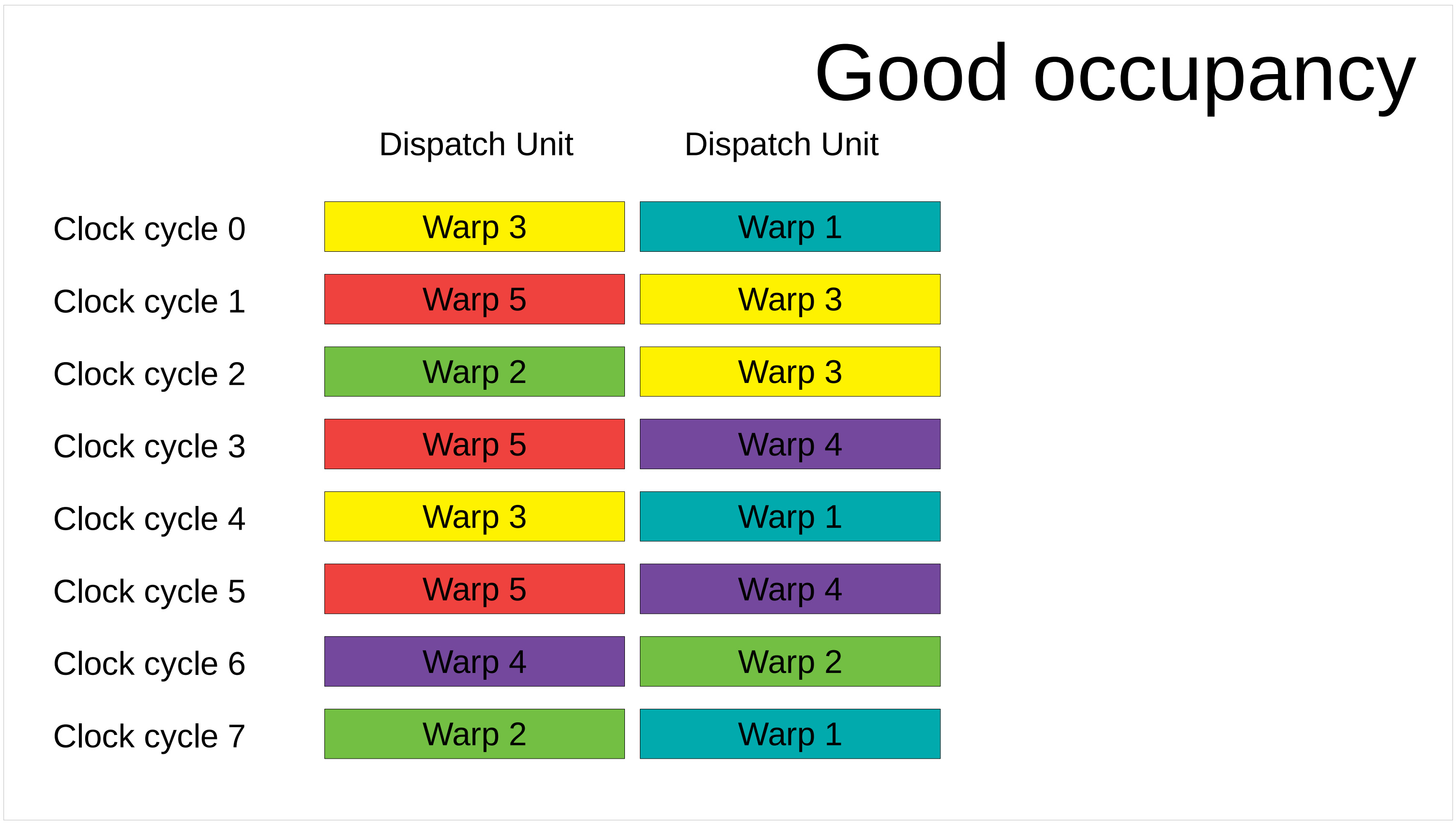
Can be limited by:

Lot of registers per thread
Many memory requests
Frequent expensive instructions
Using shared memory

Bad occupancy



With bad occupancy, many cycles wil leave the hardware unused.



Good occupancy

	Dispatch Unit	Dispatch Unit
Clock cycle 0	Warp 3	Warp 1
Clock cycle 1	Warp 5	Warp 3
Clock cycle 2	Warp 2	Warp 3
Clock cycle 3	Warp 5	Warp 4
Clock cycle 4	Warp 3	Warp 1
Clock cycle 5	Warp 5	Warp 4
Clock cycle 6	Warp 4	Warp 2
Clock cycle 7	Warp 2	Warp 1

Good occupancy instead utilises the hardware in full. This also allows the SM to hide latency, as I explained before.

Occupancy

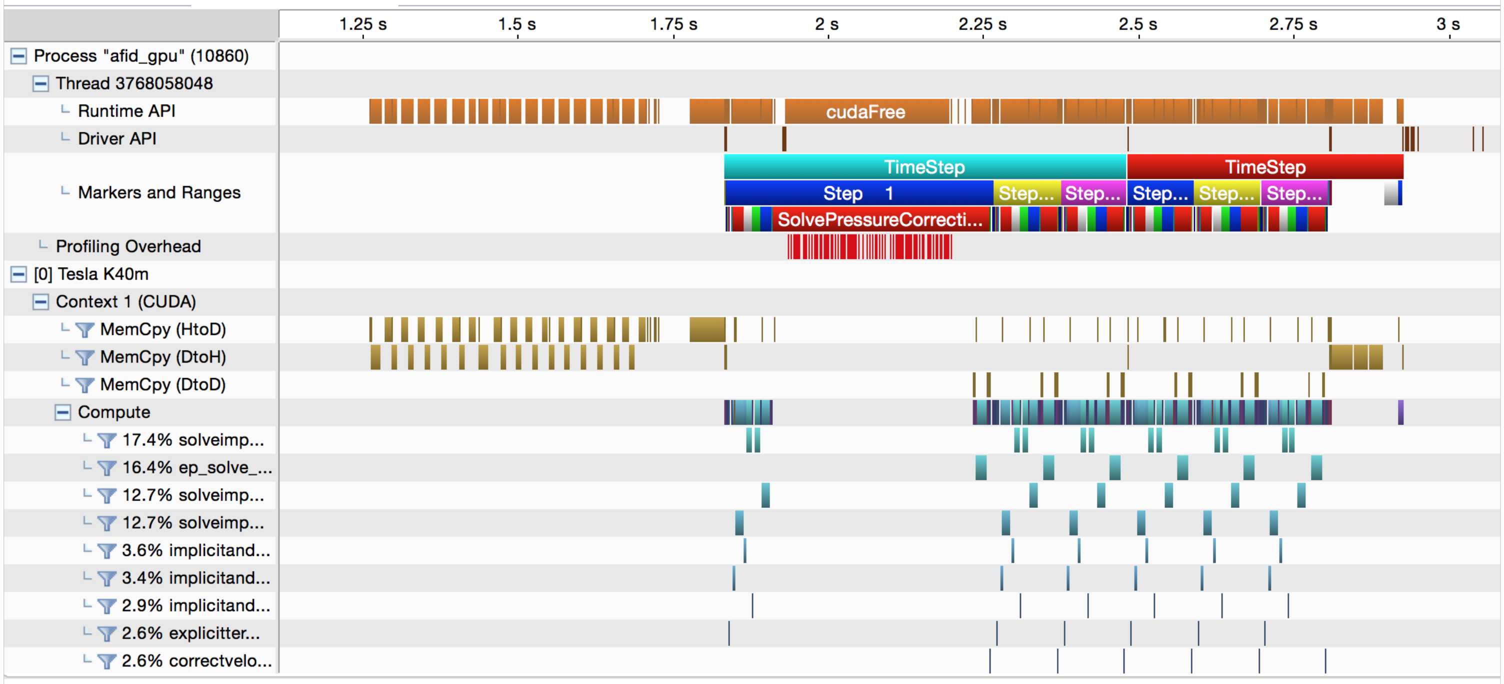
Avoid low occupancy

Best performance is not necessarily 100% occupancy!

Only need enough to saturate the memory bus.

100% occupancy is not a requirement for best performance. However, low occupancy is definitely a sign that your implementation can be improved.

Demonstration!

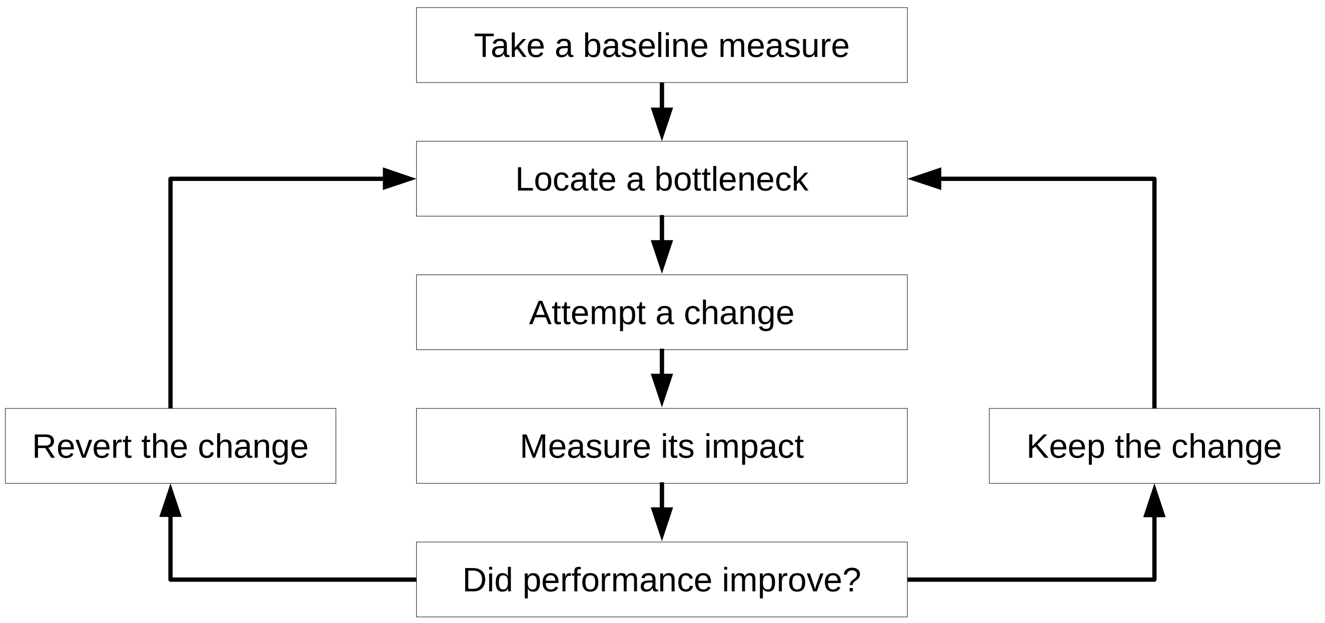


Please see the video of the lecture. I also explain there how to get the profiler going.

Part 3/3:

What can we do to make code run faster?

Remember the optimisation cycle!



I’m going to just list some optimisations you can do, but in general you should always follow the optimisation cycle, and keep on measuring. The things I mention here are potential ways you can improve, not techniques that will always improve performance.

GPU Optimisation is a matter of balance

Thread Complexity

Instruction Throughput

Memory Bandwidth

With many things on the GPU, you’ll find that modifying one thing also affects the others. For example, increasing thread complexity can reduce memory bandwidth, but in some cases reduce instruction throughput because fewer warps can execute simultaneously on the SM.

10 things you can use to make your code run faster

(+ choosing proper launch parameters)

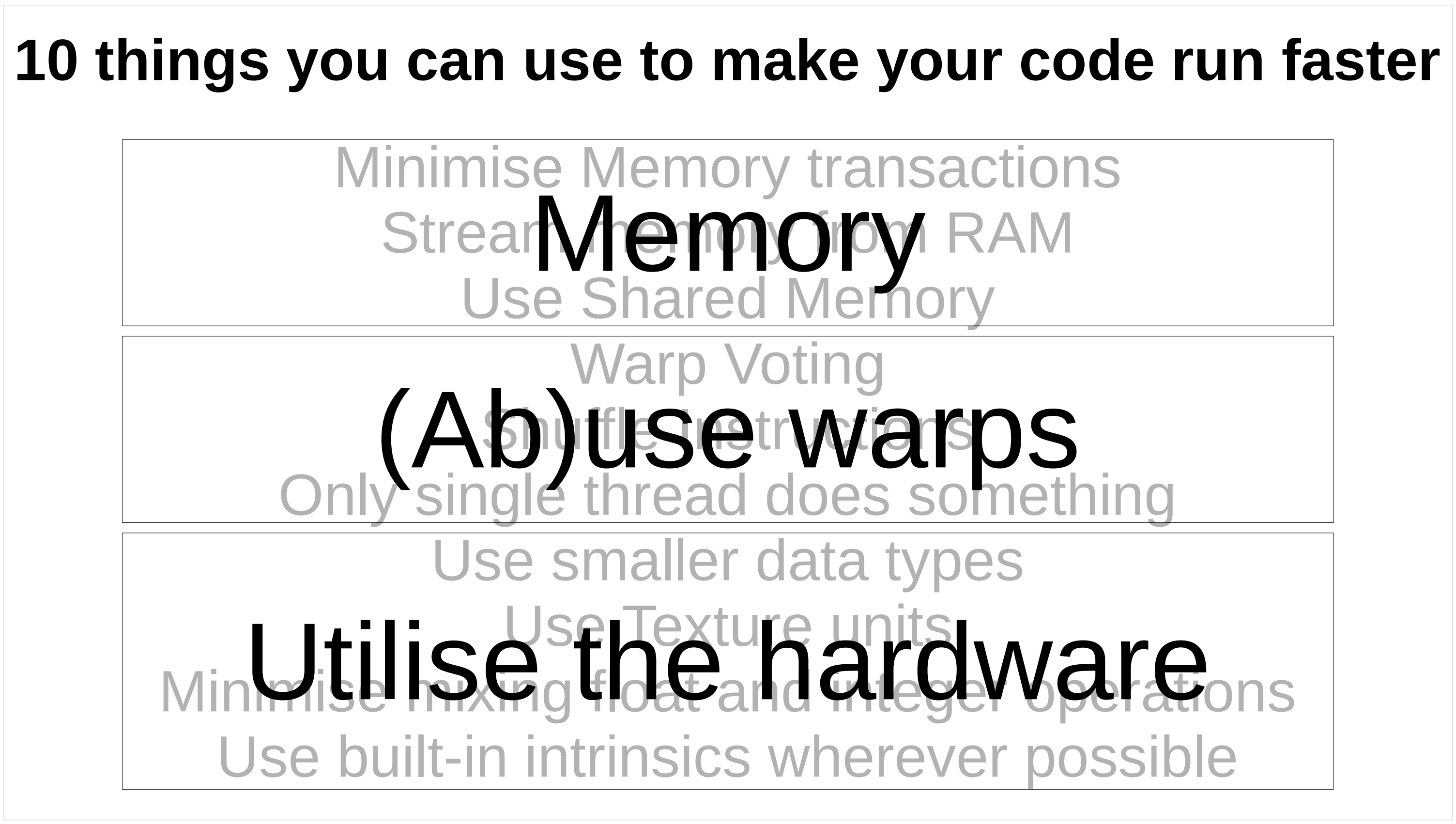
(+ ensuring memory accesses are coalesced)

(+ minimising thread divergence)

So here’s a list of 10 things you can do to improve performance, minus a few of the things I already mentioned before.

10 things you can use to make your code run faster

Minimise Memory transactions
Use Shared Memory
Stream memory from RAM
Warp Voting
Shuffle Instructions
Only single thread does something
Use smaller data types
Use Texture units
Minimise mixing float and integer operations
Use built-in intrinsics wherever possible



10 things you can use to make your code run faster

Minimise Memory transactions
Stream memory from RAM
Use Shared Memory

Memory

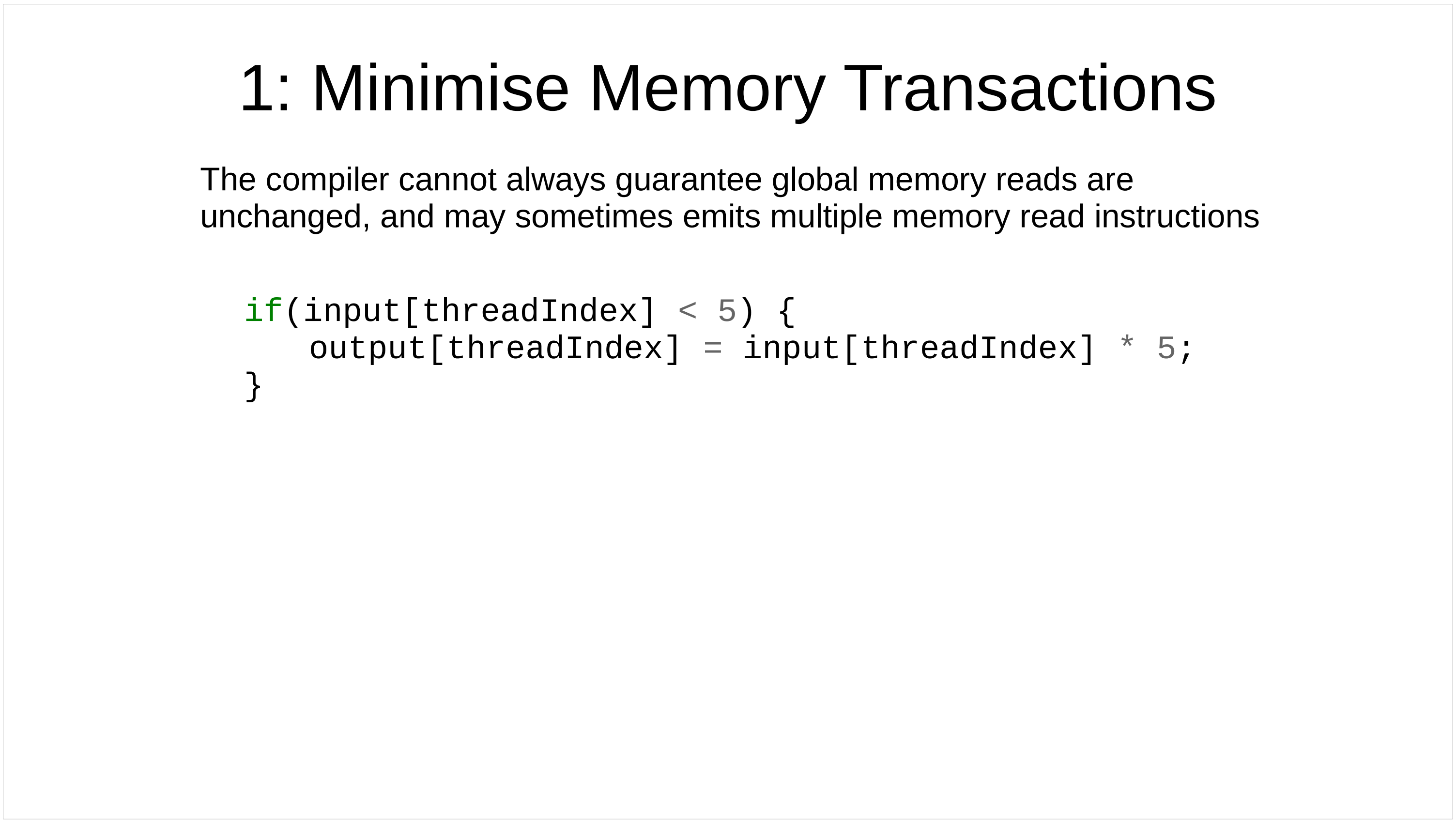
Warp Voting
Shuffle instructions
Only single thread does something

(Ab)use warps

Use smaller data types
Use Texture units
Minimise mixing float and integer operations
Use built-in intrinsics wherever possible

Utilise the hardware

Basically they can be categorised into three main groups.

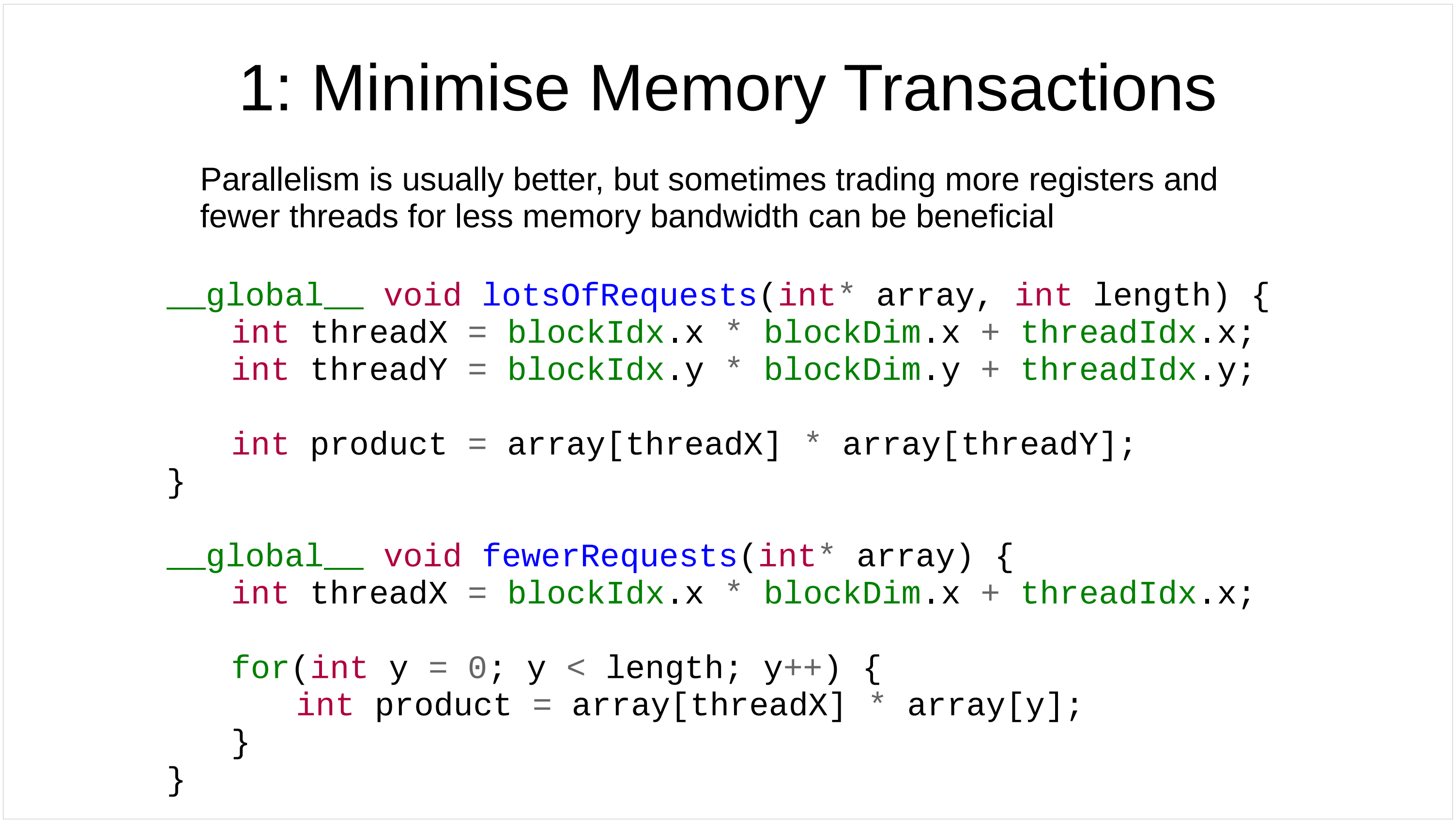


1: Minimise Memory Transactions

The compiler cannot always guarantee global memory reads are unchanged, and may sometimes emits multiple memory read instructions

```
if(input[threadIndex] < 5) {
   output[threadIndex] = input[threadIndex] * 5;
}
```

First of all, memory transactions quickly put a lot of pressure on the memory bus. You really need to think twice about reading and writing to and from main memory. Even reducing a single memory transaction can already make a big difference there.

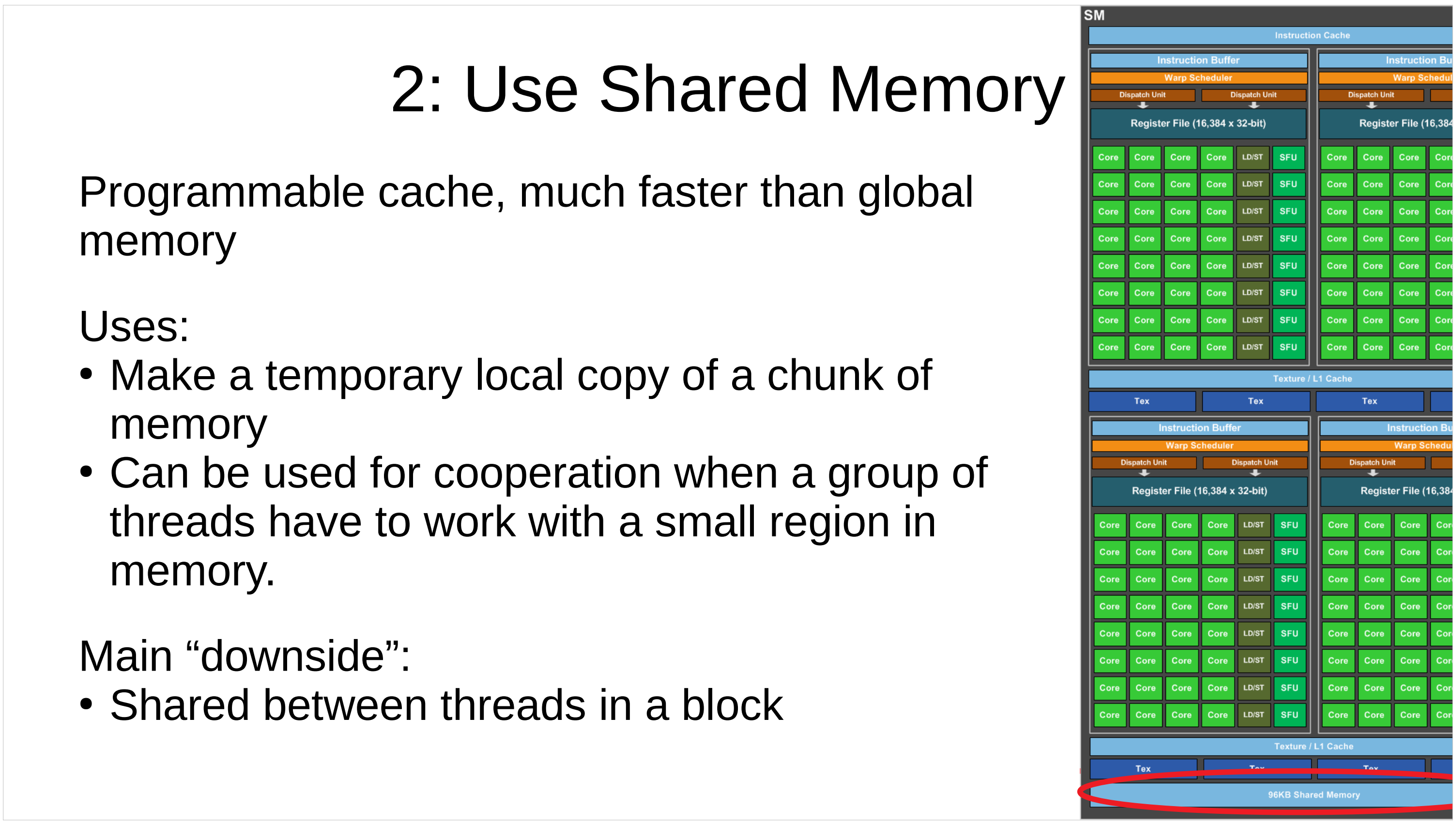


1: Minimise Memory Transactions

Parallelism is usually better, but sometimes trading more registers and fewer threads for less memory bandwidth can be beneficial

```
__global__ void lotsOfRequests(int* array, int length) {
    int threadX = blockIdx.x * blockDim.x + threadIdx.x;
    int threadY = blockIdx.y * blockDim.y + threadIdx.y;

    int product = array[threadX] * array[threadY];
}

__global__ void fewerRequests(int* array) {
    int threadX = blockIdx.x * blockDim.x + threadIdx.x;

    for(int y = 0; y < length; y++) {
        int product = array[threadX] * array[y];
    }
}
```

One example of such a reduction is to, rather than launching a 2D grid of threads, only launch a thread per row or column. This saves you loading one of the pieces of data many times.



2: Use Shared Memory

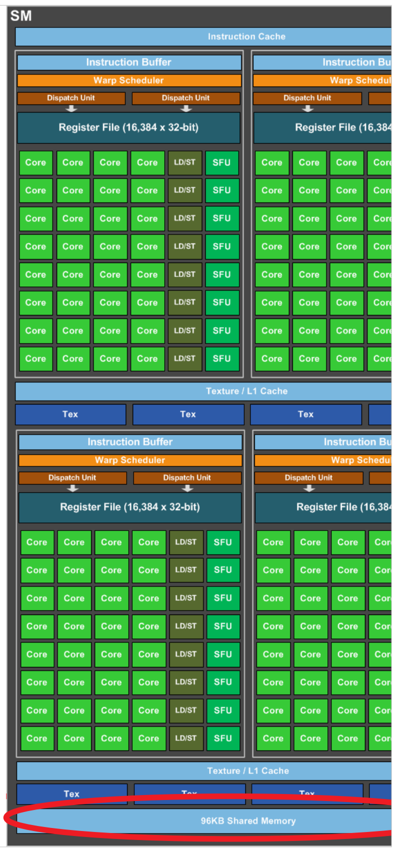
Programmable cache, much faster than global memory

Uses:

- Make a temporary local copy of a chunk of memory
- Can be used for cooperation when a group of threads have to work with a small region in memory.

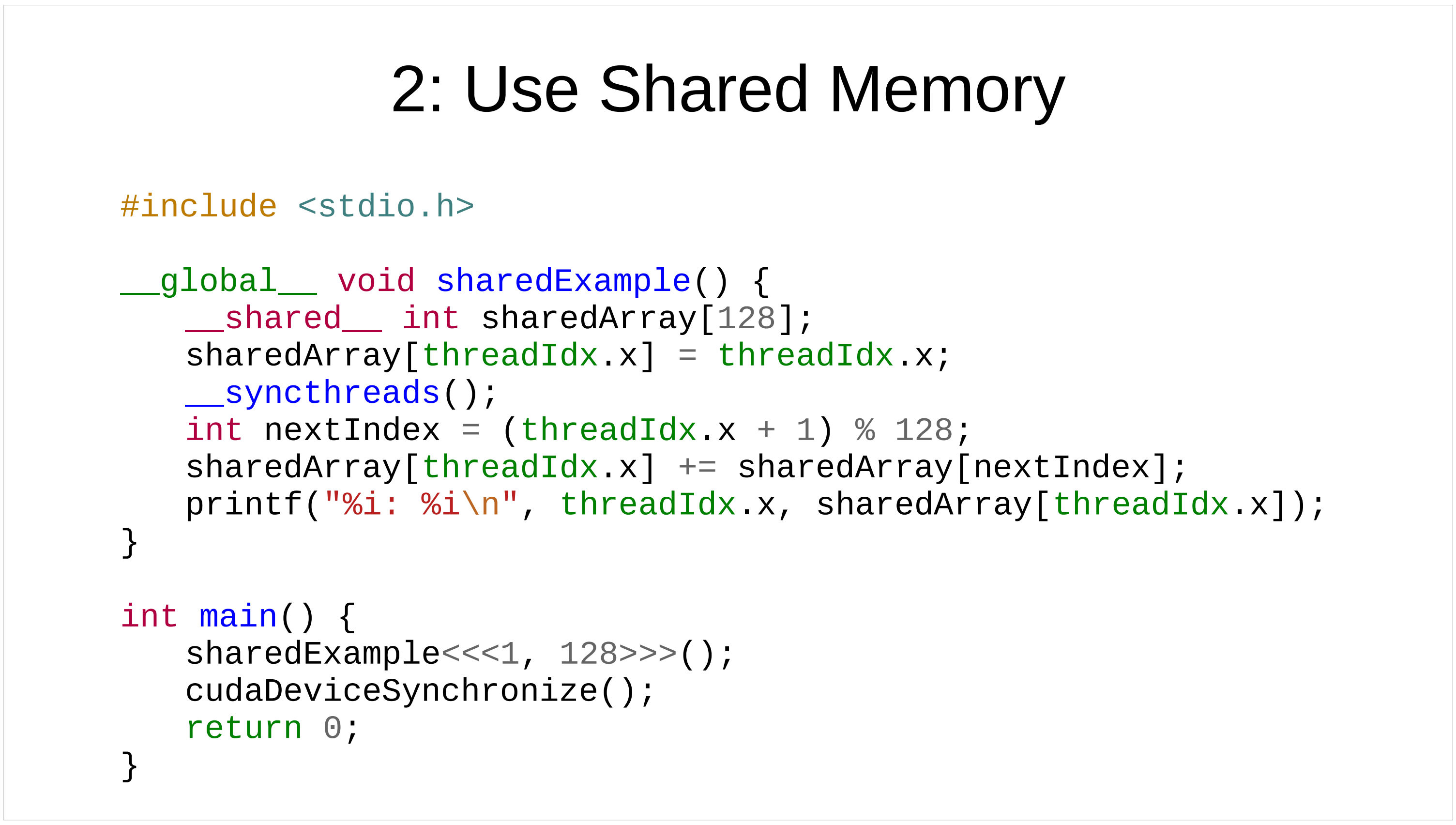
Main “downside”:

- Shared between threads in a block



Shared memory is a lot faster than global memory. As such, if you need to read and write multiple times to a piece of memory with many threads, you may want to allocate some shared memory instead, and copy the result to main memory once you’re done.

However, shared memory is allocated on a block by block basis. This means you may need to make your blocks larger to make effective use of it, which has some downsides as I showed you earlier.



2: Use Shared Memory

```
#include <stdio.h>

__global__ void sharedExample() {
    __shared__ int sharedArray[128];
    sharedArray[threadIdx.x] = threadIdx.x;
    __syncthreads();
    int nextIndex = (threadIdx.x + 1) % 128;
    sharedArray[threadIdx.x] += sharedArray[nextIndex];
    printf("%i: %i\n", threadIdx.x, sharedArray[threadIdx.x]);
}

int main() {
    sharedExample<<<1, 128>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

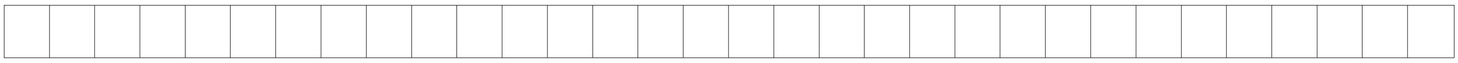
Here’s a code example where shared memory is used for threads to perform a task cooperatively.

2: Use Shared Memory

Note: Shared Memory is implemented as 32 separate memory banks.

Make sure requests to this memory are as much coalesced as possible!

Any bank conflicts that occur (threads attempting to read/write to the same bank) are handled in serial.



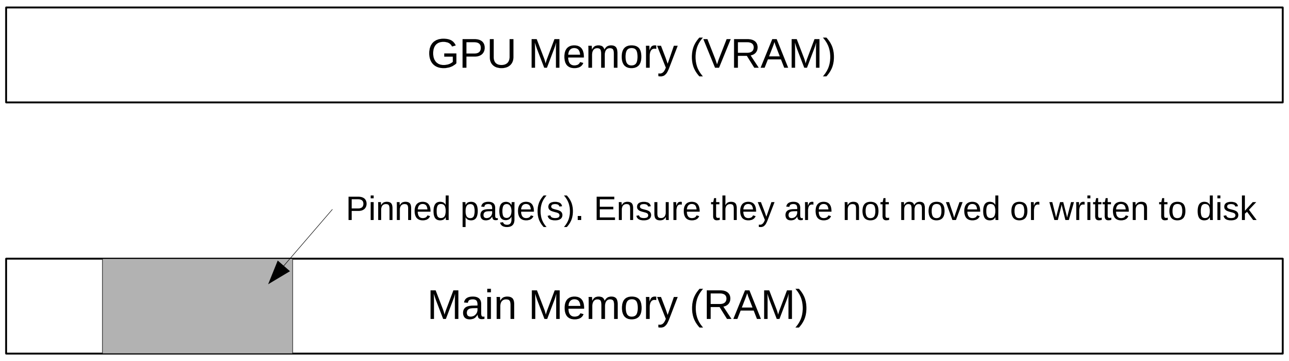
Memory is usually divided into so-called “banks” (both CPU and GPU memory). These banks can usually only handle requests in serial.

Shared memory is also divided into banks. If many threads read and write to and from the same value, individual requests need to be serviced in serial. So if you can help it, make sure your shared memory accesses are coalesced.

3: Stream memory from RAM

cudaMemcpy back and forth can take time
(especially for large buffers)

Can use “unified memory” to stream memory back and forth, *while the kernel is executing*


GPU Memory (VRAM)
Pinned page(s). Ensure they are not moved or written to disk
Main Memory (RAM)

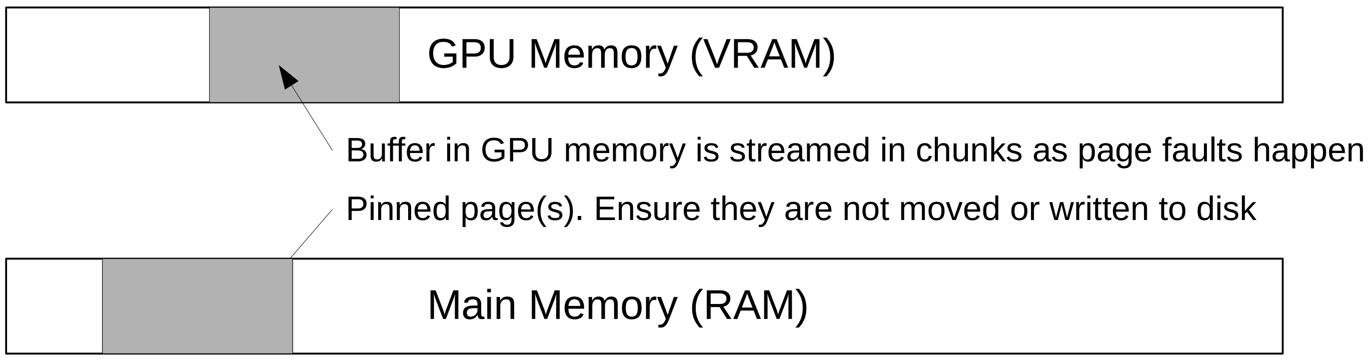
If you have a lot of data, consider streaming memory, rather than explicitly copy it to the GPU, process it, and finally copying it back.

Pascal cards and later can use hardware pagefault to redirect a memory request to main memory. The memory page (which needs to be a so-called “pinned” memory page in main (CPU) memory will subsequently be transferred to the GPU. This allows data processing to happen while data is still being copied.

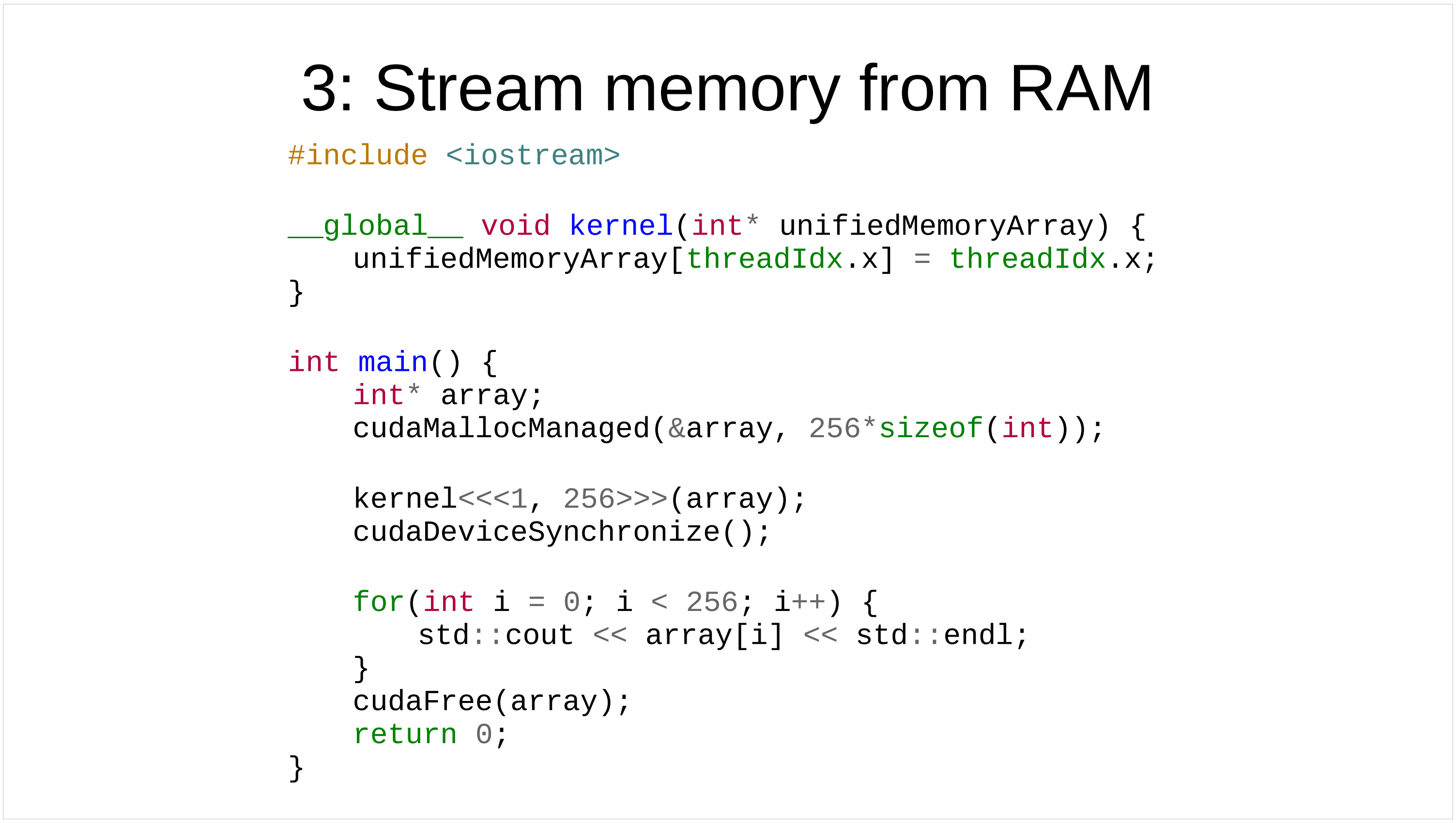
3: Stream memory from RAM

1. Use cudaMallocManaged() to allocate buffer.
2. Can access memory on the CPU **and** GPU side using the **same** pointer
3. Any necessary transfers are handled for you

.. But assumes no race conditions between the CPU and GPU side!


GPU Memory (VRAM)
Buffer in GPU memory is streamed in chunks as page faults happen
Pinned page(s). Ensure they are not moved or written to disk
Main Memory (RAM)

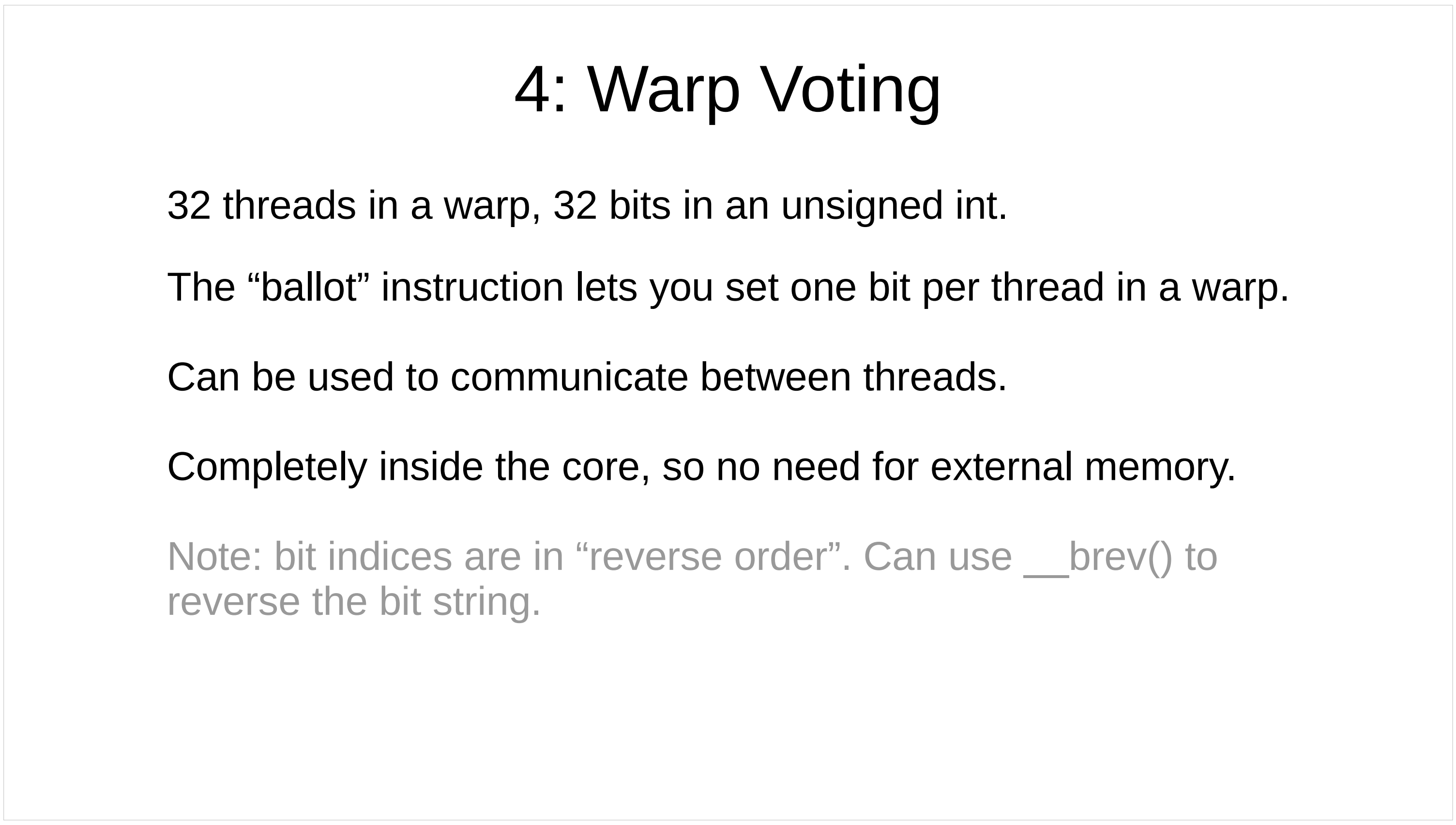
And on top of that, because there’s only a single pointer for you to use, and you don’t need to explicitly transfer memory back and forth, it’s actually easier for you to use.



3: Stream memory from RAM

```
#include <iostream>

__global__ void kernel(int* unifiedMemoryArray) {
    unifiedMemoryArray[threadIdx.x] = threadIdx.x;
}

int main() {
    int* array;
    cudaMallocManaged(&array, 256*sizeof(int));

    kernel<<<1, 256>>>(array);
    cudaDeviceSynchronize();

    for(int i = 0; i < 256; i++) {
        std::cout << array[i] << std::endl;
    }
    cudaFree(array);
    return 0;
}
```

Though be aware that when the CPU and GPU both are writing to the same memory, the result is undefined. So make sure you only process your data on one side at a time.



4: Warp Voting

32 threads in a warp, 32 bits in an unsigned int.

The “ballot” instruction lets you set one bit per thread in a warp.

Can be used to communicate between threads.

Completely inside the core, so no need for external memory.

Note: bit indices are in “reverse order”. Can use __brev() to reverse the bit string.

To some extent, the requirement that GPUs only execute threads in a warp can be seen as a limitation. However, the warp voting and shuffle instructions turn this around, and use it as an advantage instead.

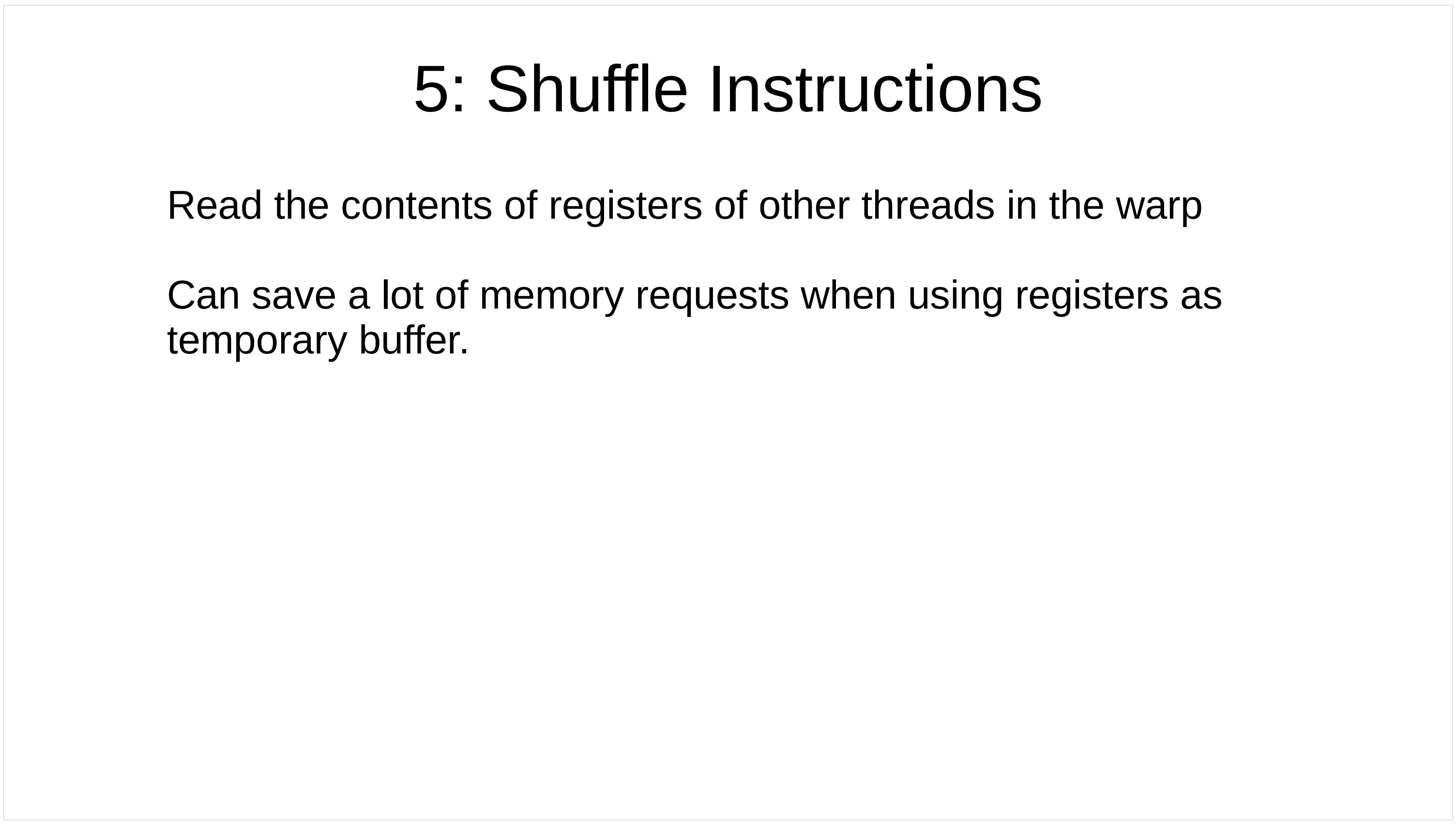
First off, warp voting allows all threads in the warp to “vote” with a boolean value. All votes are concatenated, and because a warp has 32 threads, the result is a 32-bit unsigned integer.

You can for example use this for figuring out which threads have work to do when having threads in a warp cooperate on a task. The other big advantage here is that because everything happens in-core, this means of communication is super fast. The same is true for shuffle instructions.

4: Warp Voting

```
#include <stdio.h>

__global__ void ballotExample() {
    bool needBathroom = threadIdx.x % 3 == 0;
    unsigned int votes = __ballot_sync(0xFFFFFFFF, needBathroom);
    unsigned int count = __popc(votes);
    if(threadIdx.x == 0) {
        printf("%i threads need a break.\n", count);
    }
}

int main() {
    ballotExample<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

The __popc() function is a "population count": it returns the number of bits set. In this case, that will tell you how many threads votes "true" with their __ballot_sync() call.



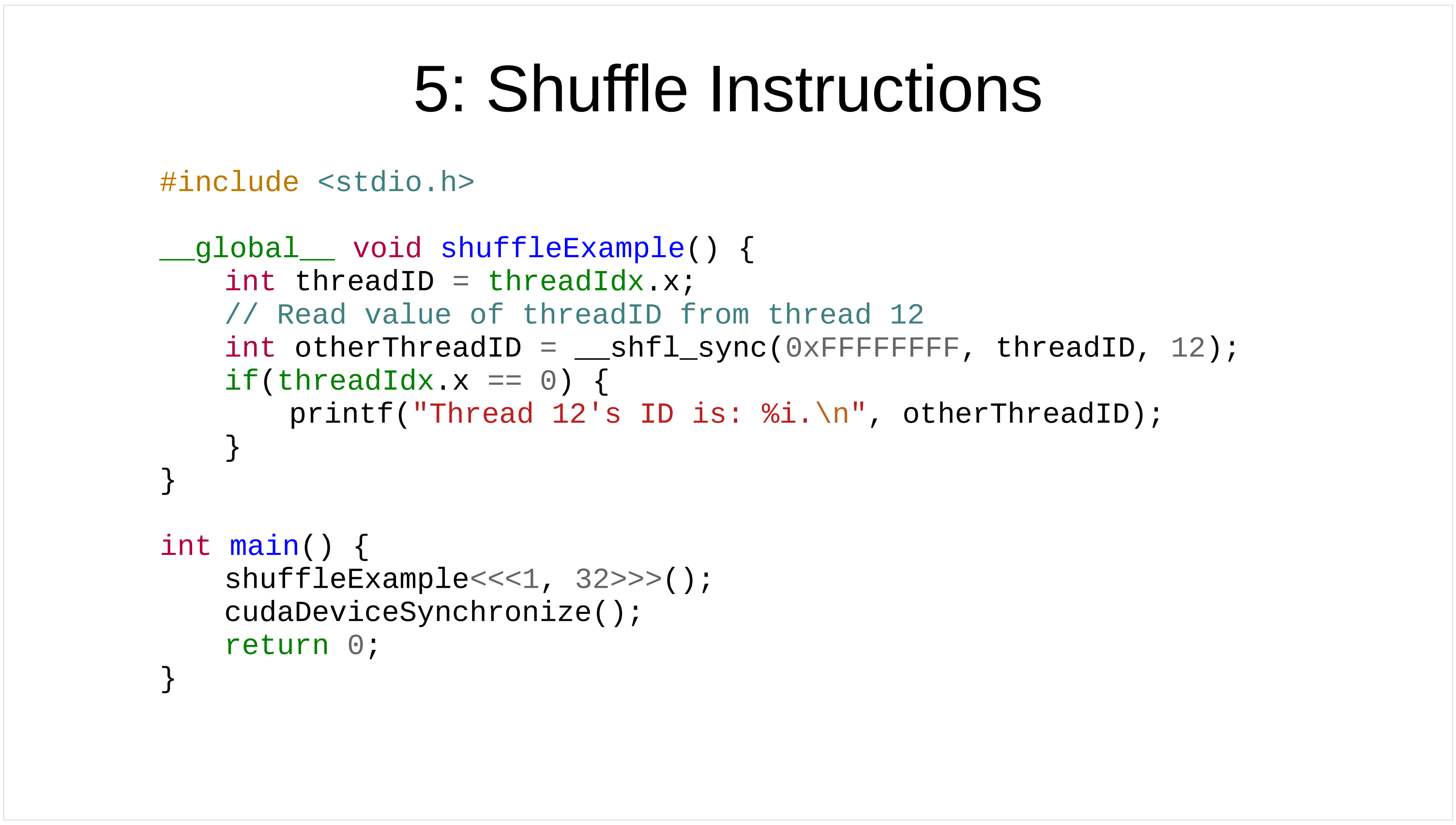
5: Shuffle Instructions

Read the contents of registers of other threads in the warp

Can save a lot of memory requests when using registers as temporary buffer.

Shuffle instructions can be a bit confusing at first, but the main idea is that threads within a warp can exchange the values of their variables (registers, to be more precise).

This allows you to, in a way, use registers as a way to temporarily store data you need fast access to. As long as one thread has a particular value, other threads can get hold of it if they need it.



5: Shuffle Instructions

```
#include <stdio.h>

__global__ void shuffleExample() {
    int threadID = threadIdx.x;
    // Read value of threadID from thread 12
    int otherThreadID = __shfl_sync(0xFFFFFFFF, threadID, 12);
    if(threadIdx.x == 0) {
        printf("Thread 12's ID is: %i.\n", otherThreadID);
    }
}

int main() {
    shuffleExample<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Here’s an example. Each thread passes in a variable, and a thread they want to read the value of that variable from. The __shfl_sync() function will return the value that that variable has in the specified thread.

6: Only one thread does something

Atomic operations can often be grouped. If we only use one thread for such an operation, we save 31 atomic instructions *per warp*.

Also useful in some cases for memory fetches (though where this should be applied depends on the architecture)

Many operations, including atomics and memory transactions, can really hammer some systems in the GPU. Fortunately, these can sometimes be simplified if only one thread in a warp performs that action.

For instance, consider a case where each thread in a warp attempts to perform an atomic addition on one variable. Rather than performing the atomic operation 32 times, you can perform a reduction within the warp (another use for shuffle instructions!), and have only one thread perform the addition “on behalf of” all threads. For atomics this can save a lot of pressure on L2 cache, where atomic operations are resolved.

```
#include <stdio.h>
__global__ void shuffleExample(int* array, int* nextIndex) {
    bool needToDoSomething = threadIdx.x % 5 == 0;
    int myValue = blockIdx.x * 100 + threadIdx.x;
    unsigned int votes = __brev(__ballot_sync(0xFFFFFFFF, needToDoSomething));
    unsigned int count = __popc(votes);
    unsigned int startIndex;
    if(threadIdx.x == 0) { startIndex = atomicAdd(nextIndex, count); }
    startIndex = __shfl_sync(0xFFFFFFFF, startIndex, 0);
    int howManyCameBeforeMe = __popc(votes >> (32 - threadIdx.x));
    if(needToDoSomething) {
        printf("%i -> %i\n", startIndex + howManyCameBeforeMe, myValue);
        array[startIndex + howManyCameBeforeMe] = myValue;
    }
}

int main() {
    int* array; int* nextIndex;
    cudaMalloc(&array, 32 * sizeof(int)); cudaMalloc(&nextIndex, sizeof(int));
    cudaMemset(nextIndex, sizeof(int), 0);
    shuffleExample<<<3, 32>>>(array, nextIndex);
    cudaDeviceSynchronize();
    return 0;
}
```

Here’s an example showing how ballot and shuffle instructions can be used. The values of needToDoSomething and myValue are just arbitrary input. We first perform a vote. This tells us how many elements in the input array to reserve. Here only one thread is performing this action, and is reserving spots in the array for all threads at once. Next, because each thread needs to know which starting index thread 0 reserved, we broadcast the start index using a shuffle instruction to all threads. Now each thread computes which index they should write to, by computing how many threads voted “true” before them [by thread index in the warp]. We finally write out the value to the array.

In this way, we can append items to an array using only a single atomic operation.

```
#include <stdio.h>
__global__ void shuffleExample(int* array, int* nextIndex) {
    bool needToDoSomething = threadIdx.x % 5 == 0;
    int myValue = blockIdx.x * 100 + threadIdx.x;
    unsigned int votes = __brev(__ballot_sync(0xFFFFFFFF, needToDoSomet
    unsigned int count = __popc(votes);
    unsigned int startIndex;
    if(threadIdx.x == 0) { startIndex = atomicAdd(nextIndex, count); }
    startIndex = __shfl_sync(0xFFFFFFFF, startIndex, 0);
    int howManyCameBeforeMe = __popc(votes >> (32 - threadIdx.x));
    if(needToDoSomething) {
        printf("%i -> %i\n", startIndex + howManyCameBeforeMe, myValue)
        array[startIndex + howManyCameBeforeMe] = myValue;
    }
}

int main() {
    int* array; int* nextIndex;
    cudaMalloc(&array, 32 * sizeof(int)); cudaMalloc(&nextIndex, sizeof
    cudaMemset(nextIndex, sizeof(int), 0);
    shuffleExample<<<3, 32>>>(array, nextIndex);
    cudaDeviceSynchronize();
    return 0;
}
```

```
14 -> 0
15 -> 5
16 -> 10
17 -> 15
18 -> 20
19 -> 25
20 -> 30
7 -> 100
8 -> 105
9 -> 110
10 -> 115
11 -> 120
12 -> 125
13 -> 130
0 -> 200
1 -> 205
2 -> 210
3 -> 215
4 -> 220
5 -> 225
6 -> 230
```

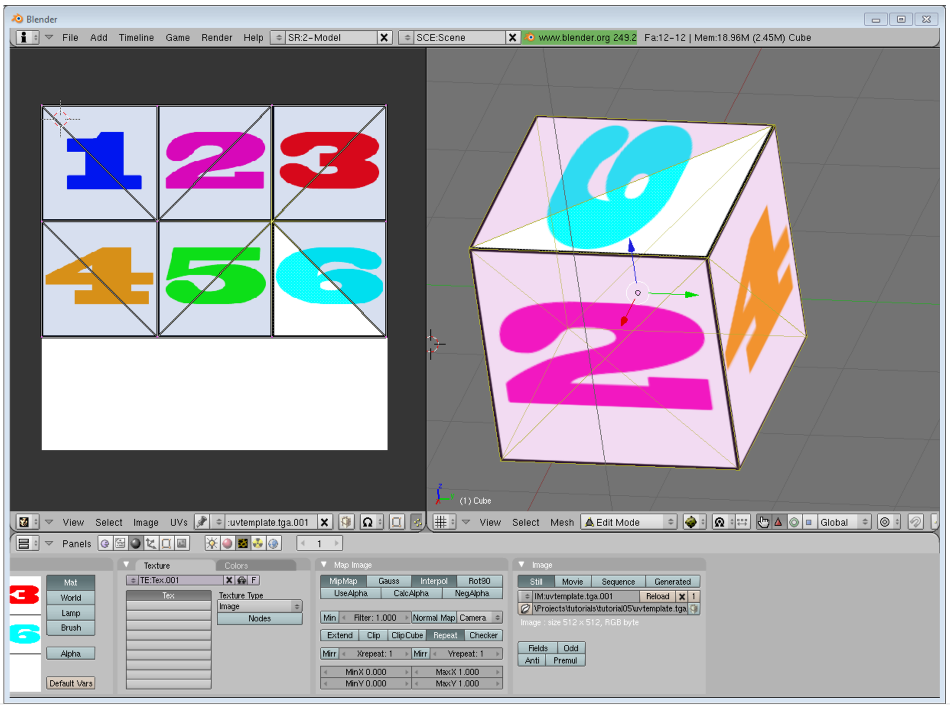
7: Use Smaller Data Types

Consider using unsigned shorts or half precision floats.

Smaller data types mean less memory bandwidth is required to fetch them from memory.

It is possible to make better use of memory bandwidth by reducing the size of your data types. For instance, you can often get away with unsigned shorts rather than ints. Likewise, depending on your application, half floats rather than single precision ones may also suffice.

8: Use Texture Units



Texture units have their own bit of cache, and are made to handle 2D data. They can even perform some fixed function operations on it, such as interpolation.

The NVCC compiler will sometimes use these units automatically, though you can make use of them explicitly if you so desire.

8: Use Texture Units

Texture units is hardware specialised in dealing with 2D data.

They also have their own memory path and cache,
albeit read-only

Can do some minor processing, such as
automatic interpolation between “pixels”.

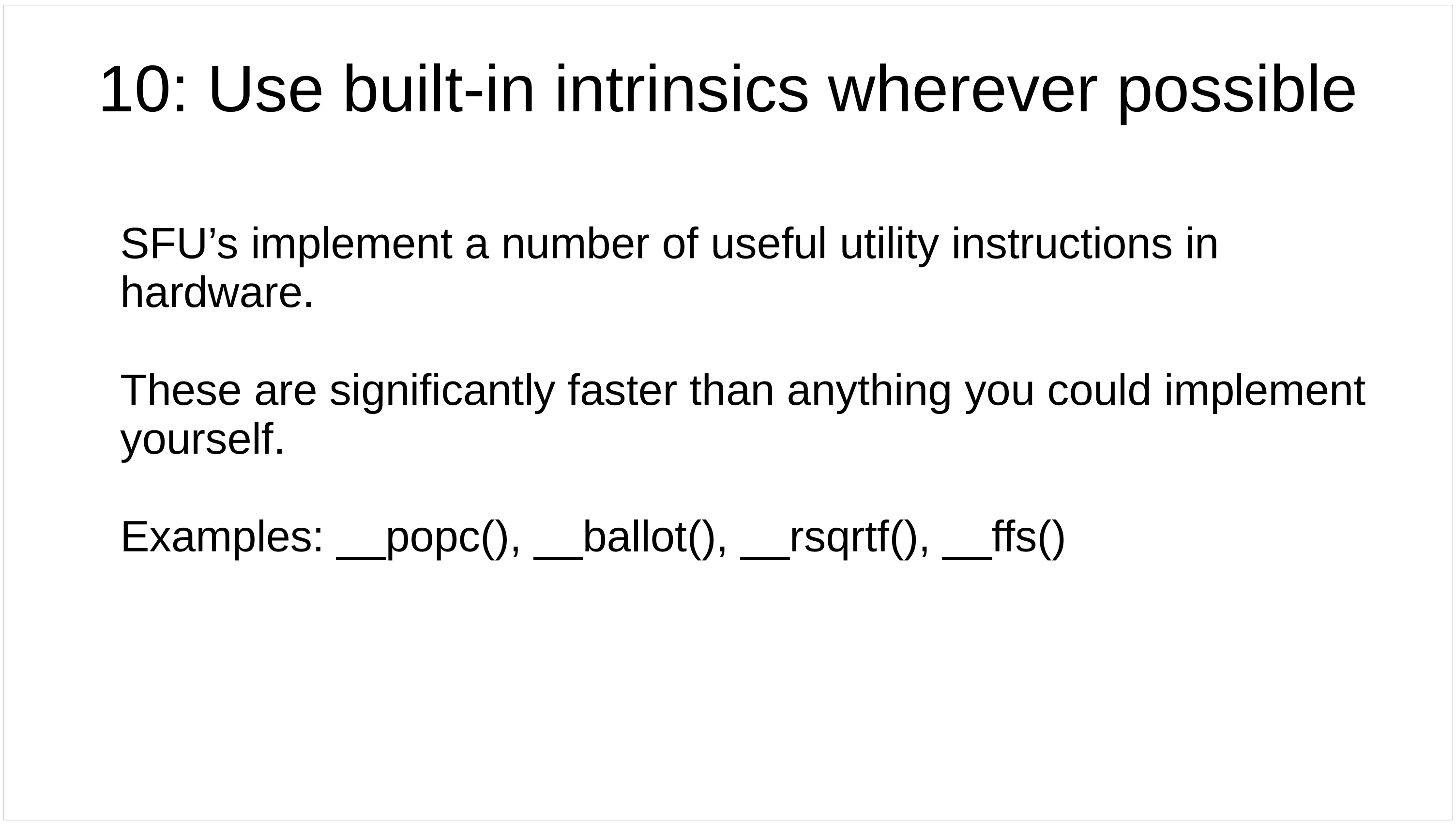
9: Minimise mixing floats and integers

On Pascal and before, intermittent float and integer instructions caused significant stalls.

Graphics processors tend to be heavily focussed on single precision floating point computations.

This is mainly true for the Pascal architecture and older, but float and integer computations tend to stall the pipeline a lot.

It’s normally not really something to look for, but in rare cases it can be a problem.



10: Use built-in intrinsics wherever possible

SFU’s implement a number of useful utility instructions in hardware.

These are significantly faster than anything you could implement yourself.

Examples: __popc(), __ballot(), __rsqrtf(), __ffs()

Graphics cards have hardware implementations of a number of utility functions (min, max, cos, sin, etc). Use these as much as possible rather than implementing your own, as those are always going to be faster.

Next week:

OpenCL

Next week: the pinnacle of API's that aims to unify all parallel hardware.