TDT4200 Parallel Programming

Bart van Blokland

Lecture 10

&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];
```

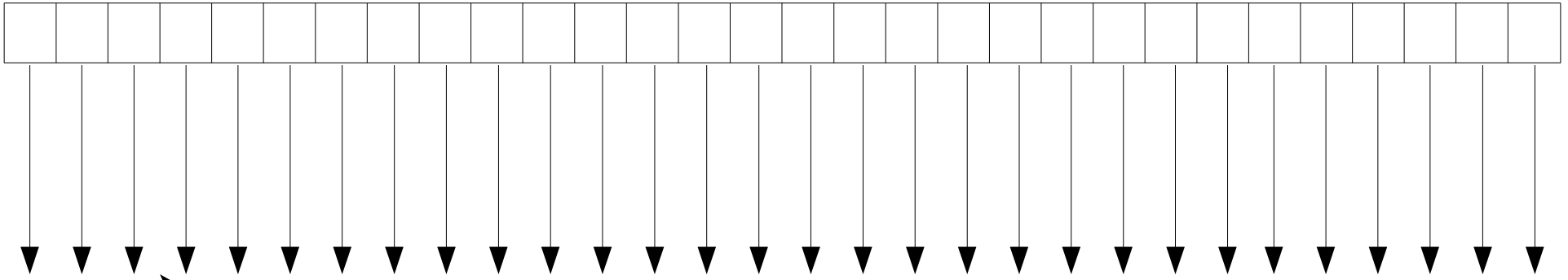
The usual way to iterate over it:

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

On the GPU:

```
int myIndex = computeThreadIndex();

doSomething(myIndex);
```



Threads
(each runs doSomething() independently)

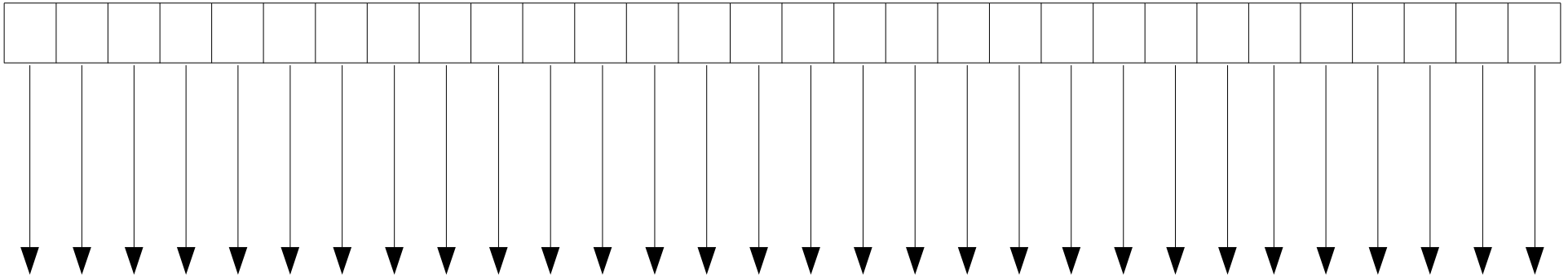
Threads launched on the GPU are independent of the amount of data you need to process.

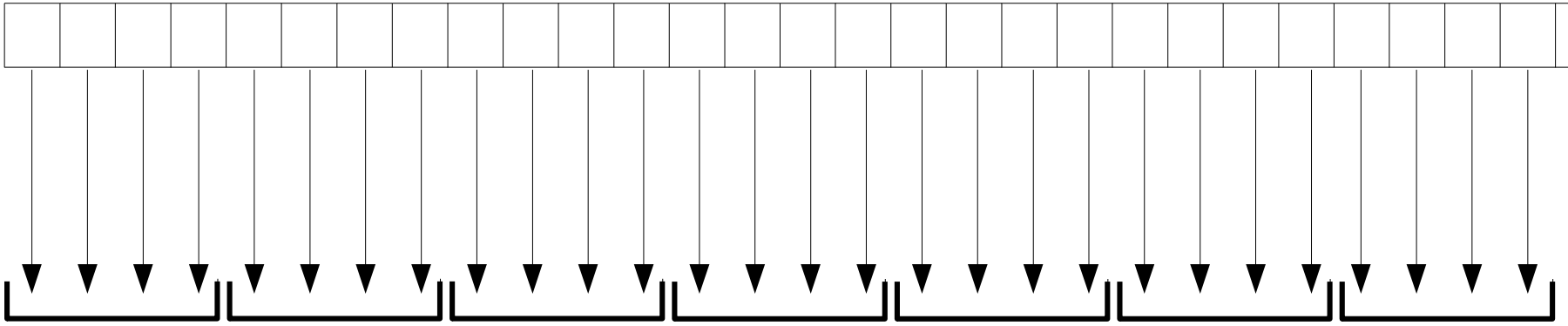
But that’s fine, right? We just launch as many threads as we need!

Unfortunately, it’s a bit more complicated:

Threads are grouped into Blocks.

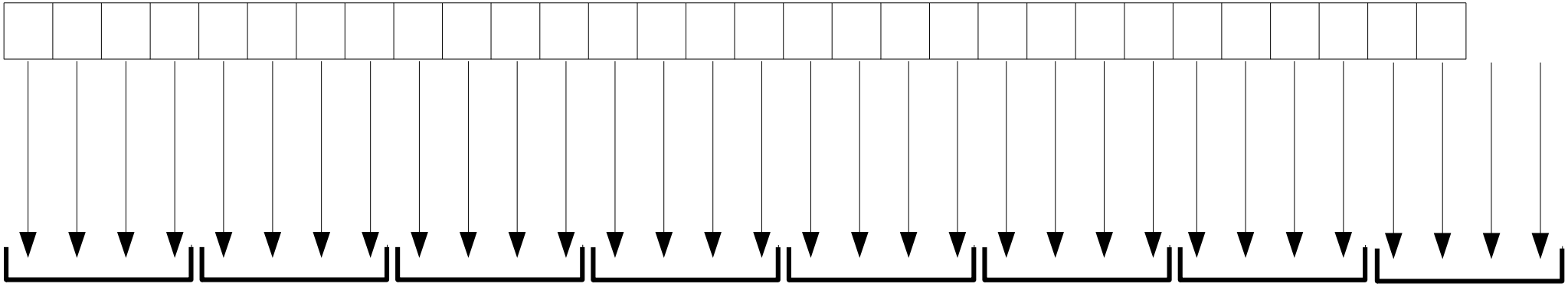
Reason: Thread Cooperation





Blocks

Too few blocks:
Some items don’t get processed.



Enough blocks:
All items get processed!

But now we have more
threads than we need..

Solution 1:

Find a block size that divides
the array in equal parts

Solution 2:

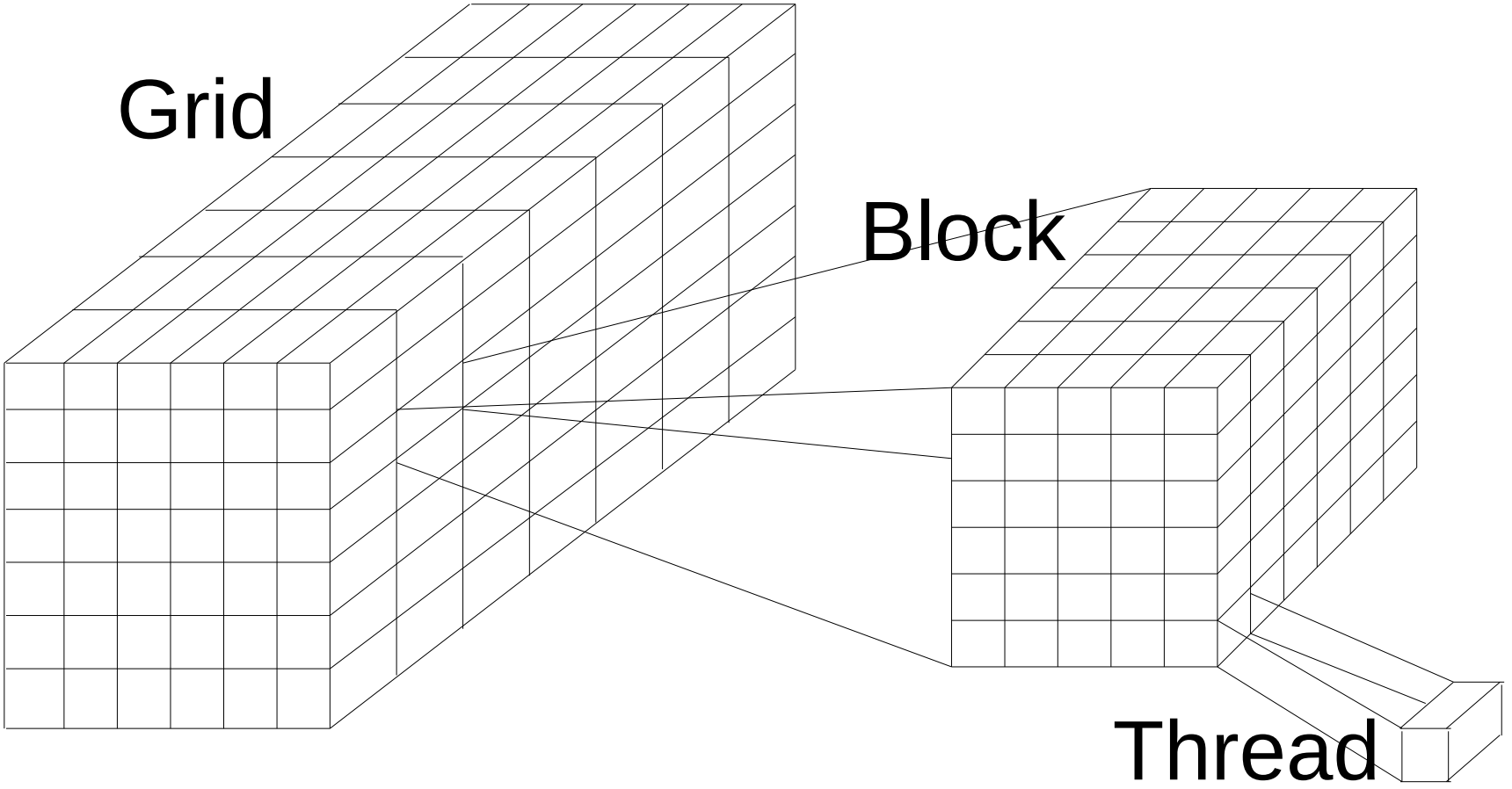
Launch more threads than the
number of elements in the array,
each thread checks for out of bounds

Solution 1:

Find a block size that divides the array into equal parts

Solution 2:

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


Grid
Block
Thread

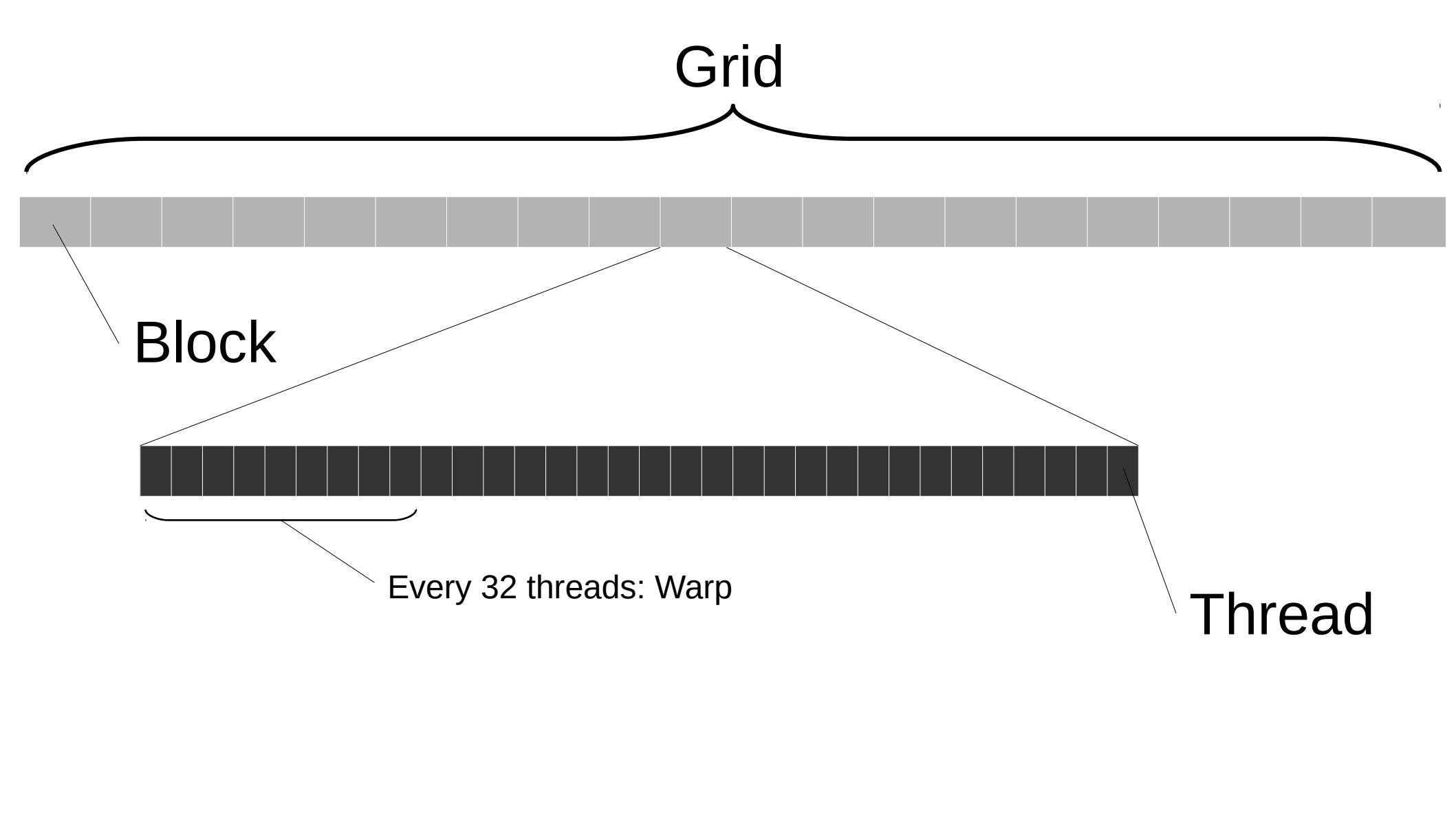
Which of these is better?

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>>>();
```


Grid
Block
Every 32 threads: Warp
Thread

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

Today:

GPU Performance

TL;DSTTL:

Use the GPU

Measure

What can cause GPU code to run slower?

How do we measure GPU performance?

What can we do to make code run faster?

Part 1/3:

What can cause GPU code to run slower?

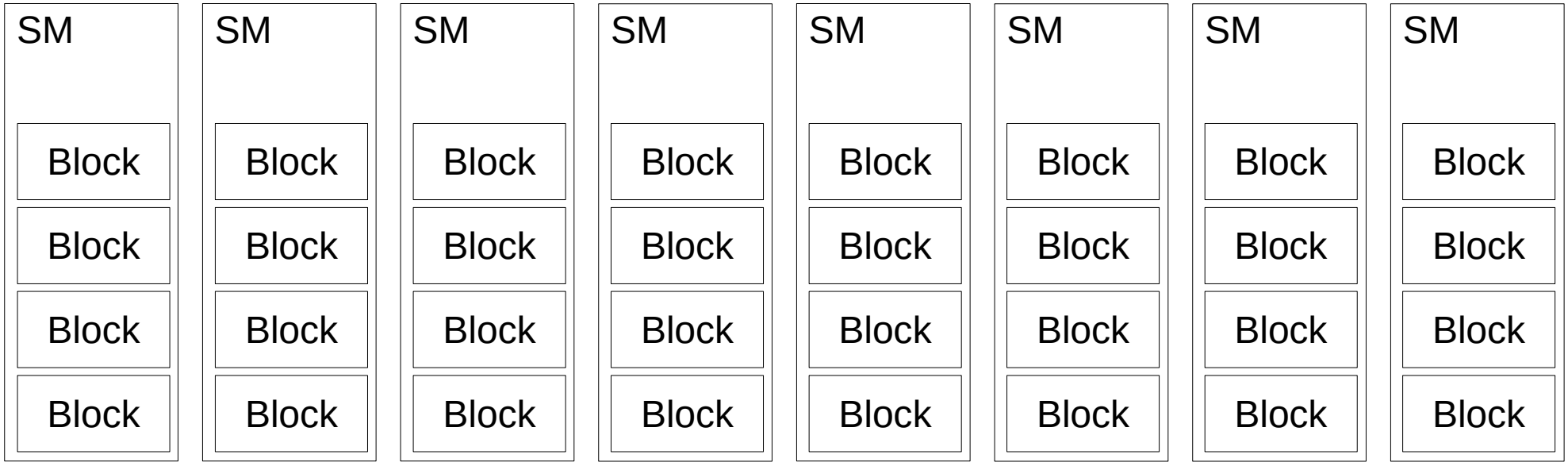
Non-Coalesced Accesses
Low Occupancy
Thread Divergence
Register Spilling
Resource Contention
Synchronization

And many more..

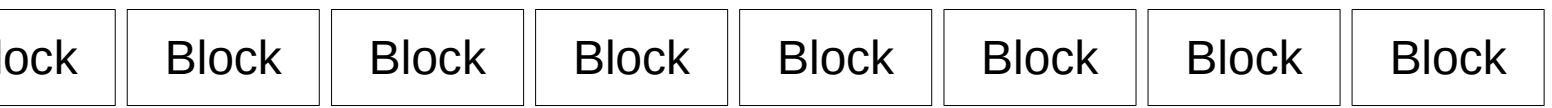
But first:

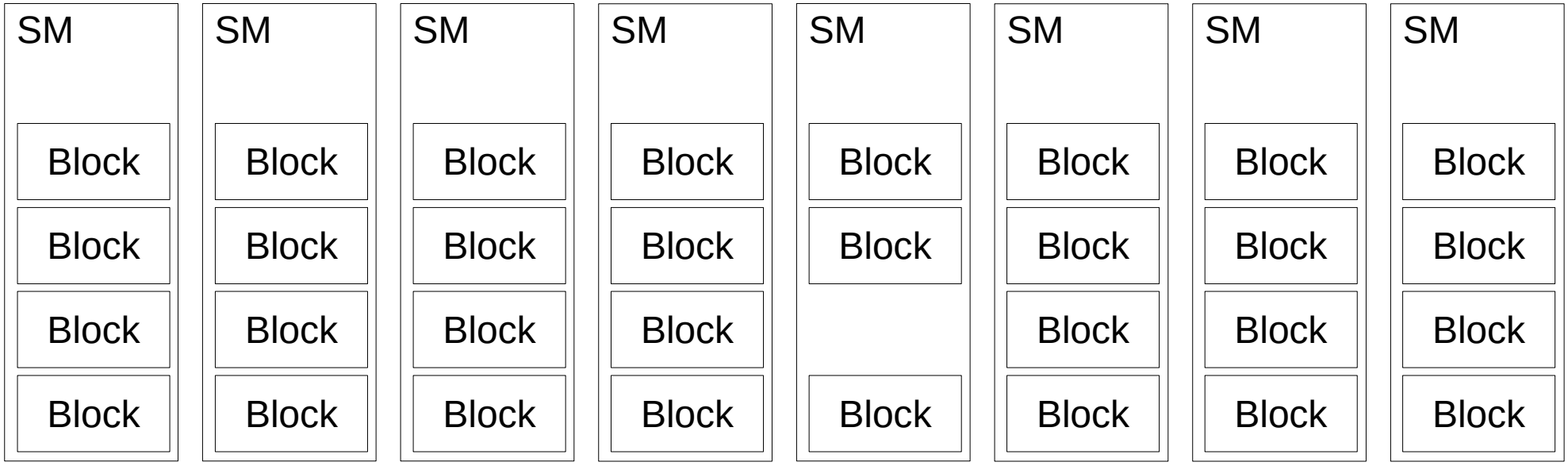
In order to understand these,
we need to understand the
hardware a little better.

How do Streaming Multiprocessors execute kernels?

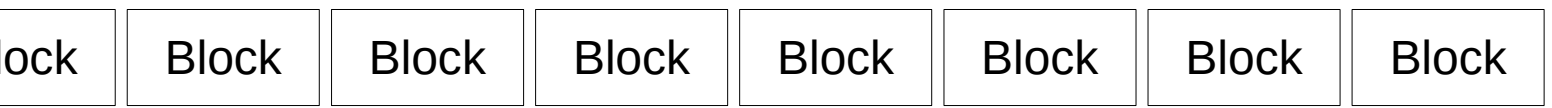


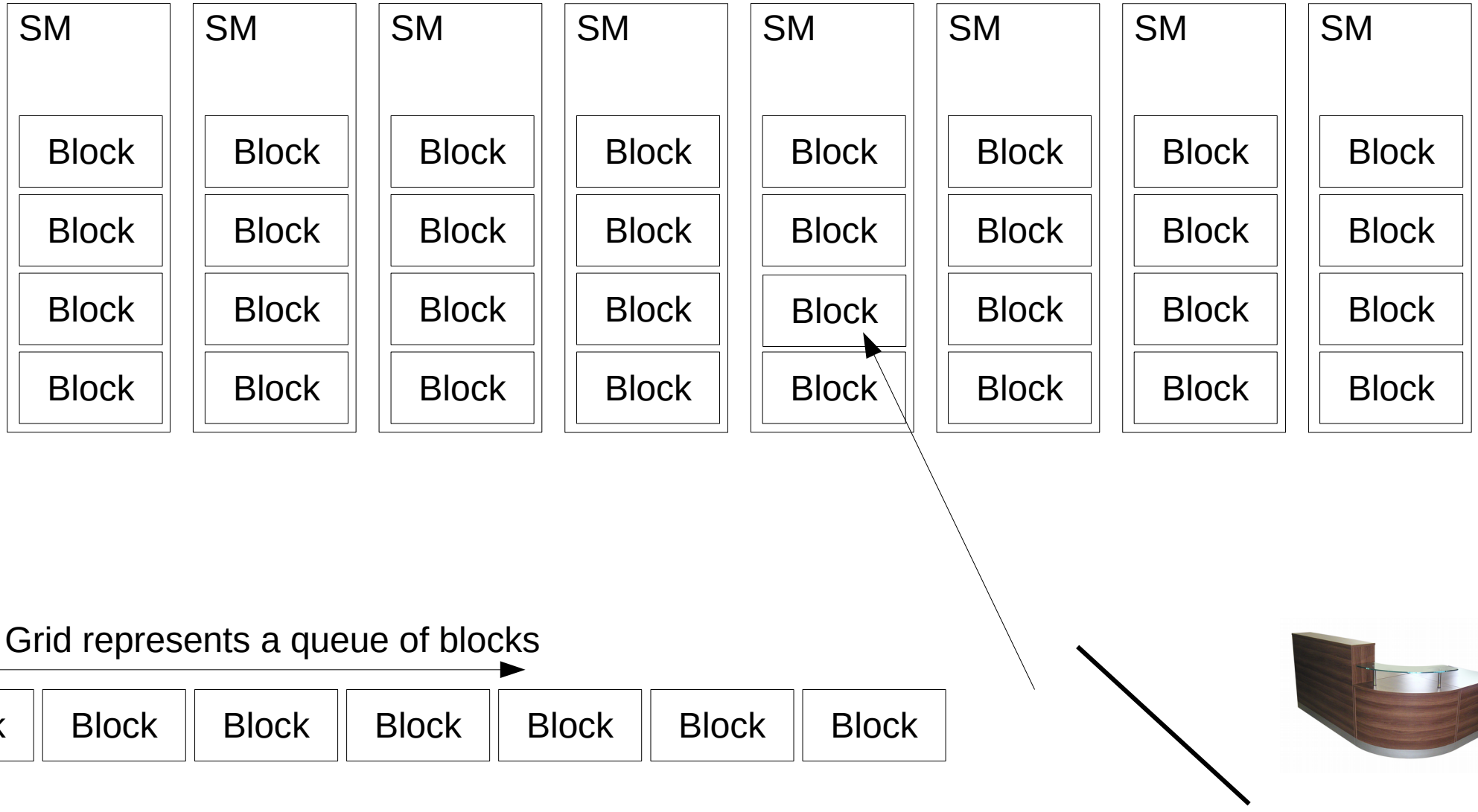
Grid represents a queue of blocks

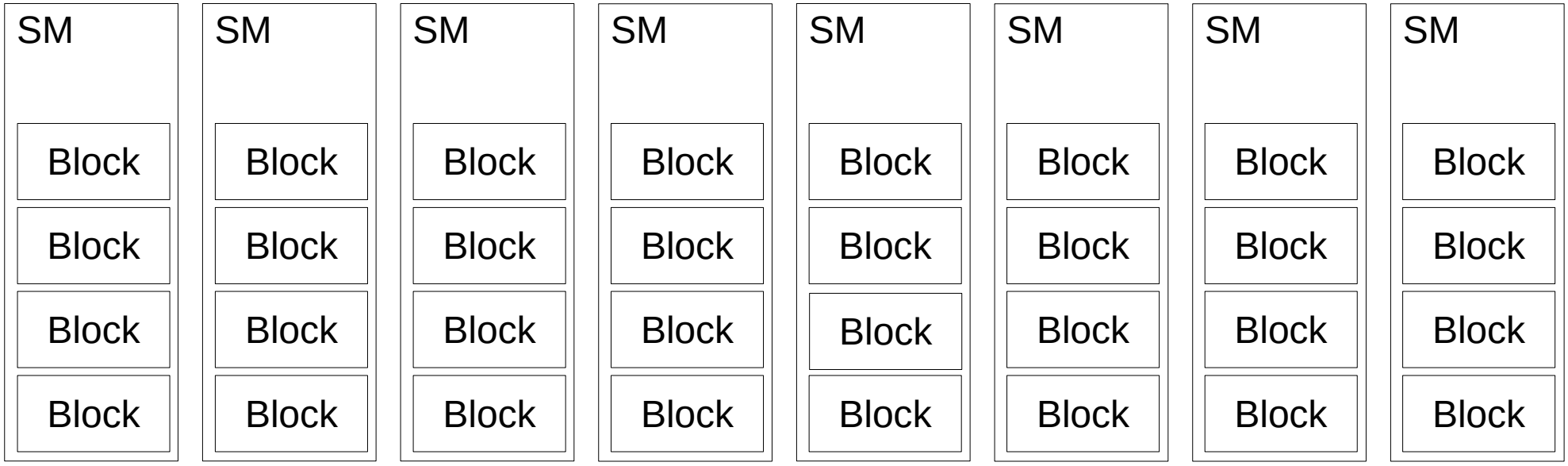




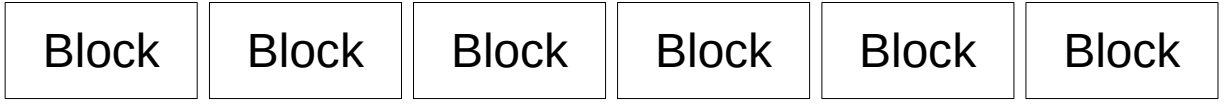
Grid represents a queue of blocks




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



Grid represents a queue of blocks



How many blocks can be active on an SM?

How many blocks can be active on an SM?

Device Limit: 32 (since Maxwell)

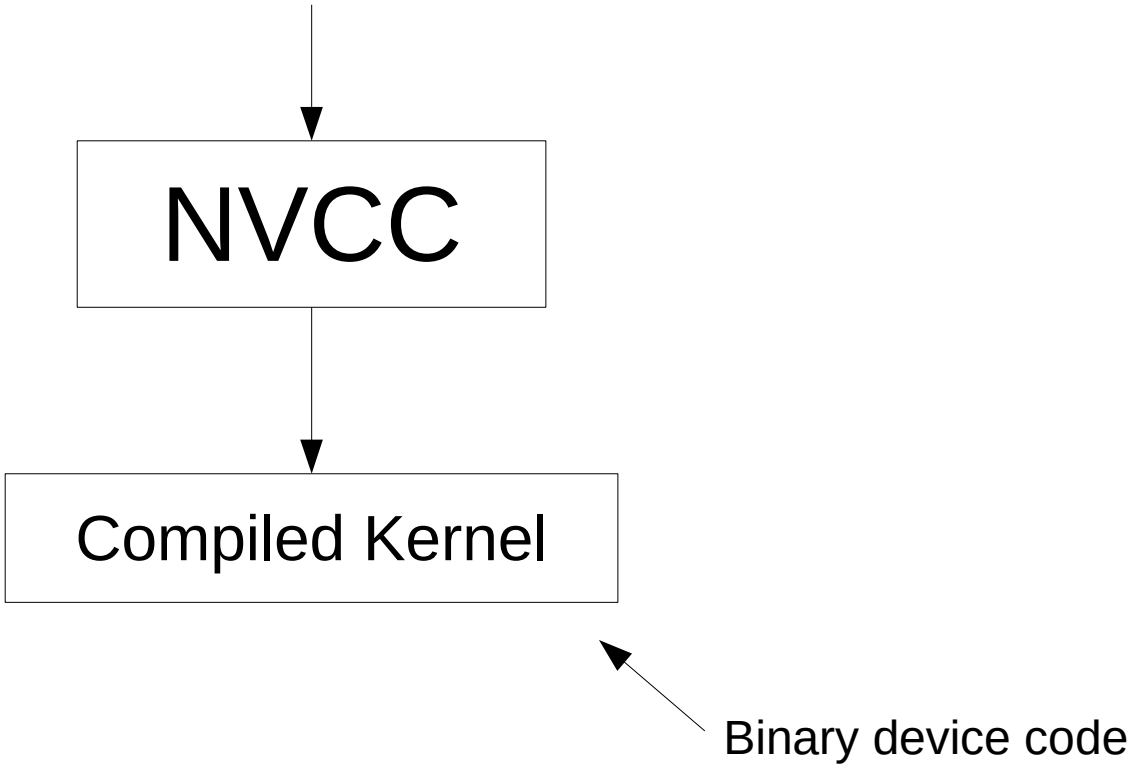
How many blocks can be active on an SM?

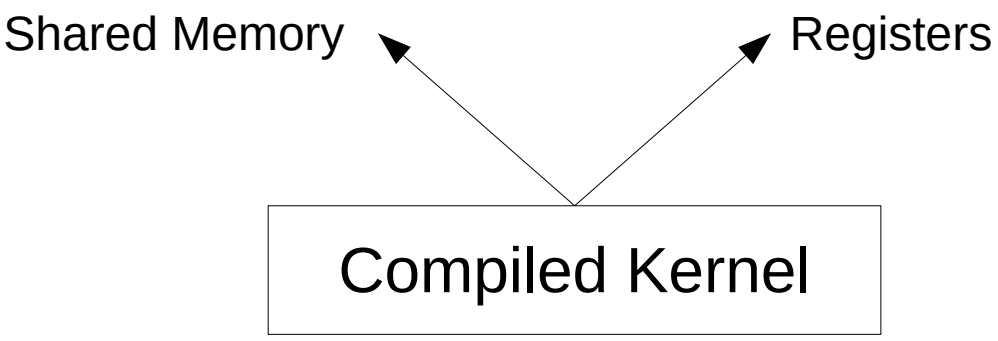
Device Limit: 32 (since Maxwell)

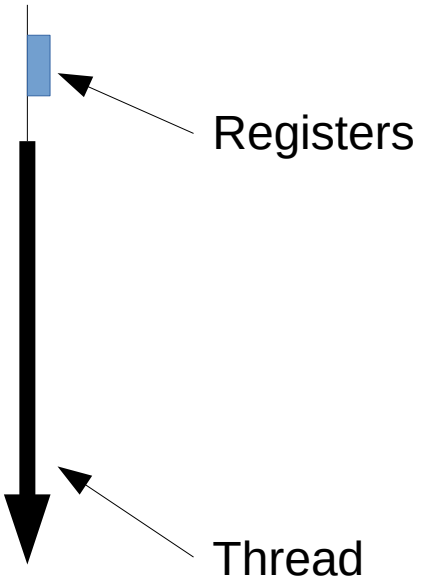
In practice: more nuanced..

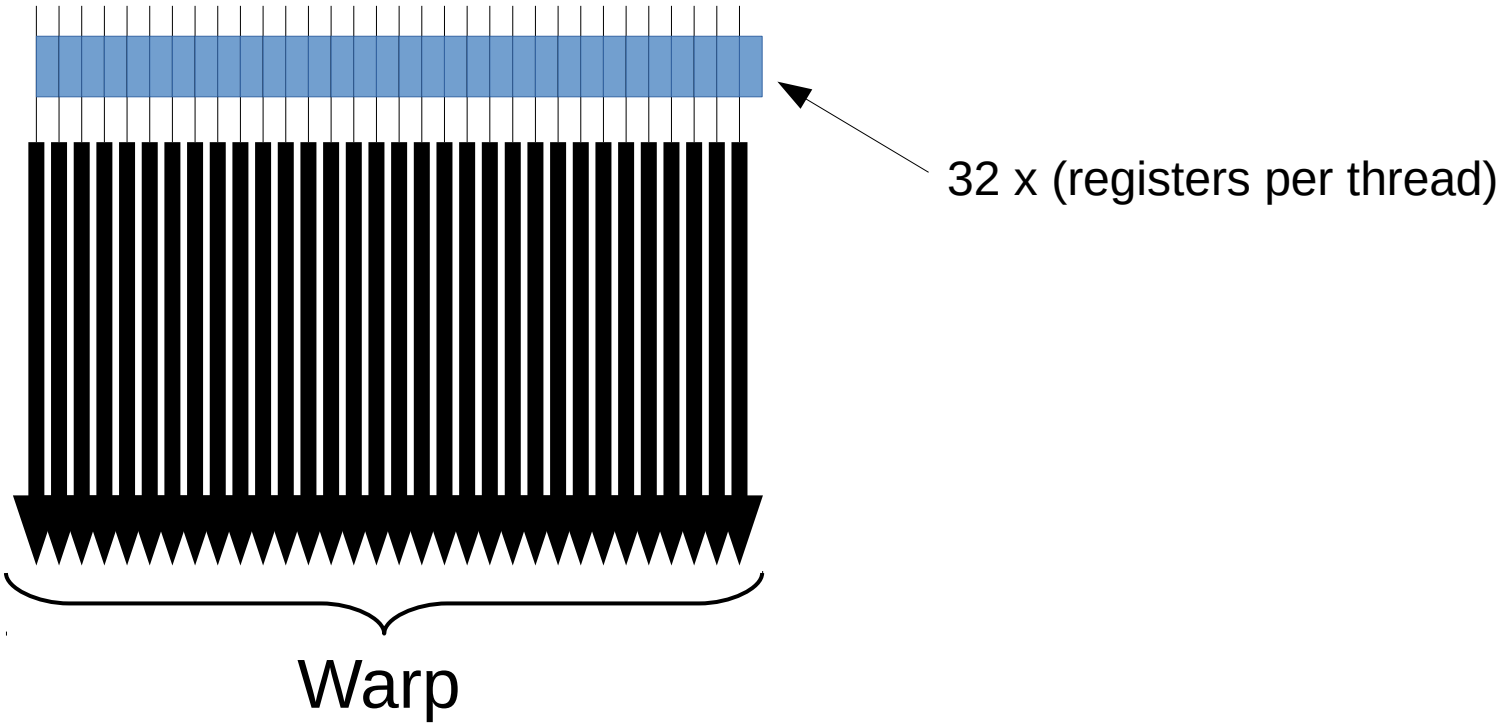
```
__global__ void wasteGPUCycles() {
     for(int i = 0; i < 9; i++) {
          int j = 9;
          j--;
          j = i;
     }
}
```

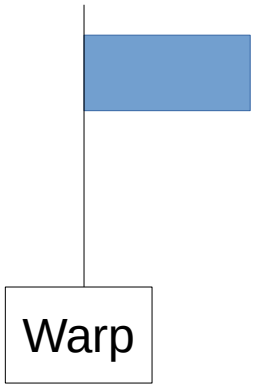
```
__global__ void wasteGPUCycles() {
     for(int i = 0; i < 9; i++) {
          int j = 9;
          j--;
          j = i;
     }
}
```

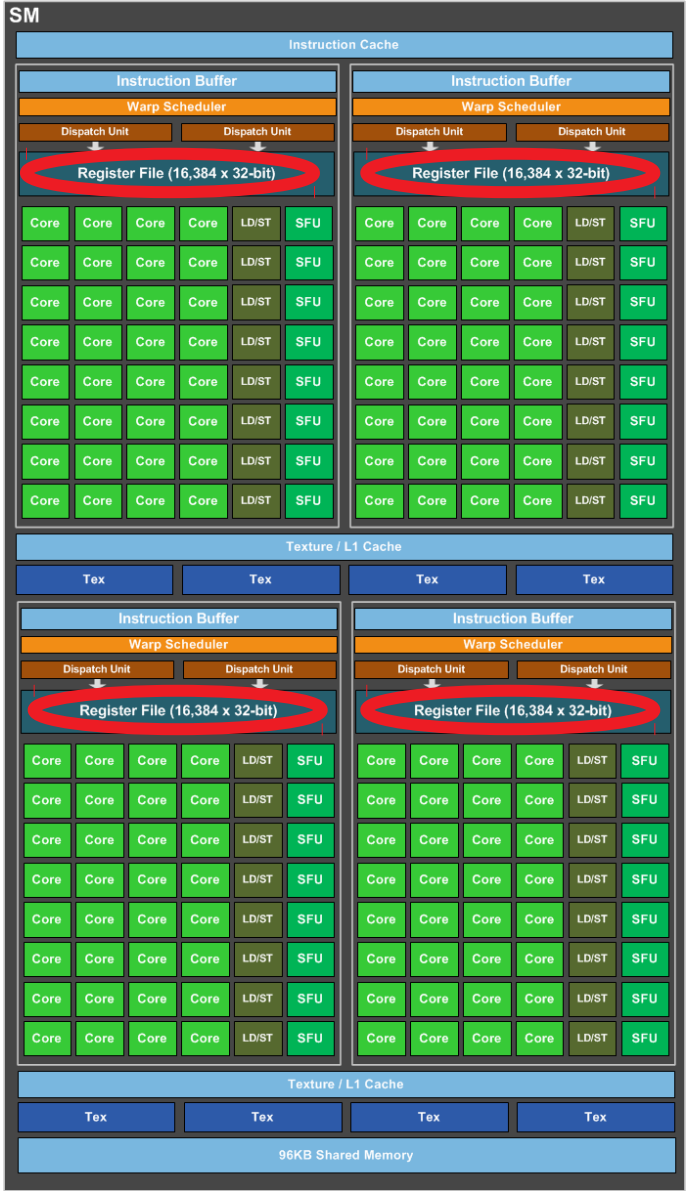



Shared Memory
Registers
Compiled Kernel

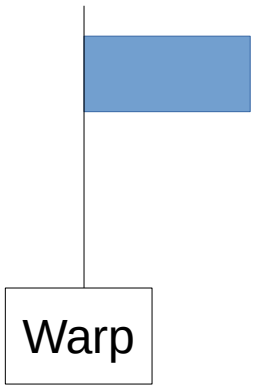

Registers
Thread

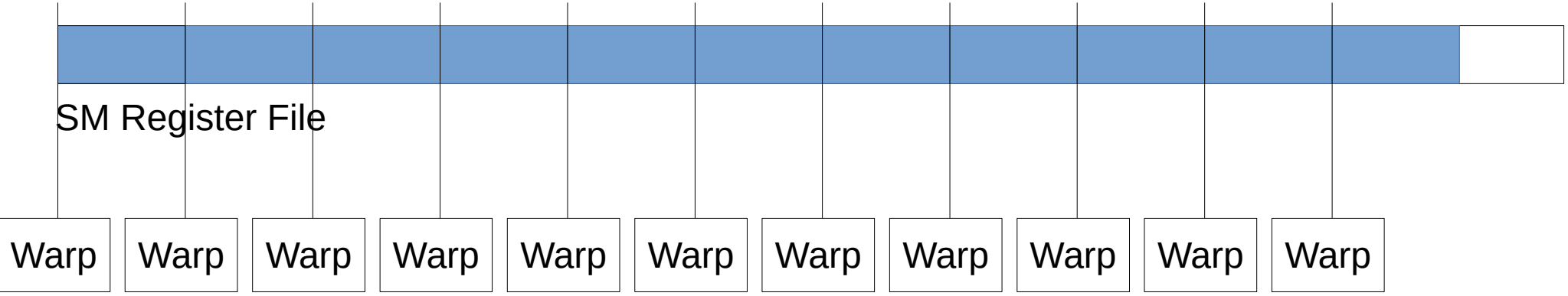

32 x (registers per thread)
Warp


Warp

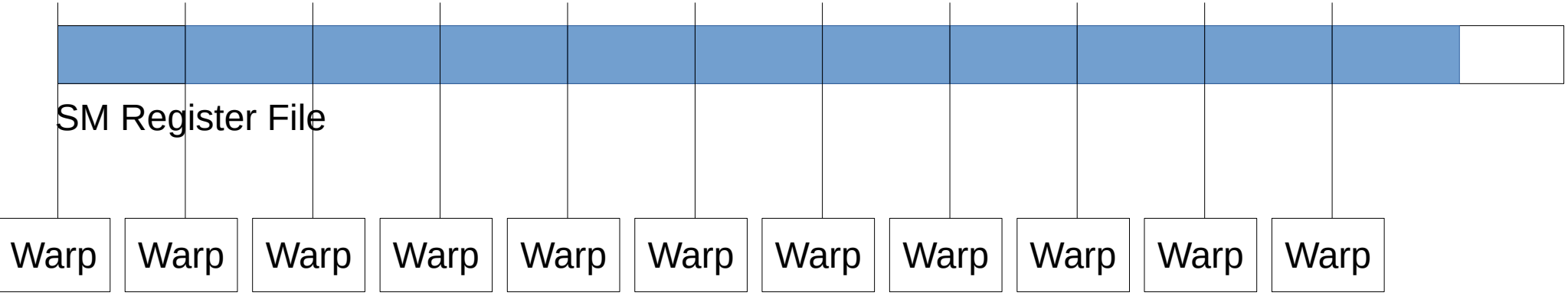

SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory

SM Register File



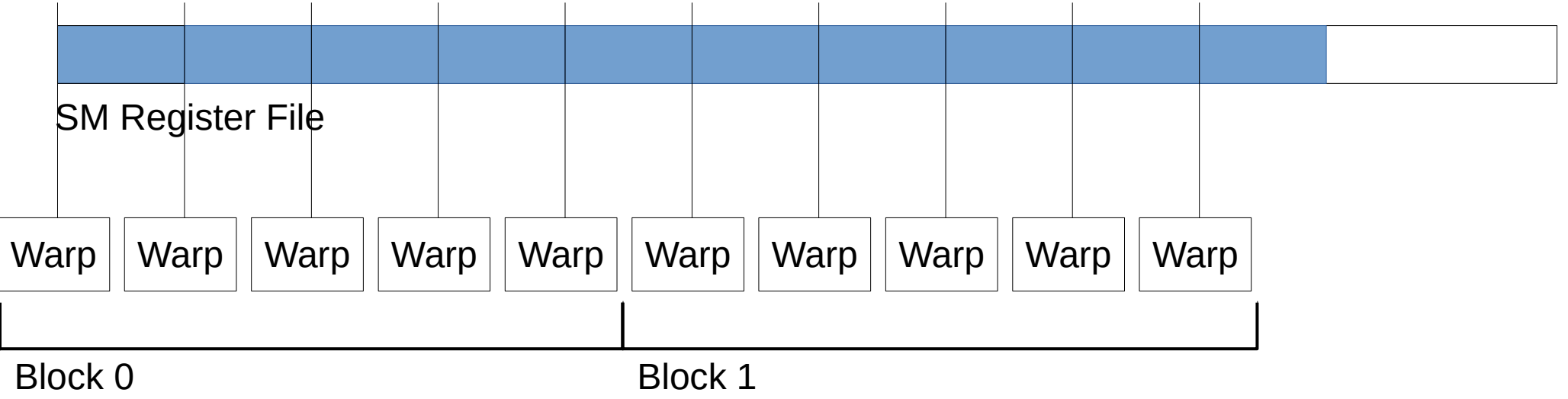


Number of warps that can execute on the SM is dependent on the number of registers needed per thread



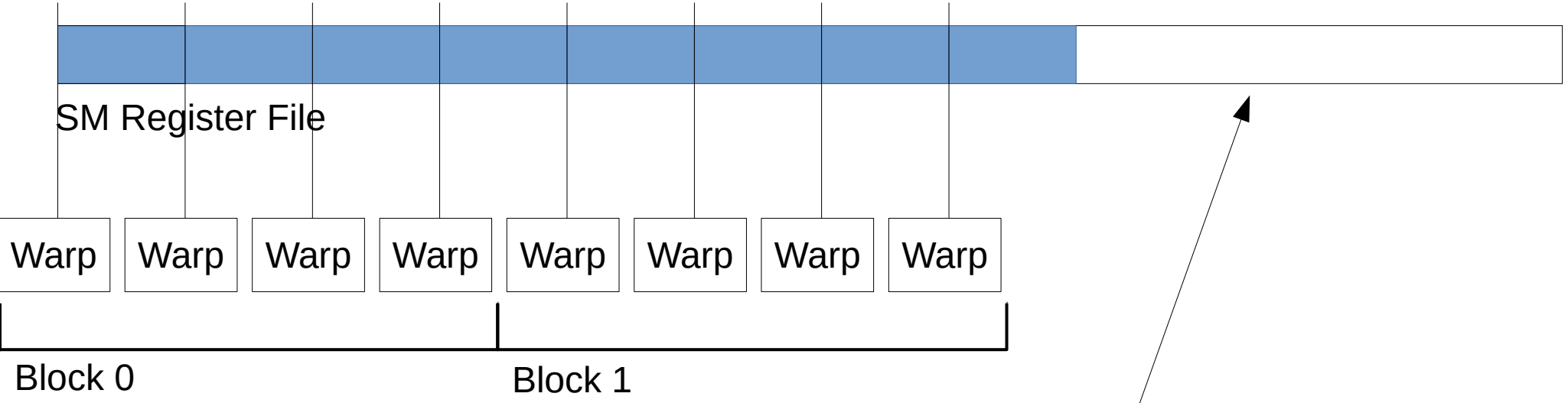
Number of warps that can execute on the SM is dependent on the number of registers needed per thread

But what about blocks?

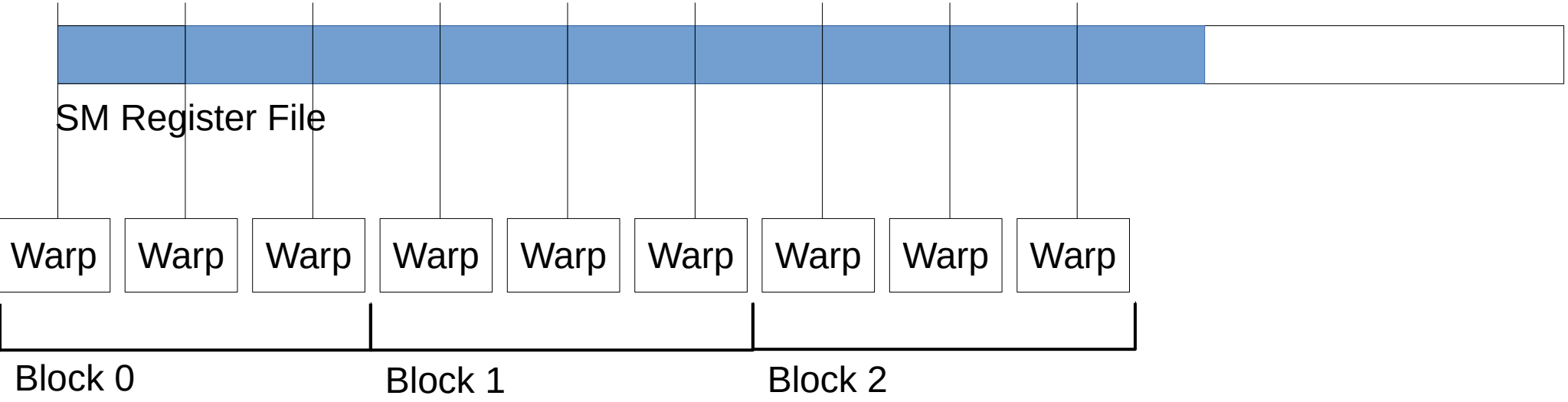


Blocks have a constant number of warps

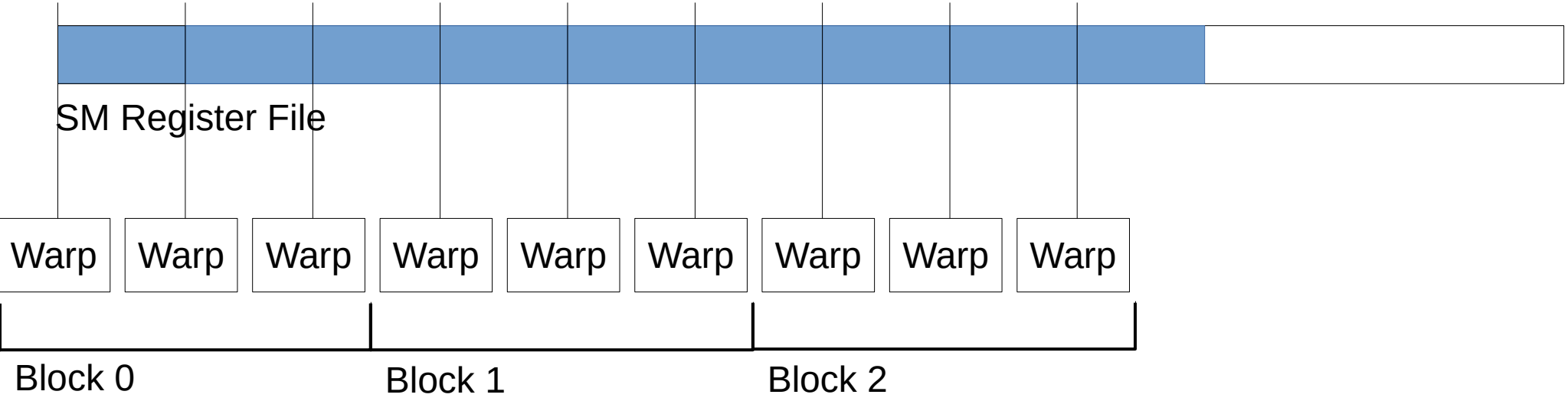
But must be assigned to an SM in full!



Depending on how many registers your threads need, your blocks may waste a lot of register capacity

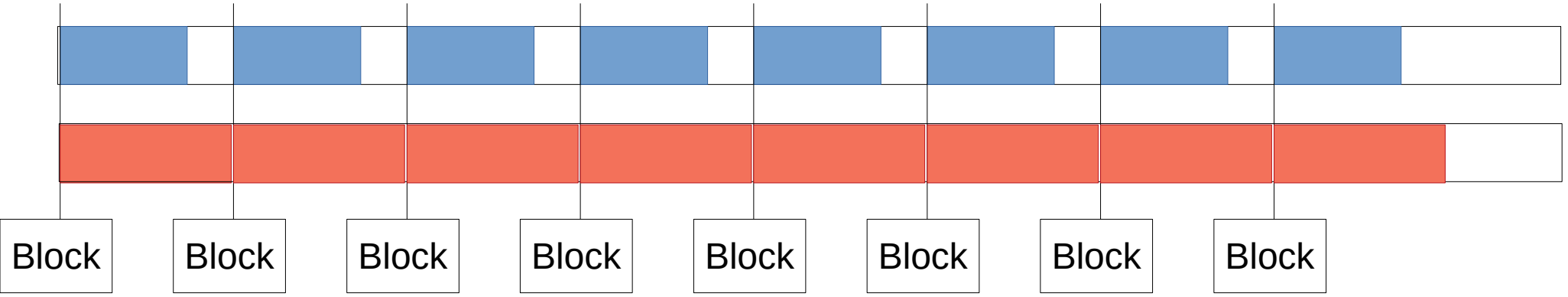


Smaller block sizes may allow more blocks to be executing on an SM simultaneously.



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

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



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

Why is this important?

More blocks allows more simultaneous execution

More simultaneous execution
usually means more throughput

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

Problem: Warps stall. A lot.

They wait, amongst others, for:

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

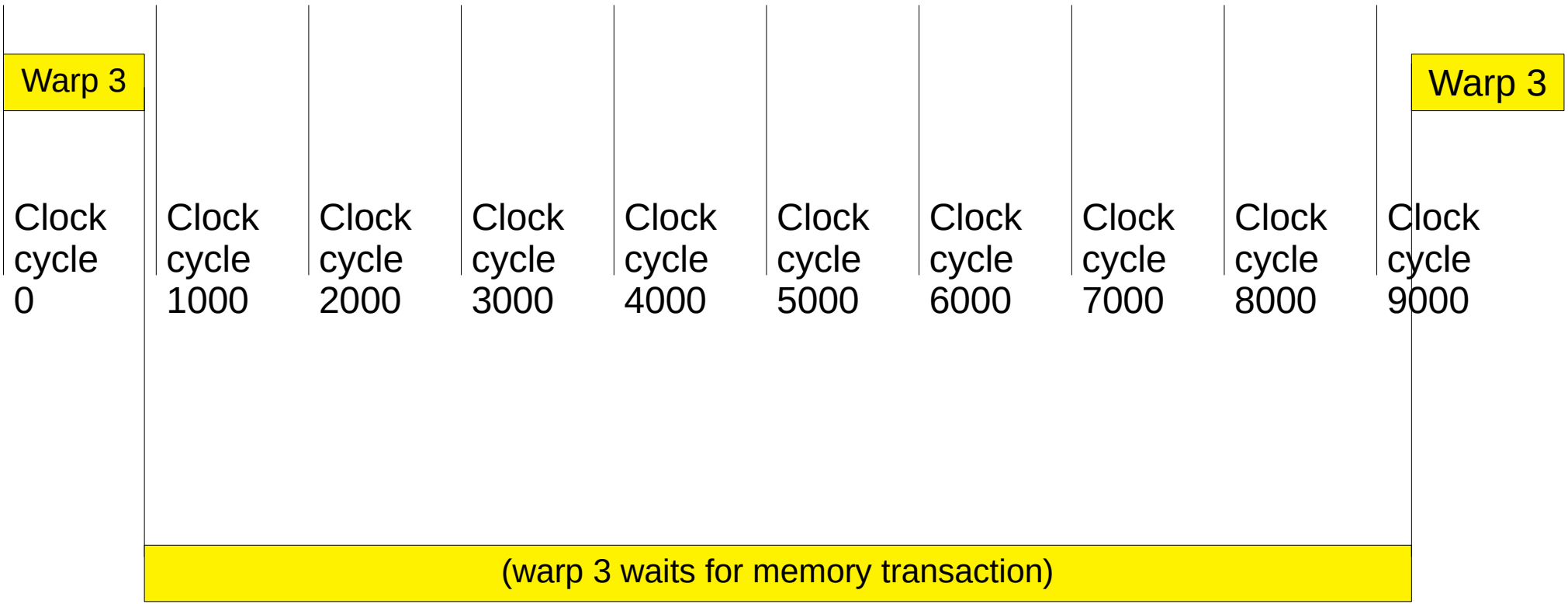
Problem: Warps stall. A lot.

And they can stall a long time...

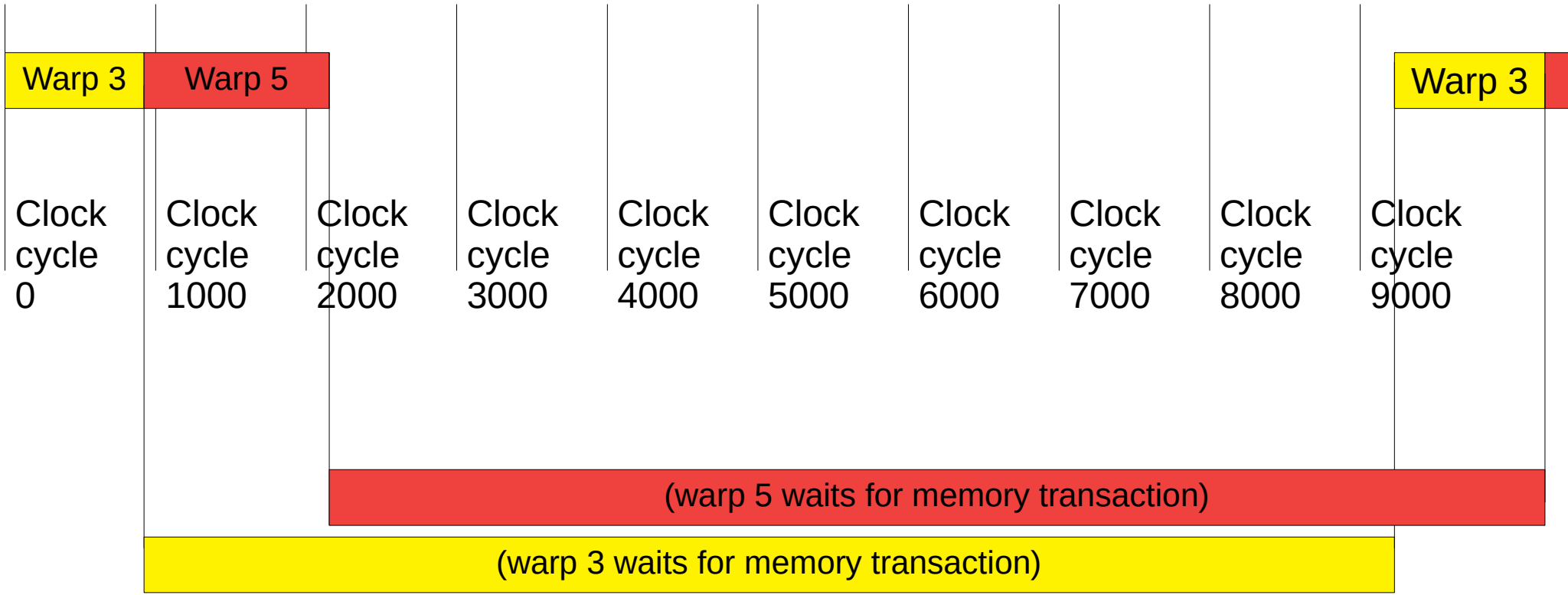
With 1000’s of cores on a die constantly requesting memory, memory transactions have Extremely high latency to ensure high bandwidth.

Problem: Warps stall. A lot.

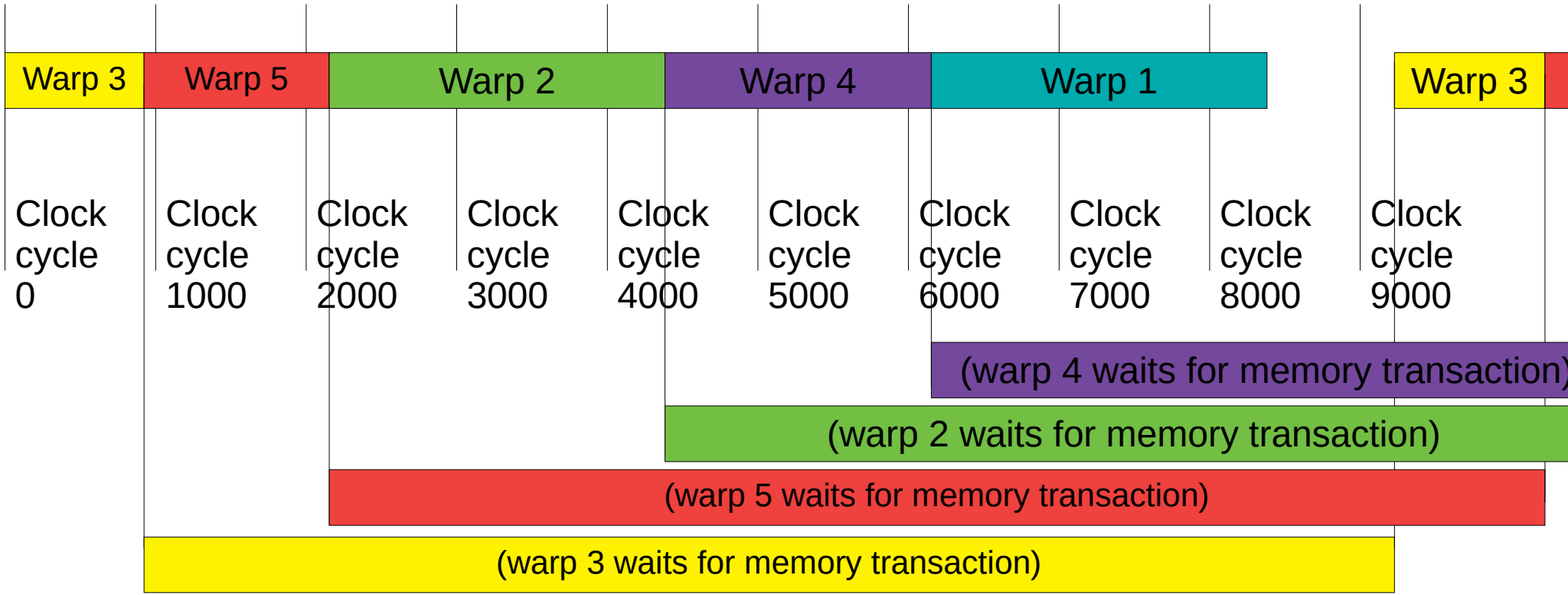
If we were to execute threads like on a CPU:



Solution: Quickly switch to other warps



Solution: Quickly switch to other warps



Solution: Quickly switch to other warps

CPU: Context switch requires dumping and loading of registers

GPU: Context switch takes zero cycles

Simultaneous Multithreading on Steroids!

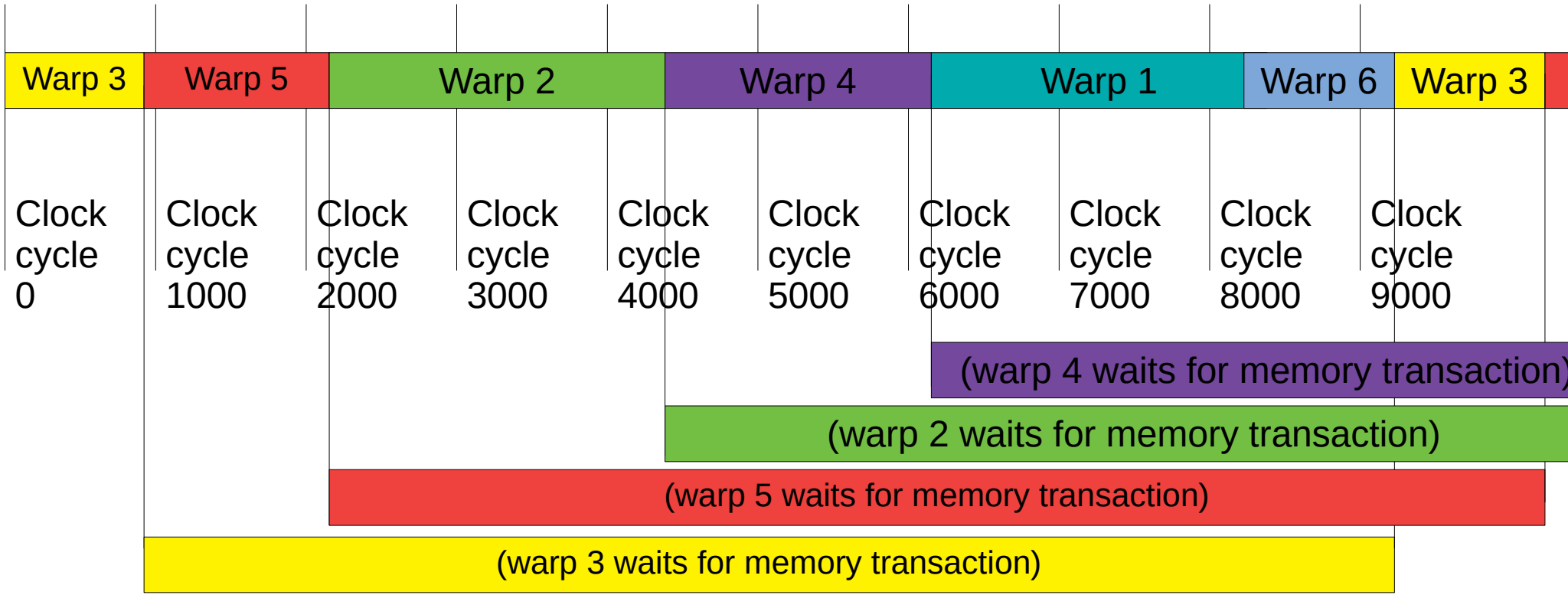
	Dispatch Unit	Dispatch Unit
Clock cycle 0	Warp 3	Warp 1
Clock cycle 1	Warp 5	Warp 3
Clock cycle 2	Warp 2	Warp 3
Clock cycle 3	Warp 5	Warp 4
Clock cycle 4	Warp 3	Warp 1
Clock cycle 5	Warp 5	Warp 4
Clock cycle 6	Warp 4	Warp 2
Clock cycle 7	Warp 2	Warp 1

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).

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



Back to our original objective:

What can make warps run slow?

Non-Coalesced Accesses
Thread Divergence
Register Spilling
Resource Contention
Synchronization

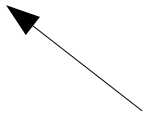
And many more..

1. Non-Coalesced Memory Requests

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

1. Non-Coalesced Memory Requests

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

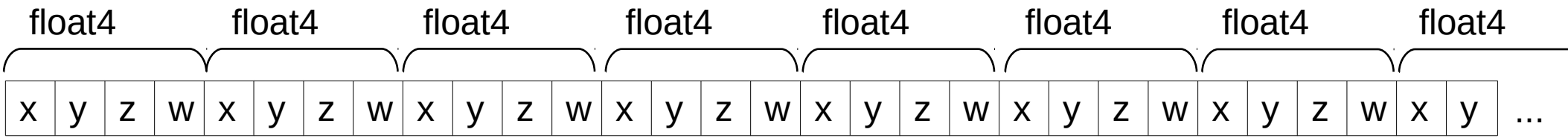


```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

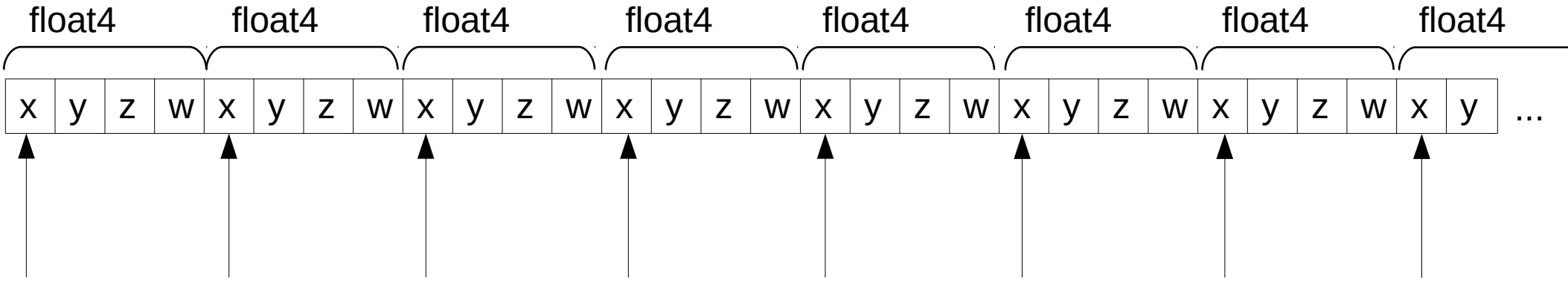
1. Non-Coalesced Memory Requests

```
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;
```

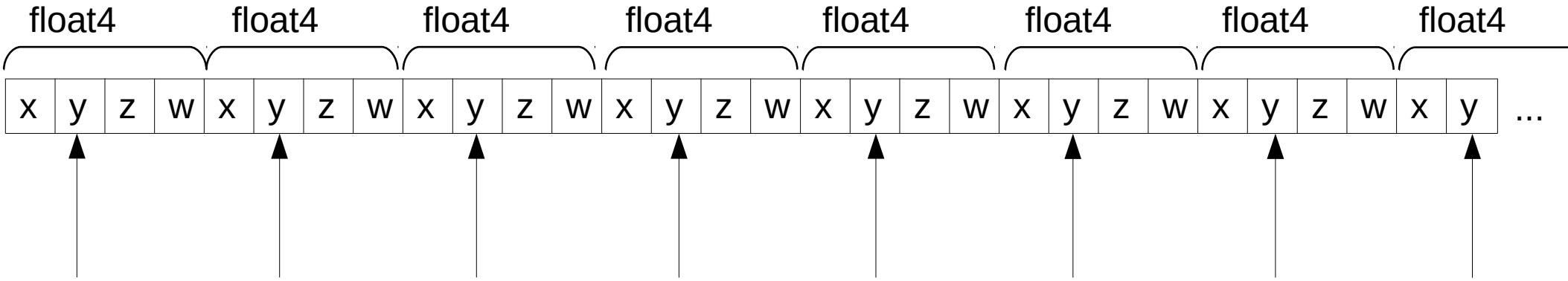


1. Non-Coalesced Memory Requests



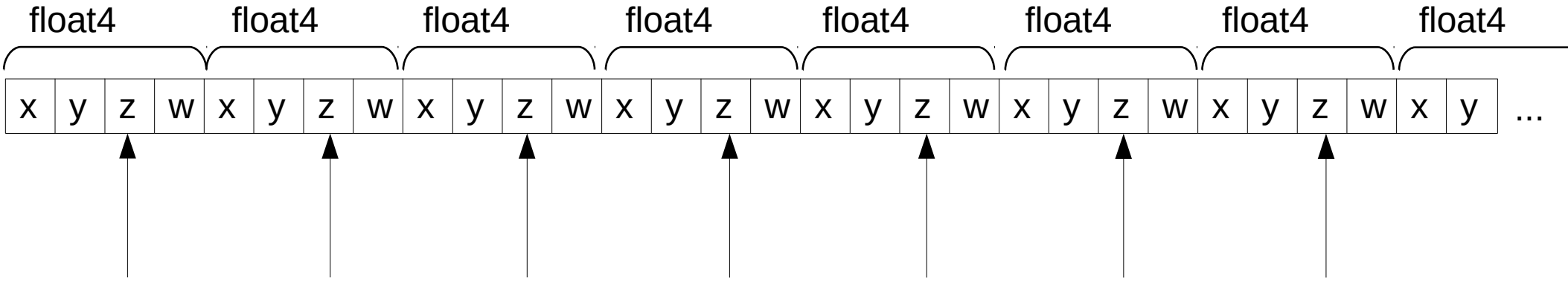

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

1. Non-Coalesced Memory Requests




```
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




```
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

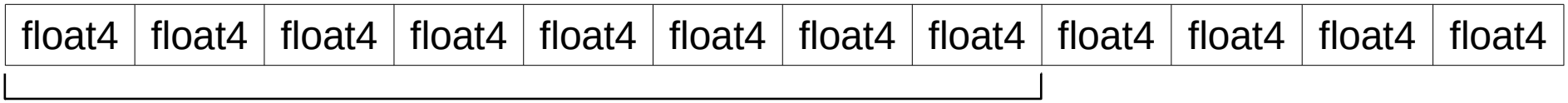
Main problem:

Cache Lines

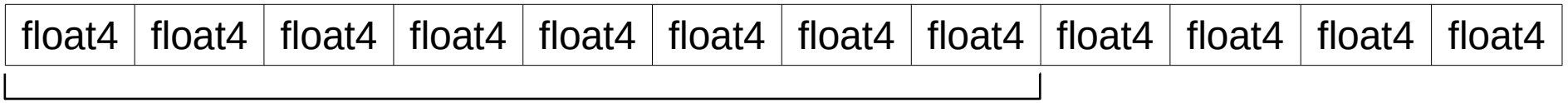
1. Non-Coalesced Memory Requests

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

1. Non-Coalesced Memory Requests



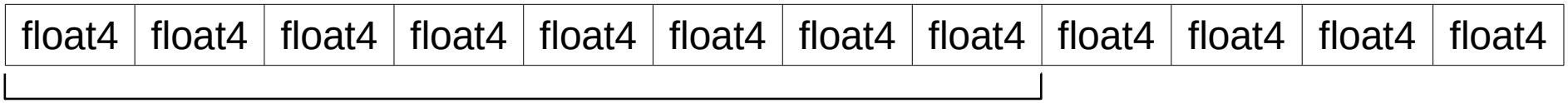
1. Non-Coalesced Memory Requests



Here’s the real kicker:

- All memory transactions are done in terms of cache lines

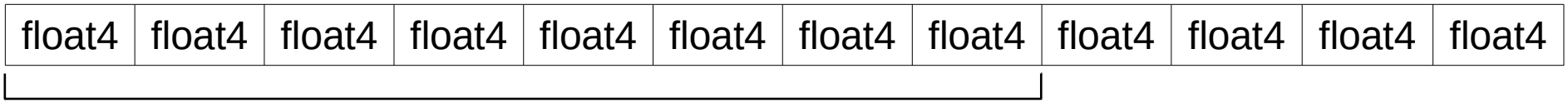
1. Non-Coalesced Memory Requests



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



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

1. Non-Coalesced Memory Requests

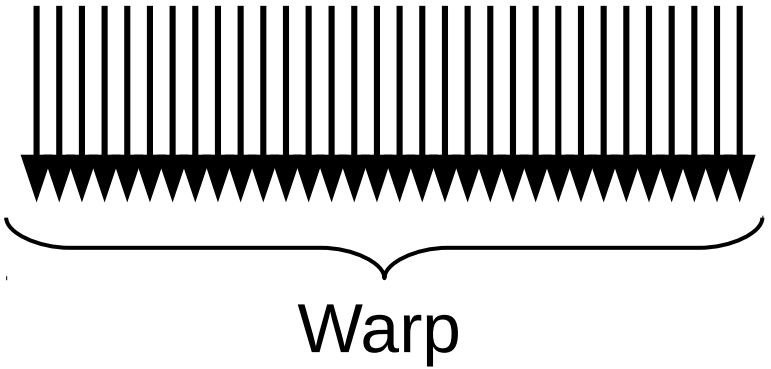
Mitigation: always ensure memory requests are within the same cache line

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

2. Thread Divergence

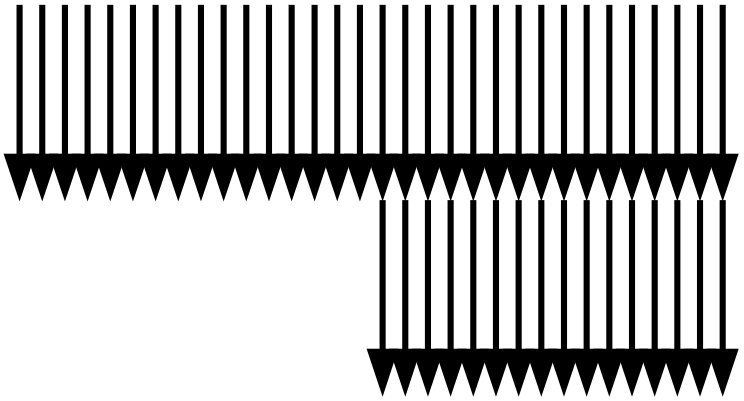
```
if(threadIndex > 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

2. Thread Divergence



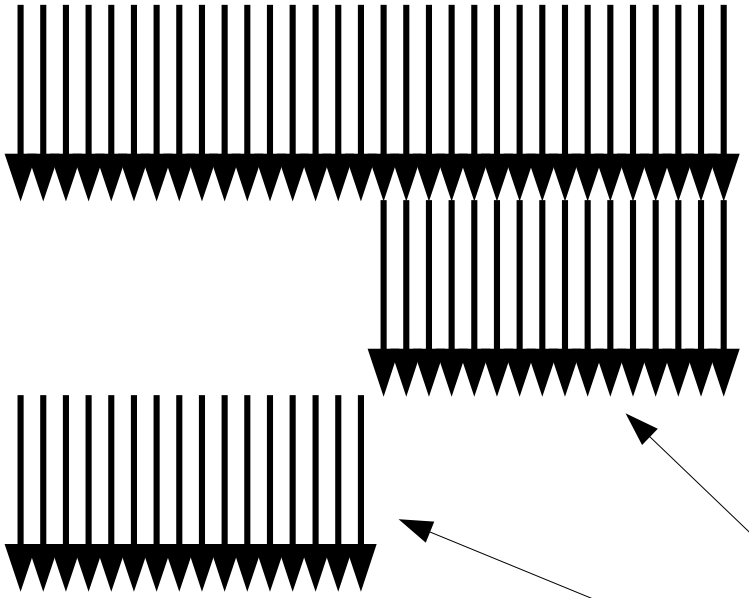

```
if(threadIndex >= 16) {
    doSomething();
} else {
    doSomethingElse();
}
```

2. Thread Divergence



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

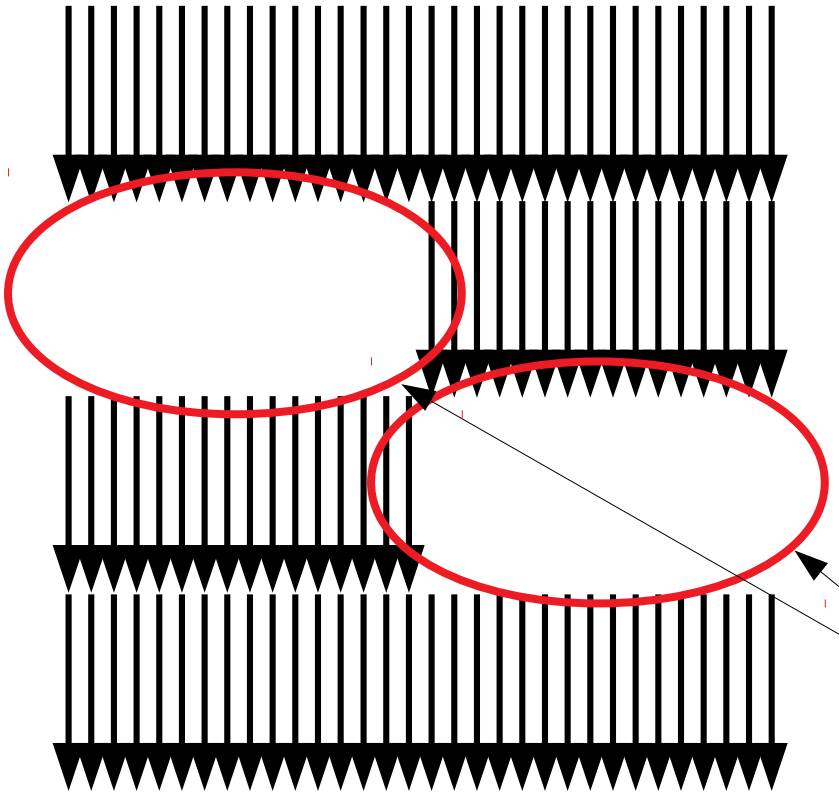
2. Thread Divergence



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

BOTH clauses need to be executed in full

2. Thread Divergence



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

While half of the threads are doing nothing!

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++) {

}
```

2. Thread Divergence

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

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.

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

.. And others

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

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

Part 2/3:

How do we measure GPU performance?

Occupancy

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

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

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

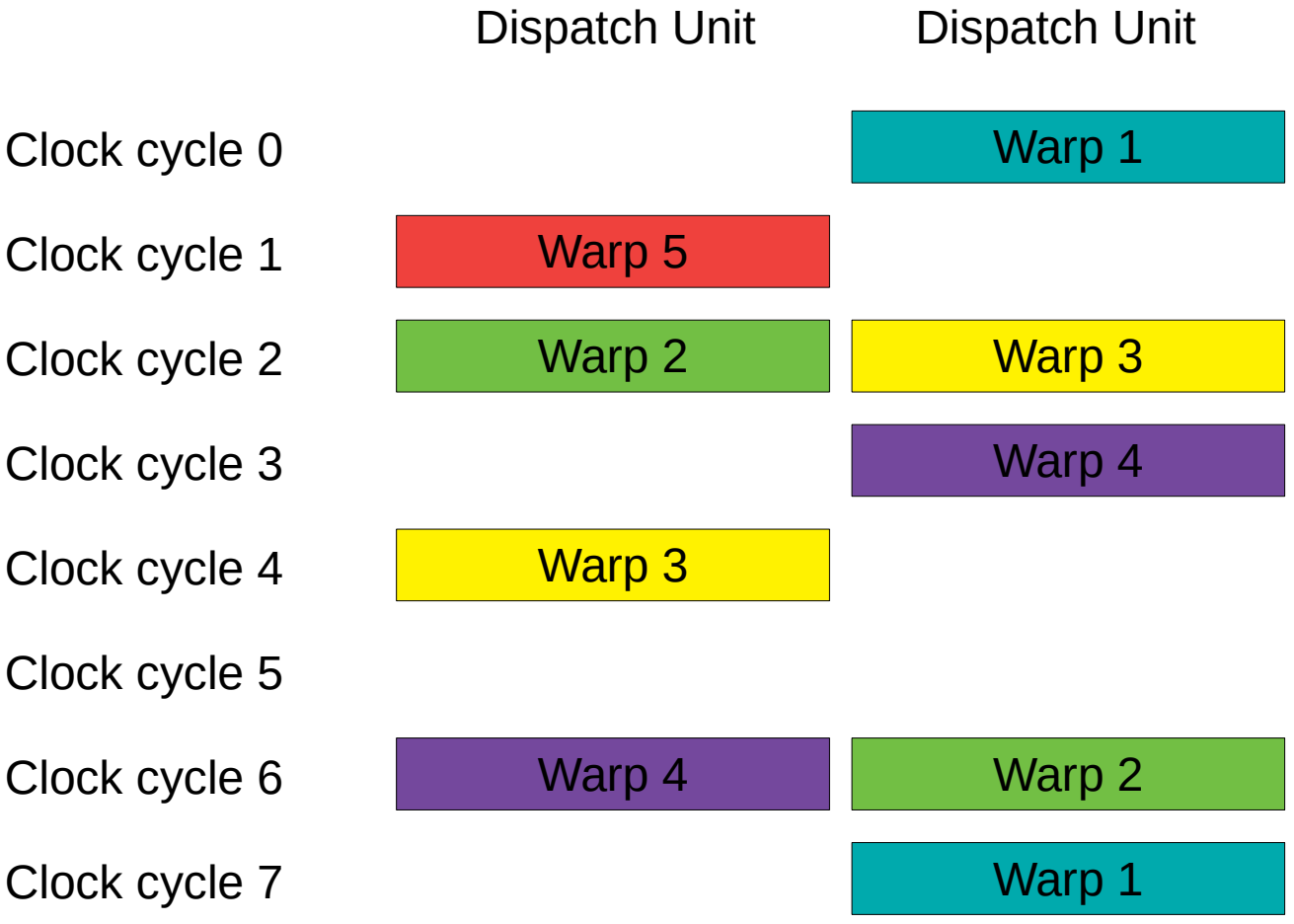
Determined by the hardware limit

Occupancy

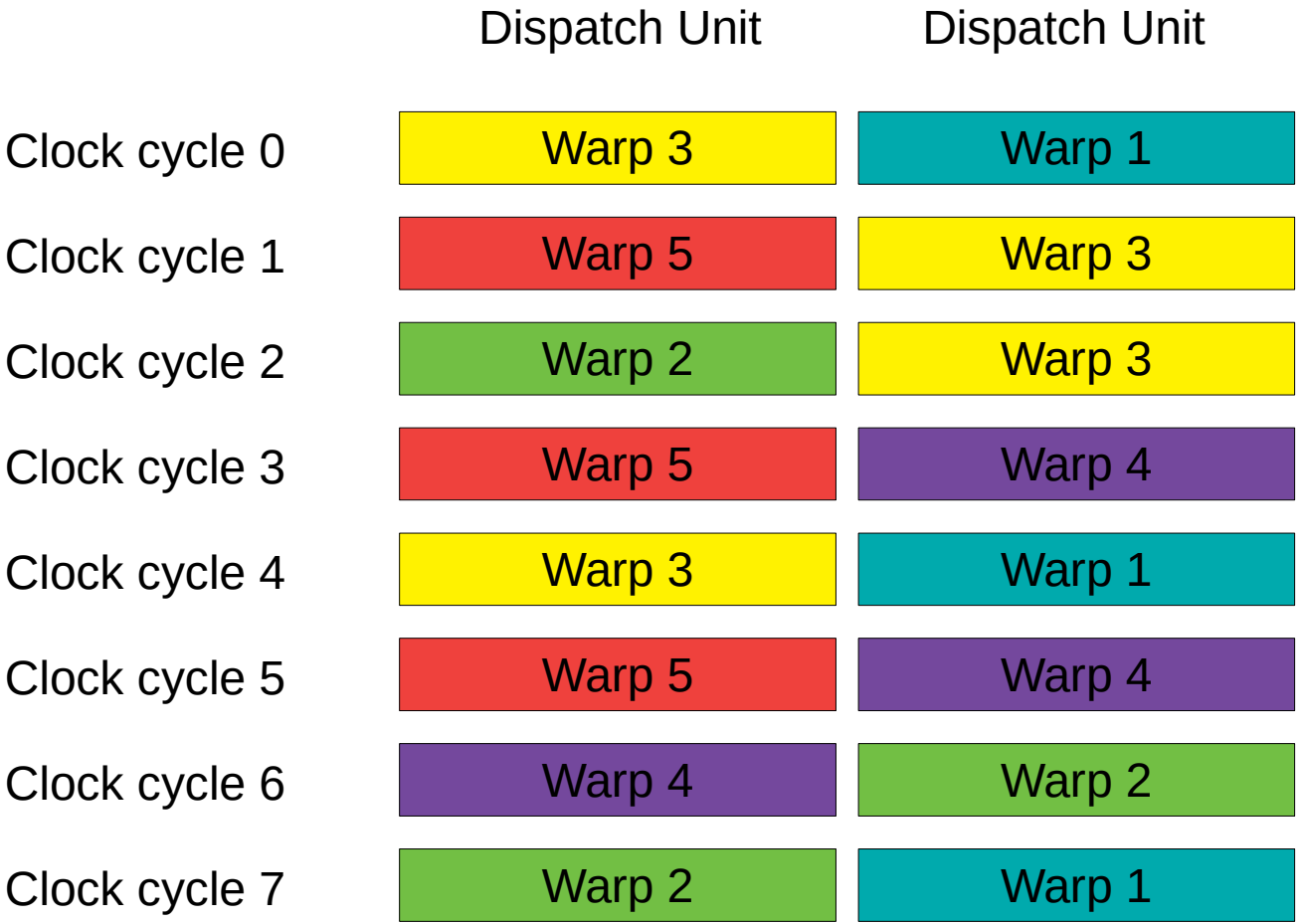
Can be limited by:

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

Bad occupancy



Good occupancy



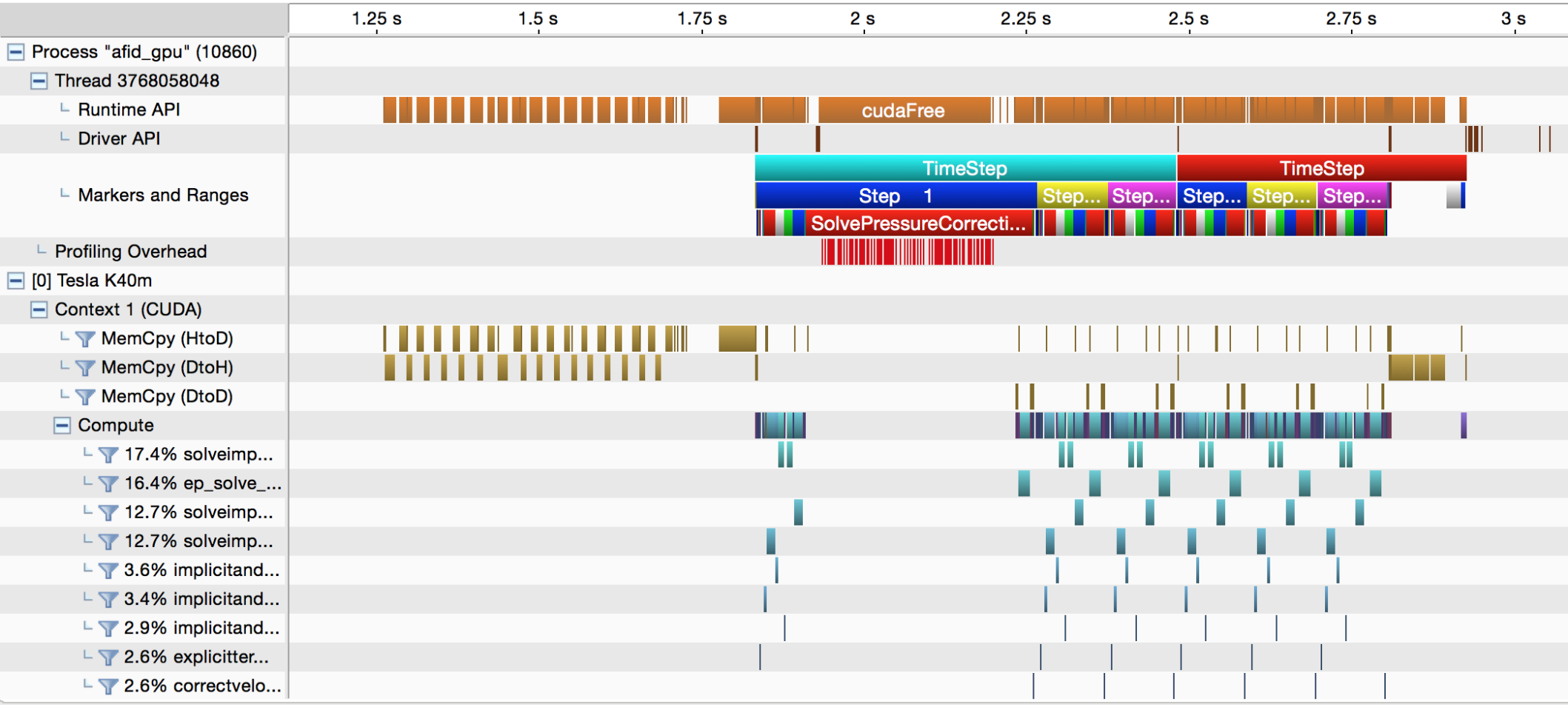
Occupancy

Avoid low occupancy

Best performance is not
necessarily 100% occupancy!

Only need enough to saturate the memory bus.

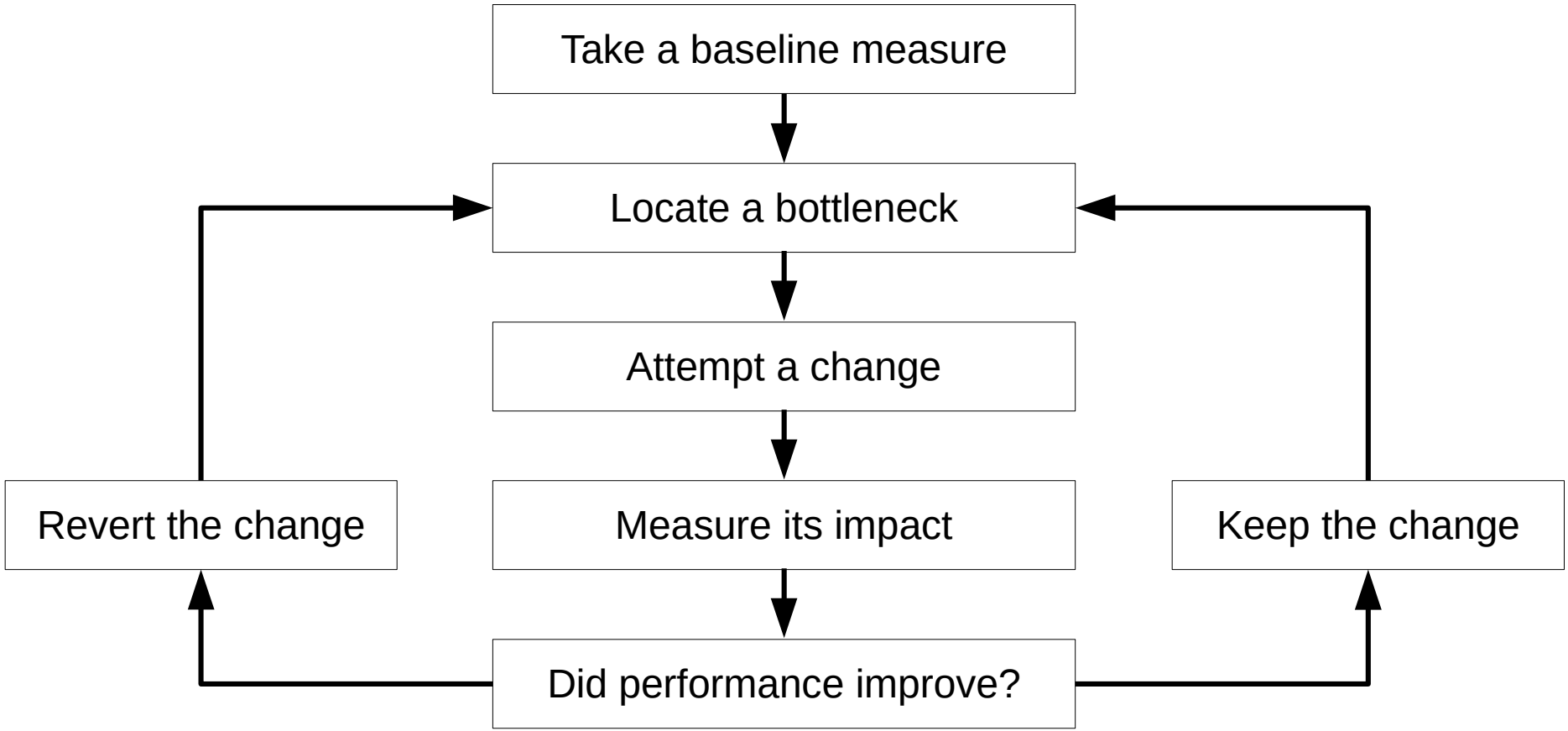
Demonstration!



Part 3/3:

What can we do to make code run faster?

Remember the optimisation cycle!



GPU Optimisation is a matter of balance

Thread Complexity

Instruction Throughput

Memory Bandwidth

10 things you can use to make your code run faster

(+ choosing proper launch parameters)

(+ ensuring memory accesses are coalesced)

(+ minimising thread divergence)

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

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;
}
```

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];
    }
}
```

2: Use Shared Memory

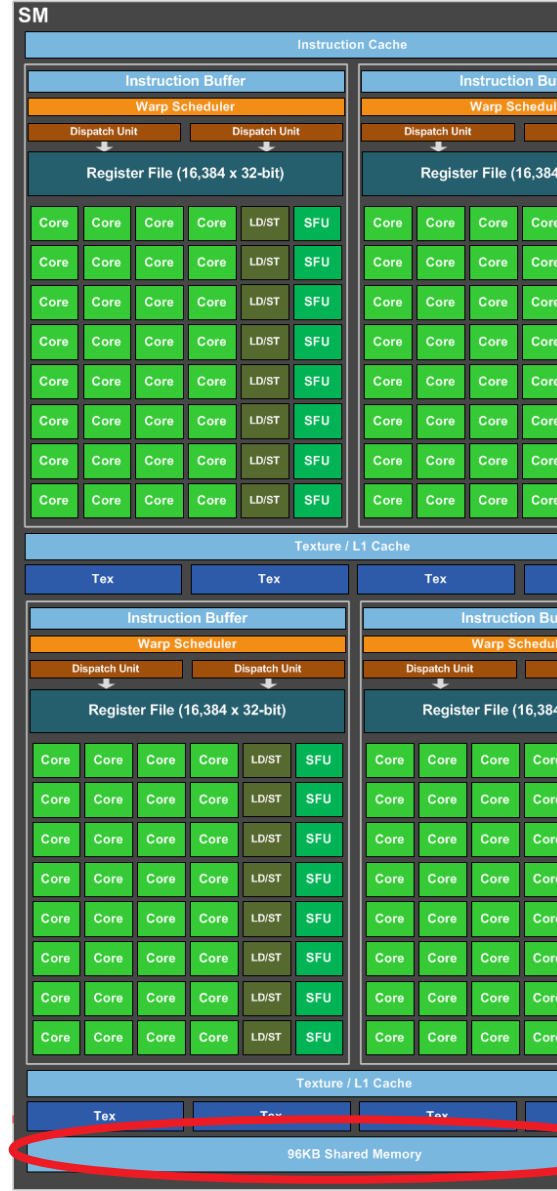
Programmable cache, much faster than global memory

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



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;
}
```

2: Use Shared Memory

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

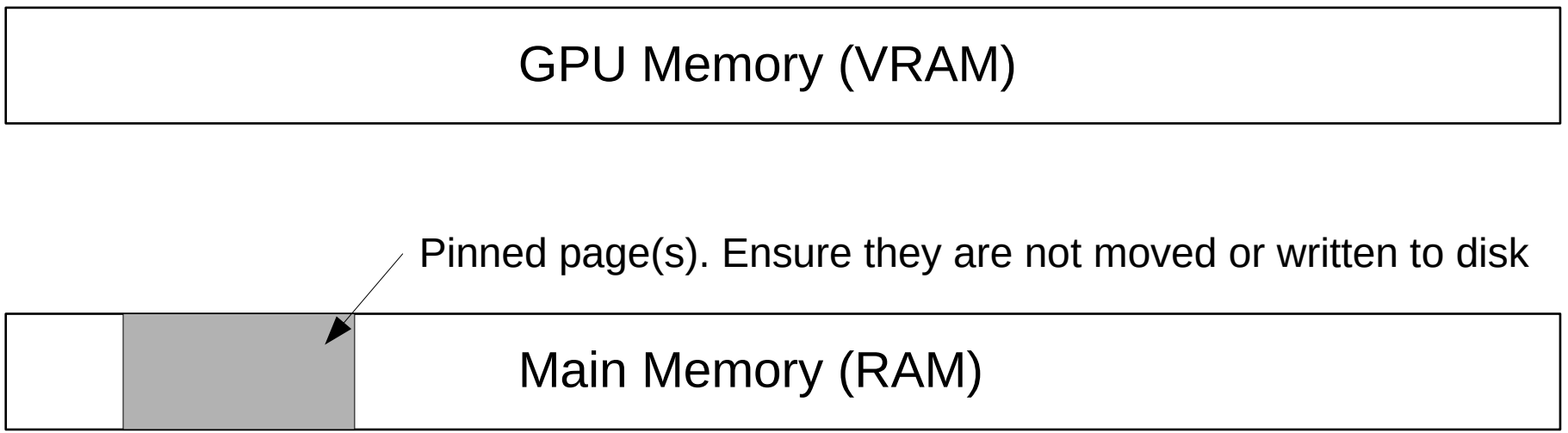
Make sure requests to this memory are as much coalesced as possible!

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

3: Stream memory from RAM

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

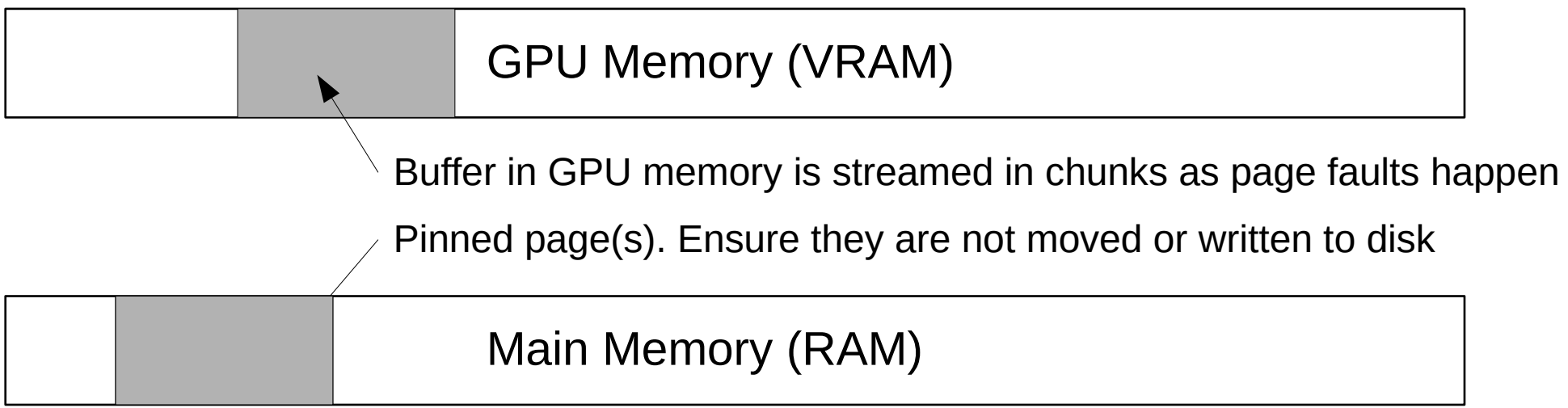
Can use “unified memory” to stream memory back and forth, *while the kernel is executing*



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!



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;
}
```

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.

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;
}
```

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.

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;
}
```

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)

```
#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;
}
```

```cpp
#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;
}
```

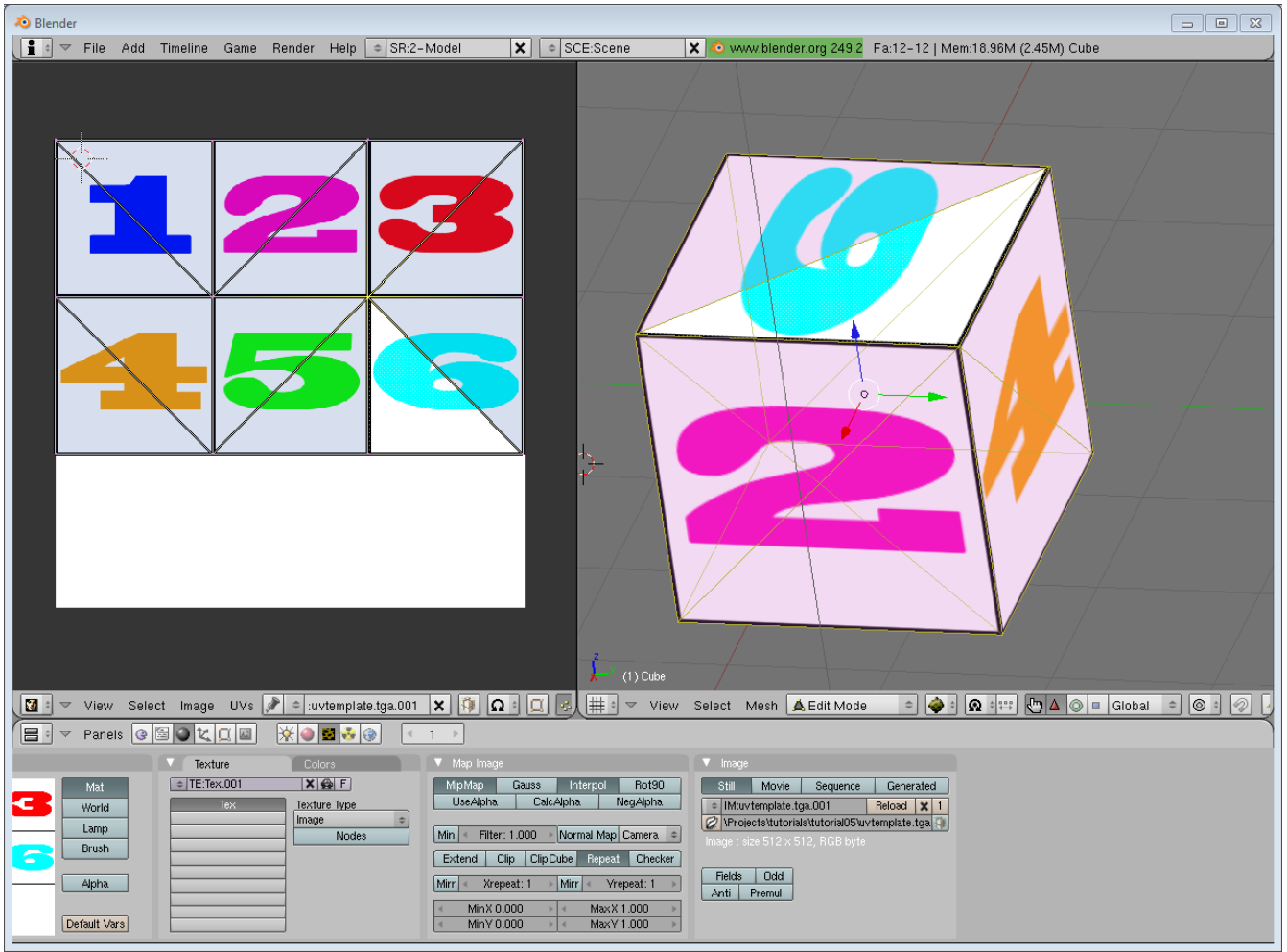
```
14 -> 0
15 -> 5
16 -> 10
17 -> 15
18 -> 20
19 -> 25
20 -> 30
7 -> 100
8 -> 105
9 -> 110
10 -> 115
11 -> 120
12 -> 125
13 -> 130
0 -> 200
1 -> 205
2 -> 210
3 -> 215
4 -> 220
5 -> 225
6 -> 230
```

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.

8: Use Texture Units



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.

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()

Next week: