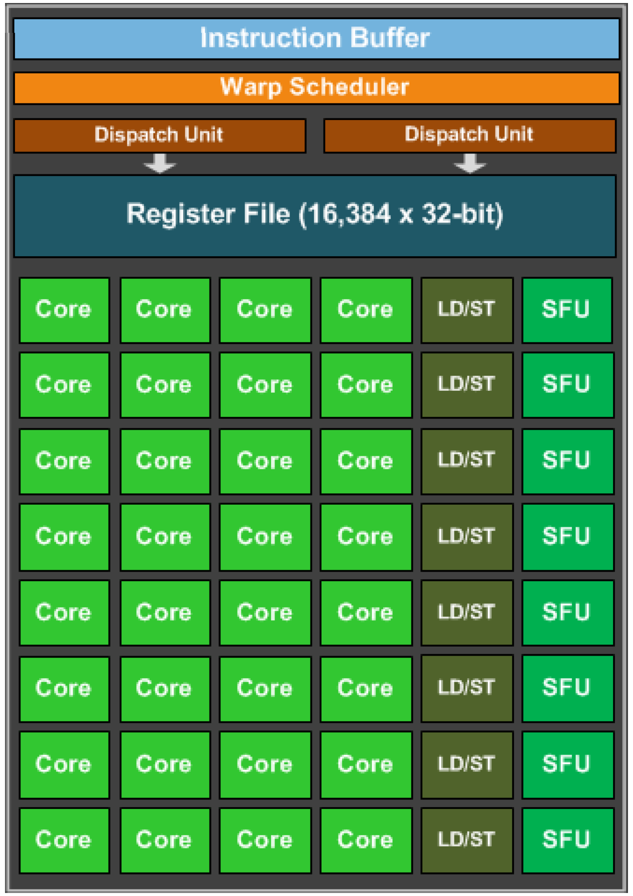
CSE113: Parallel Programming

March 17, 2023

- **Topics**:
  - Wrapping up GPUs
  - WebGPU
  - End of class



Announcements

- HW 5 is out, please get started

Announcements

- Final:
  - All day March 22 (8 AM to 8 PM)
  - Support given between 4 pm and 7 pm.
  - Ask Private Post on Piazza
  - Open note, open slides, open book
  - Open internet to an extent
    - Do not google exact answers
    - Do not ask questions on forums
    - Do not use ChatGPT or other AI tools

Previous Quiz

No previous quiz (sorry!)

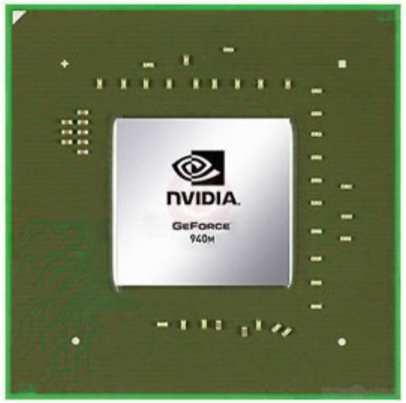
Previous Quiz

Which type of GPU will you be using for HW 5?

Nvidia ✓	13 respondents	**39 %**
Intel	15 respondents	45 %
AMD	2 respondents	6 %
Apple	7 respondents	21 %

Programming a GPU

Tiny GPU in an
embedded system



Fight!

The CPU in
my professor
workstation

Nvidia Jetson Nano (whole chip, CPU + GPU)
2 Billion transistors
10 TDP
Est. $99

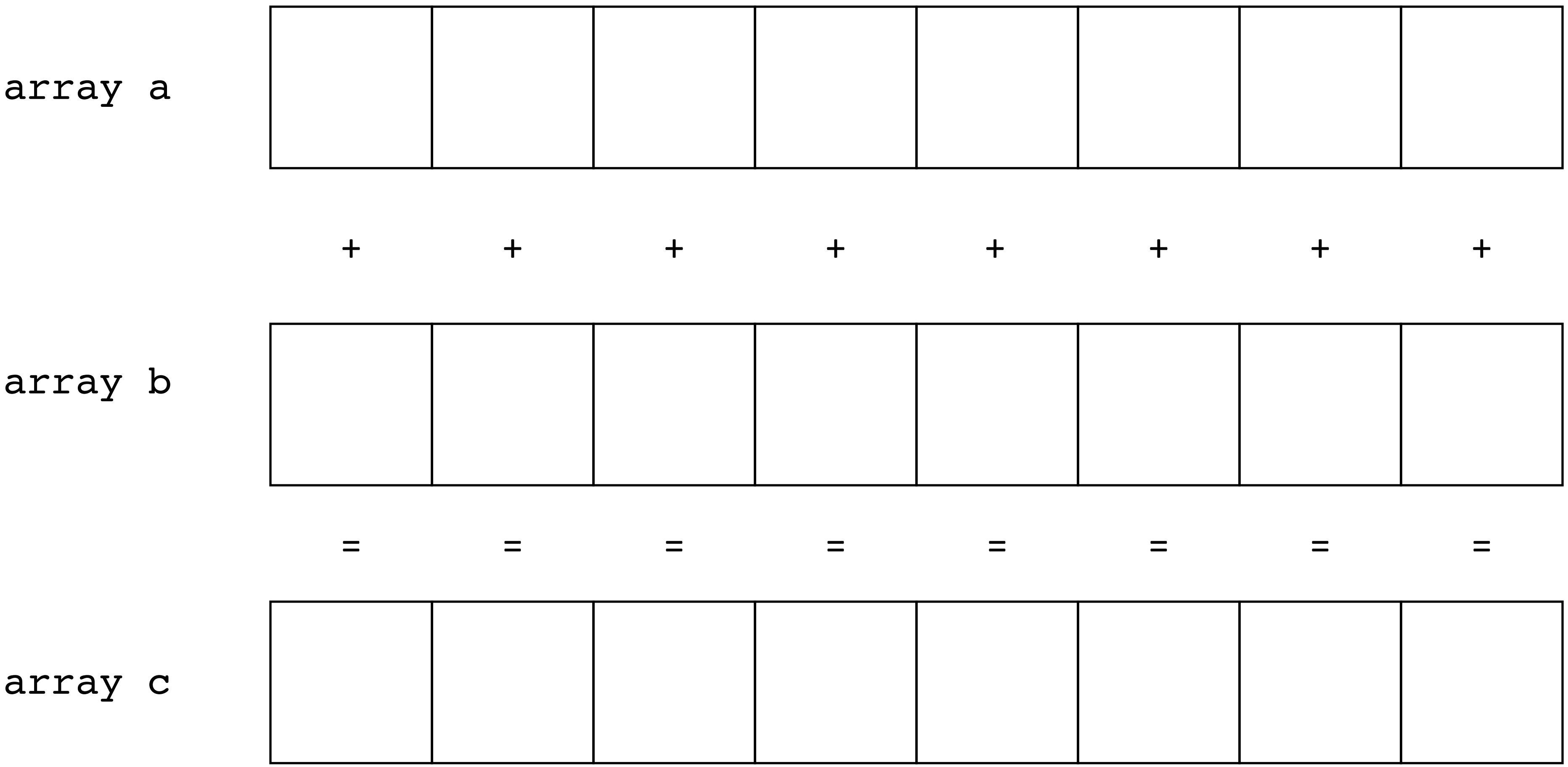
Intel i7-9700K
2.16 Billion transistors
95 TDP
Est. $316

https://www.techpowerup.com/gpu-specs/geforce-940m.c2648
https://www.alibaba.com/product-detail/Intel-Core-i7-9700K-8-Cores_62512430487.html
https://www.prolast.com/prolast-elevated-boxing-rings-22-x-22/

Embarrassingly parallel

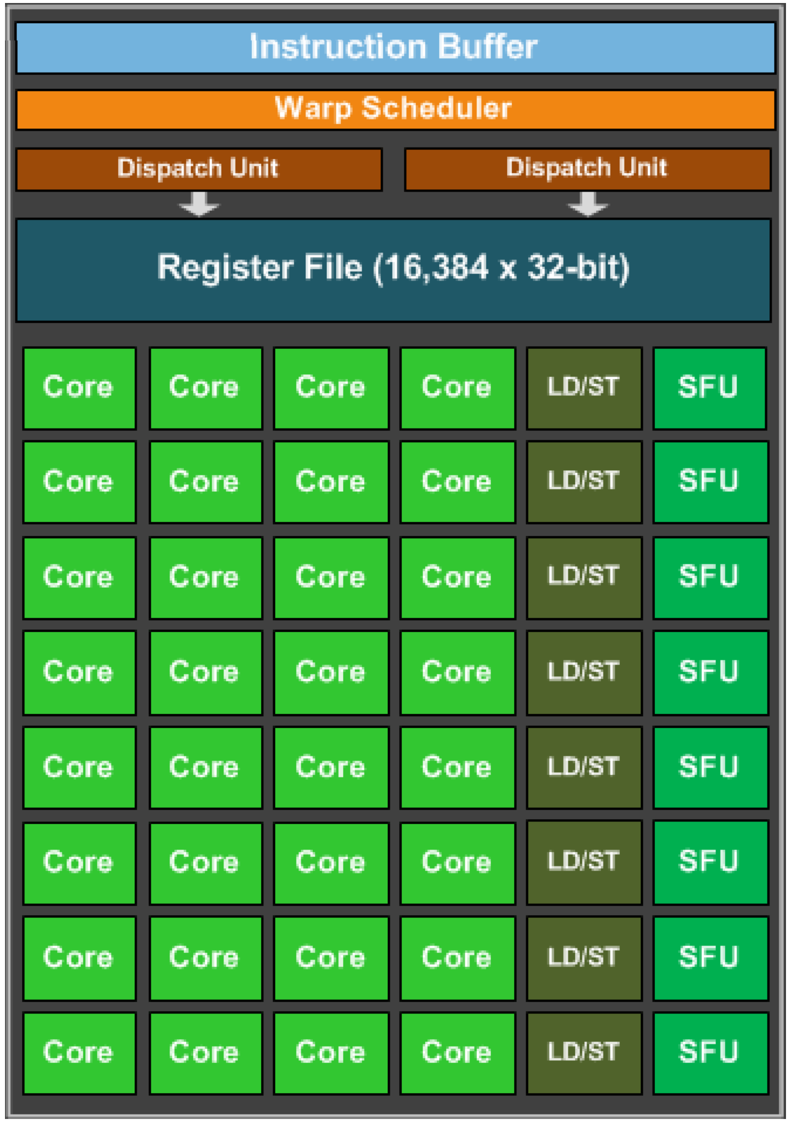
Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



32 cores!

We should parallelize our application!



https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf

First parallelization attempt

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
    int chunk_size = size/blockDim.x;
    int start = chunk_size * threadIdx.x;
    int end = start + end;
    for (int i = start; i < end; i++) {
      d_a[i] = d_b[i] + d_c[i];
   }
}
```

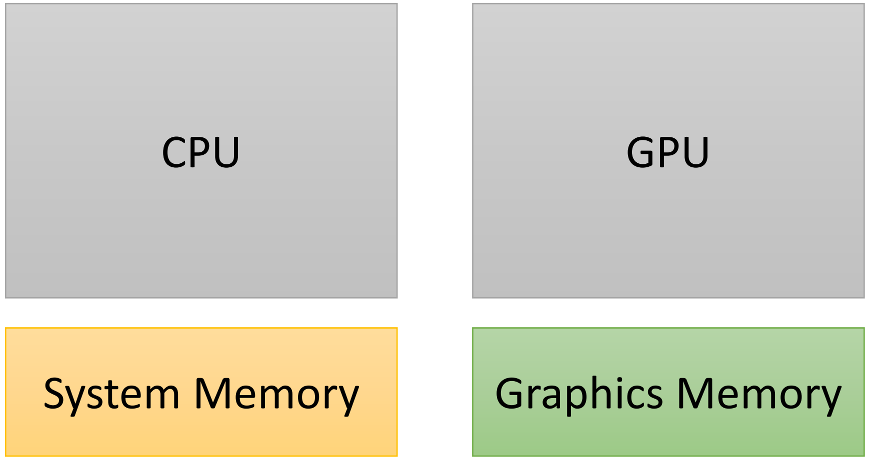
calling the function

```
vector_add<<<1,32>>>(d_a, d_b, d_c, size);
```

number of threads
thread id

GPU Memory

CPU Memory:
Fast: Low Latency
Easily saturated: Low Bandwidth
Scales well: up to 1 TB
DDR



GPU Memory:
slow: High Latency
hard to saturate: High Bandwidth
doesn't scale: 32 GB
GDDR, HBM

2-lane straight highway driven on by sports cars

Different technologies

16-lane highway on a windy road driven by semi trucks

Preemption and concurrency?

warp 0

warp 1

warp 2

GPU

Graphics Memory

We can hide latency through preemption and concurrency!

Preemption and concurrency?

warp 1

warp 2

warp 0

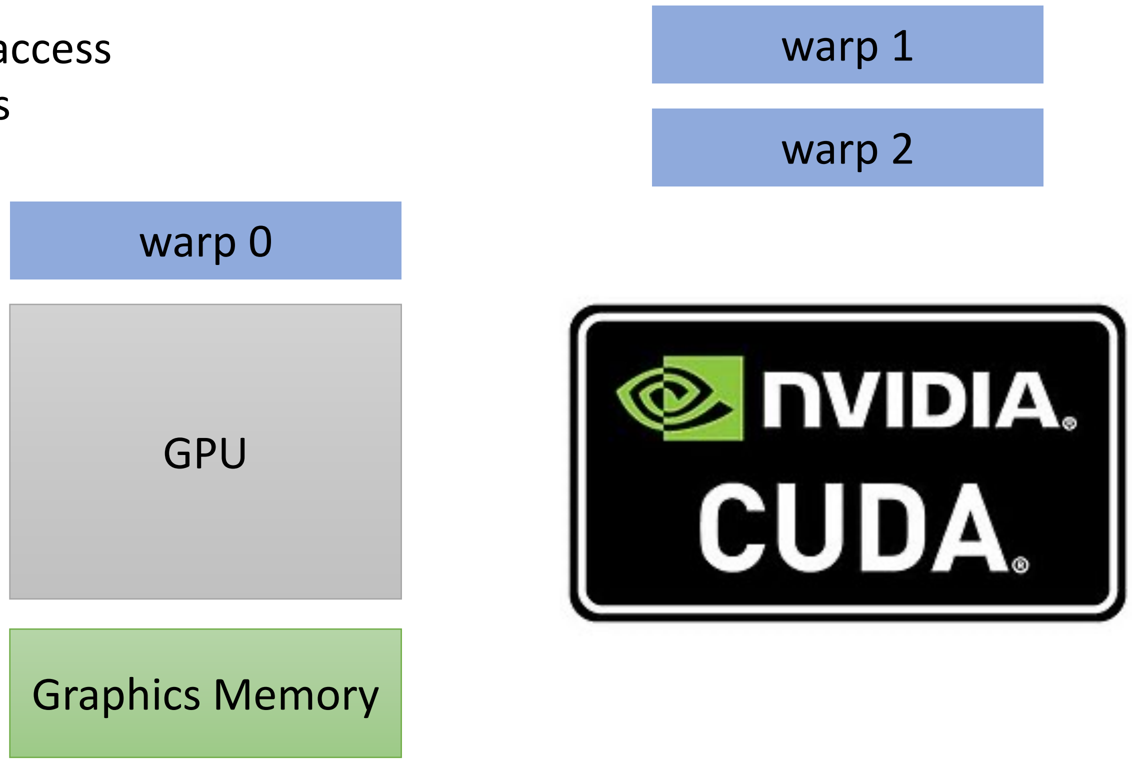
GPU

Graphics Memory

We can hide latency through preemption and concurrency!

Preemption and concurrency?

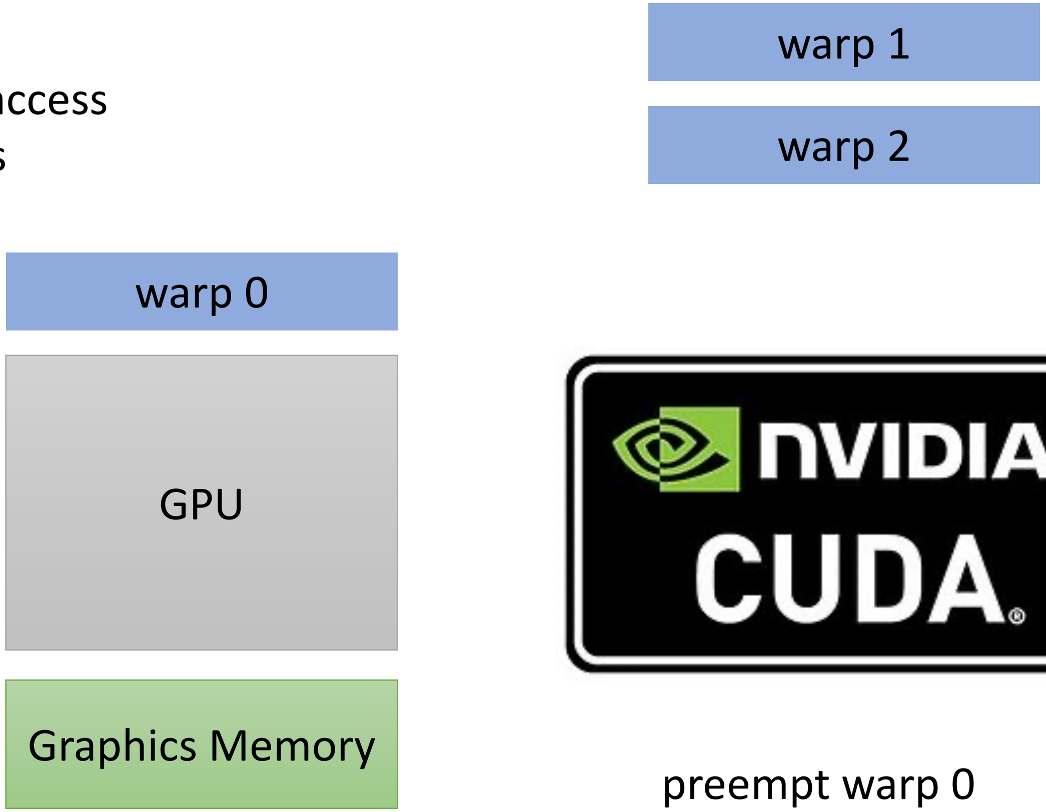
memory access
600 cycles



We can hide latency through preemption and concurrency!

Preemption and concurrency?

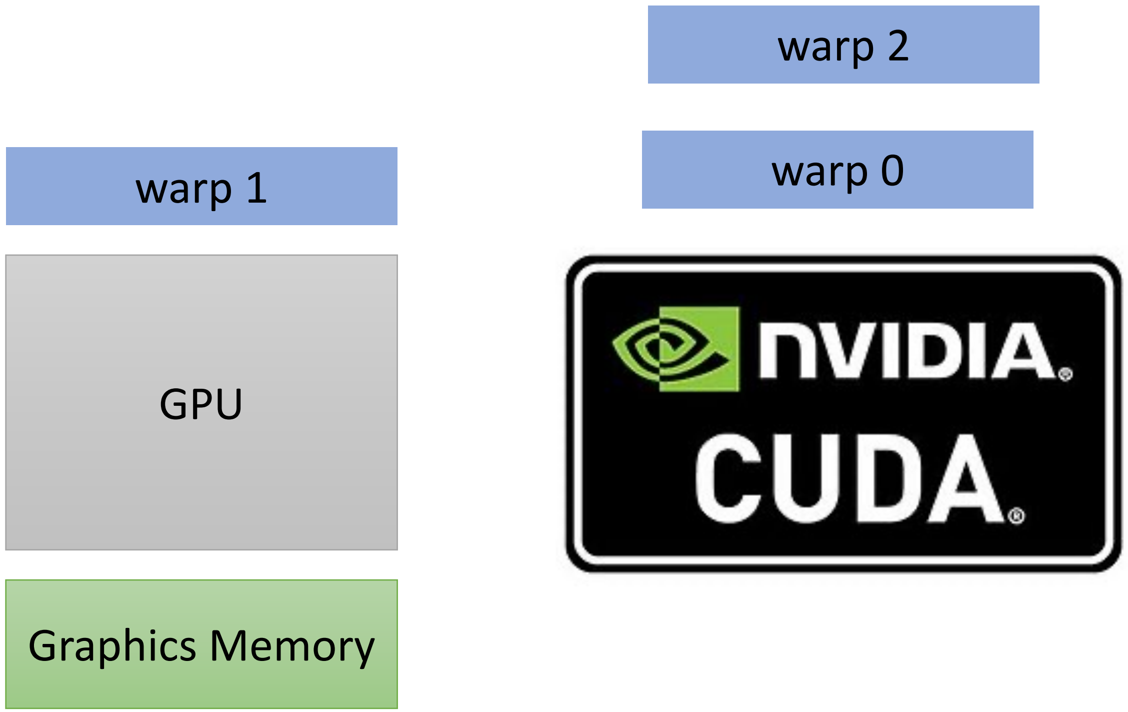
memory access
600 cycles



preempt warp 0
and put warp 1 on

We can hide latency through
preemption and concurrency!

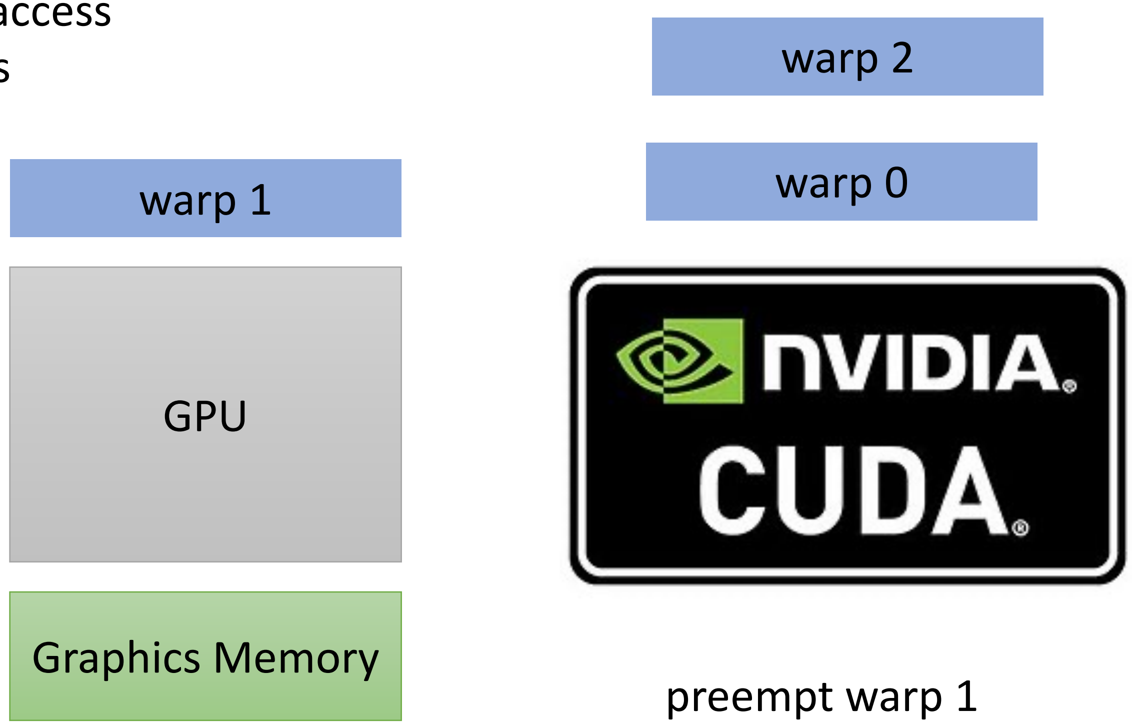
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

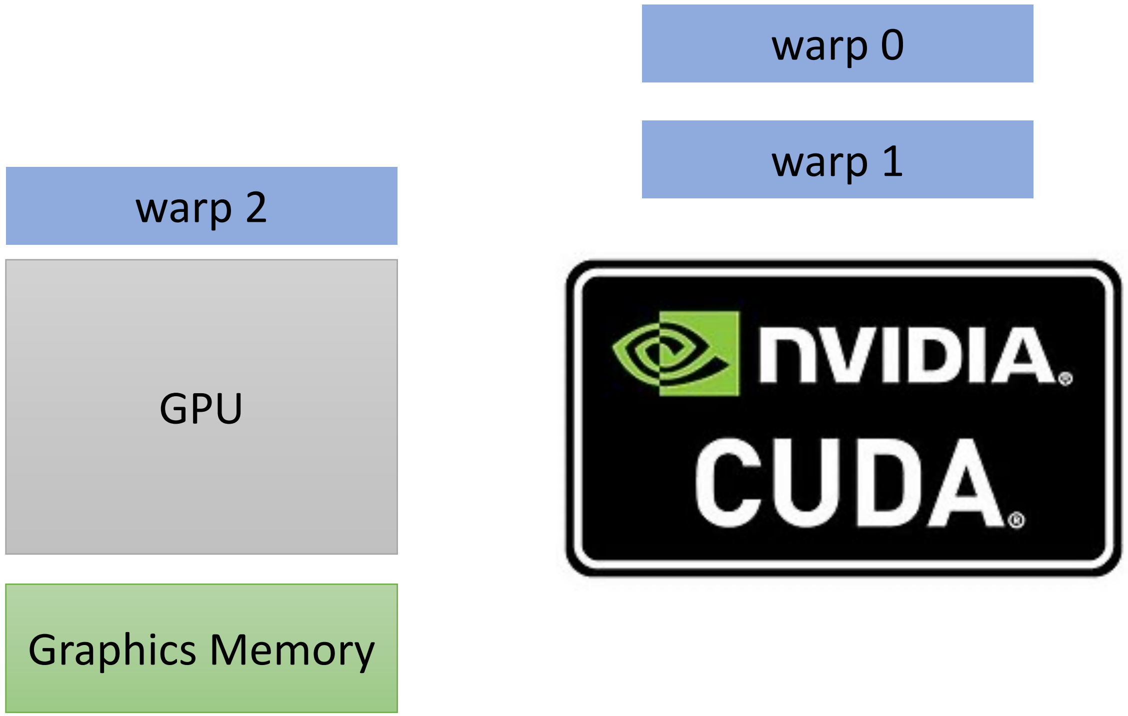
memory access
600 cycles



preempt warp 1
and put warp 2 on

We can hide latency through
preemption and concurrency!

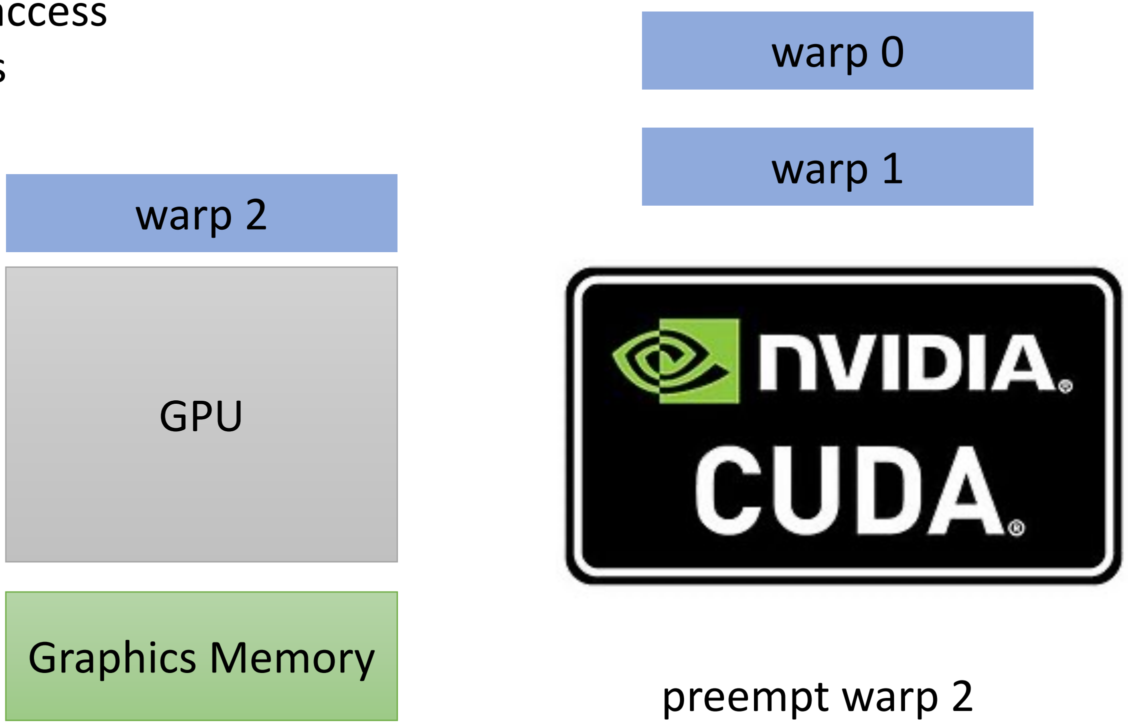
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

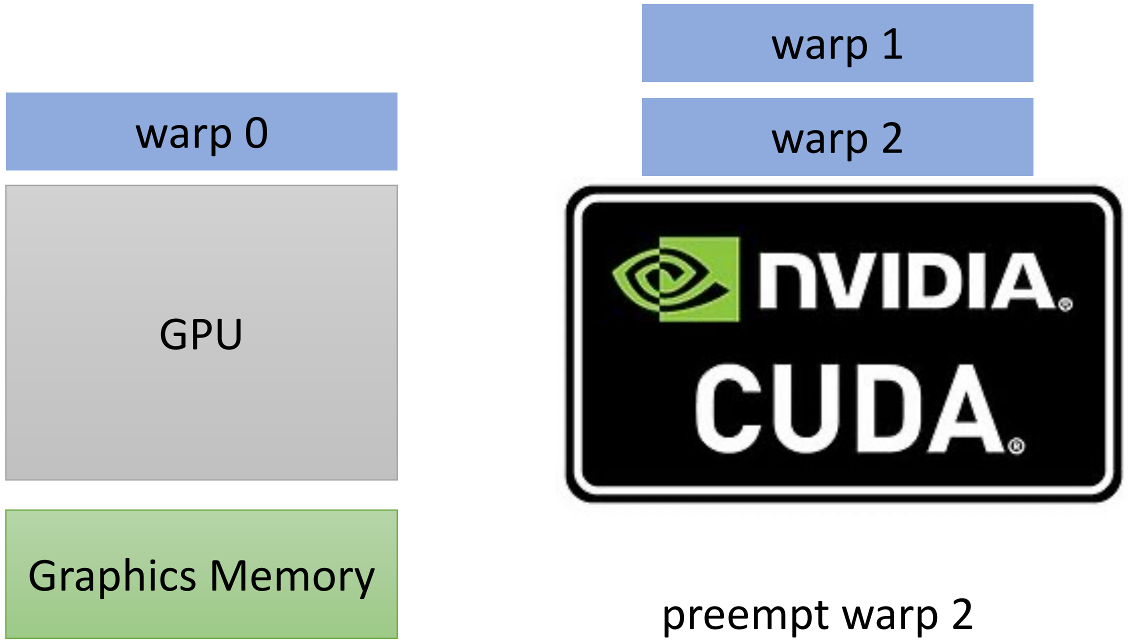


preempt warp 2
and put warp 0 on

We can hide latency through
preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!



preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Go back to our program

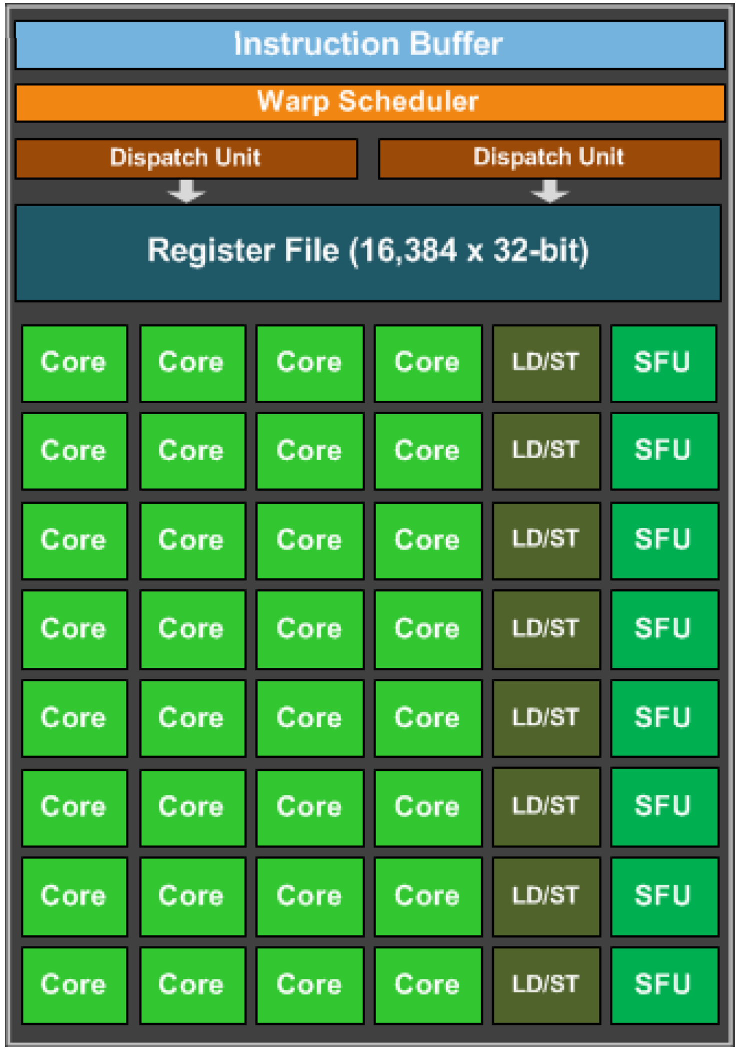
```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
    int chunk_size = size/blockDim.x;
    int start = chunk_size * threadIdx.x;
    int end = start + end;
    for (int i = start; i < end; i++) {
      d_a[i] = d_b[i] + d_c[i];
   }
 }
```

calling the function

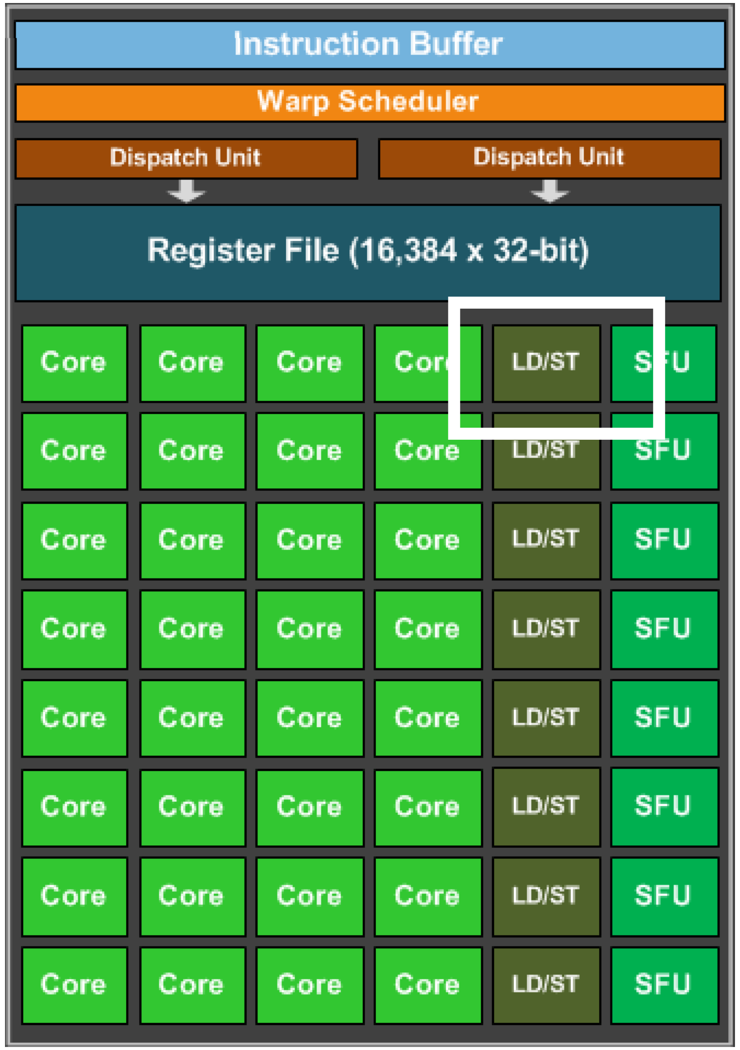
Lets launch with 32 warps

```
vector_add<<<1,1024>>>(d_a, d_b, d_c, size);
```

Optimizing memory accesses



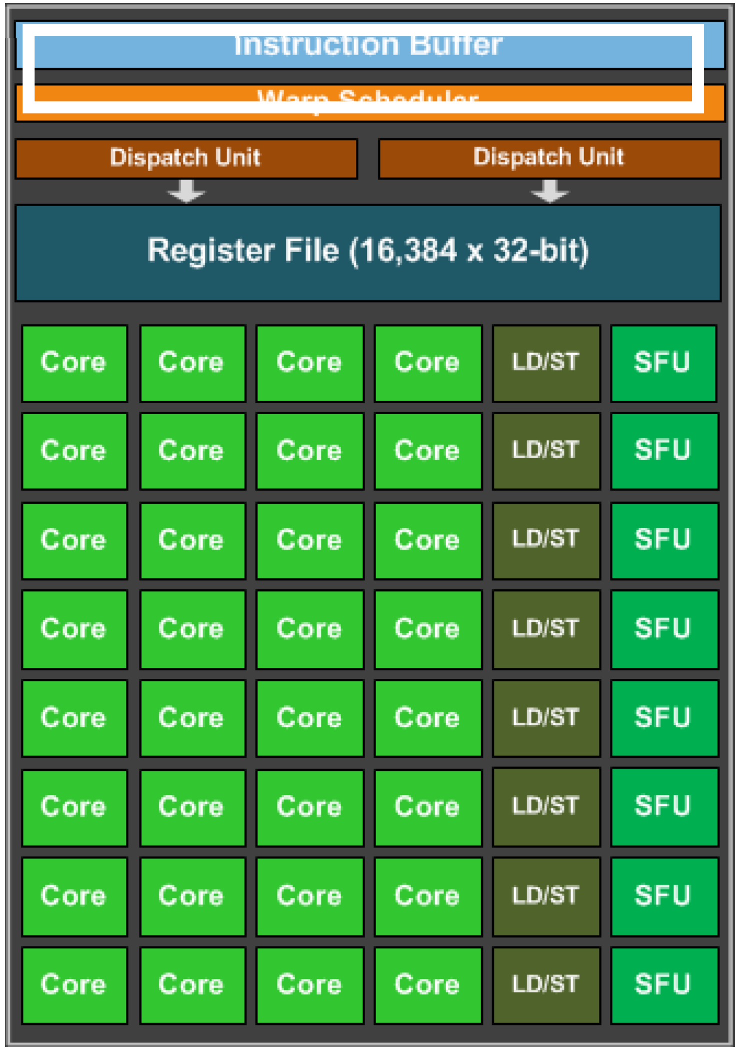
Optimizing memory accesses



this is the load/store unit. The hardware component responsible for issuing loads and stores.

Why doesn’t every core have one?

Optimizing memory accesses

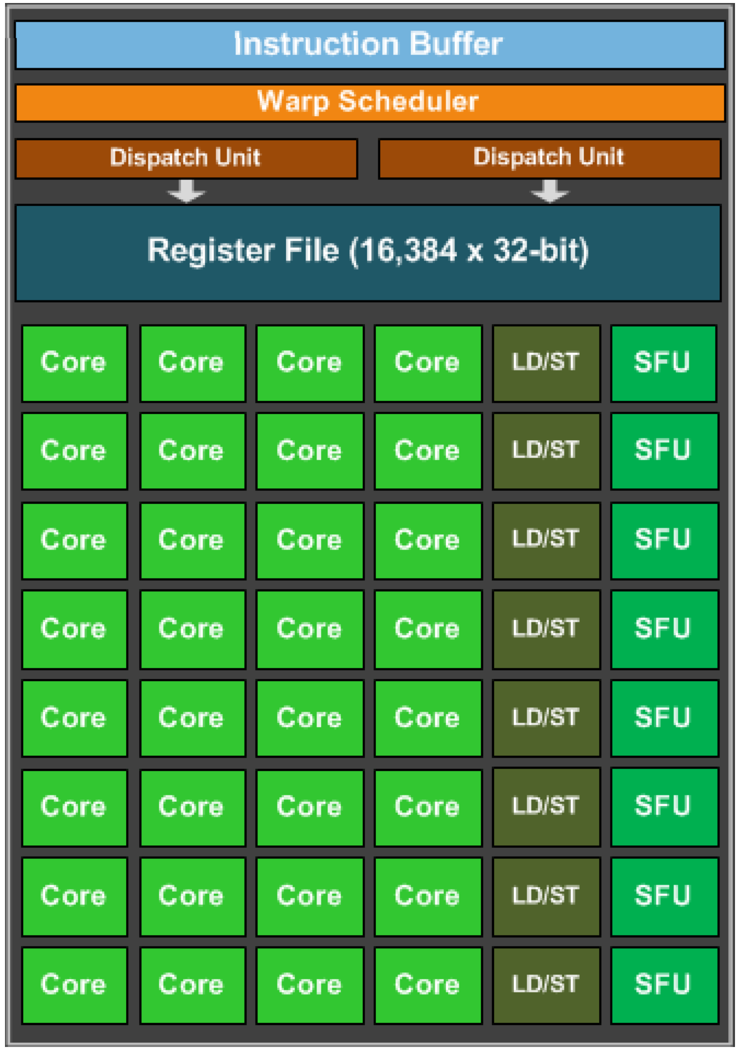


This is the instruction cache... Why doesn't every core have a instruction buffer to keep track of its program?

this is the load/store unit. The hardware component responsible for issuing loads and stores.

Why doesn't every core have one?

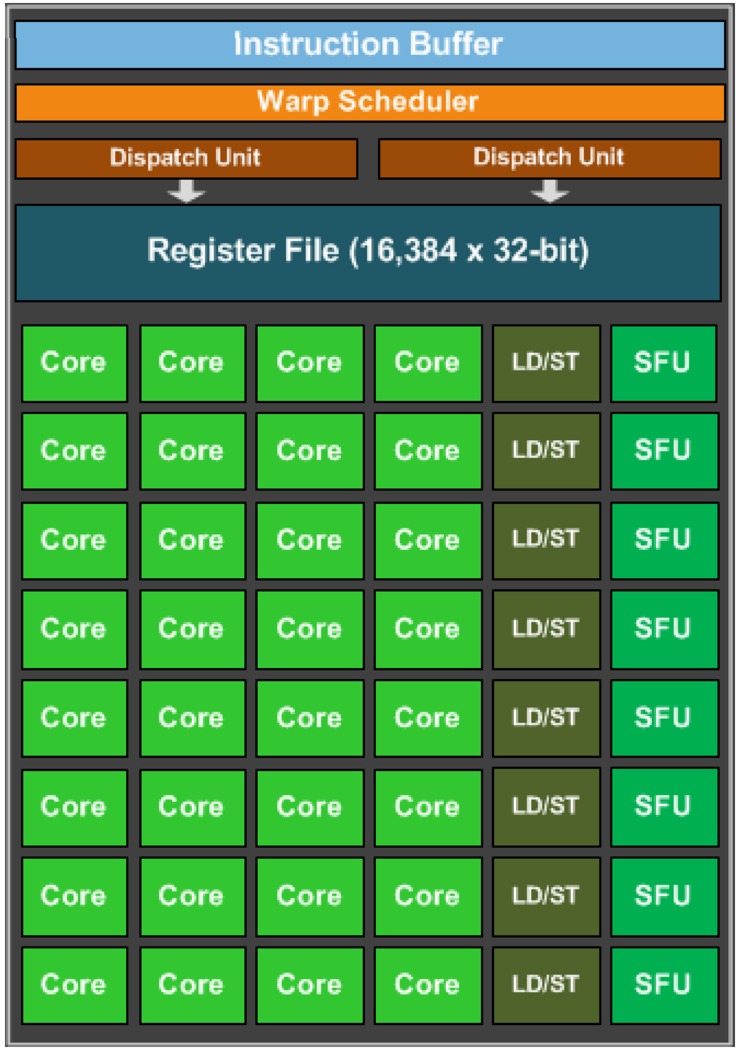
Warp execution



Groups of 32 threads are called a "warp"

They are executed in lock-step, i.e. they all execute the same instruction at the same time

Warp execution



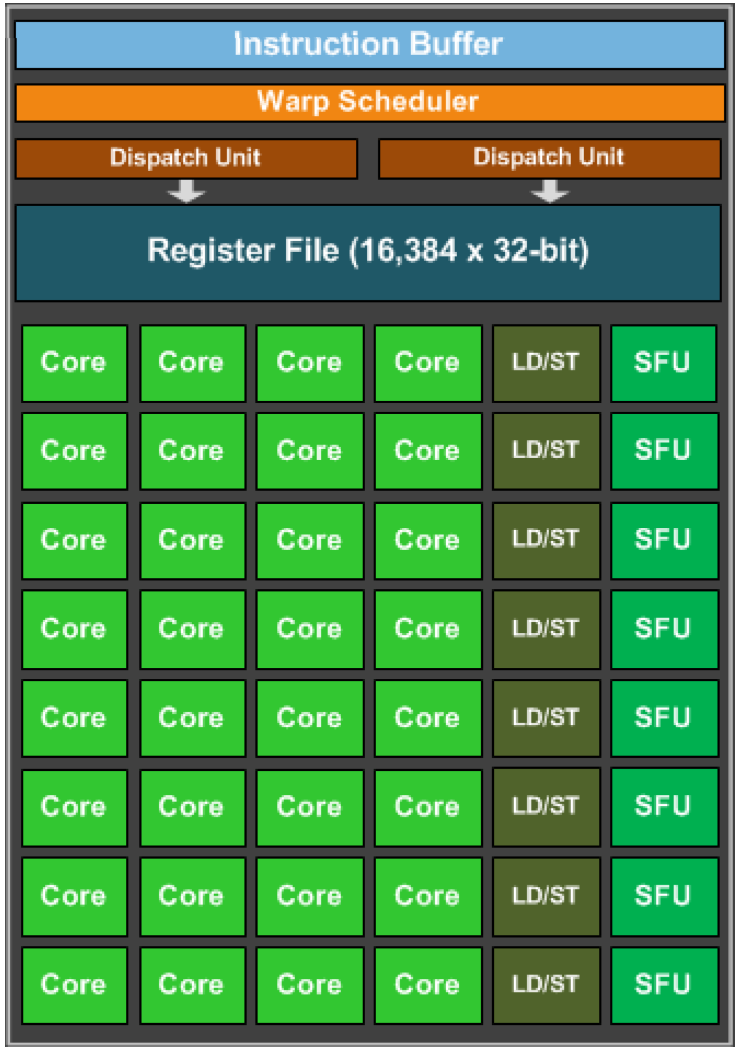
Groups of 32 threads are called a “warp”

They are executed in lock-step, i.e. they all execute the same instruction at the same time

Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution



Groups of 32 threads are called a "warp"

They are executed in lock-step, i.e. they all execute the same instruction at the same time

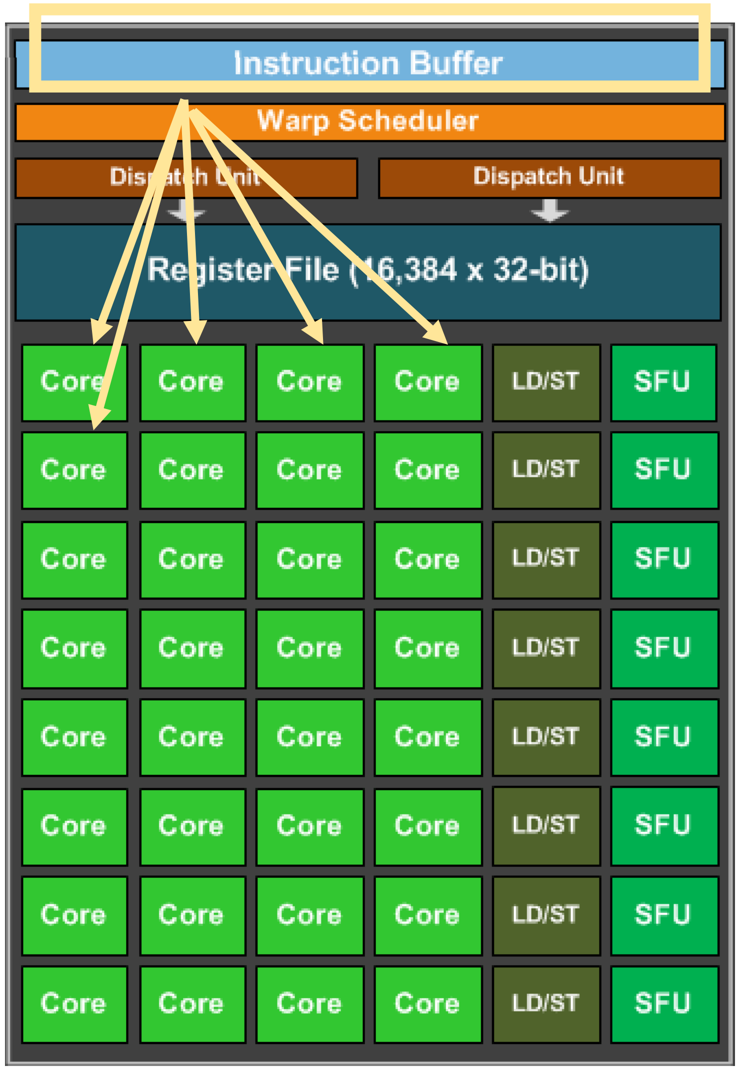
Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution

Groups of 32 threads are called a “warp”

They are executed in lock-step, i.e. they all execute the same instruction at the same time



instruction is fetched from the buffer and distributed to all the cores.

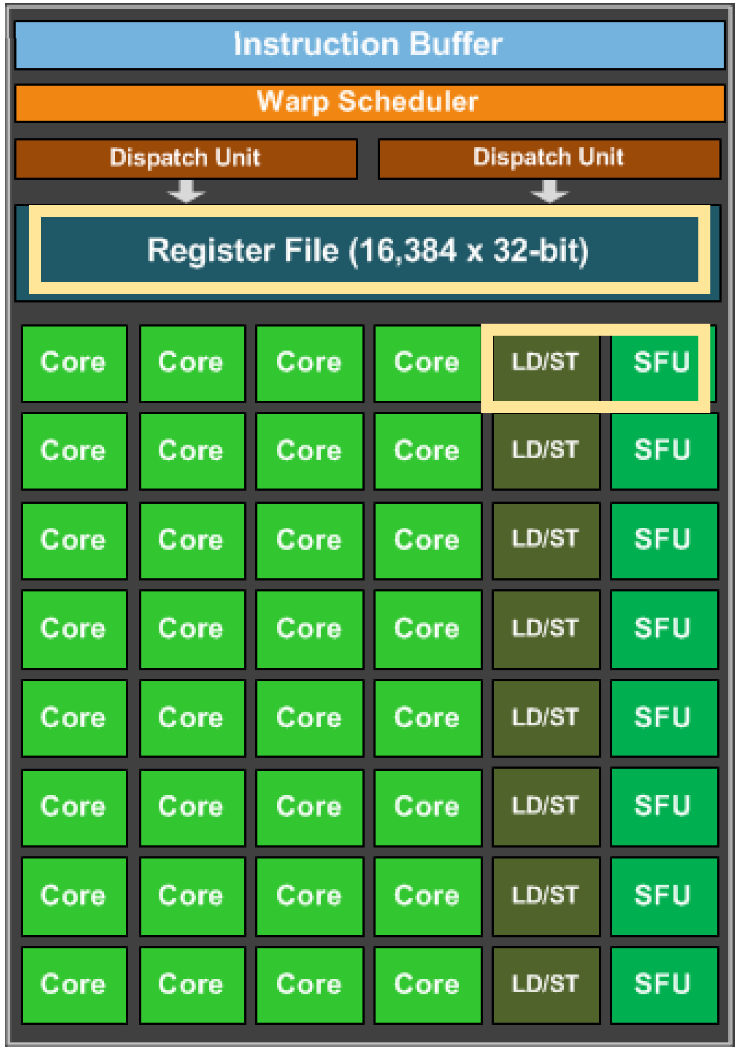
Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution

Groups of 32 threads are called a "warp"

They are executed in lock-step, i.e. they all execute the same instruction at the same time



Cores can a large register file
they share expensive HW units (load/store and special functions)

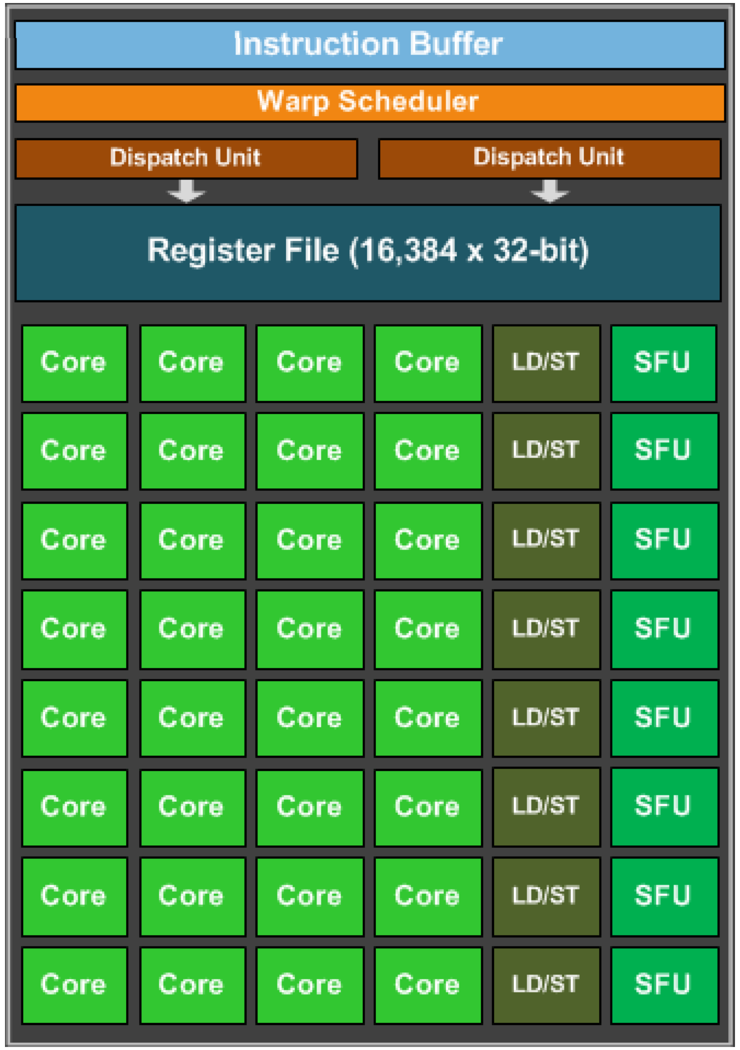
Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution

Groups of 32 threads are called a "warp"

They are executed in lock-step, i.e. they all execute the same instruction at the same time



All cores need to wait until all cores finish the first instruction

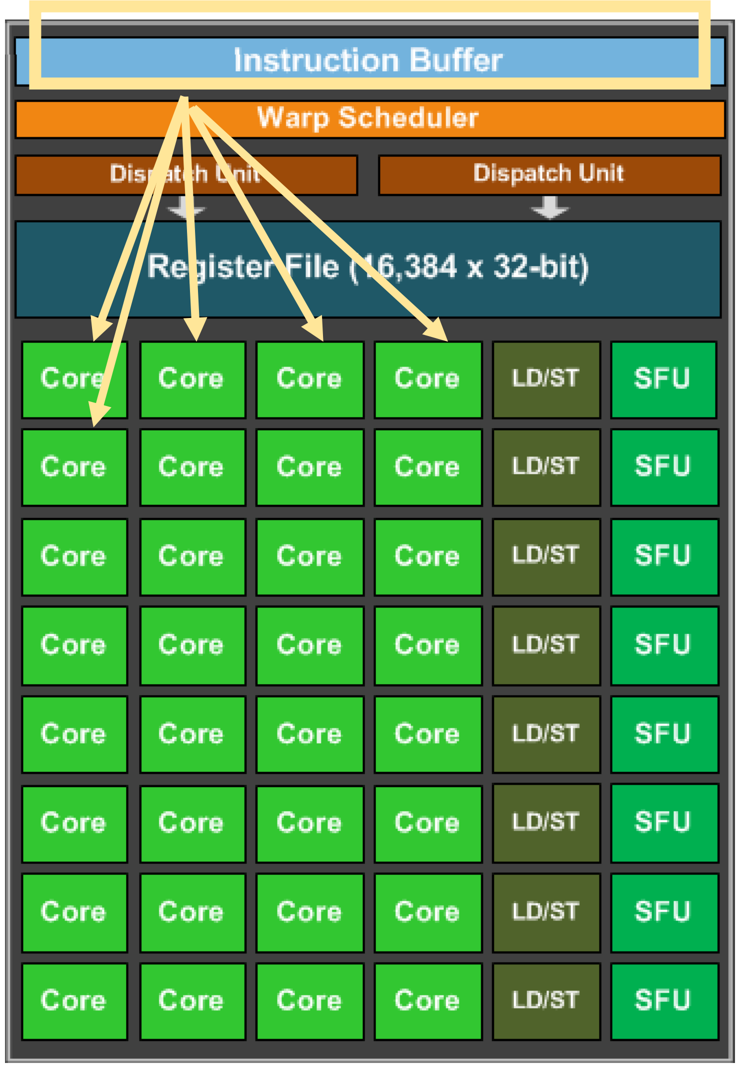
Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution

Groups of 32 threads are called a “warp”

They are executed in lock-step, i.e. they all execute the same instruction at the same time



Start the next instruction.

Program:

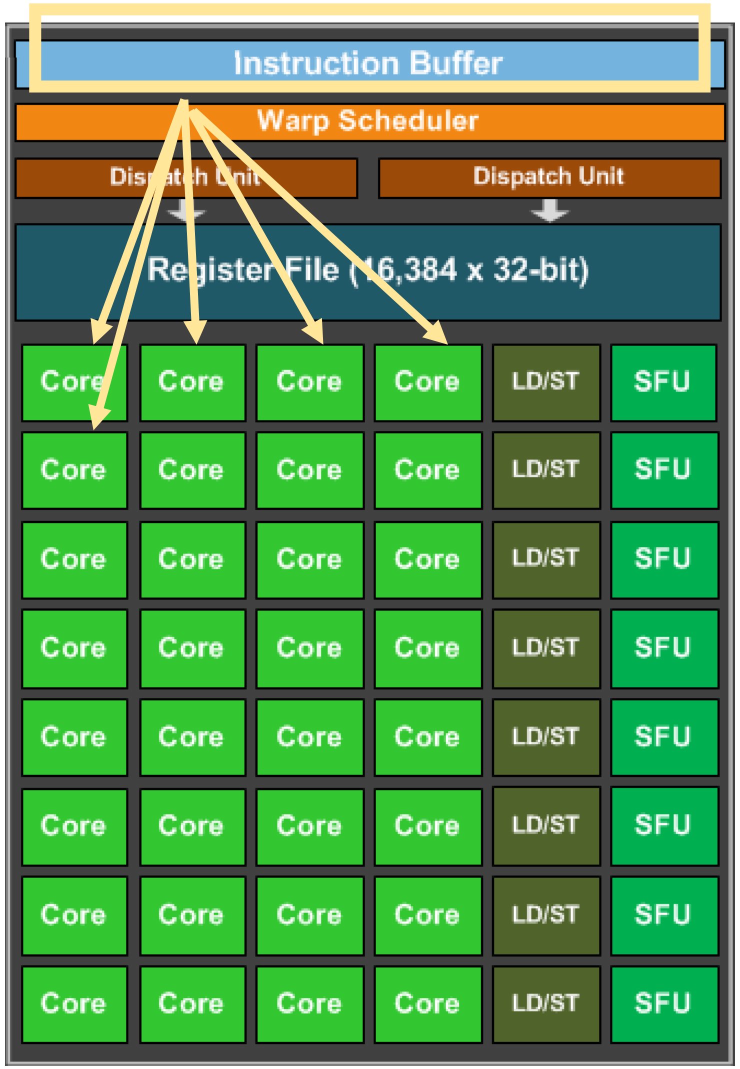
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Why would we have a programming model like this?

Warp execution

Groups of 32 threads are called a “warp”

They are executed in lock-step, i.e. they all execute the same instruction at the same time



Start the next instruction.

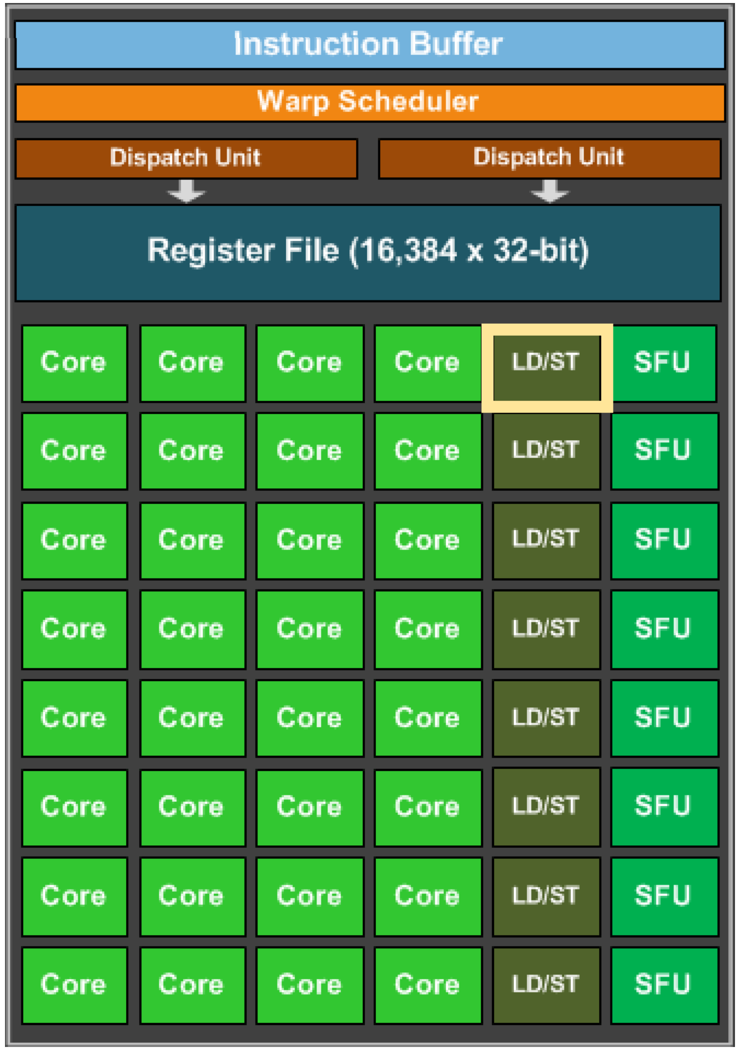
Program:

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Why would we have a programming model like this?
More cores (share program counters)
Can be efficient to share other hardware resources

Warp execution

Lets look closer at memory

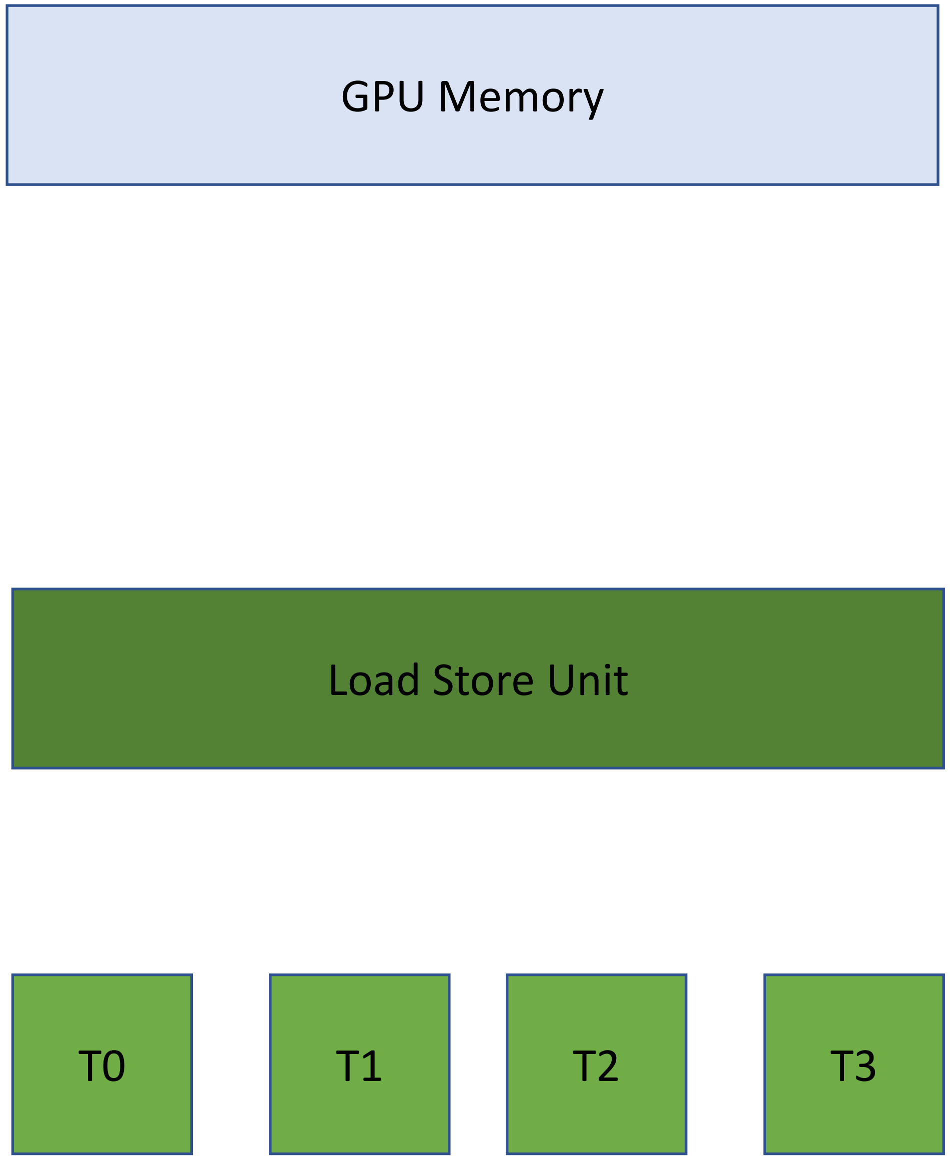


<u>Program:</u>

```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen

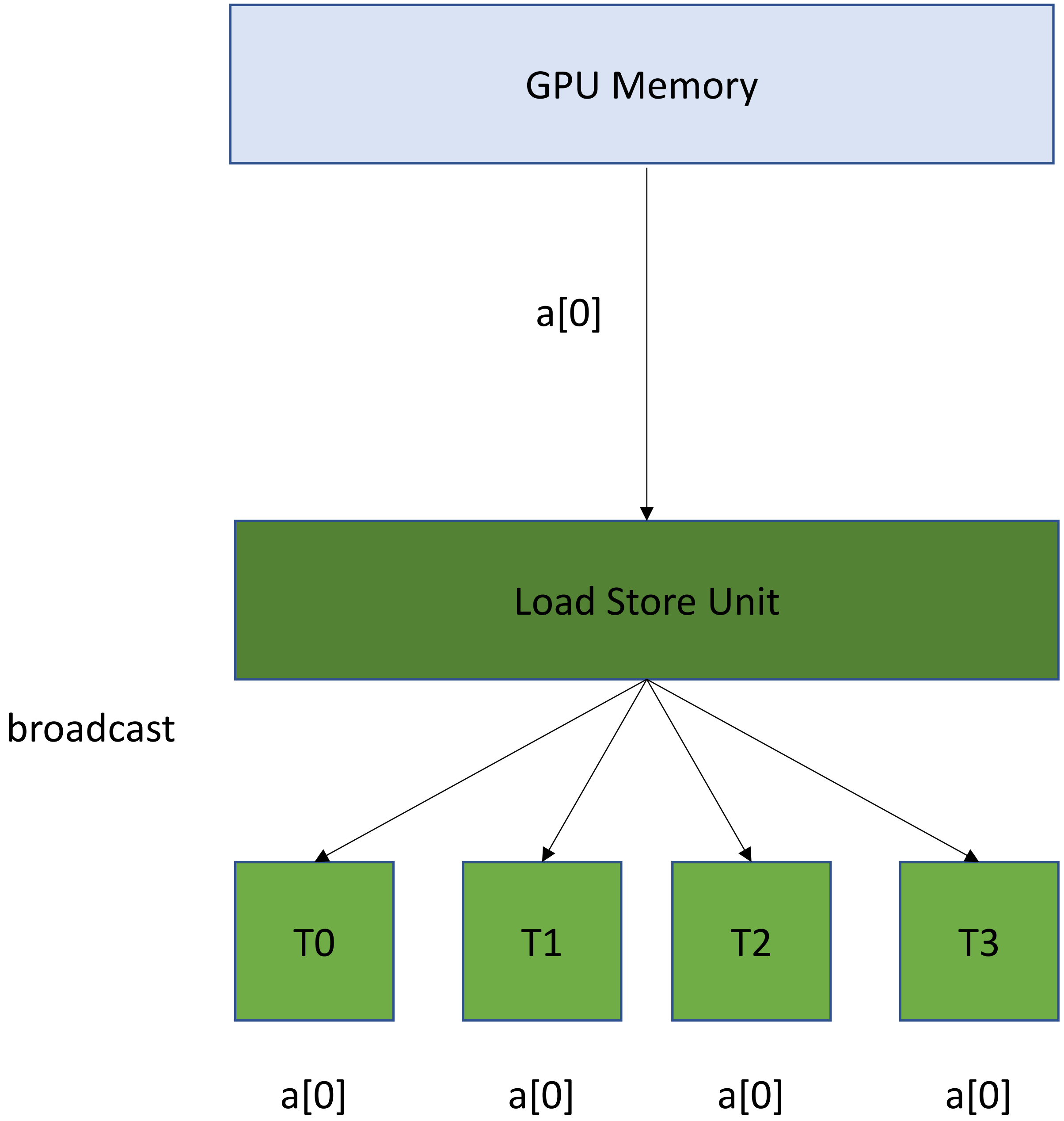


4 cores are accessing memory. What can happen

All read the same value
This is efficient: the load store unit can ask for the value and then broadcast it to all cores.

1 request to GPU memory

Efficient, but probably not too common.

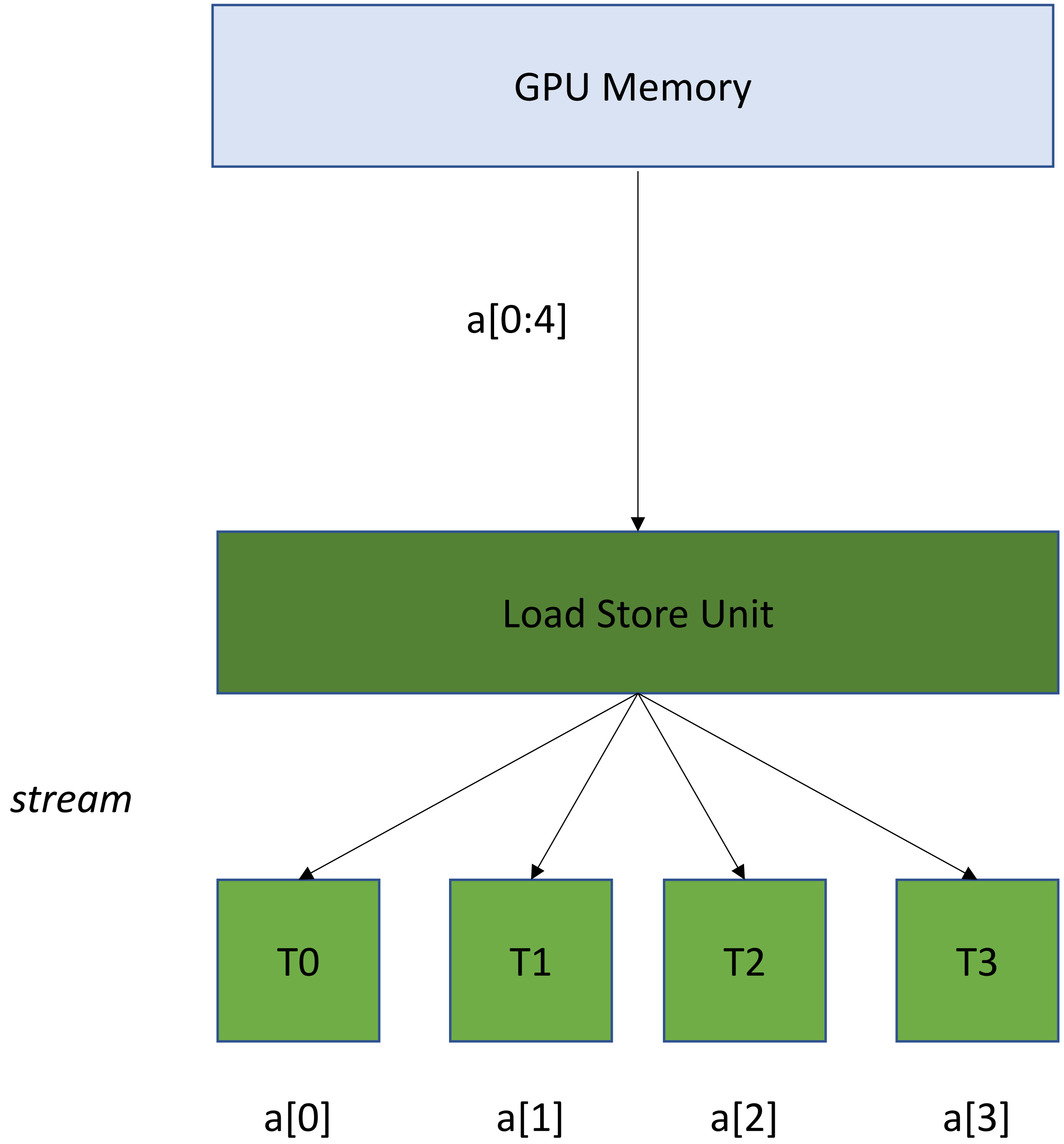


4 cores are accessing memory. What can happen

Read contiguous values
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes

Can easily distribute the values to the threads

1 request to GPU memory

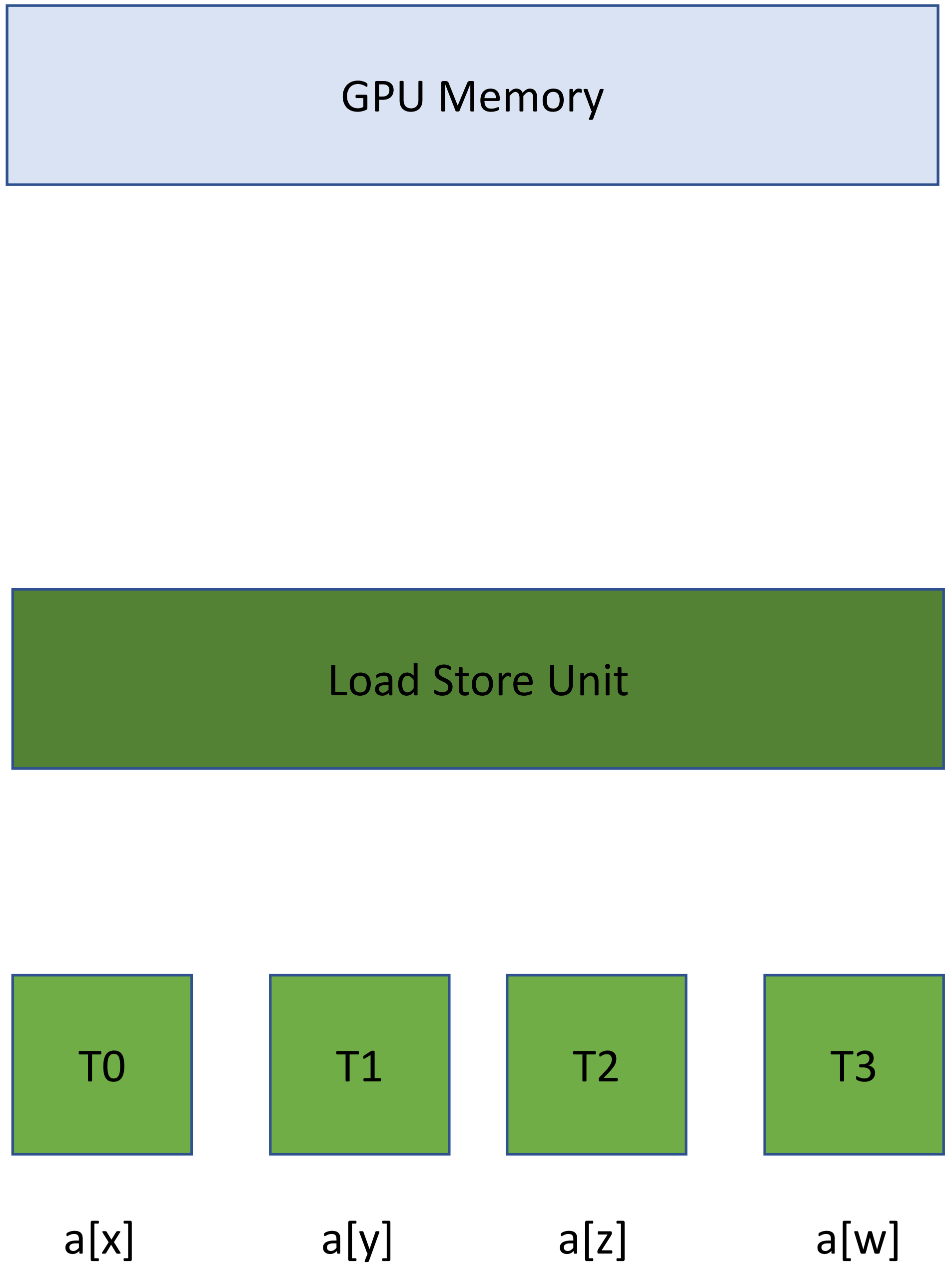


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

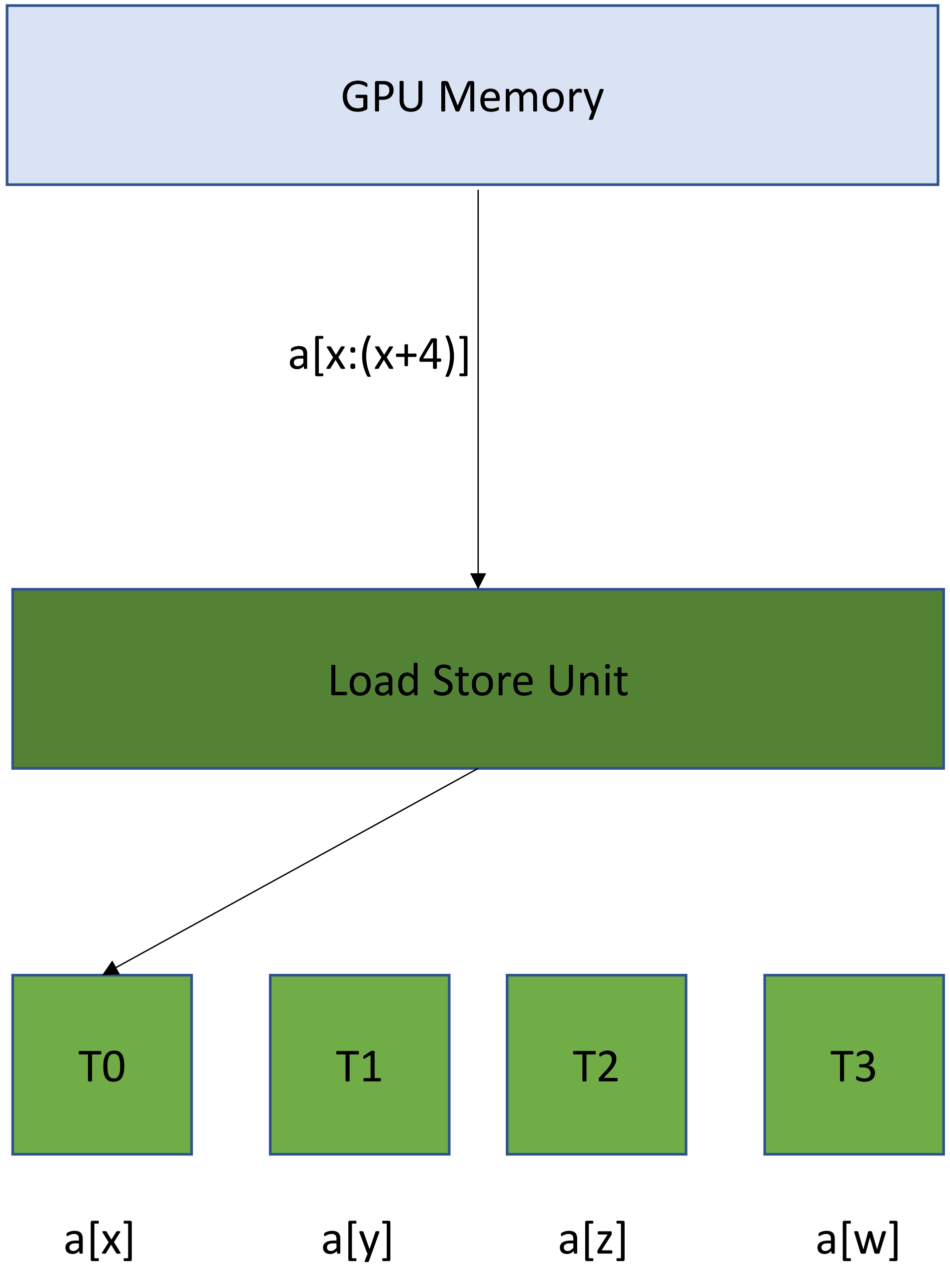


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

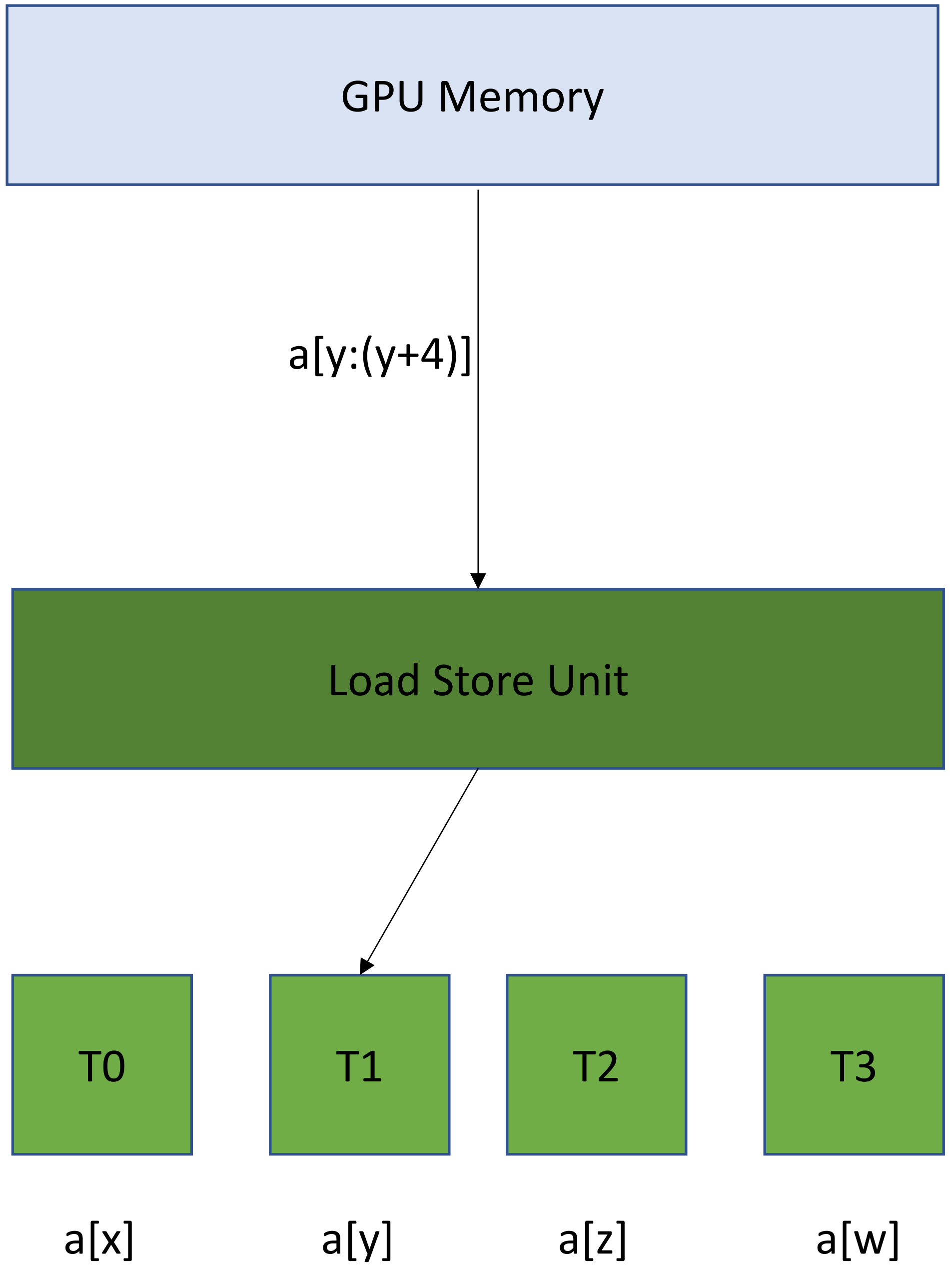


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

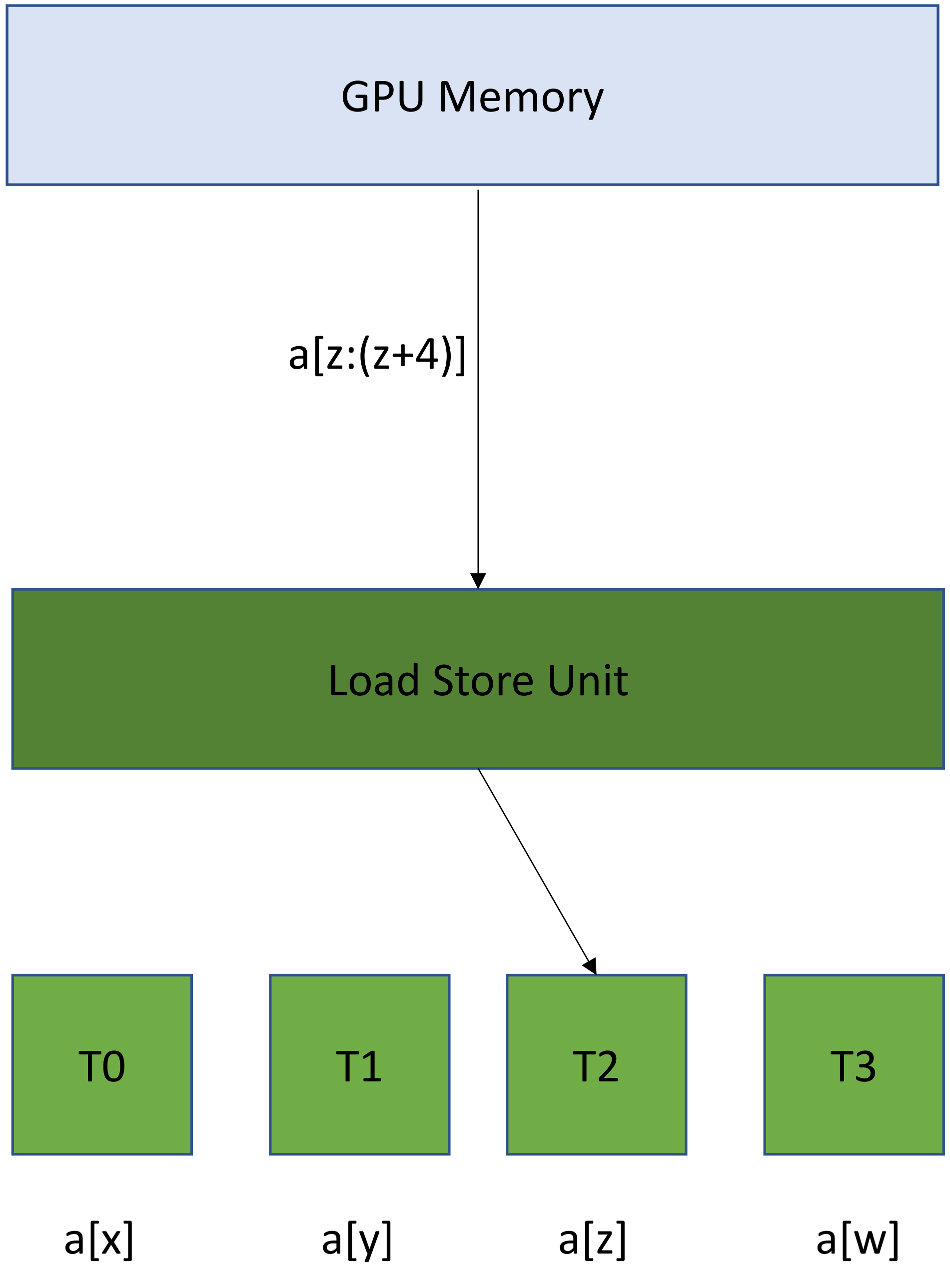


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

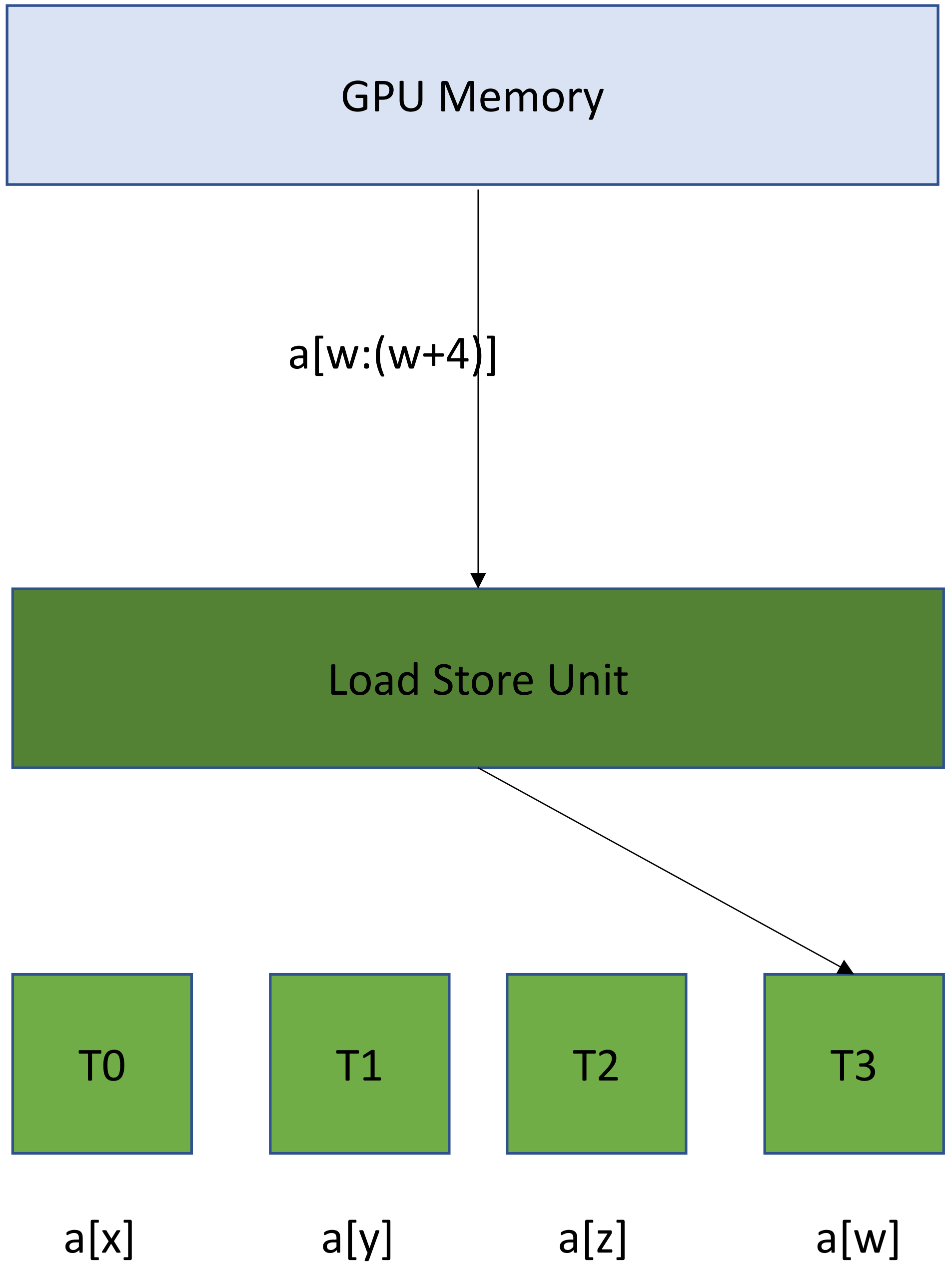


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory



Go back to our program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
    int chunk_size = size/blockDim.x;
    int start = chunk_size * threadIdx.x;
    int end = start + end;
    for (int i = start; i < end; i++) {
      d_a[i] = d_b[i] + d_c[i];
   }
}
```

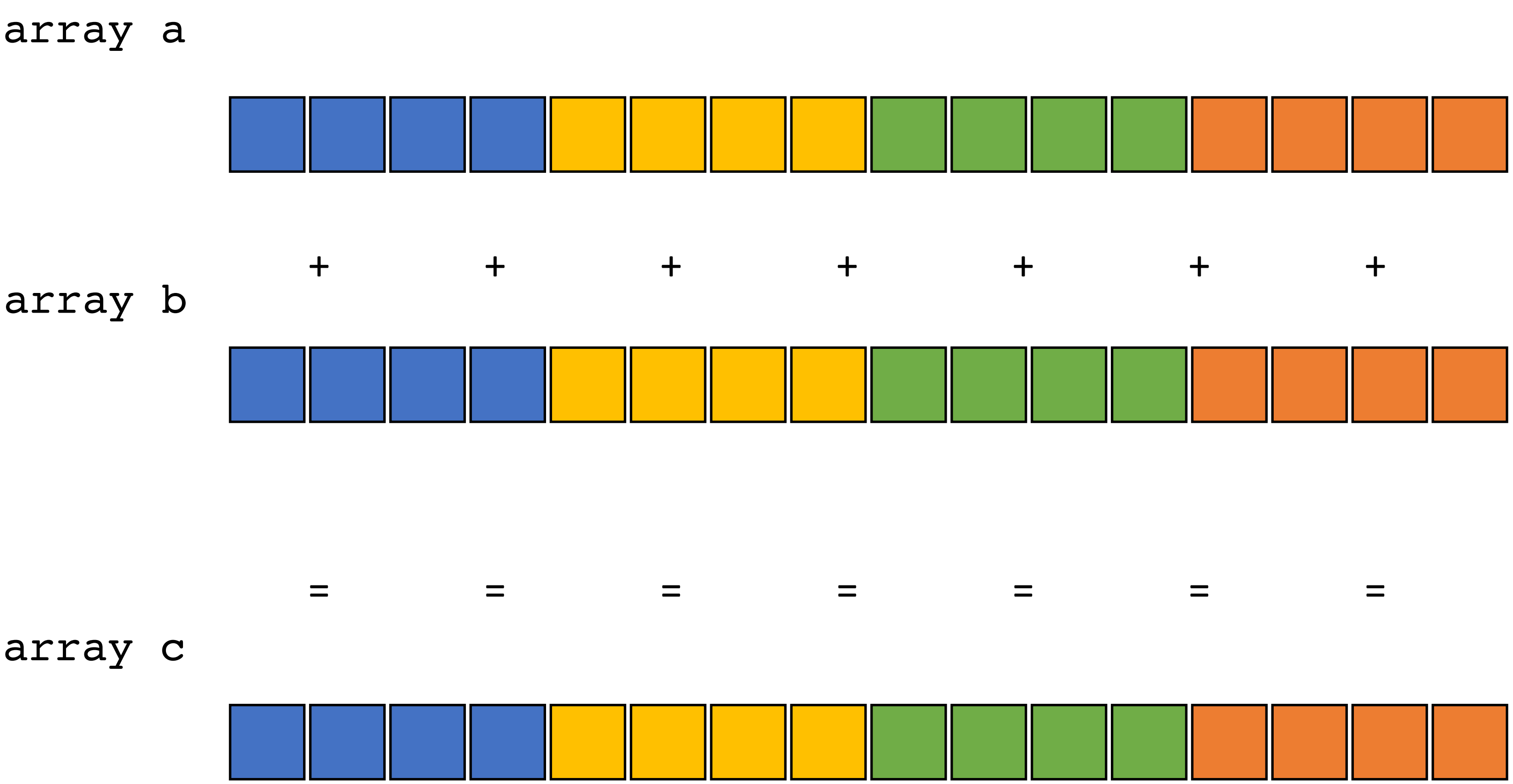
calling the function

```
vector_add<<<1,1024>>>(d_a, d_b, d_c, size);
```

Chunked Pattern

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

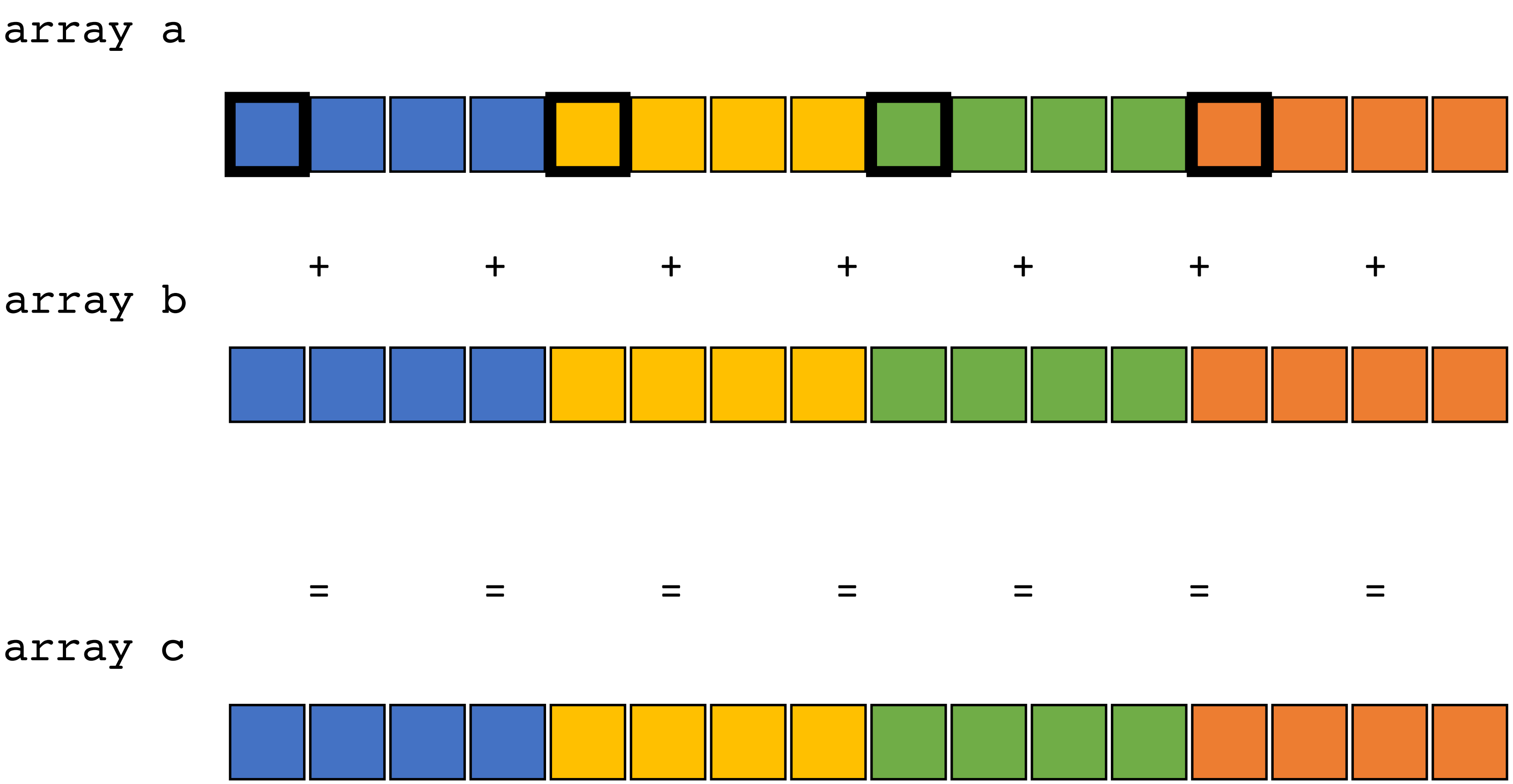


Chunked Pattern

the first element accessed by the 4 threads sharing a load store unit. What sort of access is this?

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

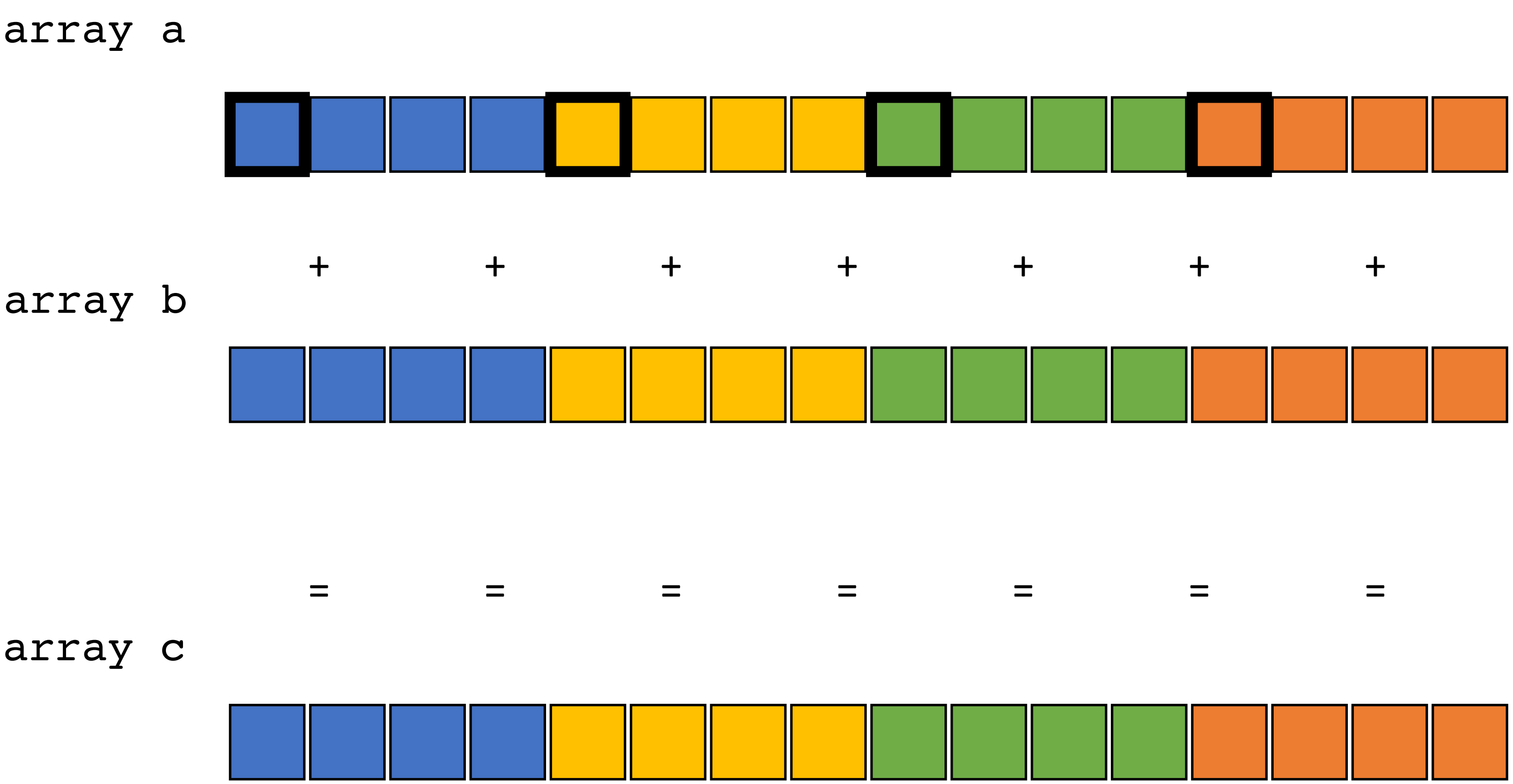


Chunked Pattern

the first element accessed by the 4 threads sharing a load store unit. What sort of access is this?

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

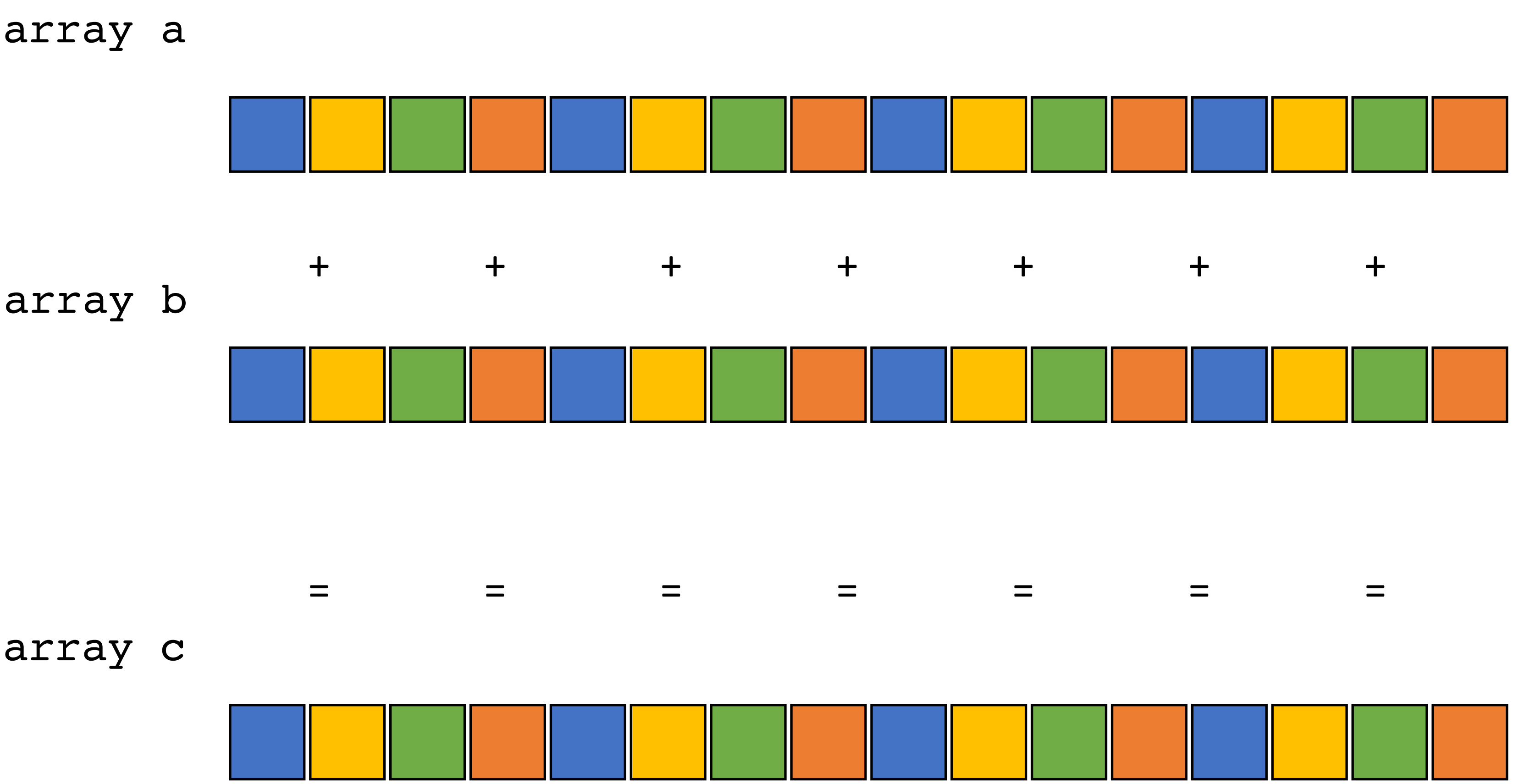


How can we fix this

Stride Pattern

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

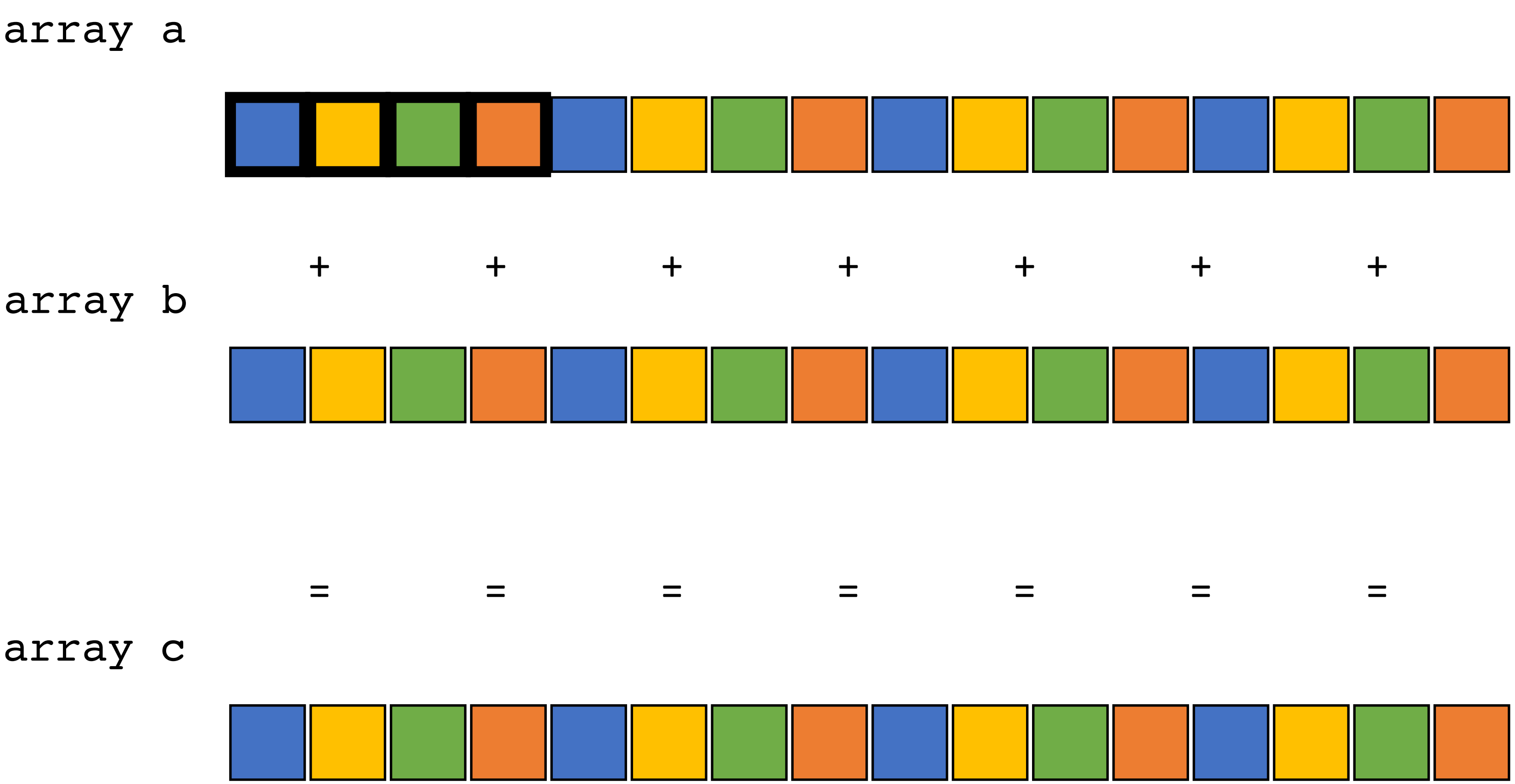


Stride Pattern

What sort of pattern is this?

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Go back to our program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
    int chunk_size = size/blockDim.x;
    int start = chunk_size * threadIdx.x;
    int end = start + end;
    for (int i = start; i < end; i++) {
      d_a[i] = d_b[i] + d_c[i];
   }
 }
```

calling the function

Lets change this to a stride pattern

```
vector_add<<<1,1024>>>(d_a, d_b, d_c, size);
```

Go back to our program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
  for (int i = threadIdx.x; i < size; i+=blockDim.x) {
     d_a[i] = d_b[i] + d_c[i];
  }
}
```

calling the function

```
vector_add<<<1,1024>>>(d_a, d_b, d_c, size);
```

Coalesced memory accesses

Lets try it! What do we think?

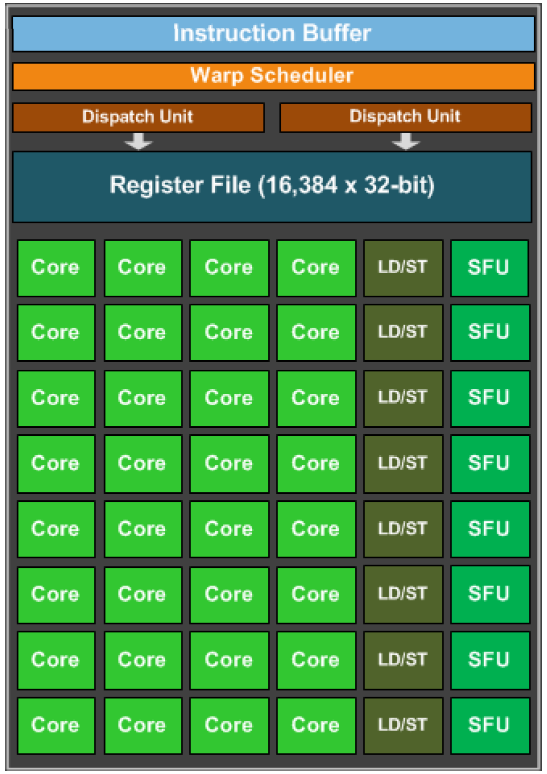
Coalesced memory accesses

Lets try it! What do we think? 😀

What else can we do?

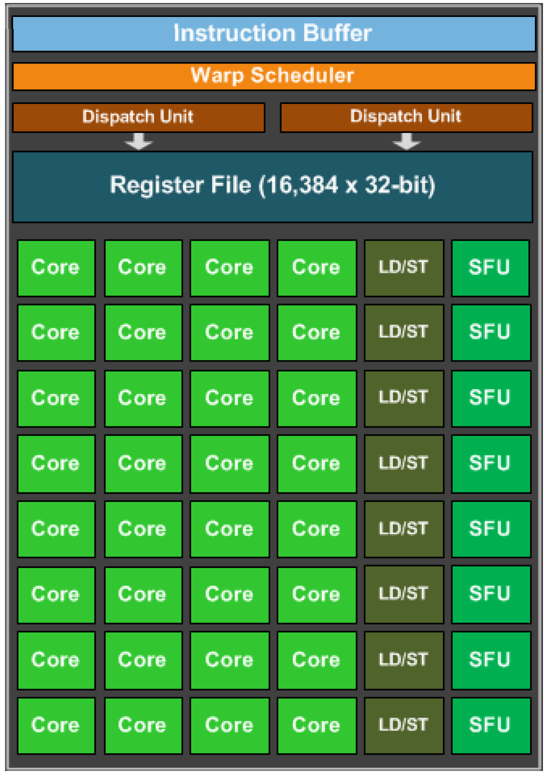
Multiple streaming multiprocessors

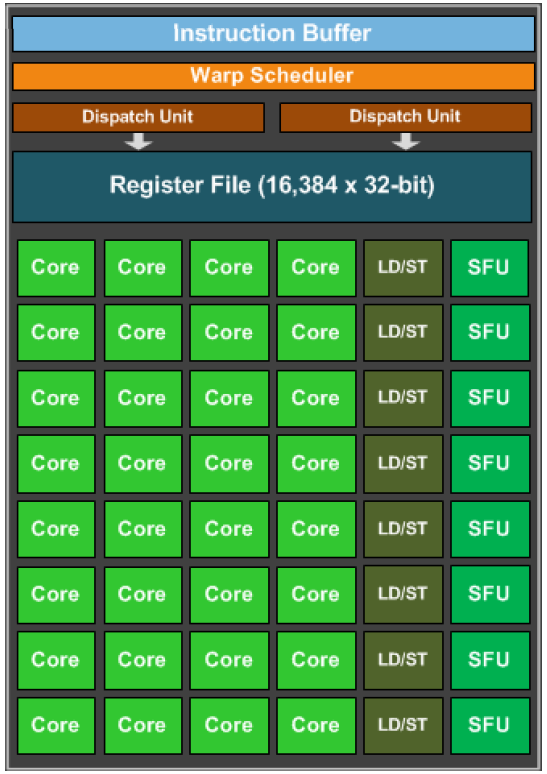
We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs big ML GPUs have 32. This little GPU has 1

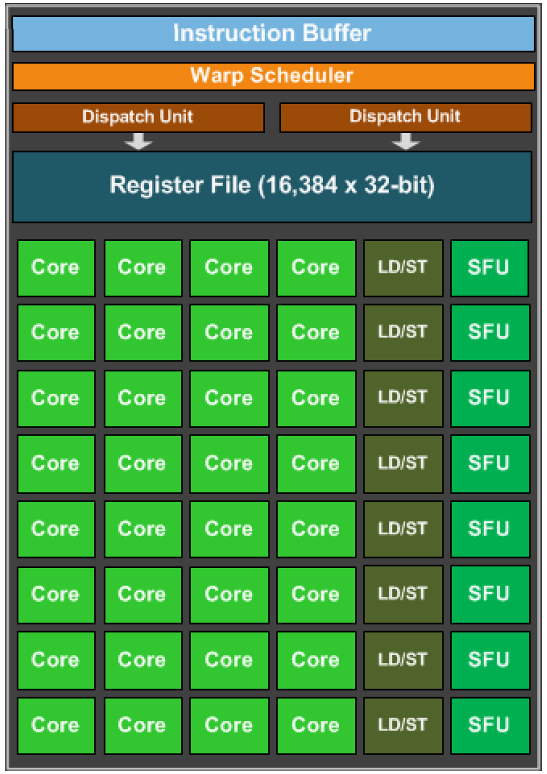


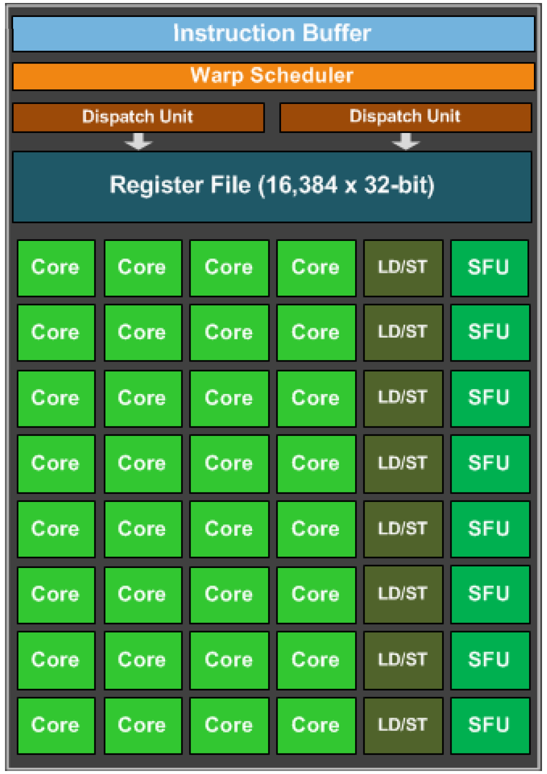
Multiple streaming multiprocessors

We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs big ML GPUs have 32. This little GPU has 1



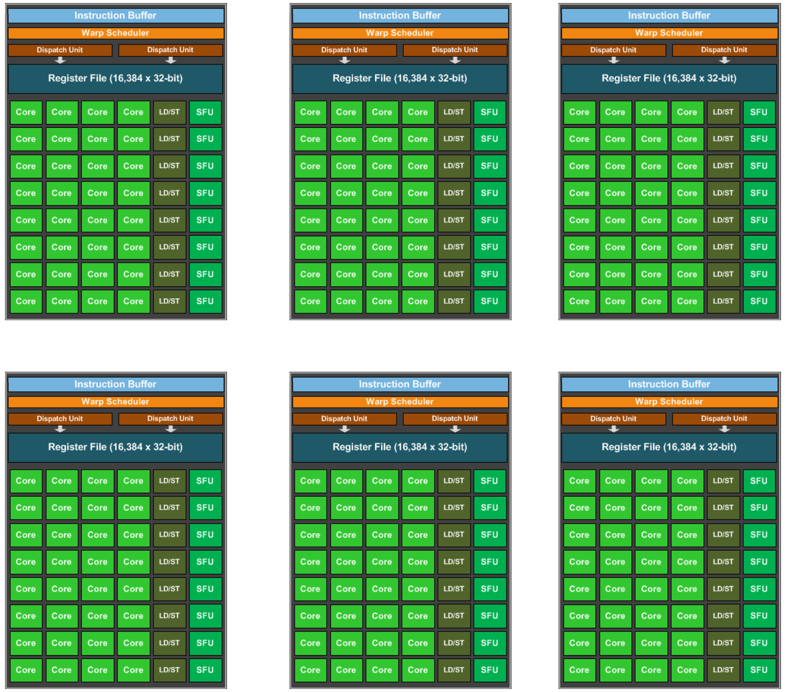






Multiple streaming multiprocessors

CUDA provides virtual streaming multiprocessors called **blocks**
Very efficient at launching and joining **blocks.**
No limit on blocks: launch as many as you need to map 1 thread to 1 data element



Go back to our program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
  for (int i = threadIdx.x; i < size; i+=blockDim.x) {
     d_a[i] = d_b[i] + d_c[i];
  }
}
```

calling the function

Launch with many thread blocks

```
vector_add<<<1,1024>>>(d_a, d_b, d_c, size);
```

Go back to our program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  d_a[i] = d_b[i] + d_c[i];
}
```

Need to recalculate some thread ids.

calling the function

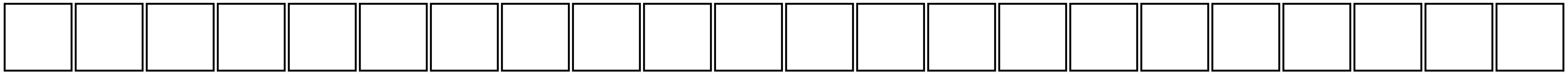
Launch with many thread blocks

```
vector_add<<<1024,1024>>>(d_a, d_b, d_c, size);
```

Now we have 1 thread for each element

```
#define SIZE (1024*1024)
```

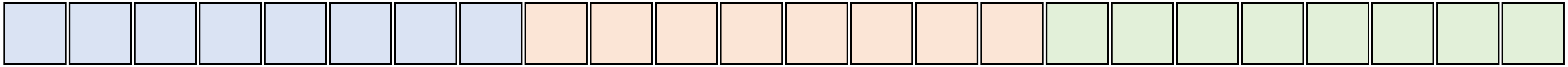
How does this work



Consider thread ids as a flattened array (which is often how they are used to index memory)

How does this work

block 0:
block 1:
block 2:

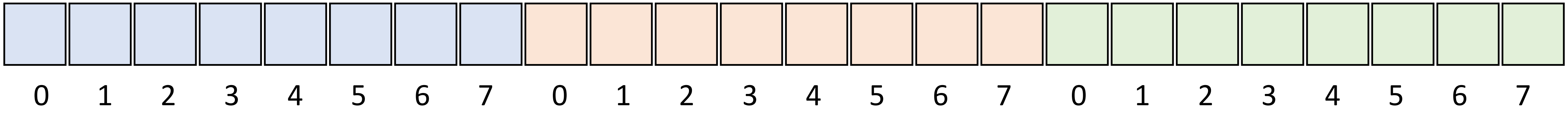


Consider thread ids as a flattened array (which is often how they are used to index memory)

Say we specify 8 threads per block (this can be up to 1024)

How does this work

block 0:
block 1:
block 2:



local thread ids

Consider thread ids as a flattened array (which is often how they are used to index memory)

Say we specify 8 threads per block (this can be up to 1024)

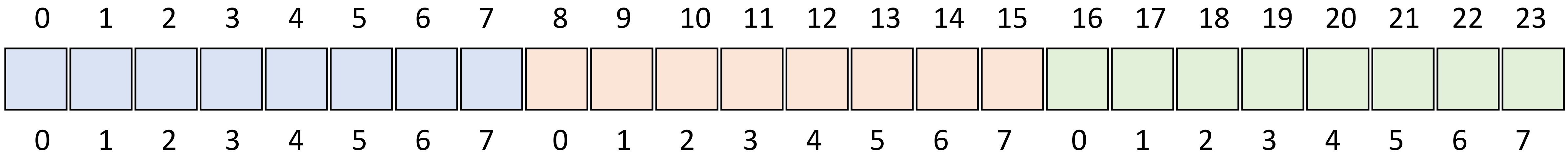
Thread ids are local to a block

Compute global id? `blockIdx.x * blockDim.x + threadIdx.x`

How does this work

block 0:
block 1:
block 2:

global thread ids



local thread ids

Consider thread ids as a flattened array (which is often how they are used to index memory)

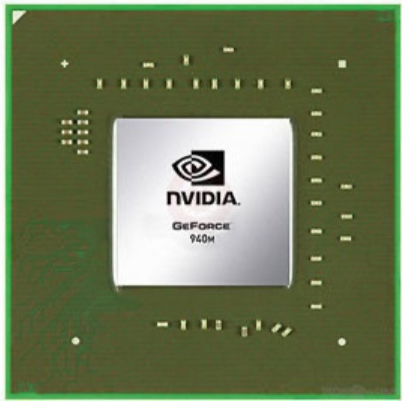
Say we specify 8 threads per block (this can be up to 1024)

Thread ids are local to a block

Compute global id? `blockIdx.x * blockDim.x + threadIdx.x`

Final Round

Tiny GPU in an embedded system



Fight!

The CPU in my professor workstation

Nvidia Jetson Nano (whole chip, CPU + GPU)
2 Billion transistors
10 TDP
Est. $99

Intel i7-9700K
2.16 Billion transistors
95 TDP
Est. $316

https://www.techpowerup.com/gpu-specs/geforce-940m.c2648
https://www.alibaba.com/product-detail/Intel-Core-i7-9700K-8-Cores_62512430487.html
https://www.prolast.com/prolast-elevated-boxing-rings-22-x-22/

WebGPU

- The language is wgsl
  - It is new, there are not many examples (and the specification changes!)
  - Official specification is here: https://www.w3.org/TR/WGSL/

WebGPU

- wgsl is NOT javascript

- Javascript is interpreted: not possible on GPUs

- wgsl is compiled
  - into Vulkan on Linux
  - into Metal on Apple
  - into HLSL on Windows

- No printing (can be difficult to debug)

WebGPU

- variables (optional types):

var <name> = <value>;

```
var cluster_dist = 3.0;
```

```
var <name> : <type> = <value>;
var cluster_dist : f32 = 3.0;
```

WebGPU

```
struct Particle {
    pos : vec2<f32>;
};
```

- types:
  - i32
  - u32
  - f32
  - vec2<f32>
  - array<*type*>

```
struct Particles {
    particles : array<Particle>;
};
```

```
var index_pos : vec2<f32> = particlesA.particles[index].pos;
```

- structures

```
var index : u32 = GlobalInvocationID.x;
```

- Built-ins (global id)

you have one thread for each particle!

WebGPU

- Built in functions:
  - arrayLength
  - sqrt
  - pow
  - distance

WebGPU

For loops:

```
for (var i : u32 = 0u; i < arrayLength(&particlesA.particles); i = i + 1u) {
...
}
```

WebGPU

- Types can be frustrating
- But compiler errors will help you, and you can do casts.

Last day of class!

- I hope after the final you take some time to reflect

Taking a class is like going on a long hike

Mutexes

Mutexes

Concurrent Data structures

Take some time
in the spring break
to enjoy the view!

Thank you!

- You are now all now experts on parallel programming!
- You’re all going to do great on the final!
- Thank you for being such great students!
- See you around!