

CSE113: Parallel Programming

Feb, 28, 2024

- **Topics:**

- Intro to GPUs



Announcements

- Midterm is graded, you should be able to see your grades
 - The TAs will go over the questions and solutions in class next Wednesday with you.
 - This lecture will not be recorded and the slides will not be uploaded.
 - If you want to see your test, please come see me or the TAs in office hours.
 - Sadly we do these things due to academic integrity violations last year.

Announcements

- Last day to turn in HW 3 was yesterday, hope you got it in!
- HW 4 is released today. We will discuss it in class a little bit today
- We will plan for HW 5 to be released on Feb 11.

Announcements

- I am gone tomorrow, so I will not have office hours, which is why I had them yesterday
- Please talk to the TAs (in person) if you want to go over your midterm.

Announcements

- Probably last day on GPUs.
- This is just the tip of the iceberg! Read CUDA by example (free textbook linked on website). Lots of good examples online once you get the basics down.
- Rest of quarter:
 - Memory models
 - Barriers
 - Concurrent Set
 - Schedulers (if time)

Previous quiz + Review

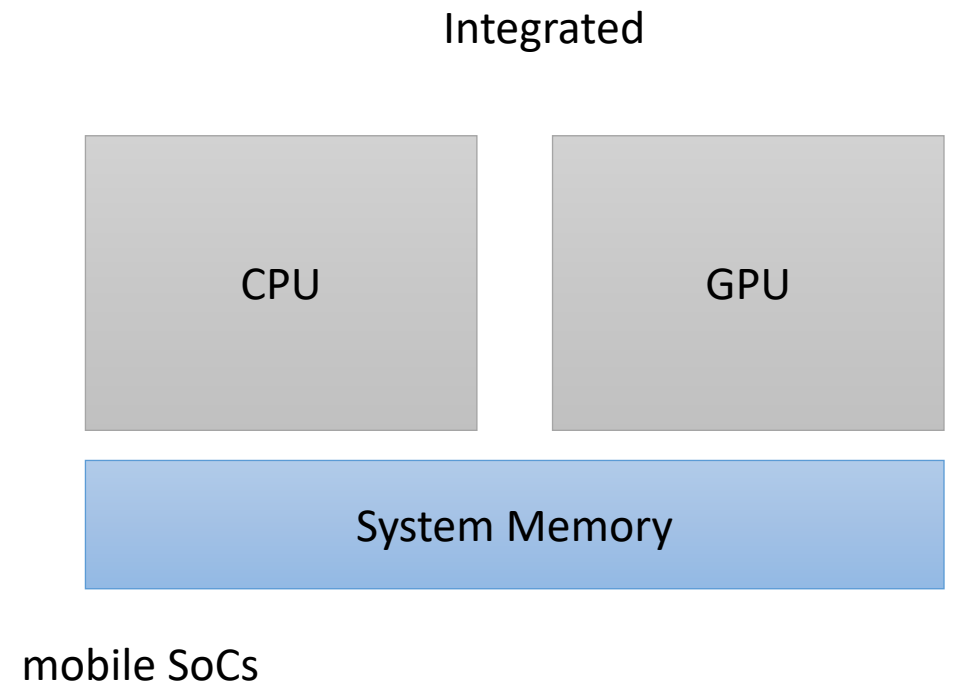
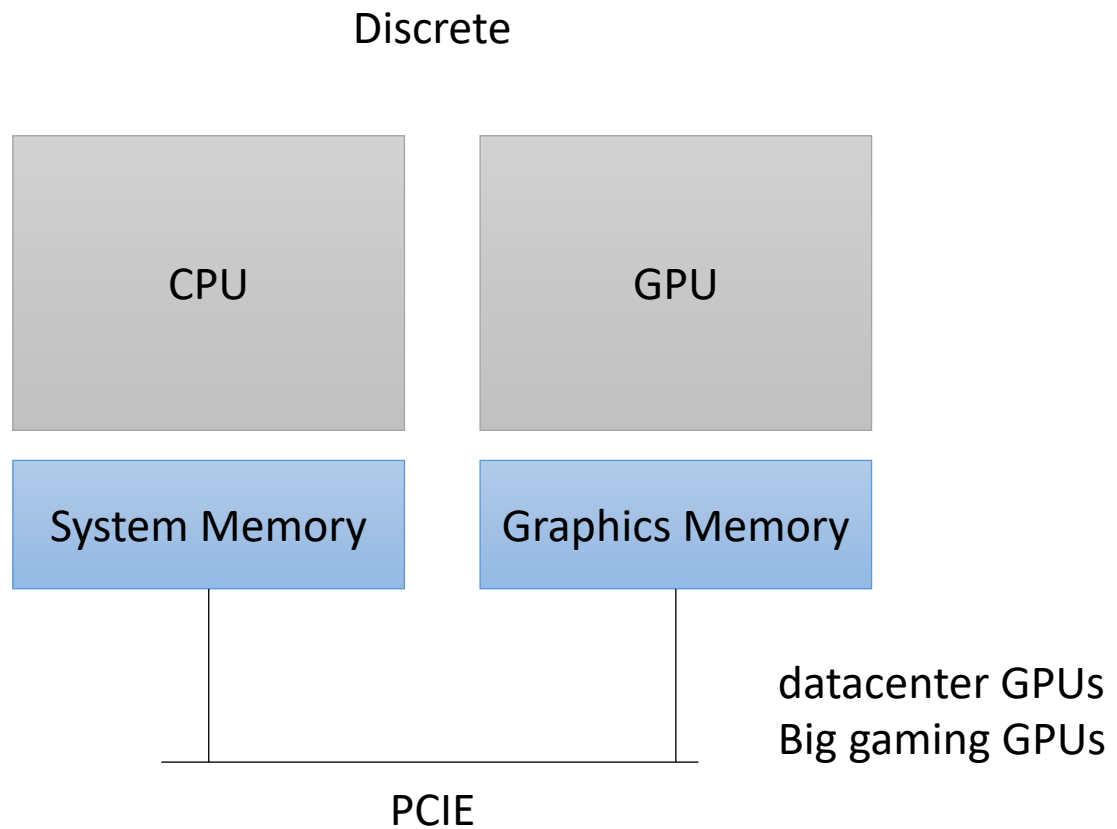
The host (CPU) will write a C++-like program that allocates and sets up memory on the GPU. The host will then call a GPU program called a kernel. Is this affirmation true?

☐ True

☐ False

GPU set up

- GPUs come in two flavors

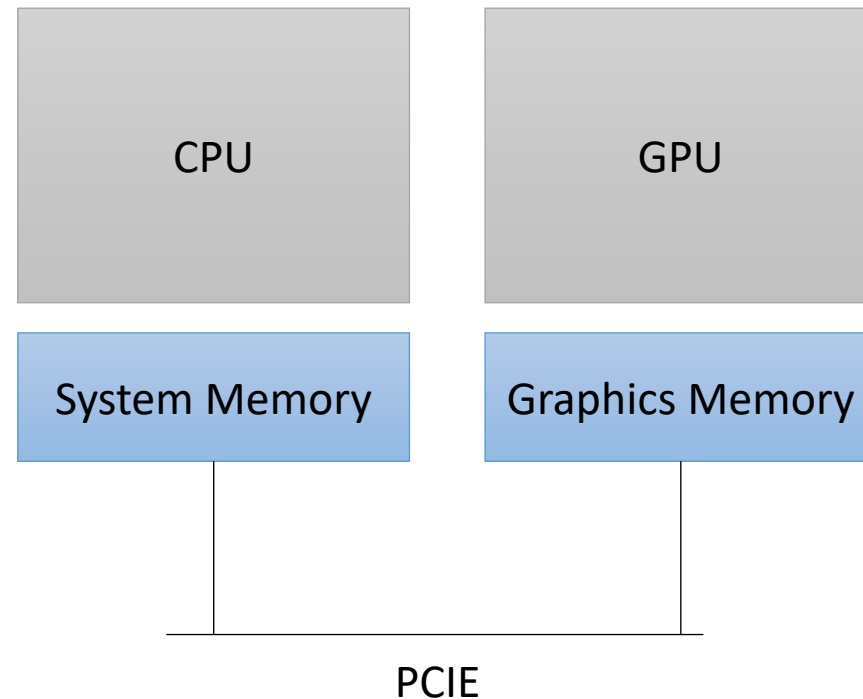


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

```
int *x = (int*) malloc(sizeof(int)*SIZE);
```

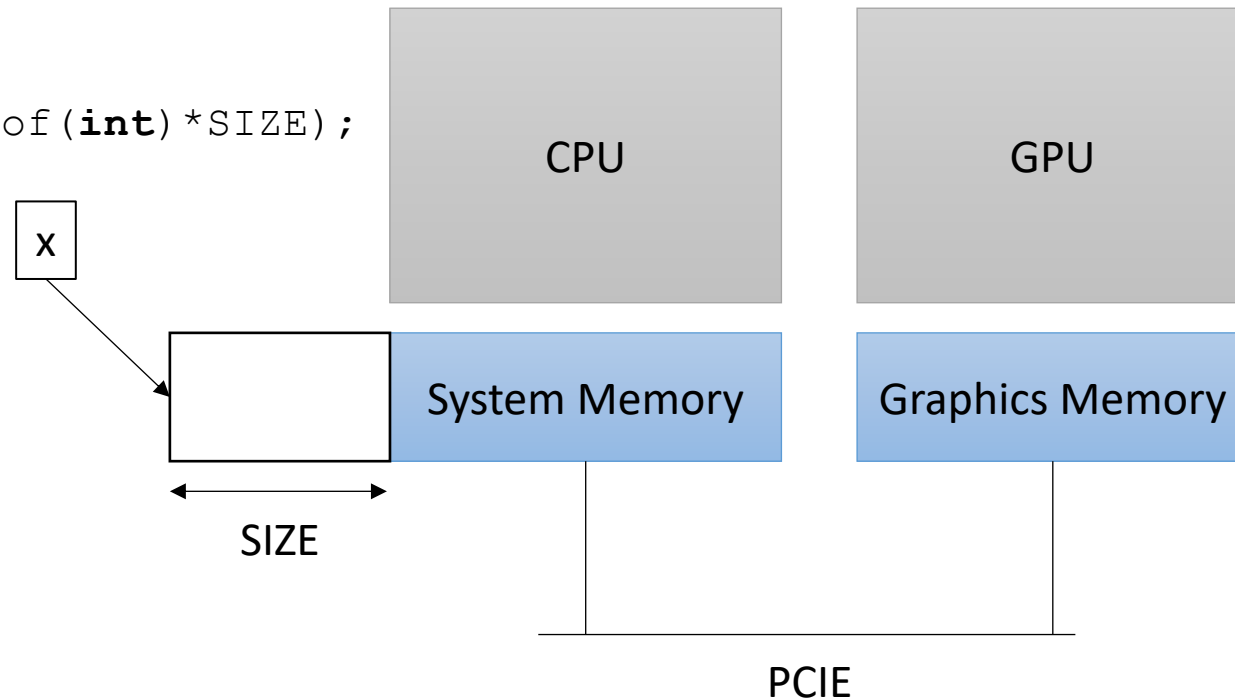


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

```
int *x = (int*) malloc(sizeof(int)*SIZE);
```

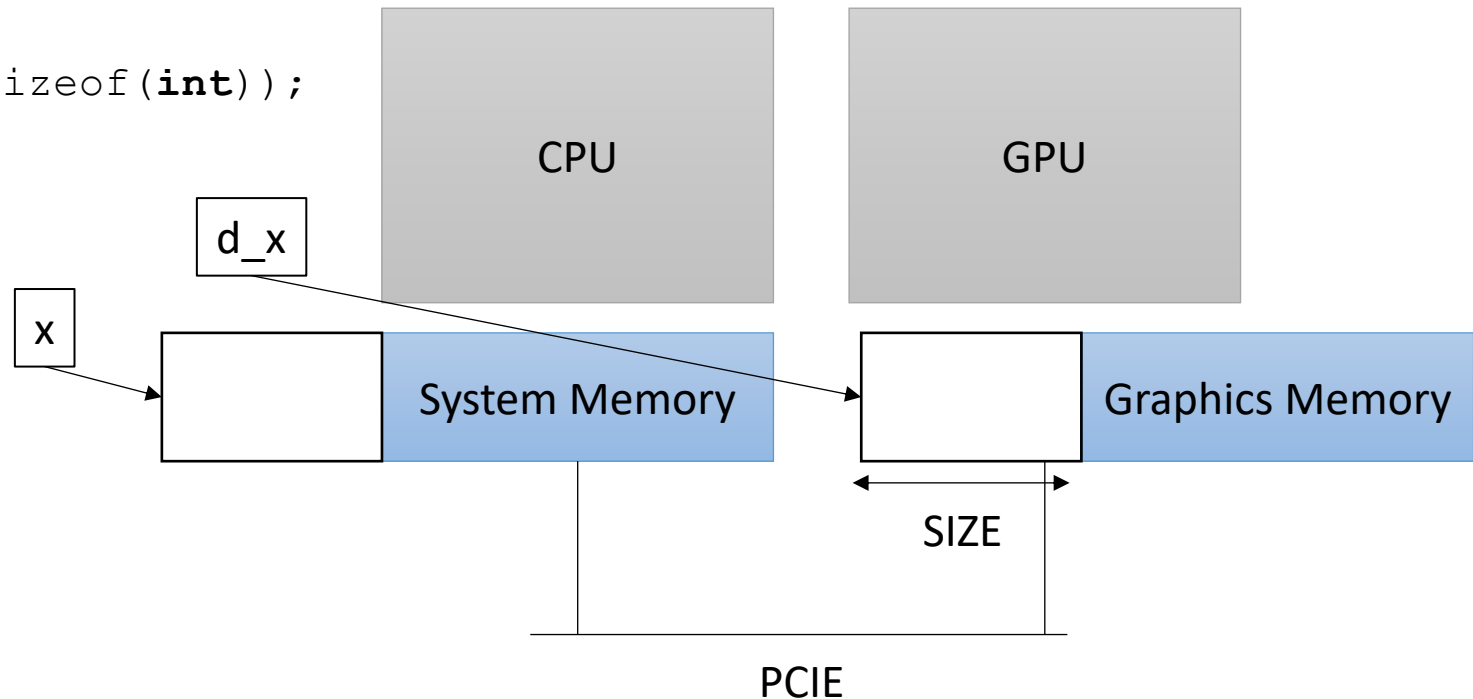


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

```
int *d_x;  
cudaMalloc(&d_x, SIZE*sizeof(int));
```



GPU set up

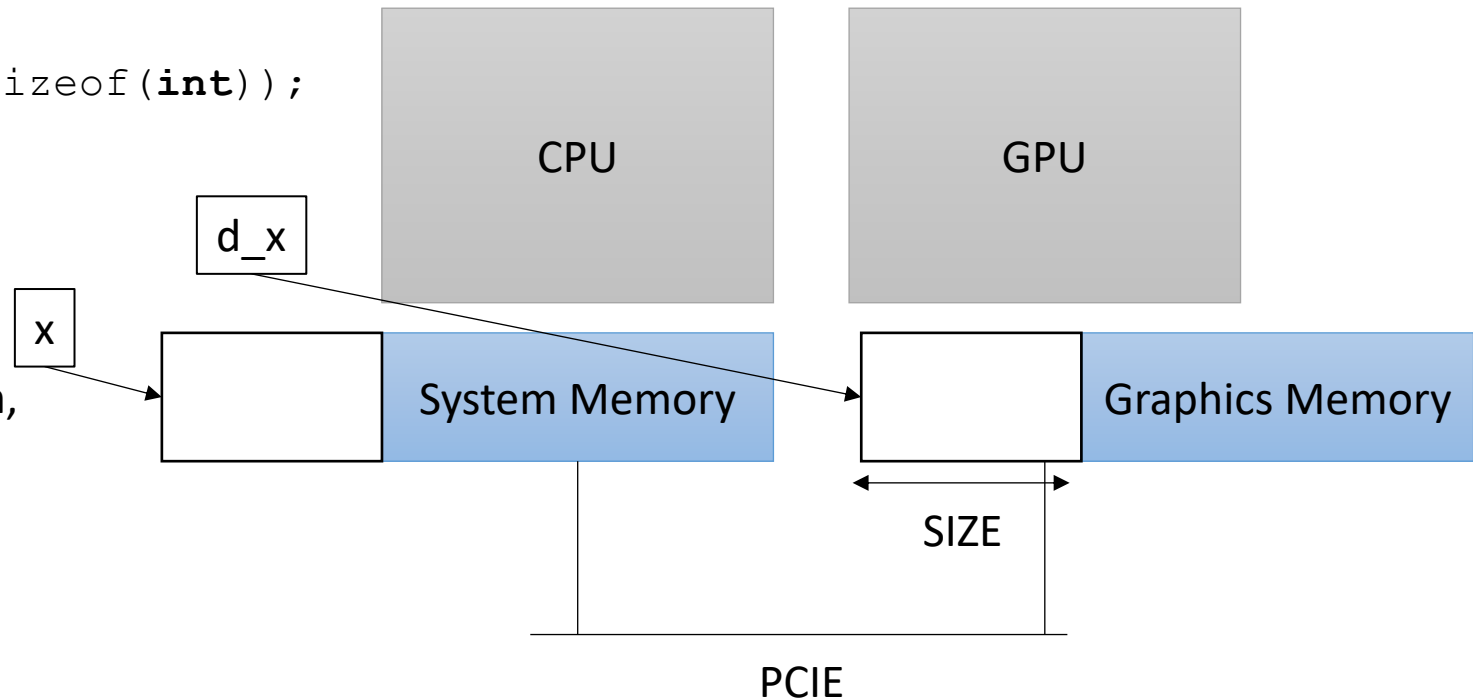
We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

```
int *d_x;  
cudaMalloc(&d_x, SIZE*sizeof(int));
```

`d_x` is a pointer, in the CPU program, that points to memory on the GPU.

We can pass the pointer around, but the CPU cannot access the data
i.e. `d_x[0]` gives an error!



Previous quiz + Review

How do we initialize memory for a variable we aim to use in the GPU computation?

- ☐ Using cudaMemcpy
- ☐ Using memcpy from C++
- ☐ Just declaring a new variable

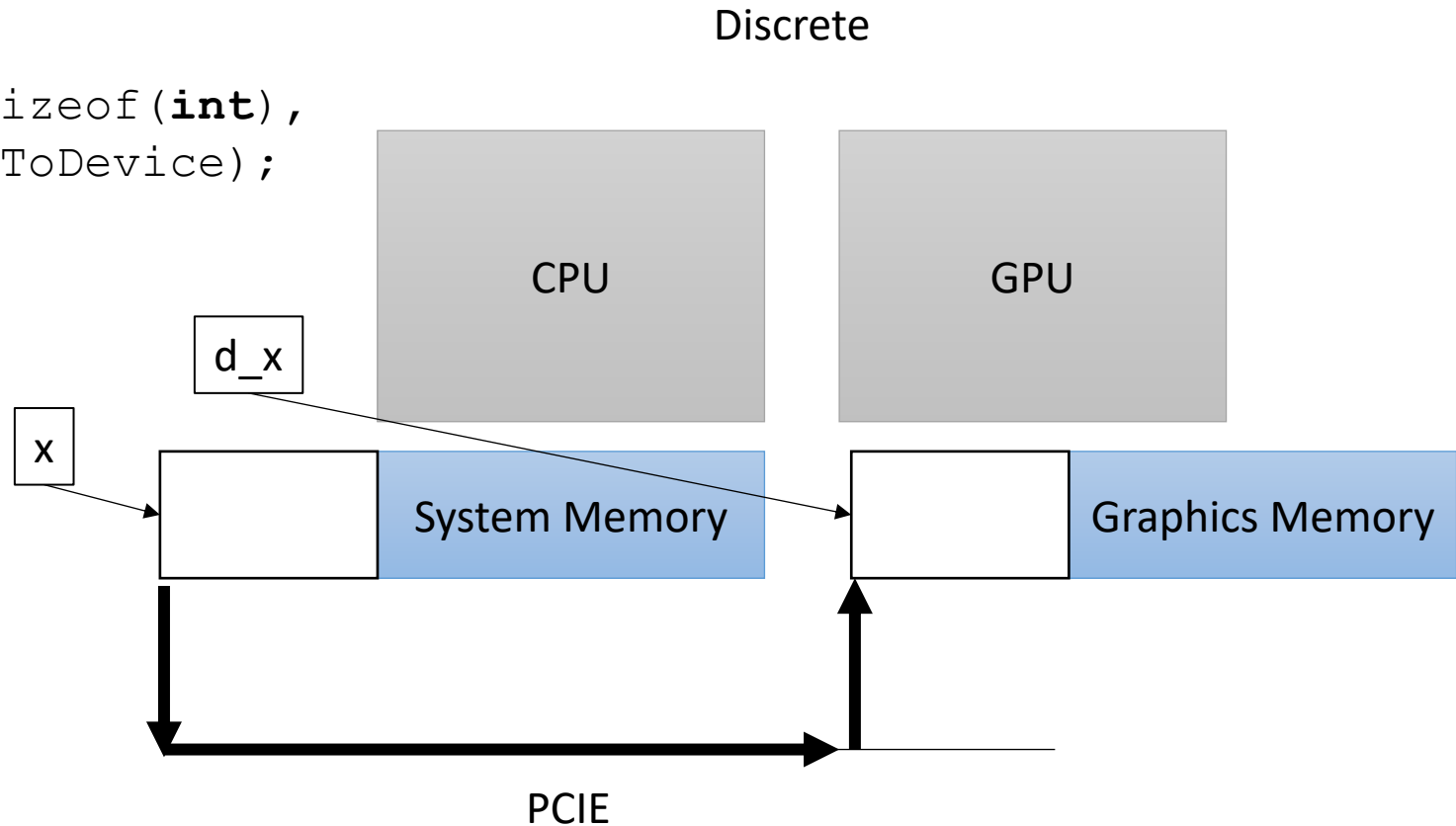
GPU set up

- Our heterogeneous, parallel, programming model

If we can't access `d_x` on the CPU, how do we initialize the memory?

GPU has no access to input devices e.g. disk

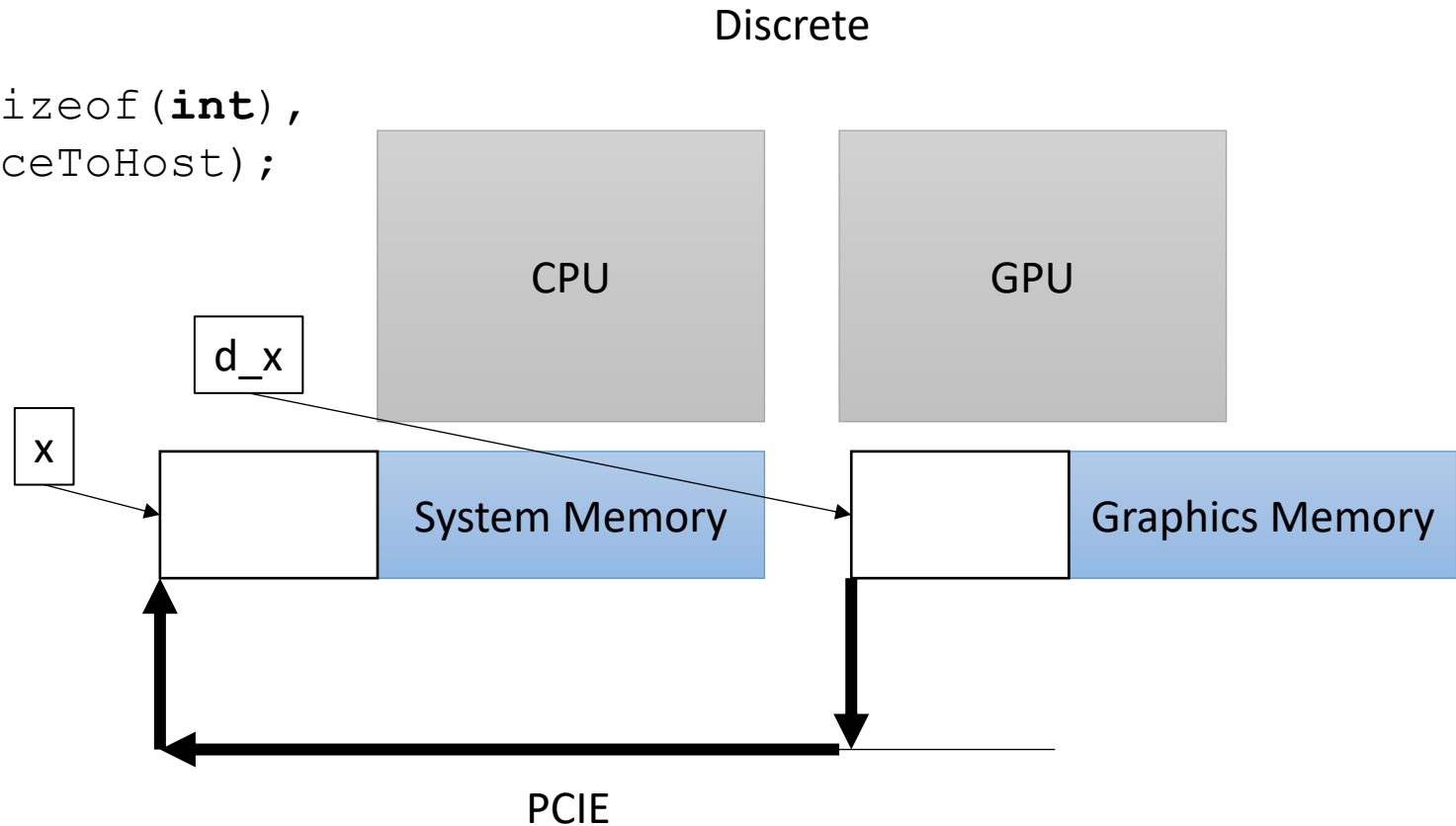
```
//initialize x on host  
cudaMemcpy(d_x, x, SIZE*sizeof(int),  
           cudaMemcpyHostToDevice);
```



GPU set up

- Our heterogeneous, parallel, programming model

```
//initialize x on host  
cudaMemcpy(x, d_x, SIZE*sizeof(int),  
           cudaMemcpyDeviceToHost);
```



Previous quiz + Review

What keyword do we need to include for a function to be executed on the GPU using CUDA?

☐ `__kernel__`

☐ `__global__`

☐ `__this__`

☐ `__gpu__`

The GPU Program

```
__global__ void vector_add(int * a, int * b, int * c, int size) {  
    for (int i = 0; i < size; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```


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

Kernel constraints

- What can and can't you do in a GPU kernel?

Kernel constraints

- What can and can't you do in a GPU kernel?
 - Print?
 - File I/O?
 - C++ standard library? E.g. vectors?
 - Memory allocation?
 - Atomics?

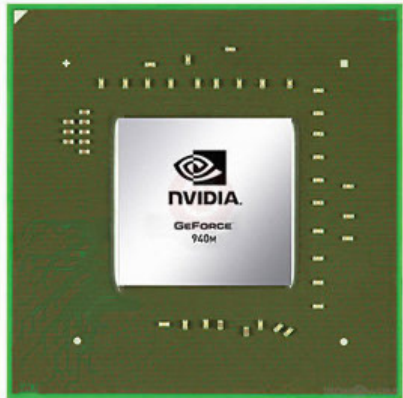
Previous quiz + Review

Using a few sentences, give examples of workloads that benefit from GPU parallelism.

Recalling where we left off

Programming a GPU

Tiny GPU in an
embedded system



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

Fight!



<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/>

The CPU in
my research
workstation



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

Programming a GPU

- The problem: Vector addition

Parallel Schedules

array a



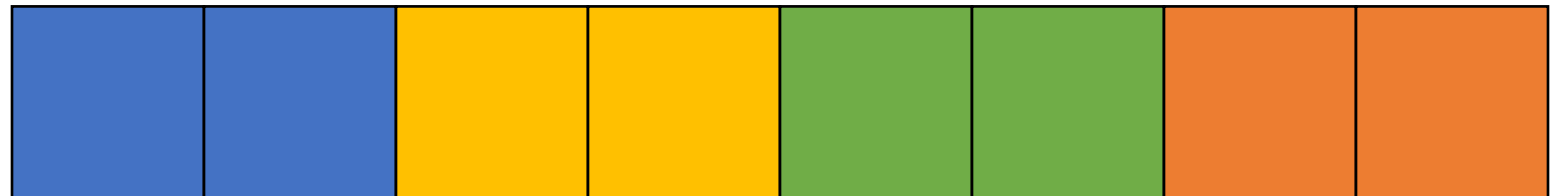
+ + + + + + + +

array b



= = = = = = = =

array c



Lets set up the CPU

- CPU code

First GPU attempt

- Basic sequential GPU program
 - Issues?
 - Key insights?

woah, 32 cores!

We should parallelize our application!



Second GPU attempt

- Use 32 cores
 - Issues?
 - Key insights?

GPU Memory

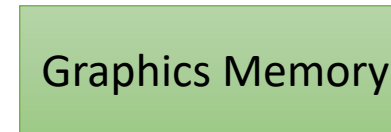
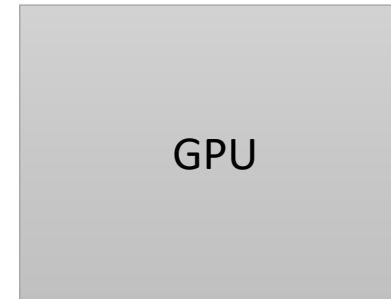
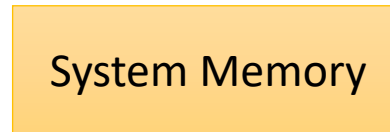
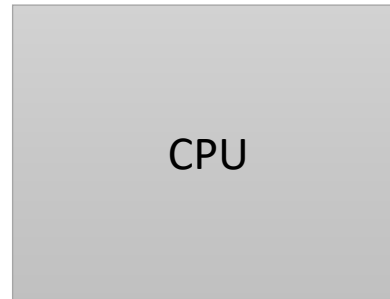
CPU Memory:

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

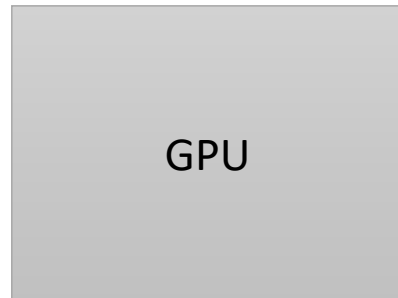
*16-lane highway on a windy
road driven by semi trucks*

Preemption and concurrency?

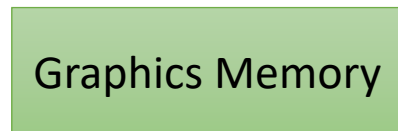
warp 0

A blue rectangular box containing the text "warp 0".

GPU

A gray rectangular box containing the text "GPU".

Graphics Memory

A green rectangular box containing the text "Graphics Memory".

Preemption and concurrency?

warp 0

all threads load from memory.

GPU

Graphics Memory

Preemption and concurrency?

warp 0

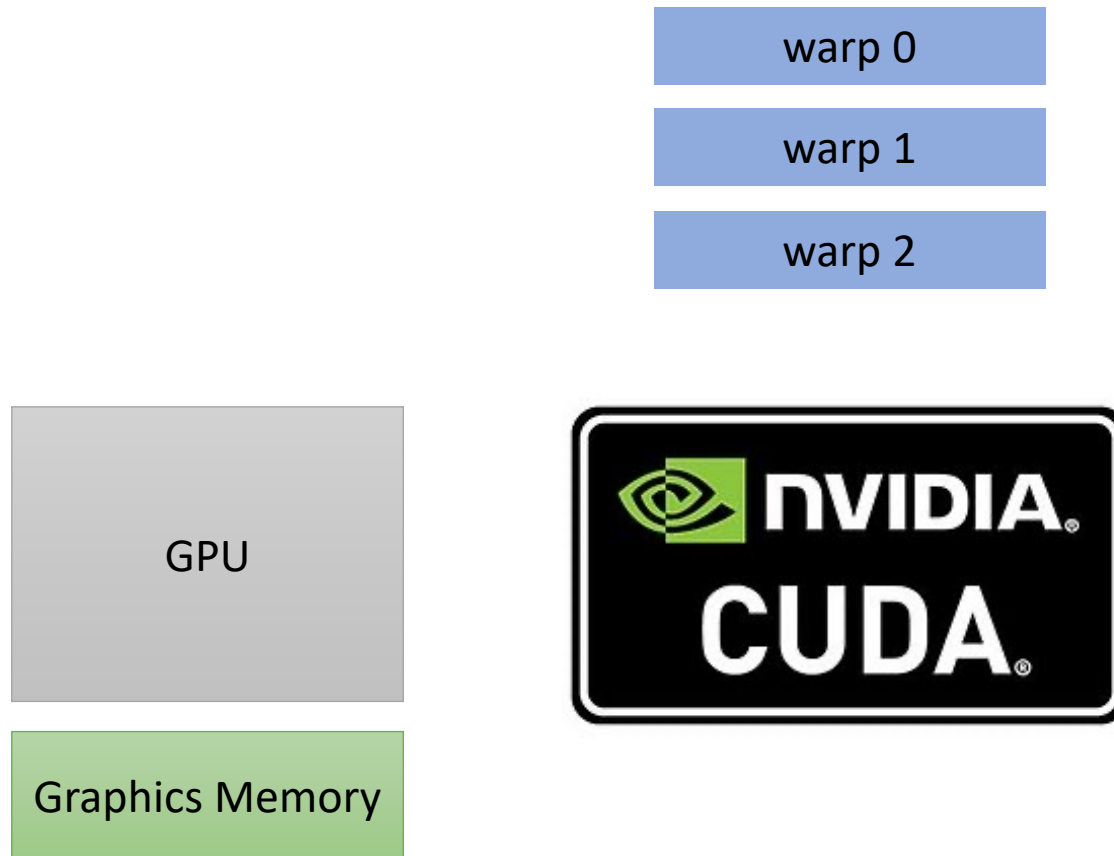
all threads load from memory.

600 cycles!

GPU

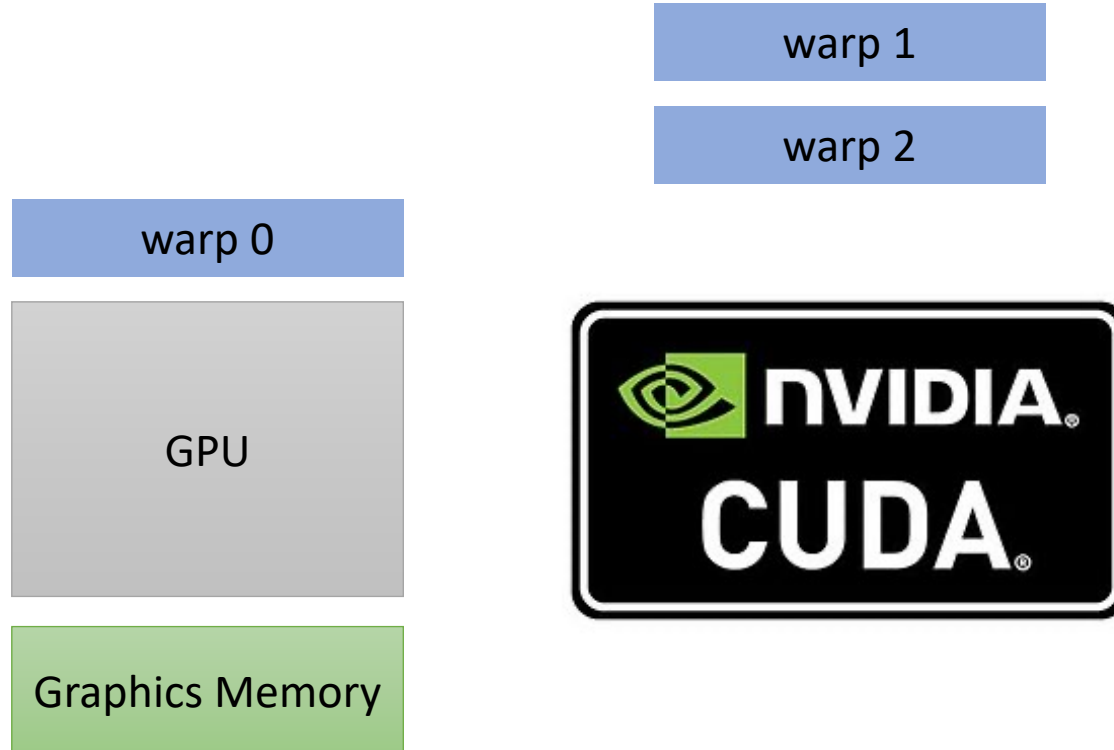
Graphics Memory

Preemption and concurrency?



We can hide latency through
preemption and concurrency!

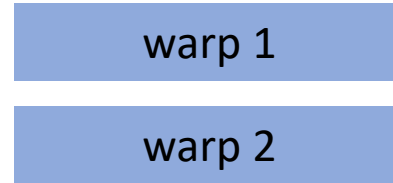
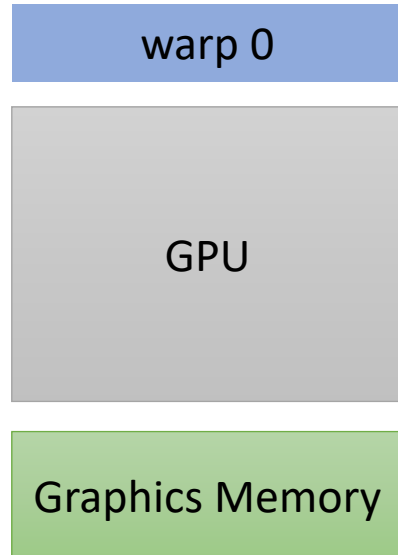
Preemption and concurrency?



We can hide latency through
preemption and concurrency!

Preemption and concurrency?

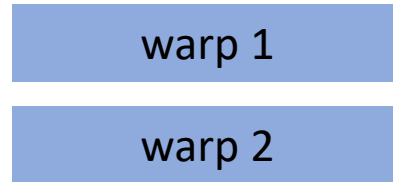
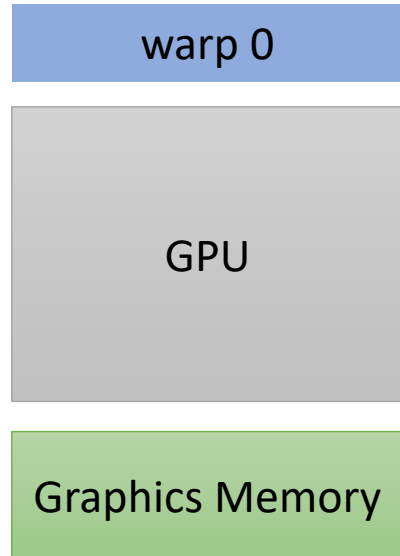
memory access
600 cycles



We can hide latency through
preemption and concurrency!

Preemption and concurrency?

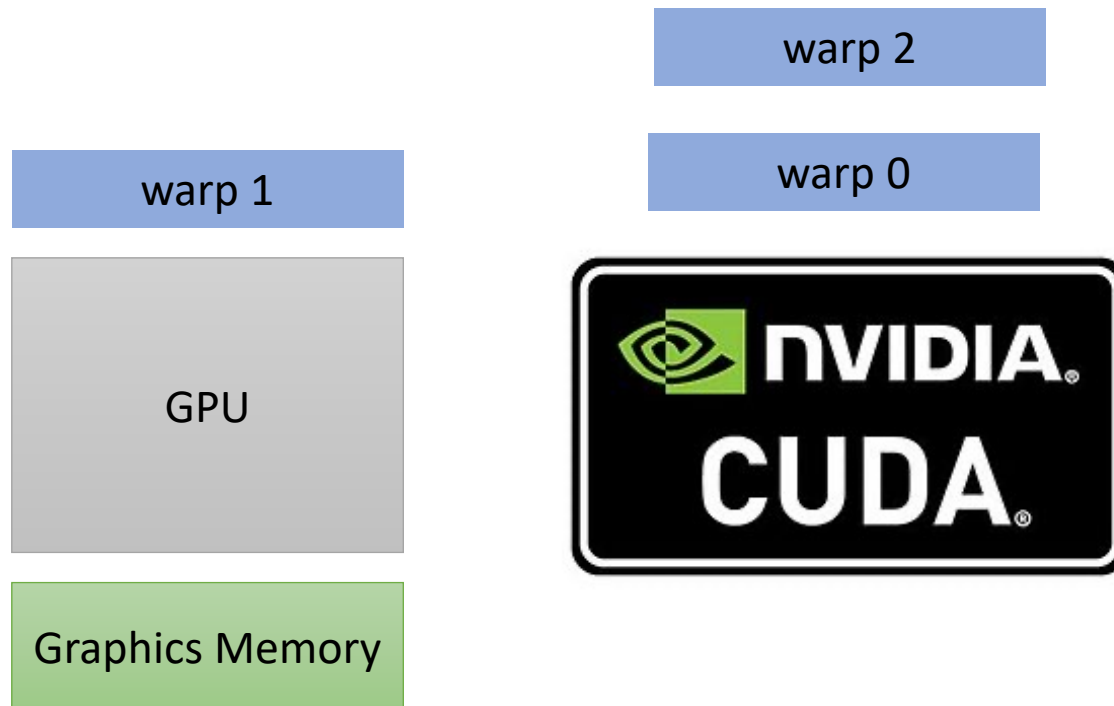
memory access
600 cycles



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

warp 1

GPU

Graphics Memory

warp 2

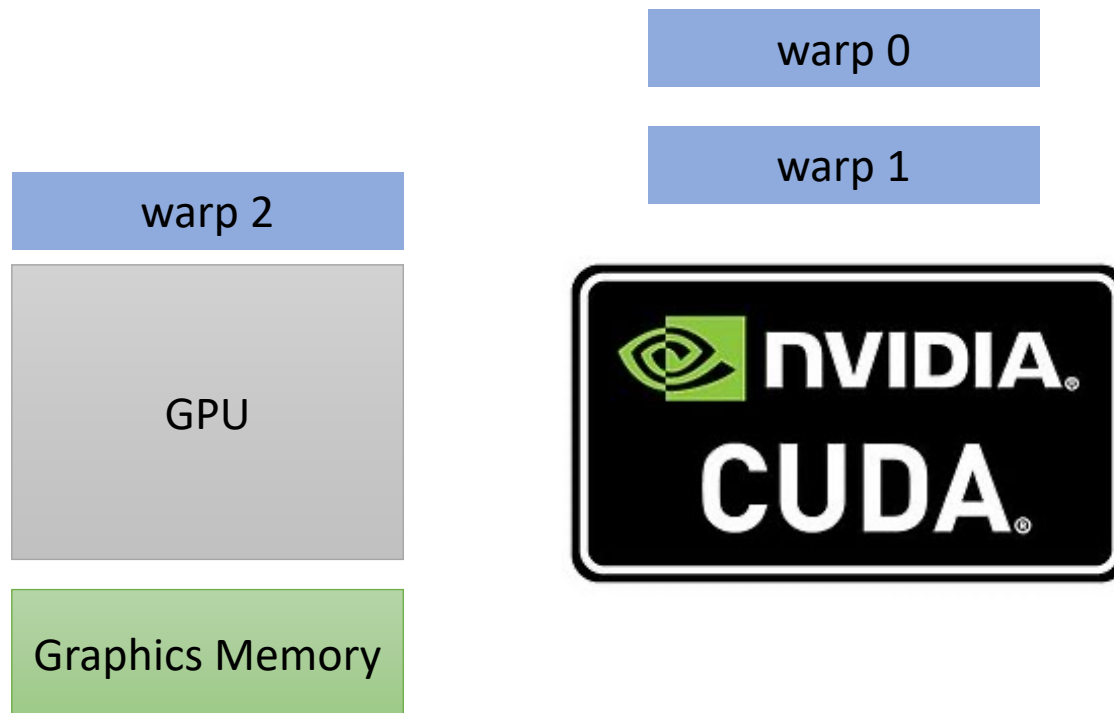
warp 0



preempt warp 1
and put warp 2 on

We can hide latency through
preemption and concurrency!

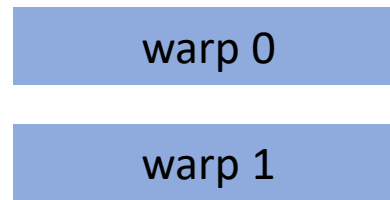
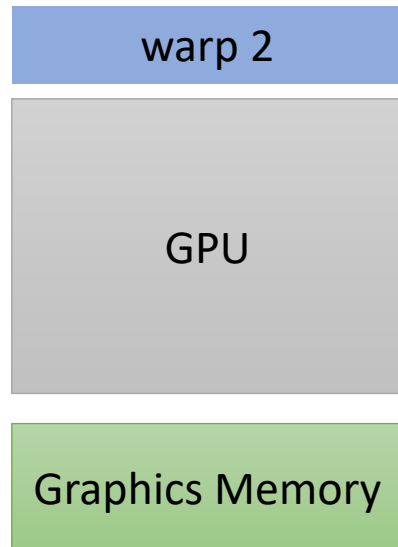
Preemption and concurrency?



We can hide latency through
preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

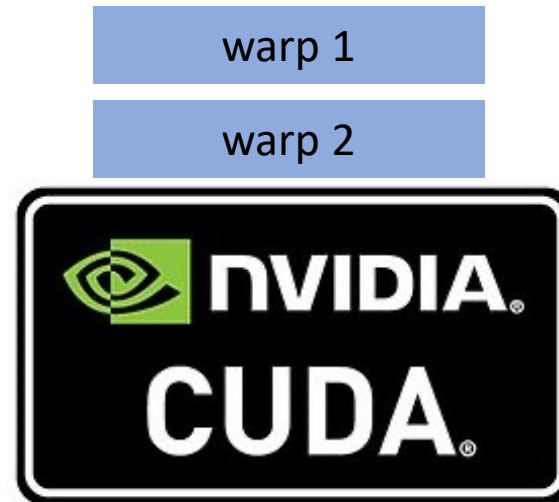
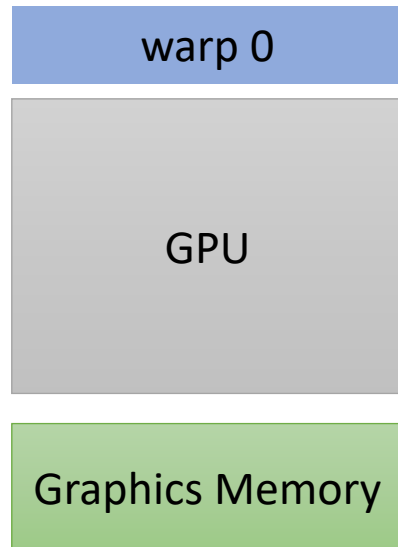


preempt warp 2
and put warp 0 on

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);
```

Concurrent warps

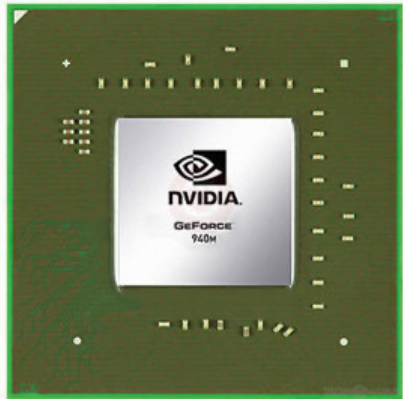
Lets try it! What do we think?

Next steps

- Back to the architecture

These cores don't come for free...

Tiny GPU in an
embedded system



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

Fight!



<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/>

The CPU in
my research
workstation

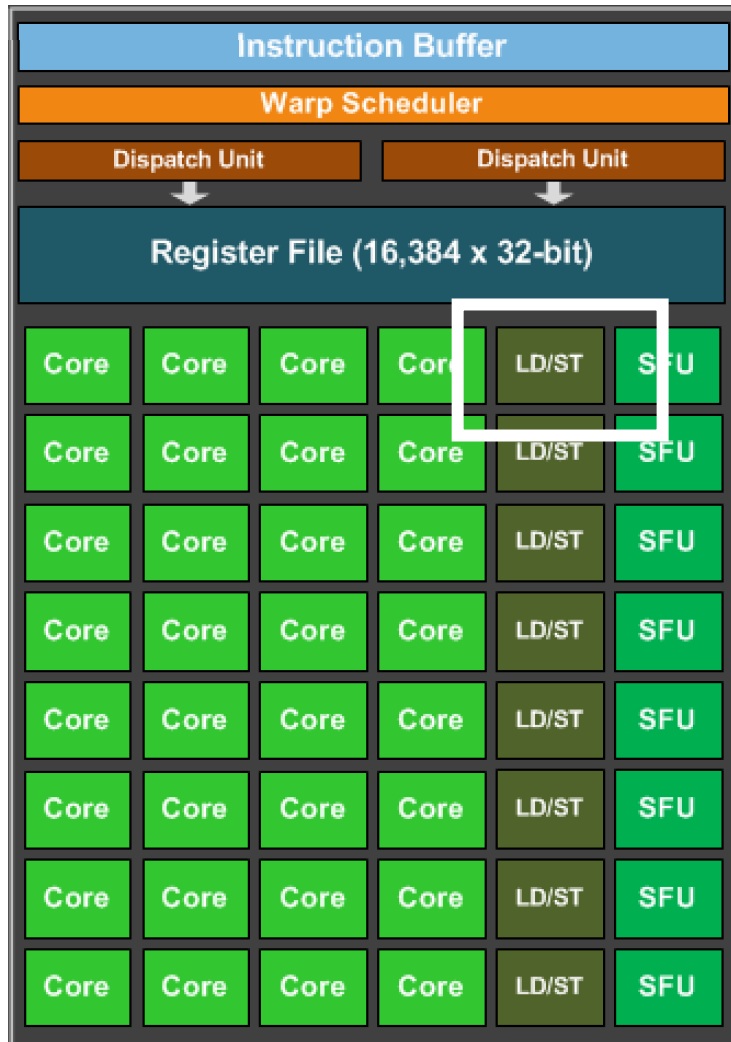


Intel i7-9700K
2.16 Billion transistors
95 TDP
Est. \$316

Optimizing memory accesses



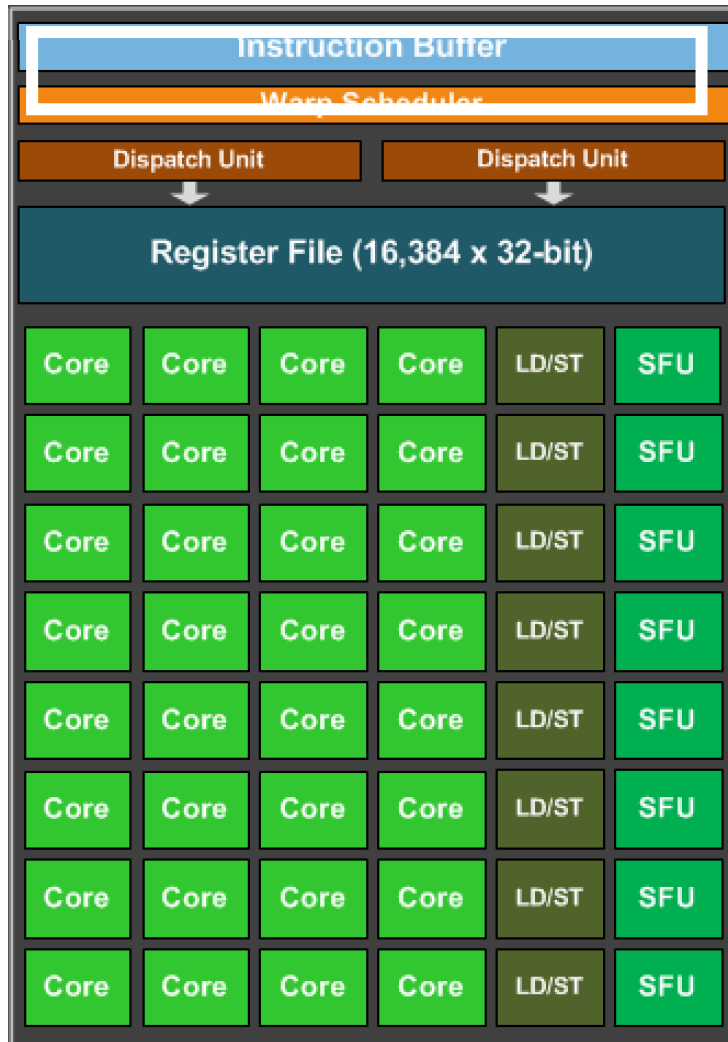
Optimizing memory accesses



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

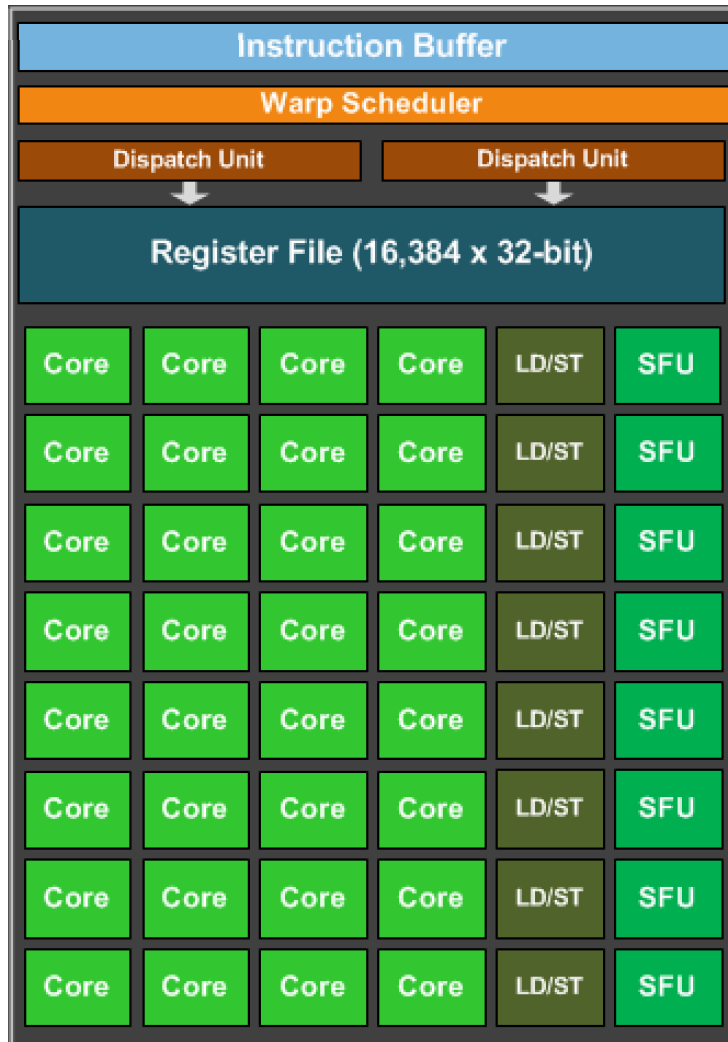


This is the instruction cache... Why doesn't every core have a instruction buffer to keep track of its program?

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

Why doesn't every core have one?

Warp execution



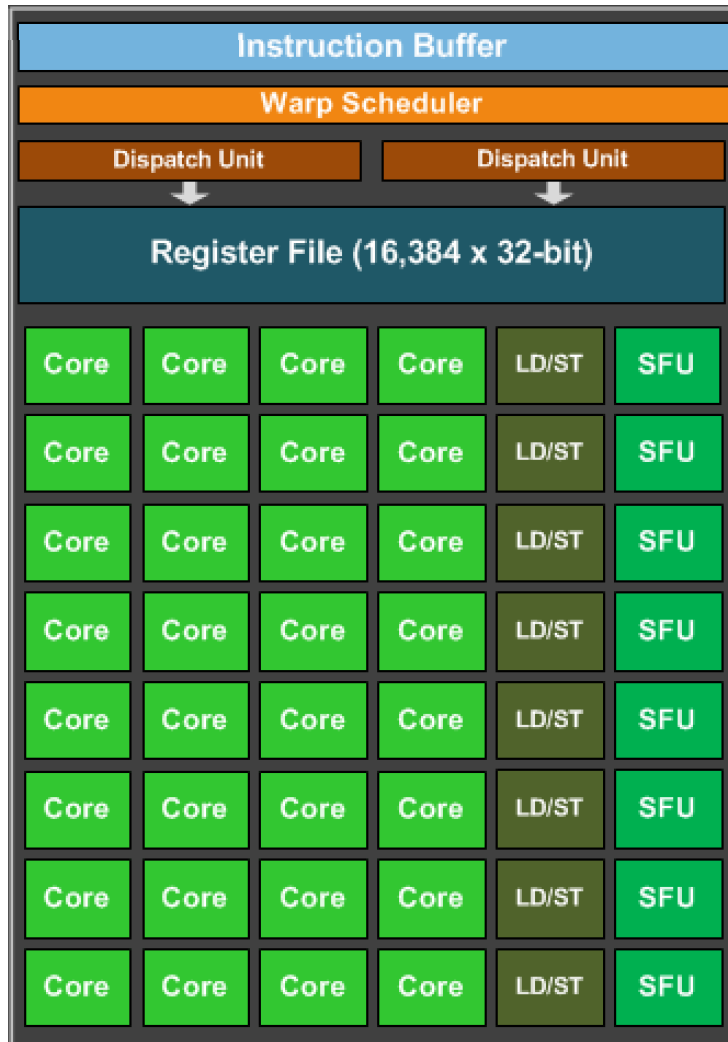
Groups of 32 threads are called a “warp”

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



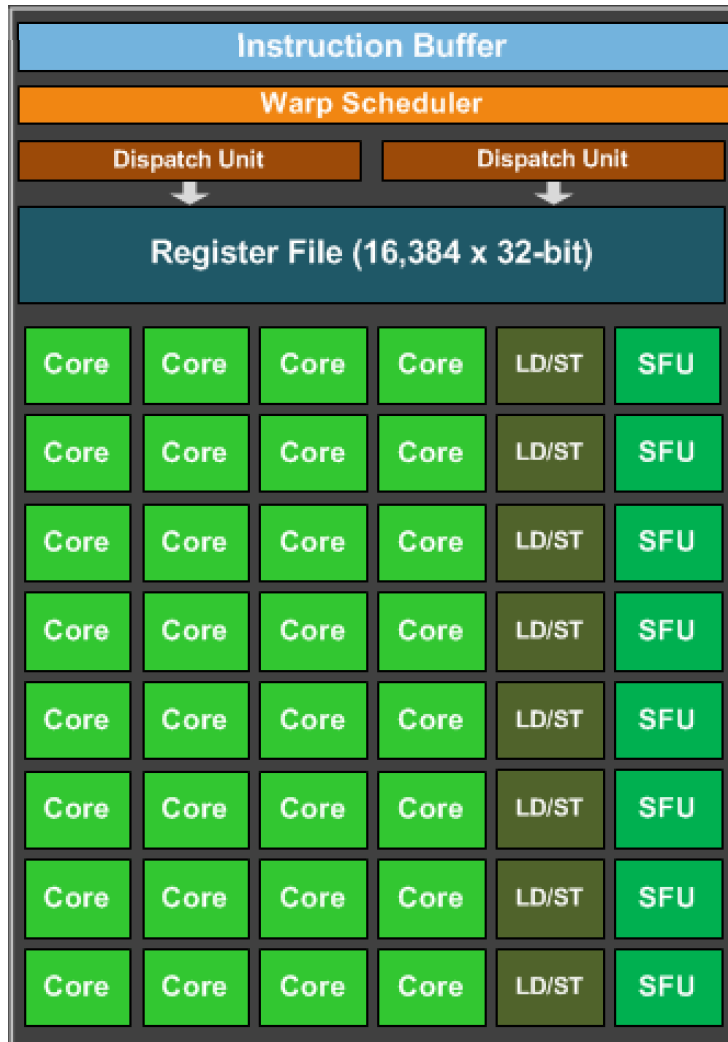
Program:

```
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”

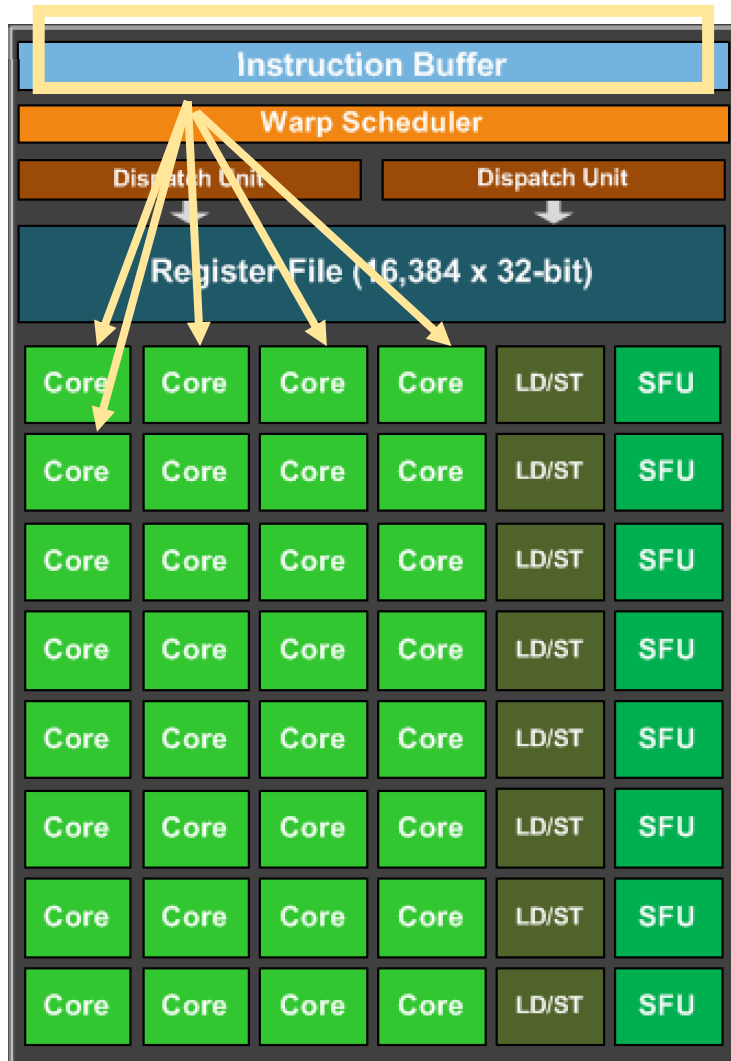
They are executed in lock-step, i.e. they all execute the same instruction at the same time



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.

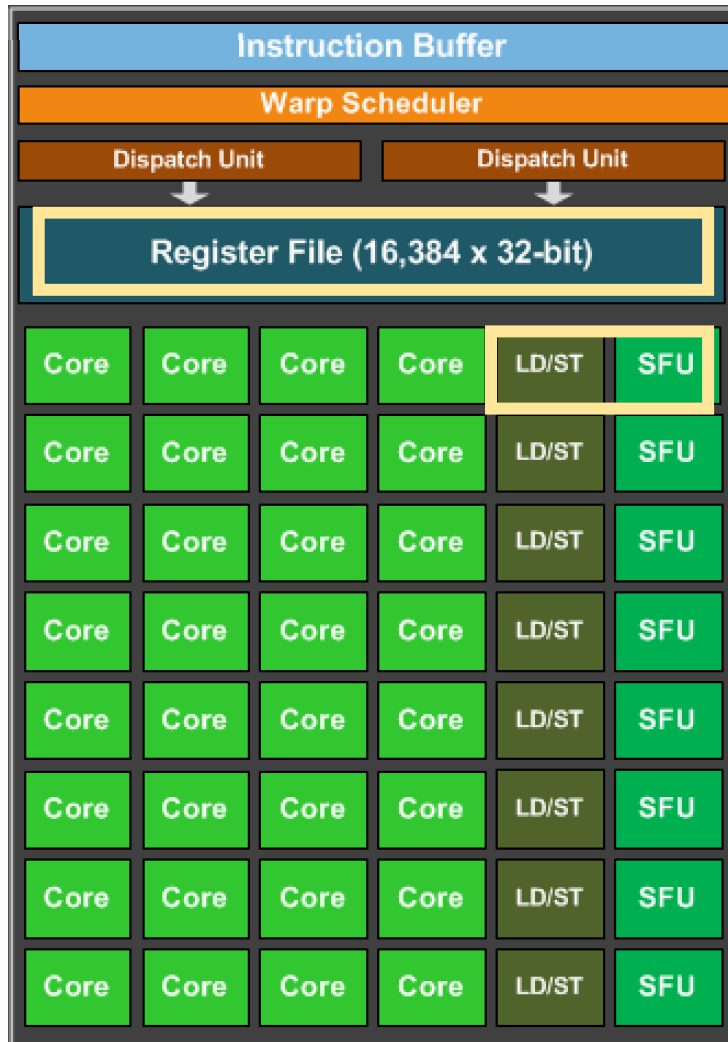
Program:

```
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)

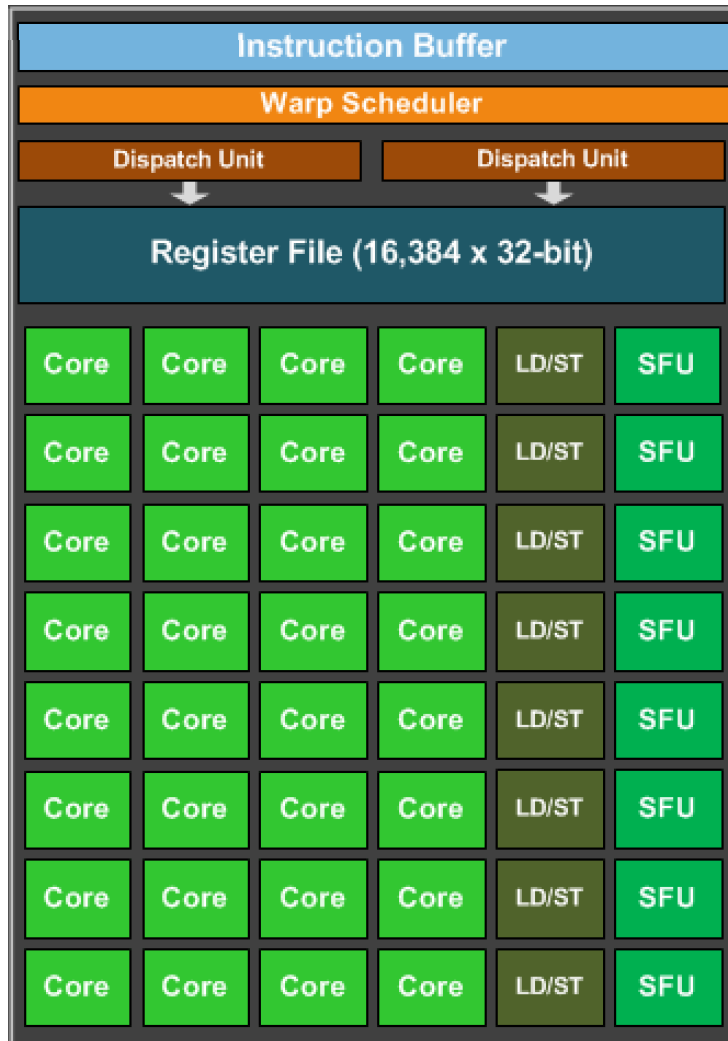
Program:

```
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

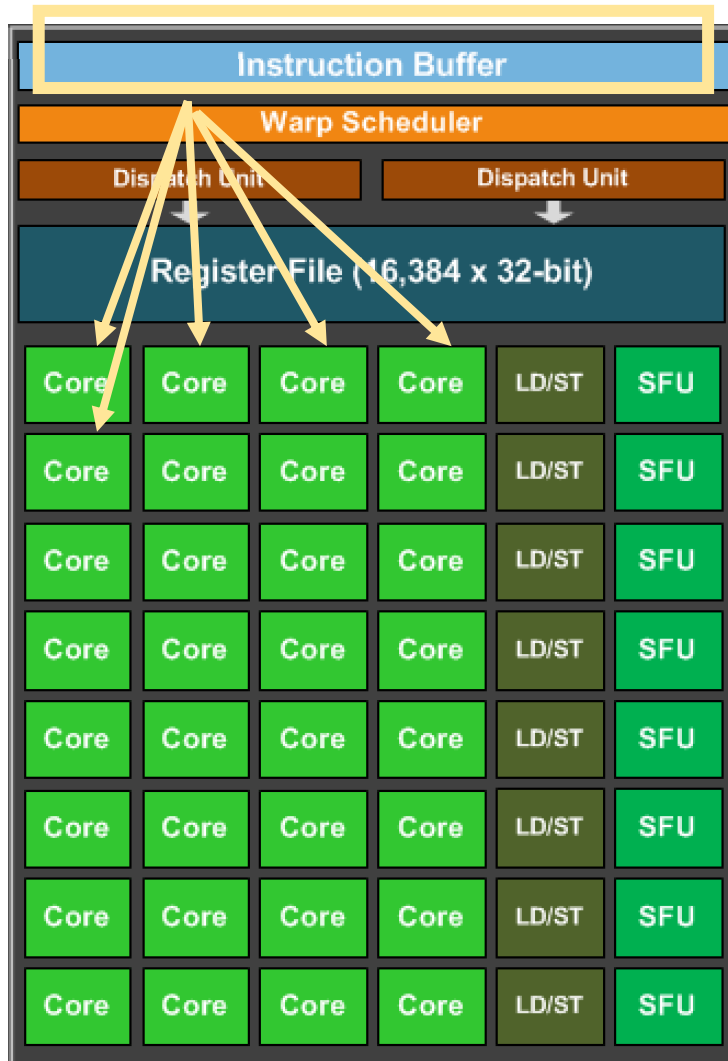


All cores need to wait until all cores finish the first instruction

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:

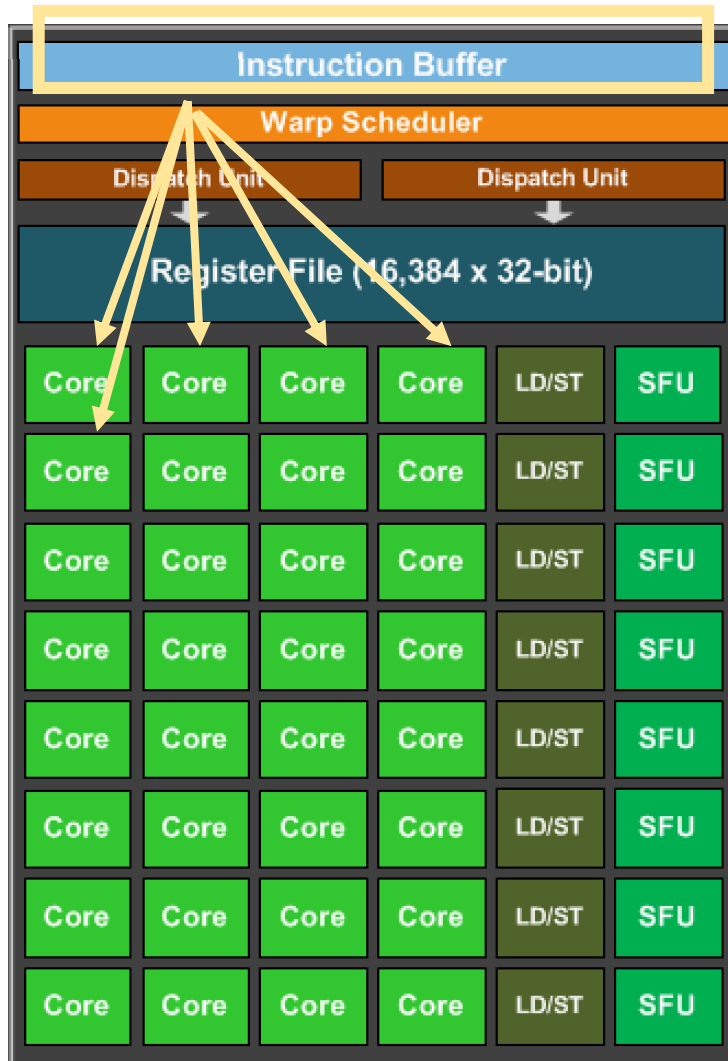
```
int variable1 = b[0];  
int variable2 = c[0];  
int variable3 = variable1 + variable2;  
a[0] = variable3;
```

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;
```

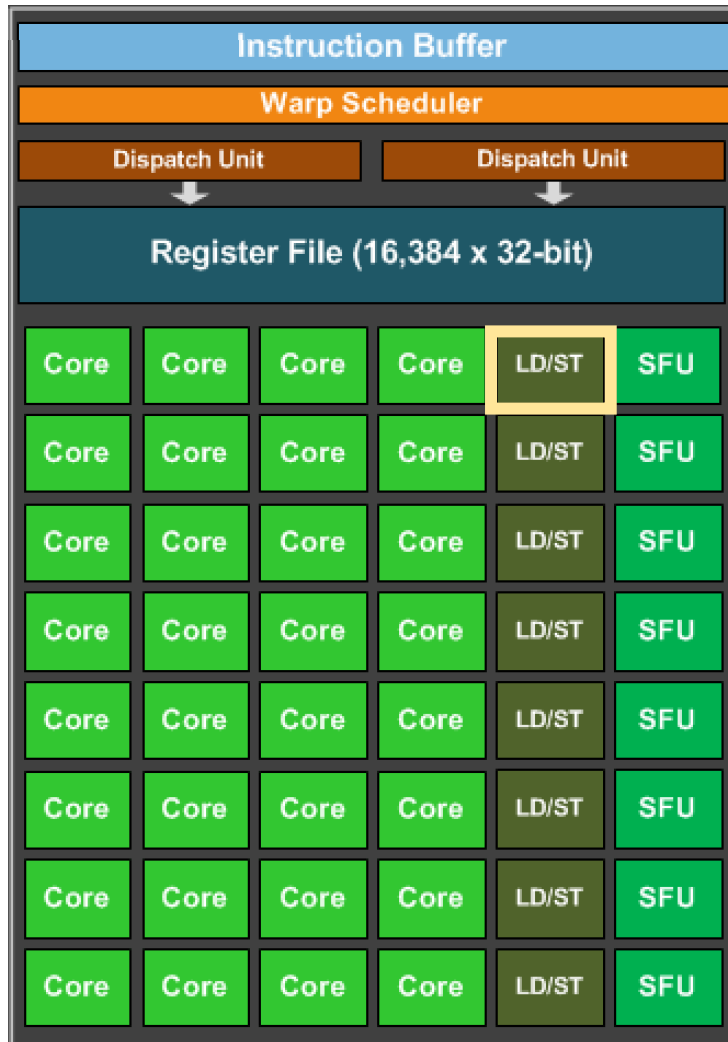
Why would we have a programming model like this?

More cores (share program counters)

Can be efficient to share other hardware resources

Warp execution

Lets look closer at memory

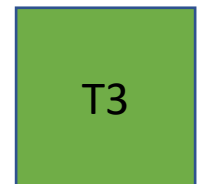
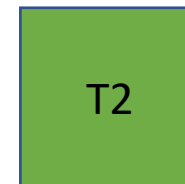
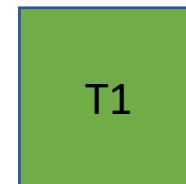
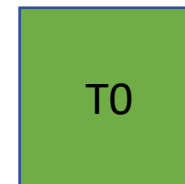
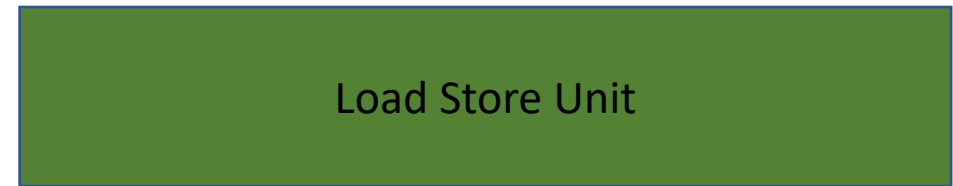
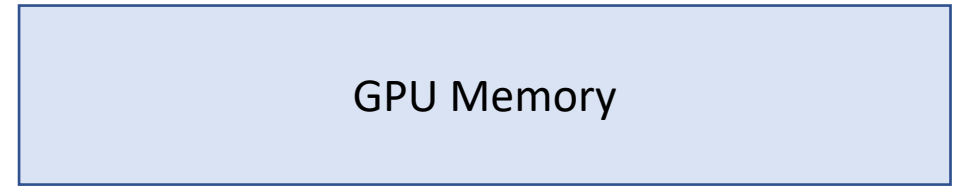


Program:

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

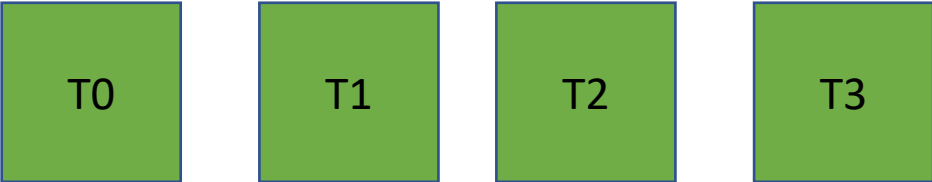
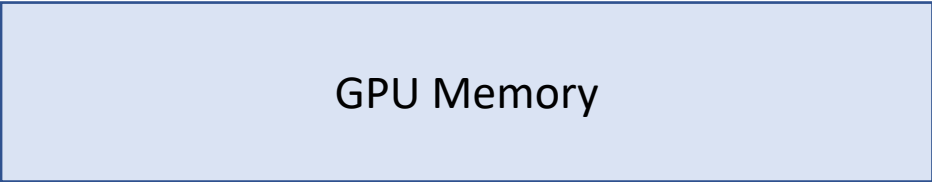
4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen



4 cores are accessing memory. What can happen

All read the same value

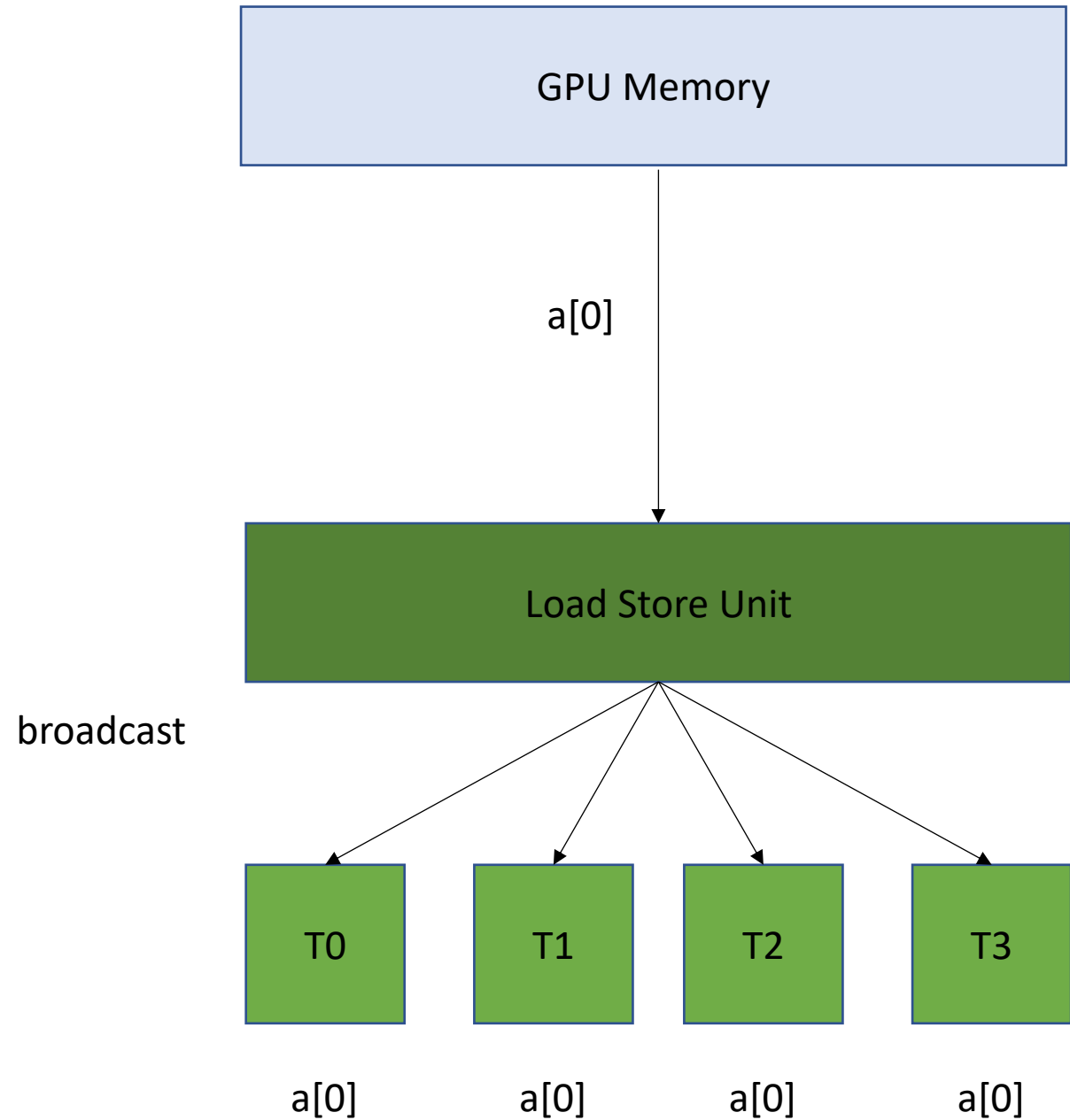


a[0] a[0] a[0] a[0]

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.



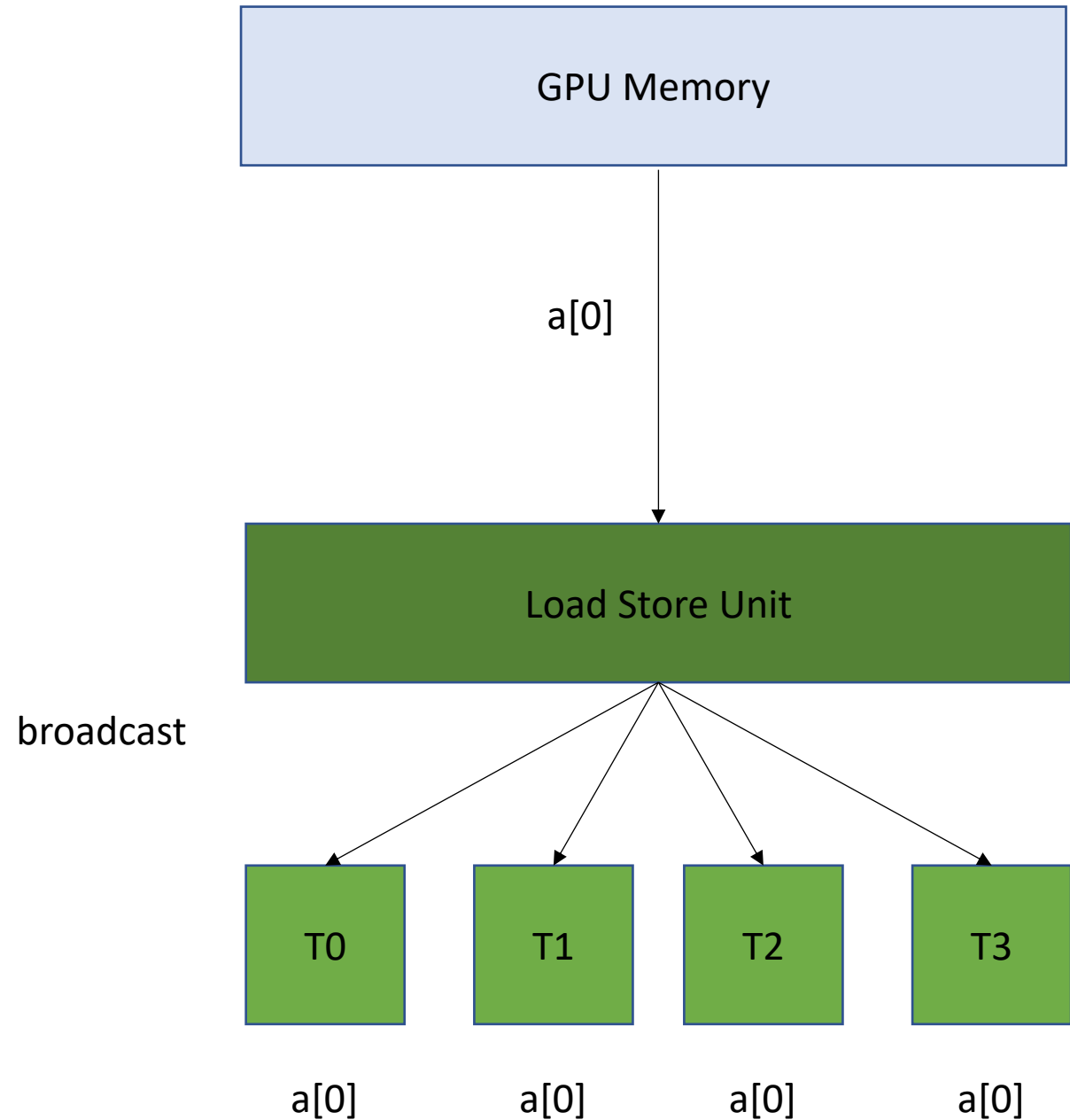
4 cores are accessing memory. What can happen

All read the same value

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

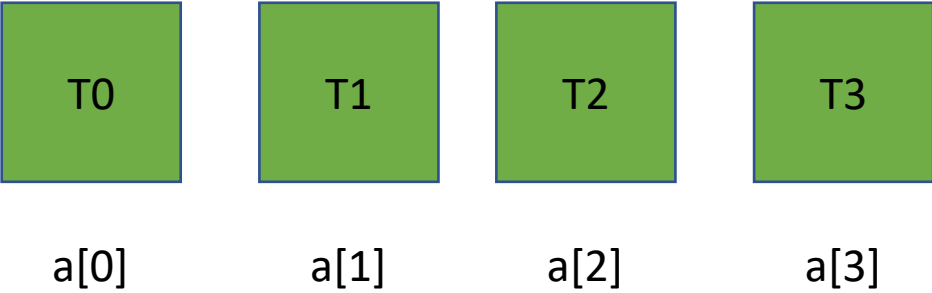
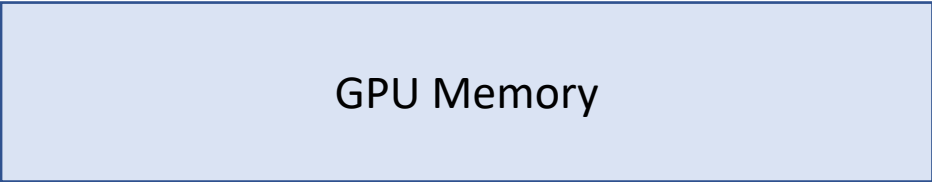
1 request to GPU memory

Efficient, but probably not too common.



4 cores are accessing memory. What can happen

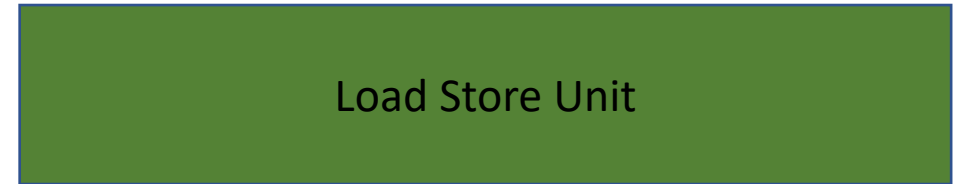
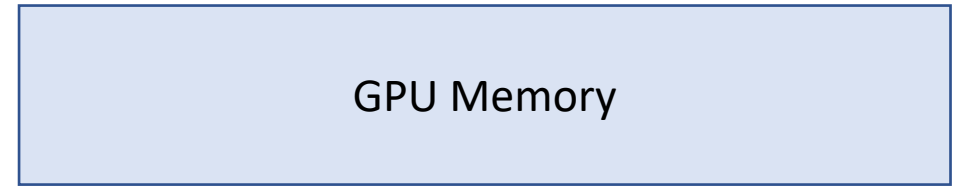
Read contiguous values



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



a[0]

a[1]

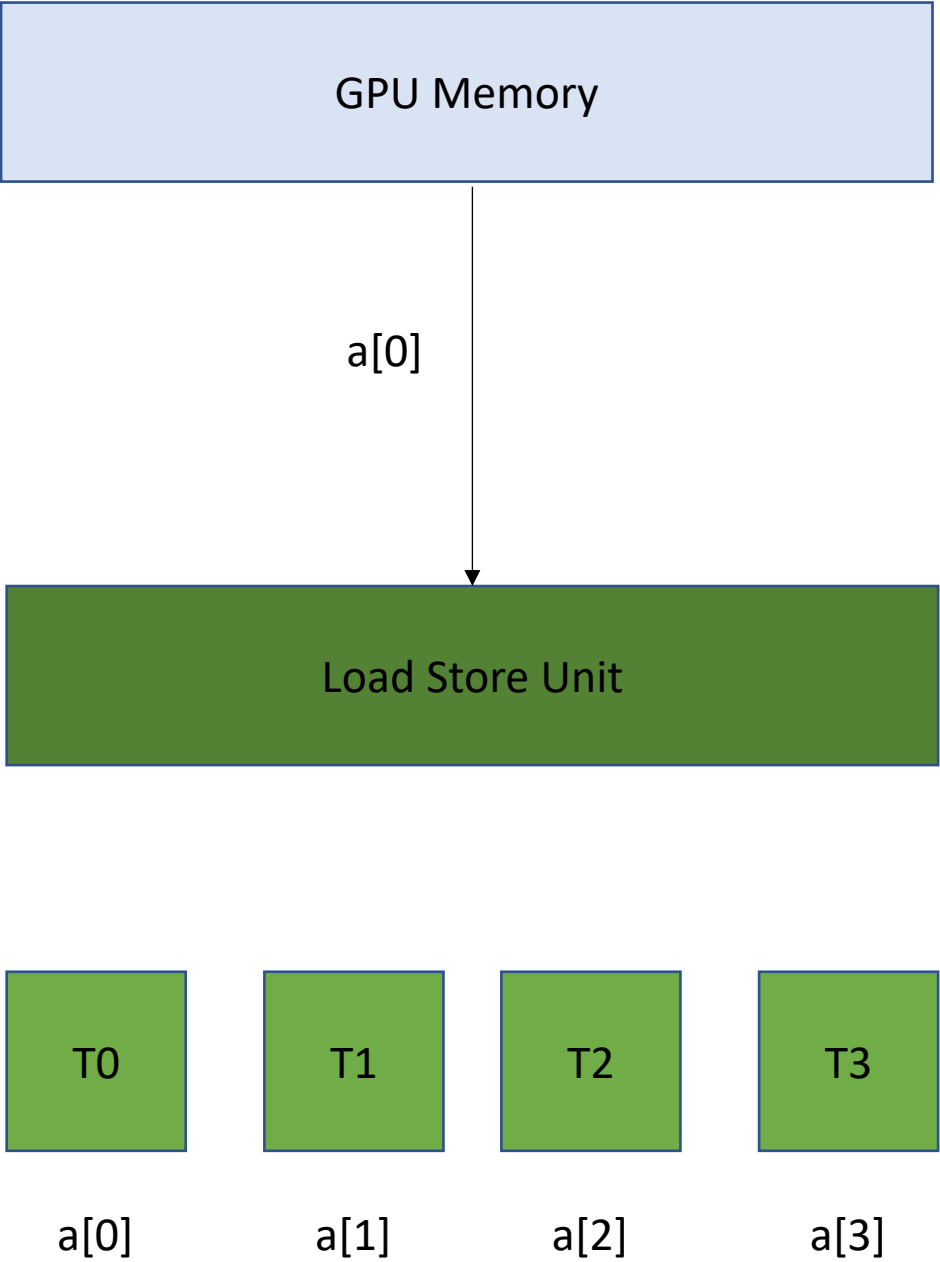
a[2]

a[3]

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 (for this example)

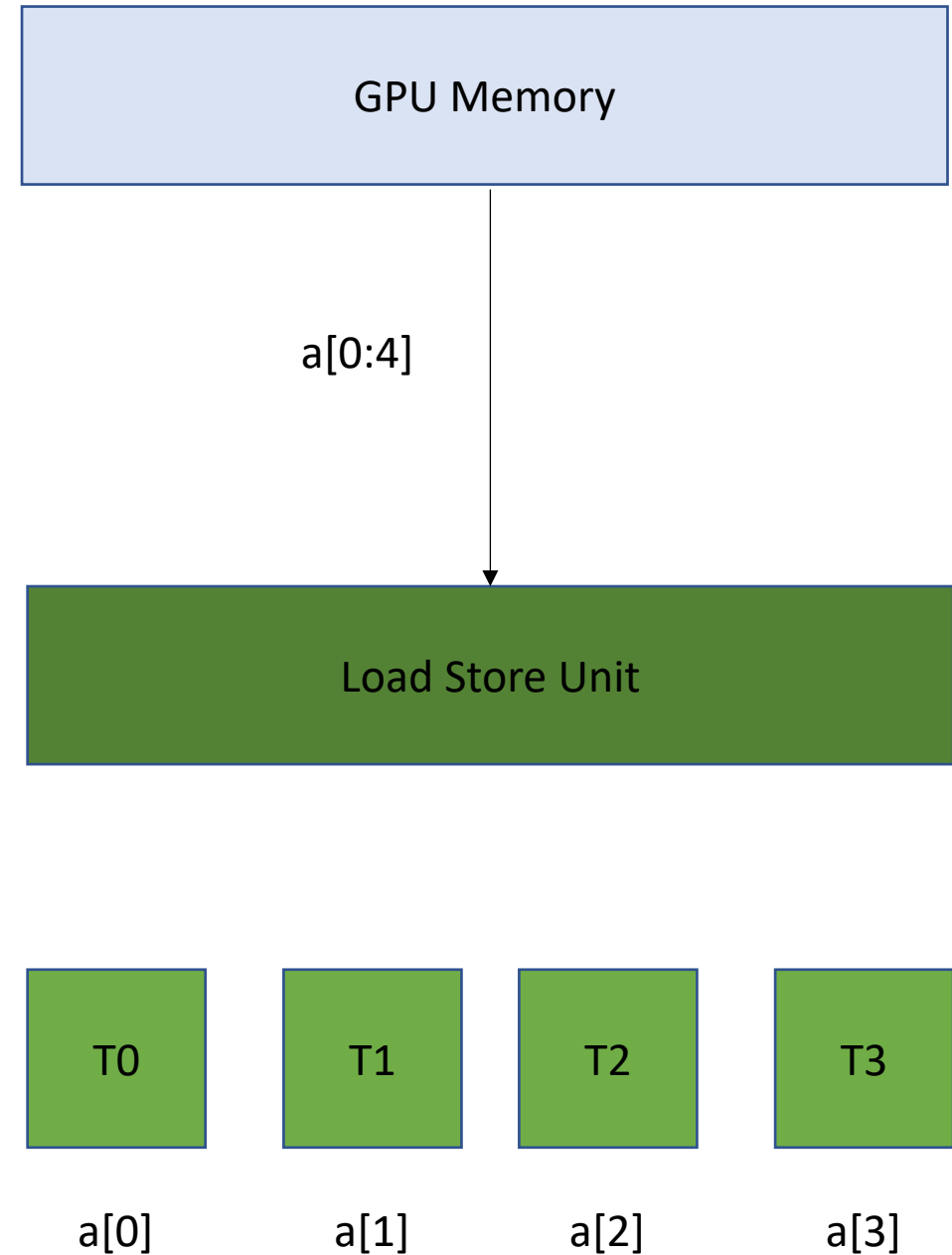


4 cores are accessing memory. What can happen

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



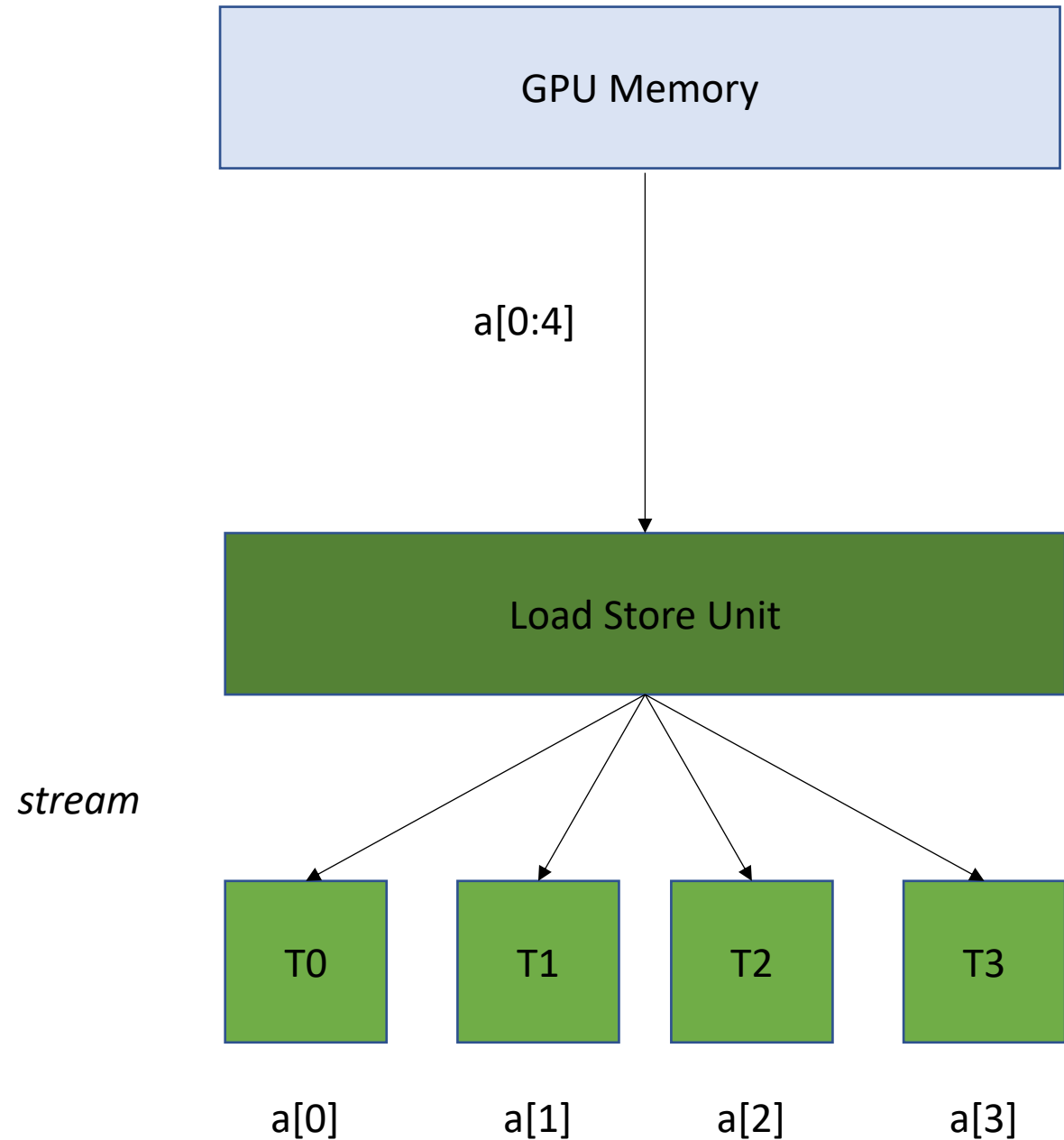
4 cores are accessing memory. What can happen

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



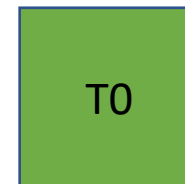
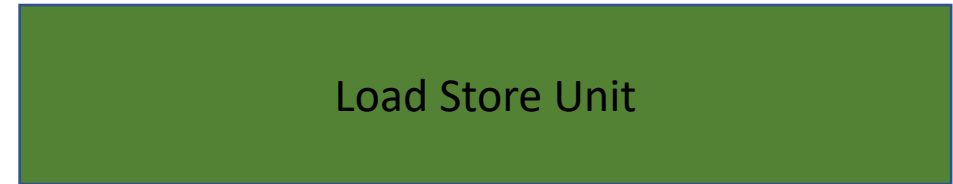
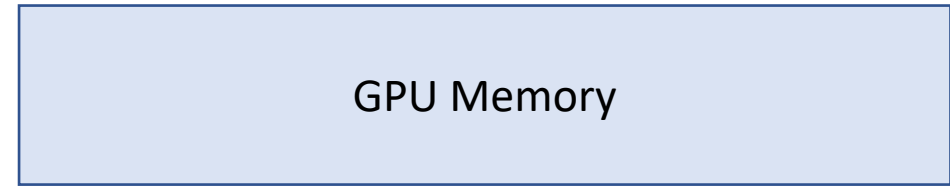
4 cores are accessing memory. What can happen

Read non-contiguous values

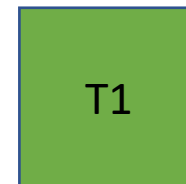
Not good!

Accesses are Serialized.

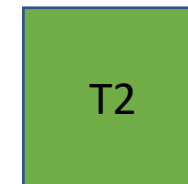
You need 4 requests to GPU memory



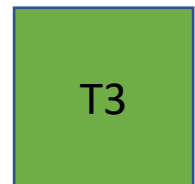
a[x]



a[y]



a[z]



a[w]

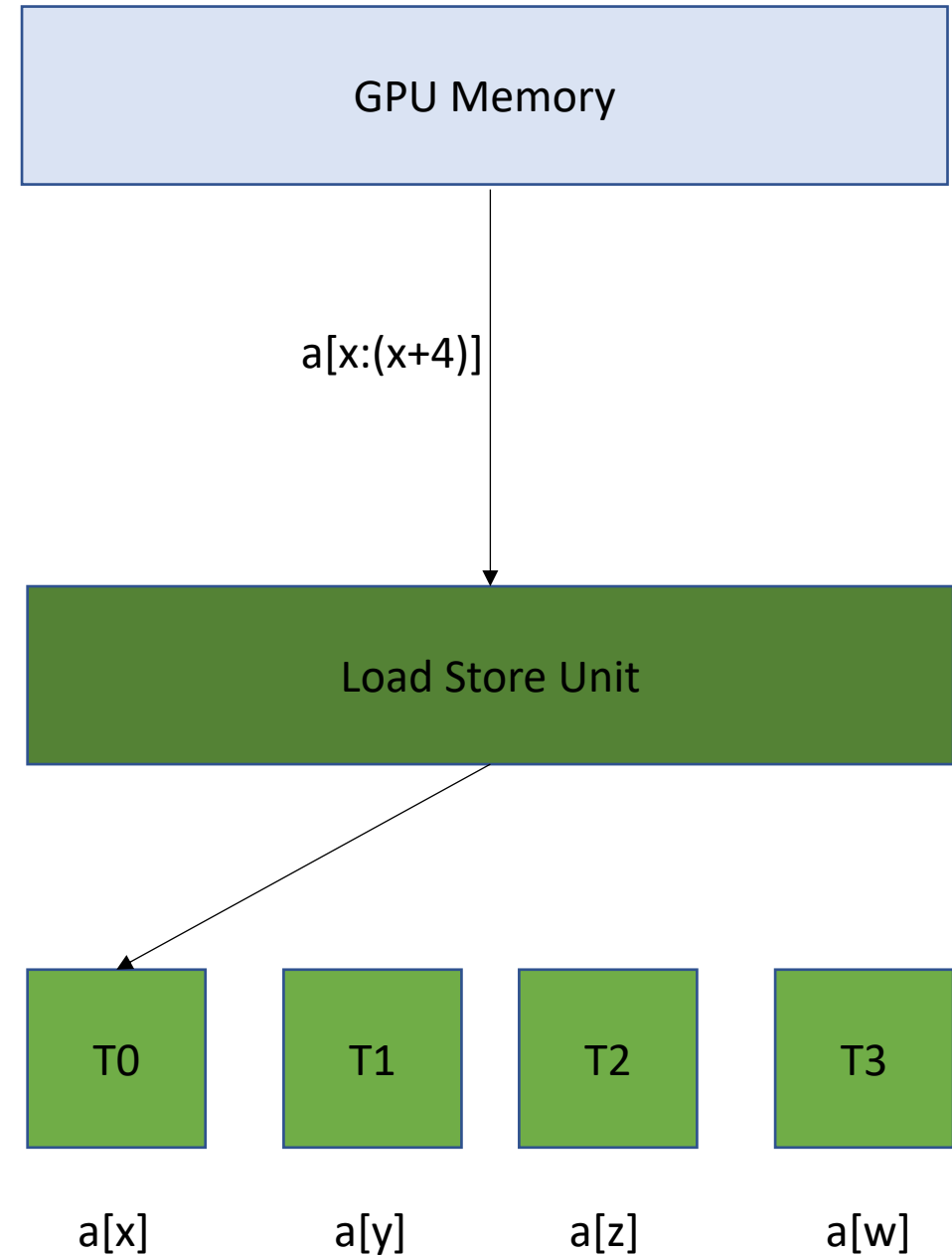
4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.

You need 4 requests to GPU memory



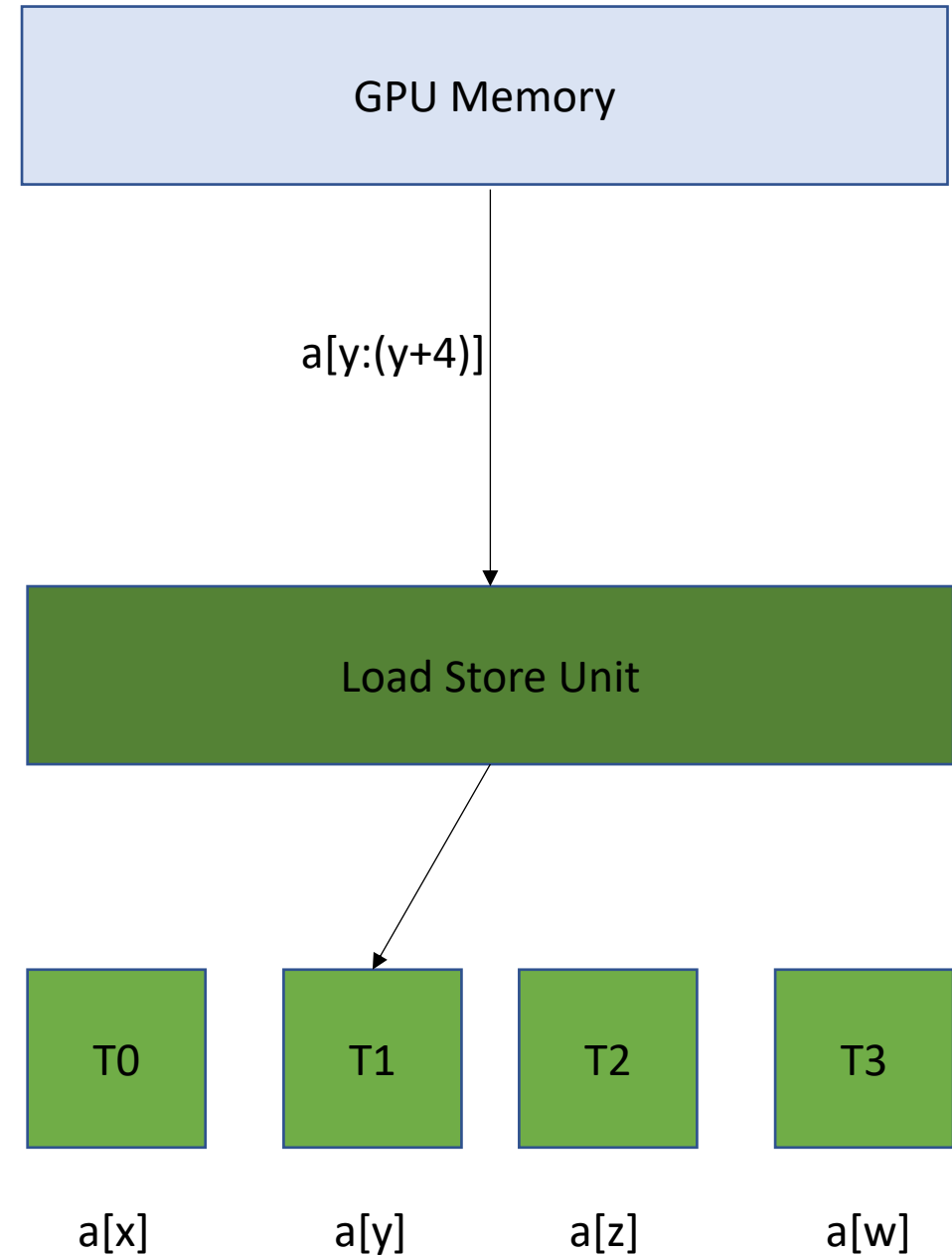
4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.

You need 4 requests to GPU memory



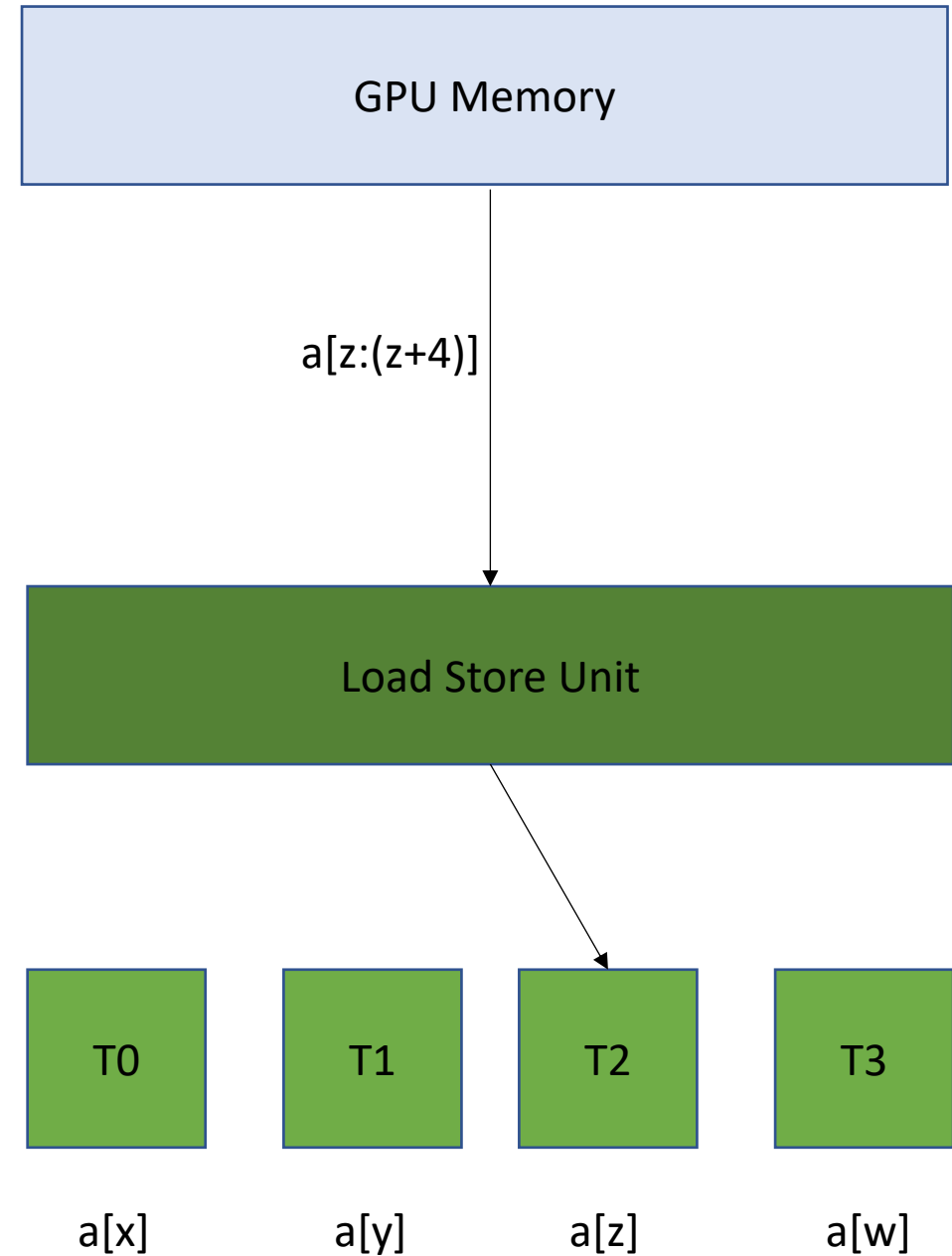
4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.

You need 4 requests to GPU memory



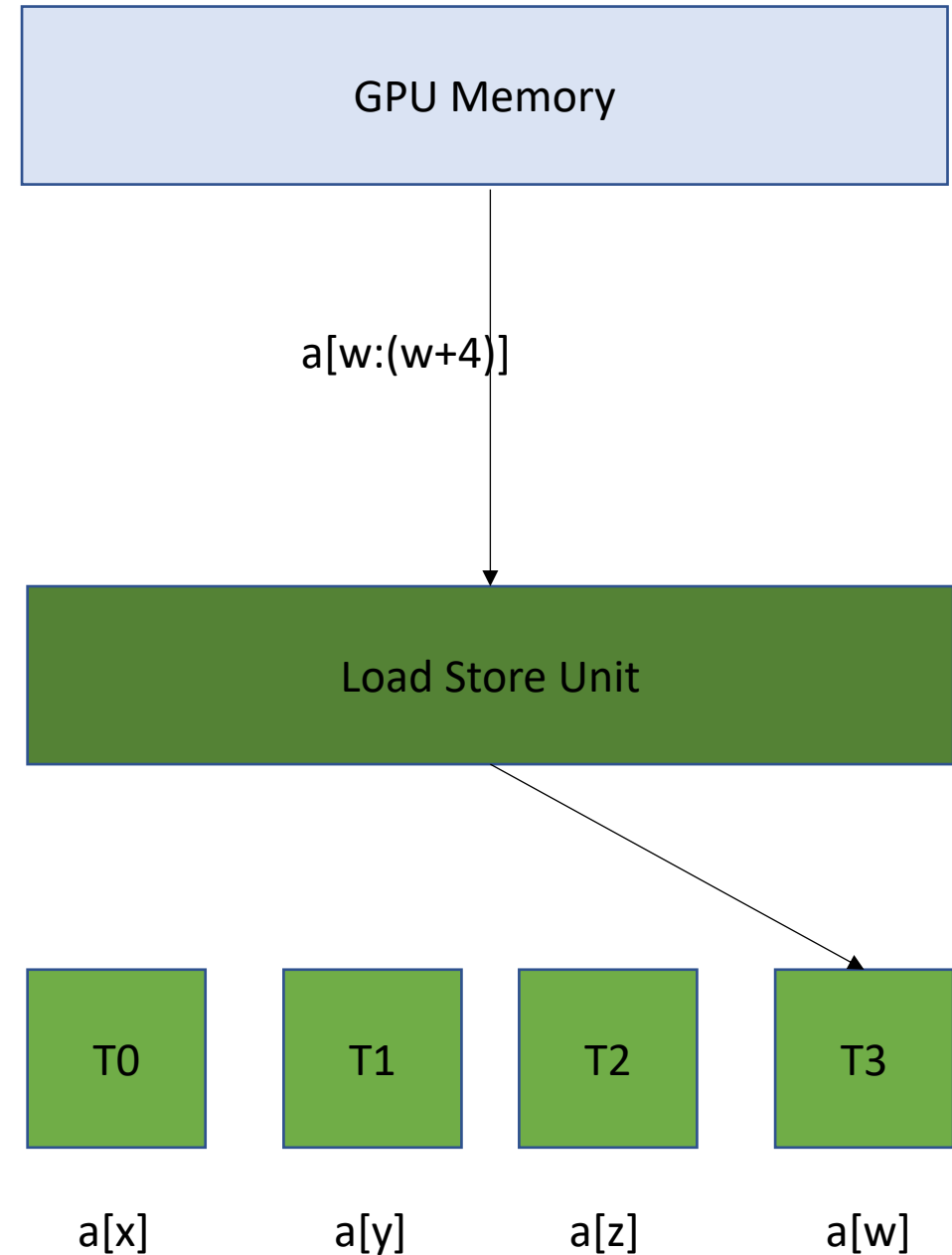
4 cores are accessing memory. What can happen

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

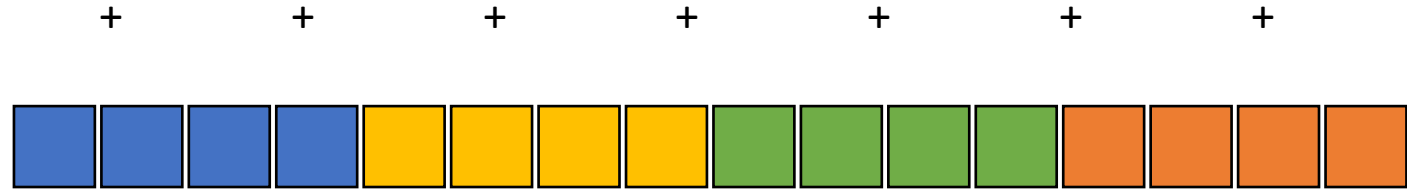
```
vector_add<<<1,32>>>>(d_a, d_b, d_c, size);
```


Chunked Pattern

array a



array b



array c



Computation
can easily be
divided into
threads

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

+ + + + + + +
=
=
=
=
=
=
=
=

Chunked Pattern

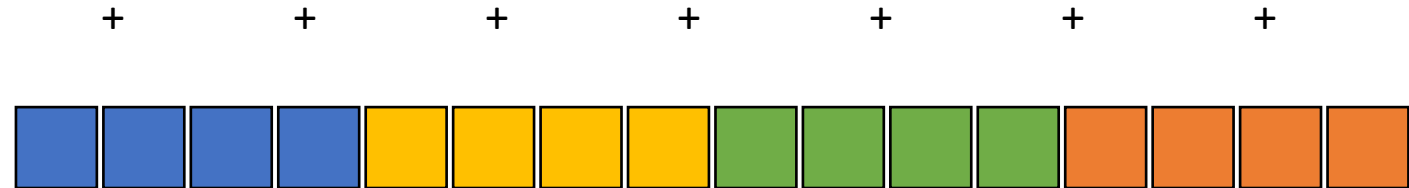
the first element accessed
by the 4 threads sharing a
load store unit. What
sort of access is this?

array a



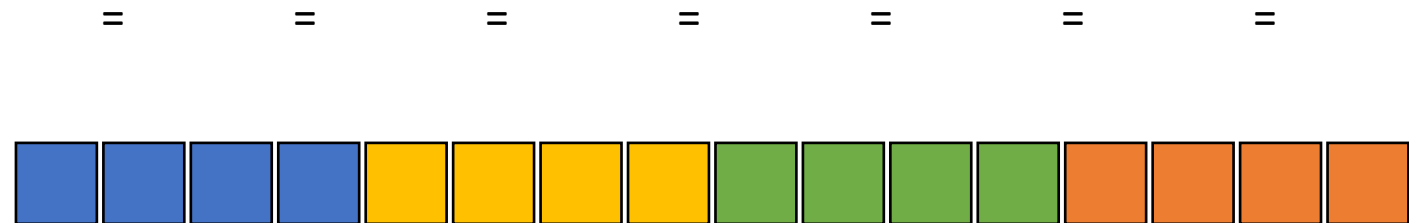
Computation
can easily be
divided into
threads

array b



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

array c



Chunked Pattern

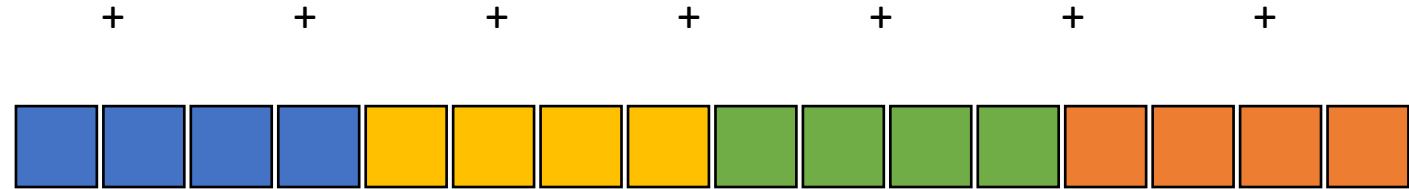
the first element accessed
by the 4 threads sharing a
load store unit. What
sort of access is this?

array a



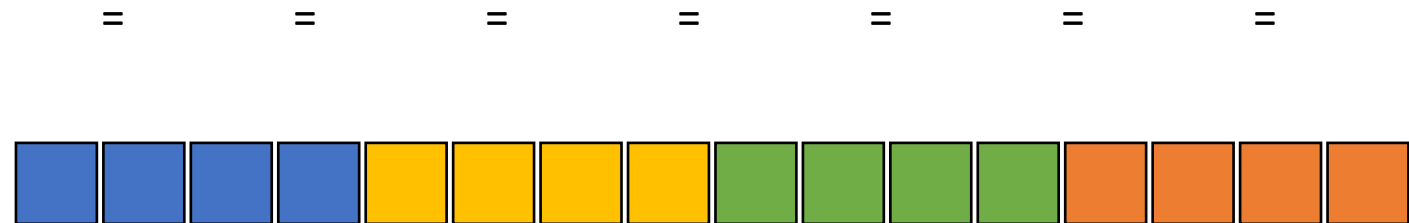
Computation
can easily be
divided into
threads

array b



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

array c



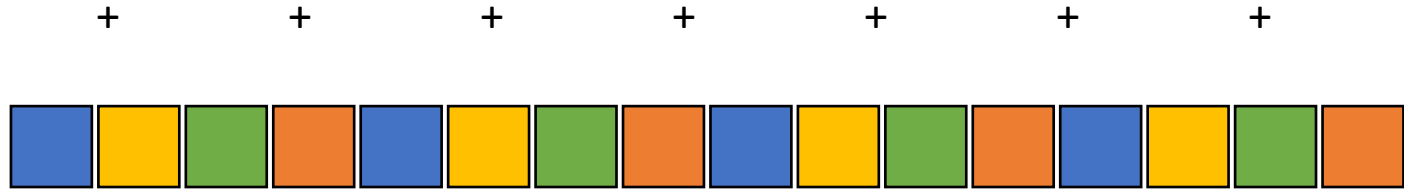
How can we fix this

Stride Pattern

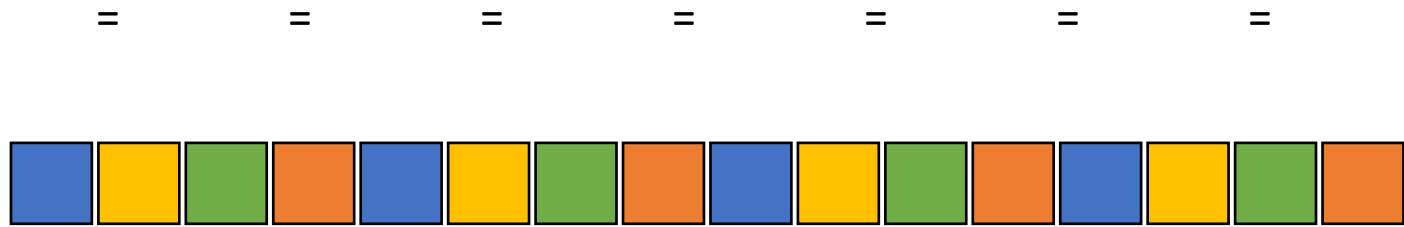
array a



array b



array c



Computation
can easily be
divided into
threads

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

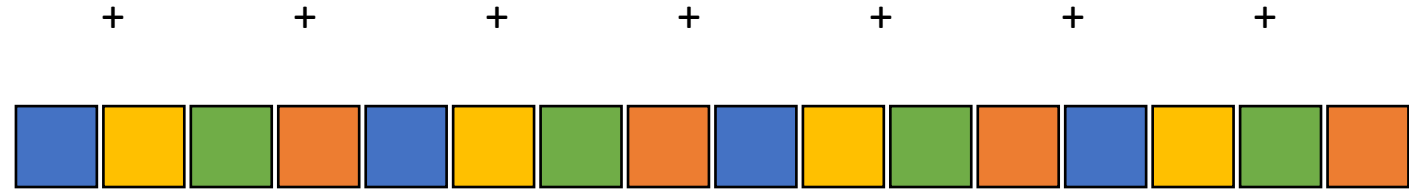
Stride Pattern

What sort of pattern is this?

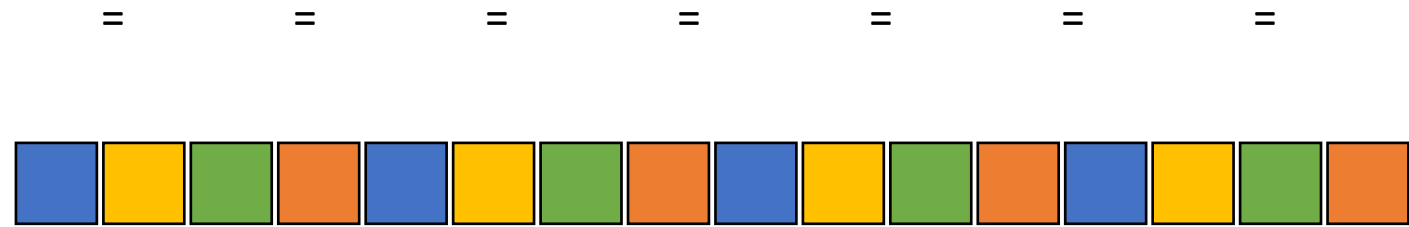
array a



array b



array c



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);
```

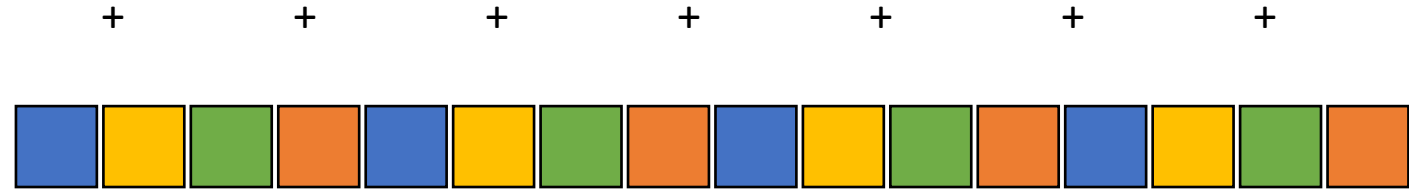
Stride Pattern

What sort of pattern is this?

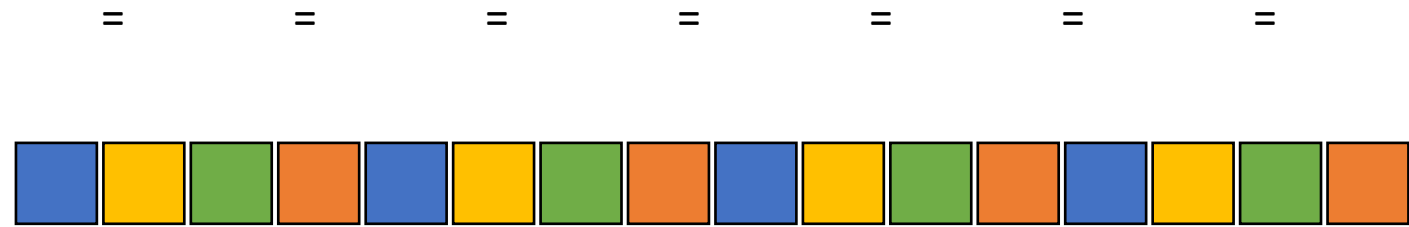
array a



array b



array c



Computation
can easily be
divided into
threads

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

Coalesced memory accesses

Lets try it! What do we think?

Coalesced memory accesses

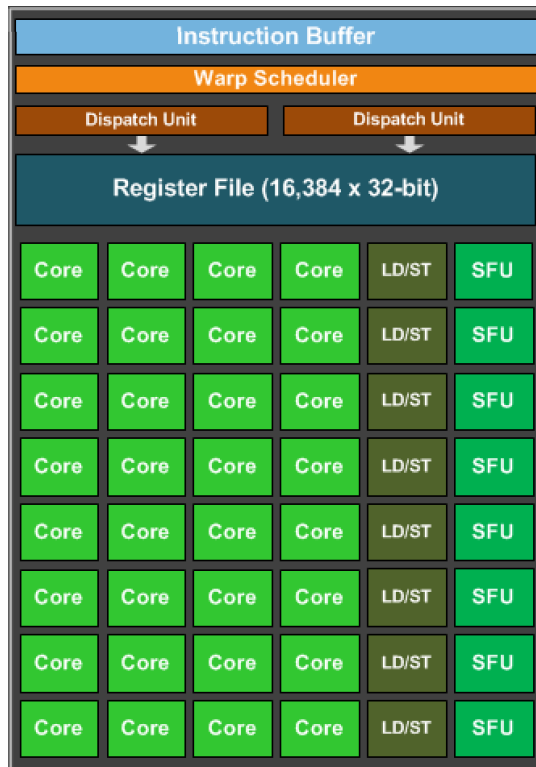
Lets try it! What do we think?



What else can we do?

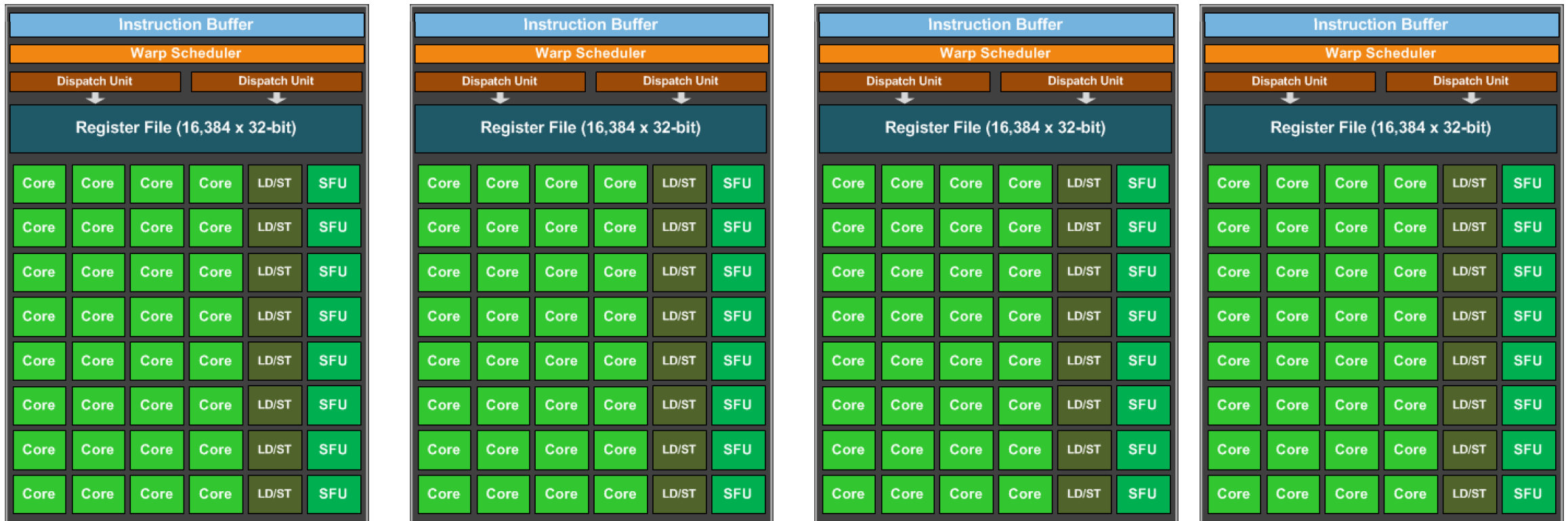
Multiple streaming multiprocessors

*We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs
big ML GPUs have 80.*



Multiple streaming multiprocessors

*We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs
big ML GPUs have 80. This little GPU has 4*



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];  
}
```

calling the function

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

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

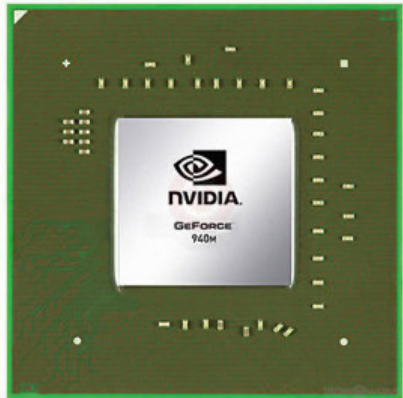
Need to recalculate some thread ids.

Launch with many thread blocks

Now we have 1 thread for each element

Final Round

Tiny GPU in an
embedded system



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

Fight!



<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/>

The CPU in
my professor
workstation



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

Extra thoughts

Extra thoughts

- Why threadIdx.x?
- Because you can pass in a 3D range (x,y,z). Historically threads mapped to pixels (or voxels)

Consequences of Warps

- Branches can cause significant performance impacts, Demo

Locks on GPU?

What is the right way to program GPUs?

- Still an open question!

Homework 5 - first look

- It is the first time offering this homework, so feedback is very welcome and we will be generous with support.
- Thanks to Mingun Cho who prepared the initial assignment! (and thanks to the TA's last year who polished it)



Homework 5 - first look

- Prerequisites
 - Google Chrome
 - should be stable on Windows and Mac
 - if you are running linux, please try things out ASAP
- We need some security set up

Homework 5 - first look

- Javascript shared array buffer:
 - How javascript threads can actually share memory
 - Similar to memory in C++

Shared memory and high-resolution timers were effectively **disabled at the start of 2018** [↗](#) in light of **Spectre** [↗](#). In 2020, a new, secure approach has been standardized to re-enable shared memory.

With a few security measures, `postMessage()` will no longer throw for `SharedArrayBuffer` objects and shared memory across threads will be available:

As a baseline requirement, your document needs to be in a **secure context**.

Your application will be in a secure context (you are writing and running locally!)

Homework 5 - first look

- You will generate some SSL keys and run in a local server

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -outcert.pem -days 365 -nodes
```

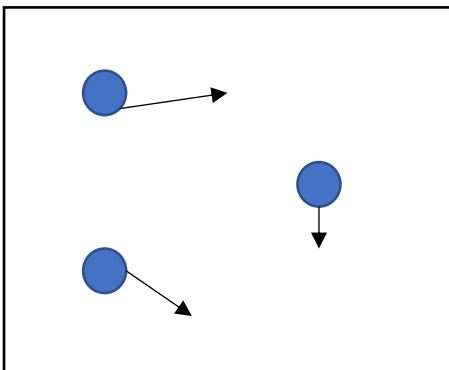
```
python3 serve.py
```

Homework 5 - first look

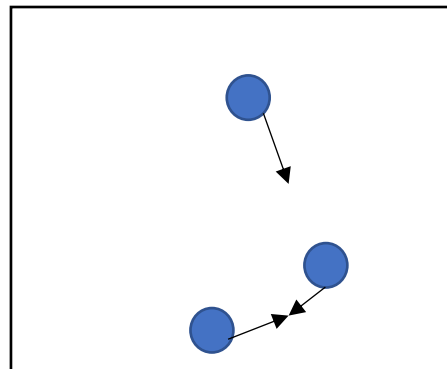
- Let's have a look!

Homework 5- first look

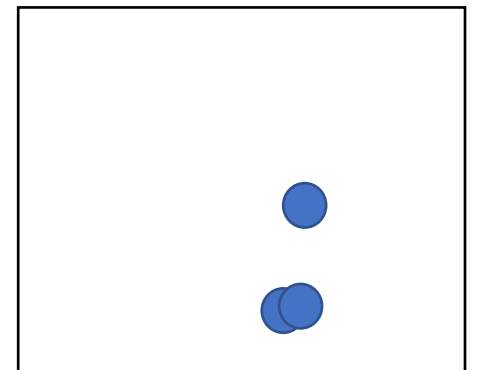
- Your assignment:
 - N-body simulation
- Each particle interacts with every other particle



time = 0



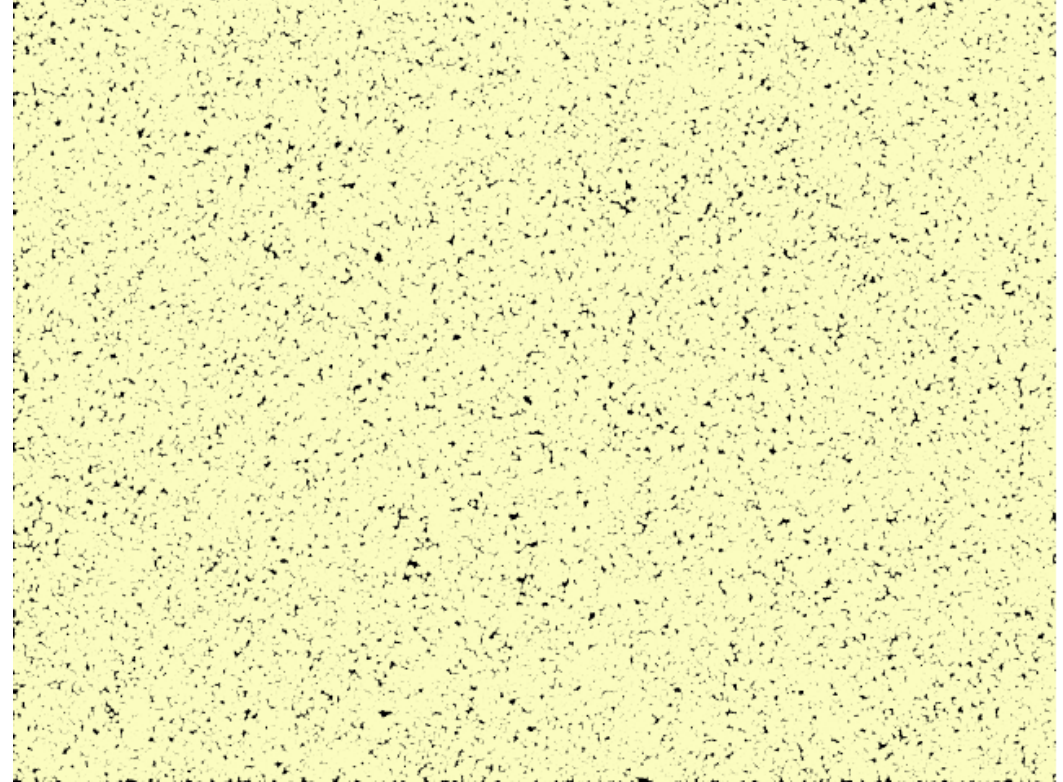
time = 1



time = 2

Examples

- Gravity:
- Boids:
 - <https://en.wikipedia.org/wiki/Boids>



Your homework

- N-body require a little bit of physics background so we will do something simpler.
 - If you want to explore with physics please feel free
- Local attraction clustering:
 - For each particle: find your closest neighbor
 - You can take one step in the x direction and one step in the y direction towards your closest neighbor.

Your homework

- Part 1 of your homework will do this on a single javascript thread
- Demo

Your homework

- Looks good, but with more particles, things start to go slower...

Your homework

- Looks good, but with more particles, things start to go slower...
- Part 2 of the homework is to implement with multiple CPU threads using javascript webworkers
 - Should get a linear speedup
- Part 3 is to implement with webGPU
 - Should get a BIG speedup!
- You need to explore how many particles you can simulate while keeping a 60 FPS framerate.

Let's look at the code and see some javascript

Shared Array Buffer

- Like Malloc, allocates a "pointer" to a contiguous array of bytes
- Can pass the "pointer" to different threads
- Need to instantiate a typed array to access the values

Web Workers

- How to do multi-threading in javascript
- Async
 - Concurrent (executes on the same thread)
 - Good for I/O and user interactions
- Web Workers will execute on multiple cores
 - Better for compute intensive applications
 - Better performance

How to use?

- Create a new worker with a file
 - Doesn't do anything yet
- File contains a function: “on message”
- Main file calls “post message” to start the thread along with arguments
- Worker sends a message back to the main file, it can catch the data

WebGPU

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

WebGPU

- wgsi is NOT javascript
- Javascript is interpreted: not possible on GPUs
- wgsi is compiled
 - into Vulkan on Linux
 - into Metal on Apple
 - into HLSL on Windows
- No printing (so GPU code 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

- types:

- i32
- u32
- f32
- vec2<f32>
- array<type>

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

```
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.