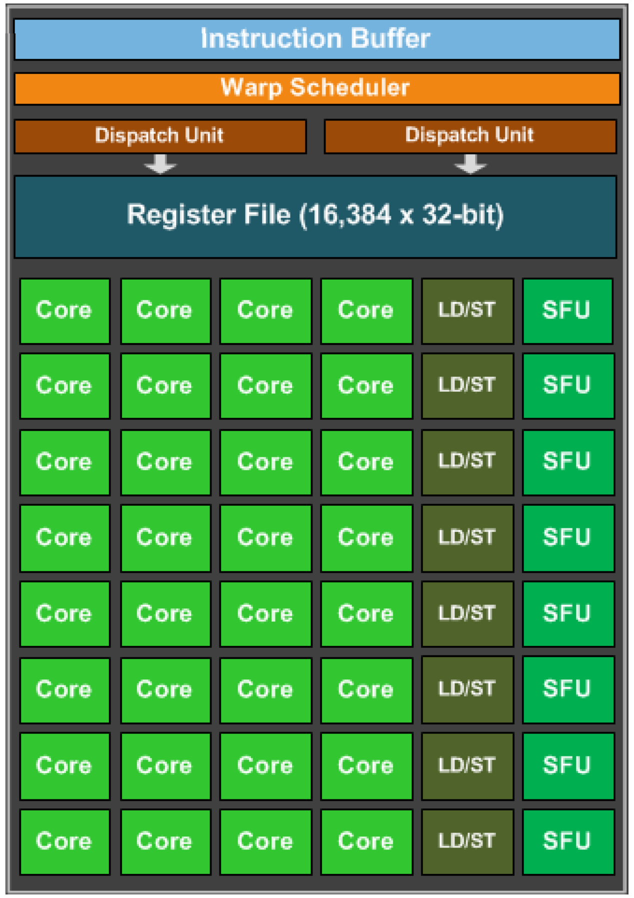
CSE113: Parallel Programming

March 15, 2023

- **Topics**:
    - GPU programming continued

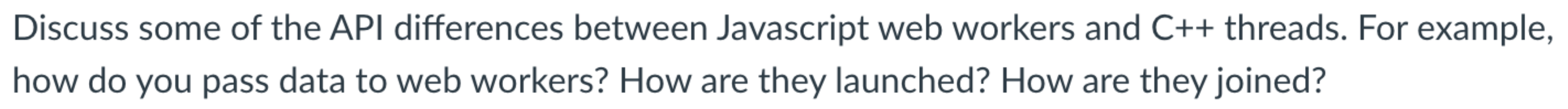


Announcements

- extra day on Homework 4 (You can turn it in by the end of today)
- HW 5 is out, you should be able to get started
- Last two days of class!

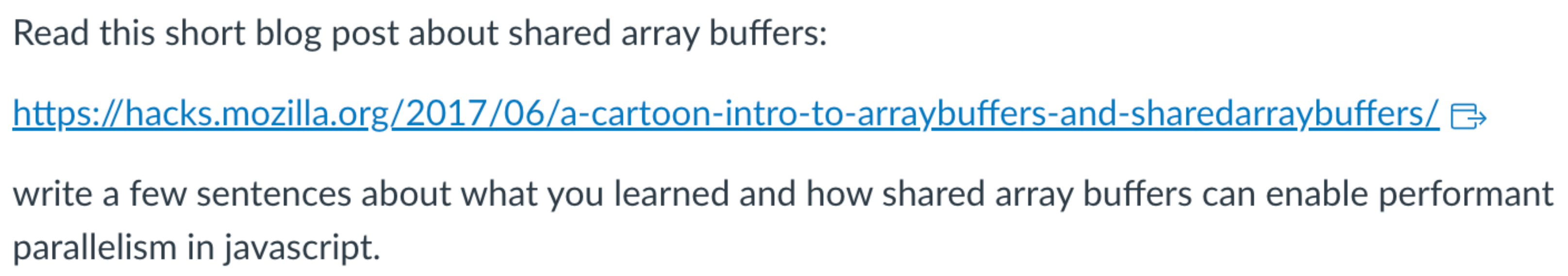
Previous Quiz

Previous Quiz



Discuss some of the API differences between Javascript web workers and C++ threads. For example, how do you pass data to web workers? How are they launched? How are they joined?

Previous Quiz

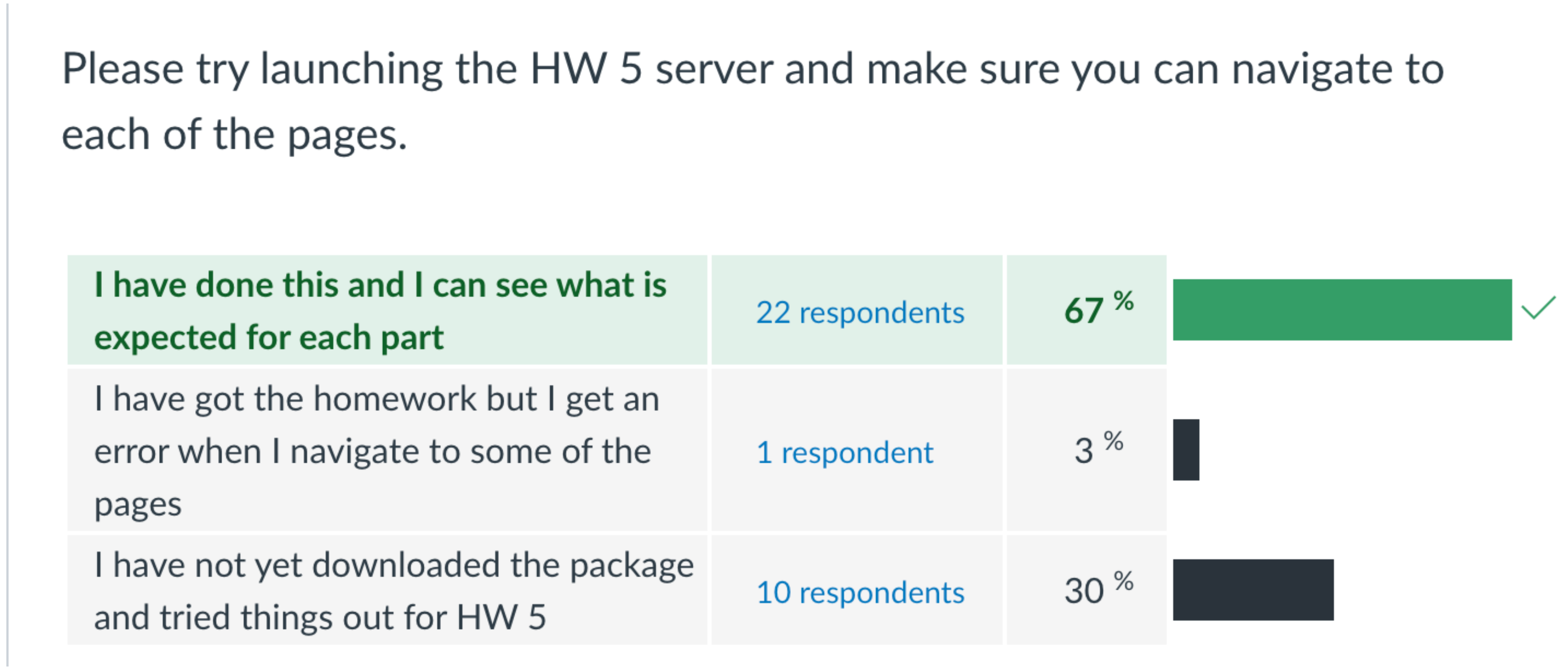


Read this short blog post about shared array buffers:

https://hacks.mozilla.org/2017/06/a-cartoon-intro-to-arraybuffers-and-sharedarraybuffers/

write a few sentences about what you learned and how shared array buffers can enable performant parallelism in javascript.

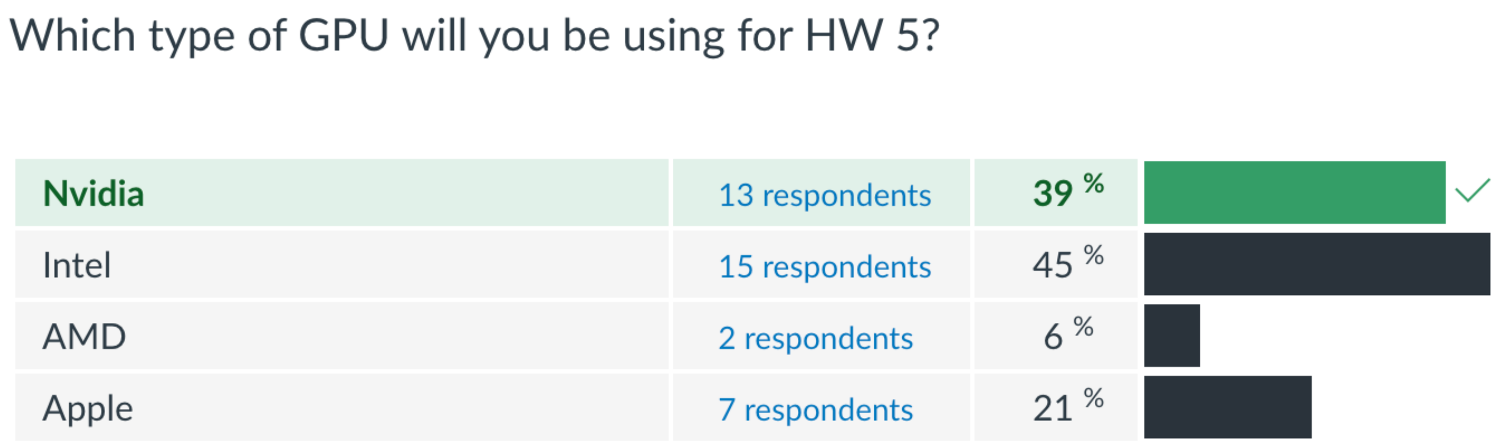
Previous Quiz



Please try launching the HW 5 server and make sure you can navigate to each of the pages.

I have done this and I can see what is expected for each part	22 respondents	**67 %**	✓
I have got the homework but I get an error when I navigate to some of the pages	1 respondent	3 %	
I have not yet downloaded the package and tried things out for HW 5	10 respondents	30 %	

Previous Quiz

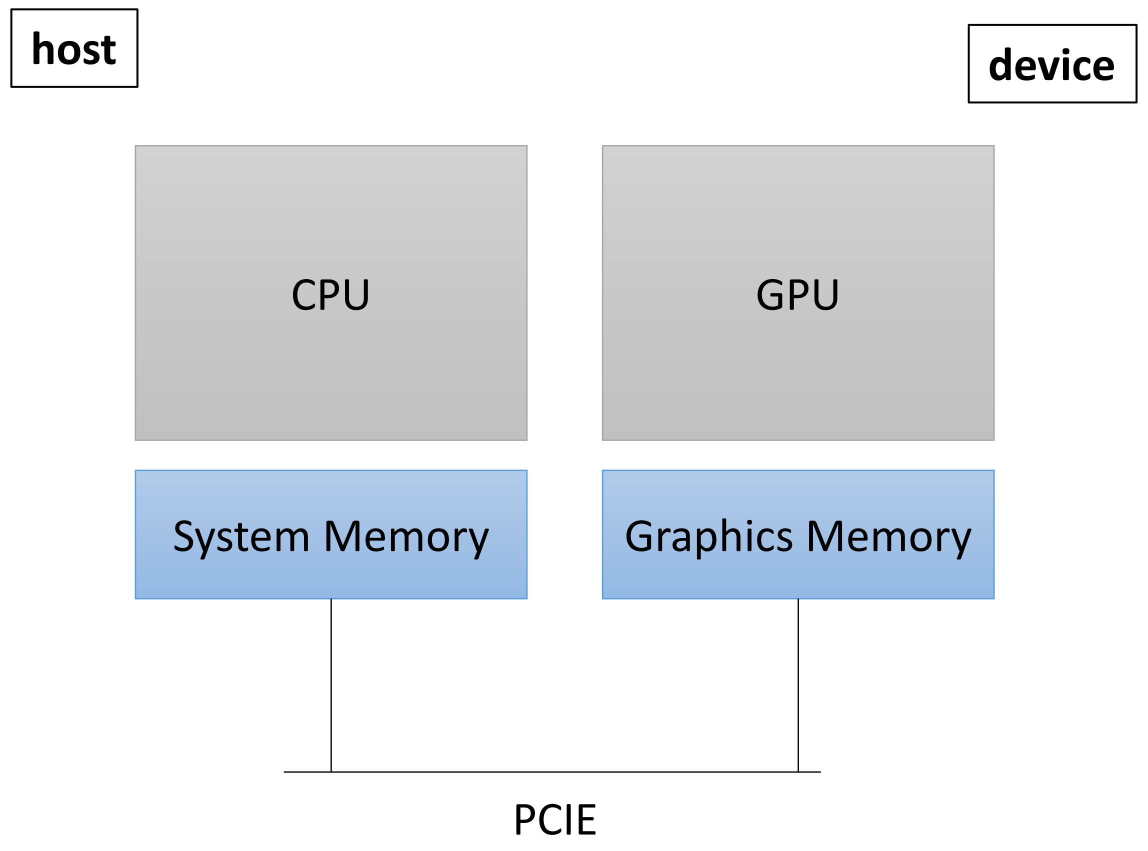


Which type of GPU will you be using for HW 5?

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

GPU set up

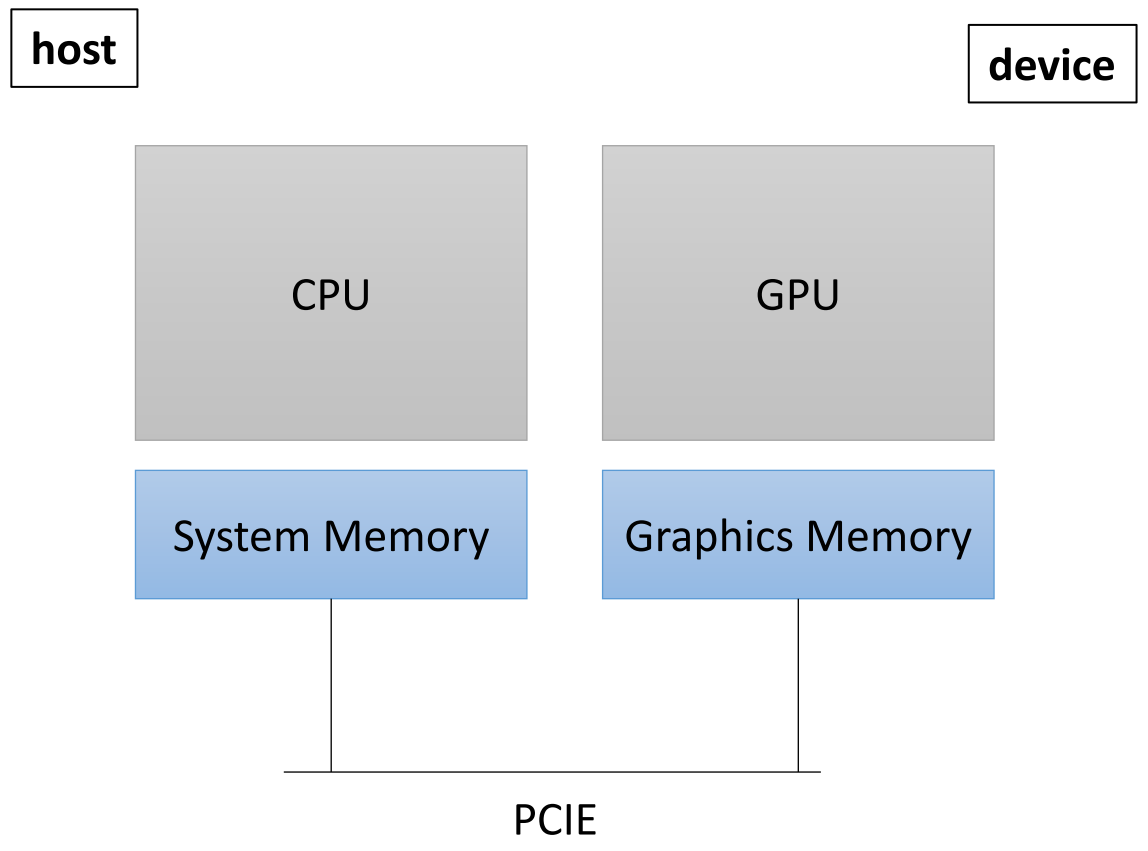
- Our heterogeneous, parallel, programming model



GPU set up

- Our heterogeneous, parallel, programming model

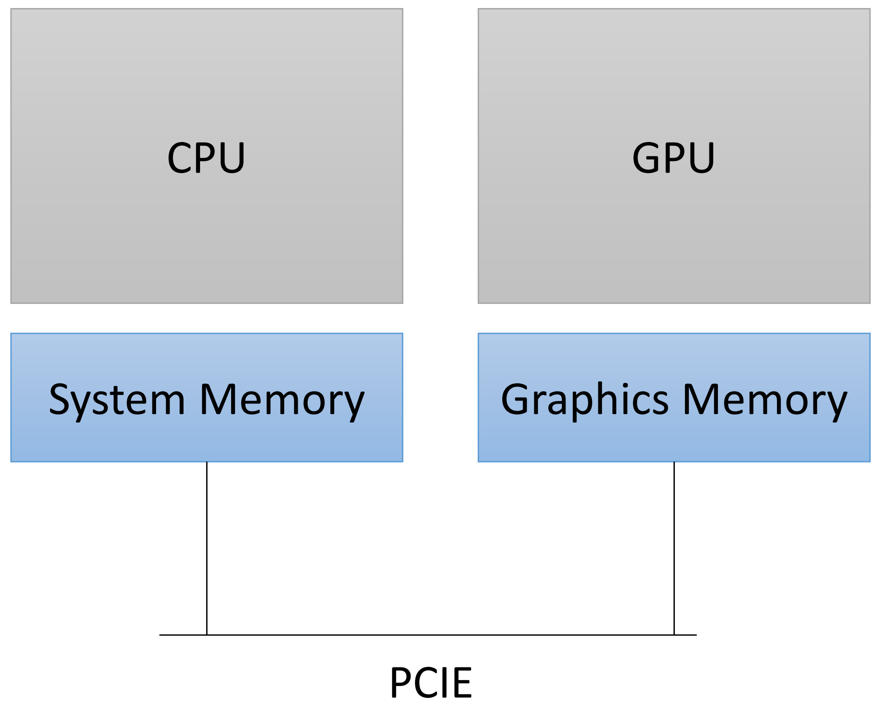
The host (CPU) will write a C++-like program that allocates and sets up memory on the GPU. The host will then call a GPU program called a kernel.



GPU set up

How do we allocate memory on a CPU?

- Our heterogeneous, parallel, programming model

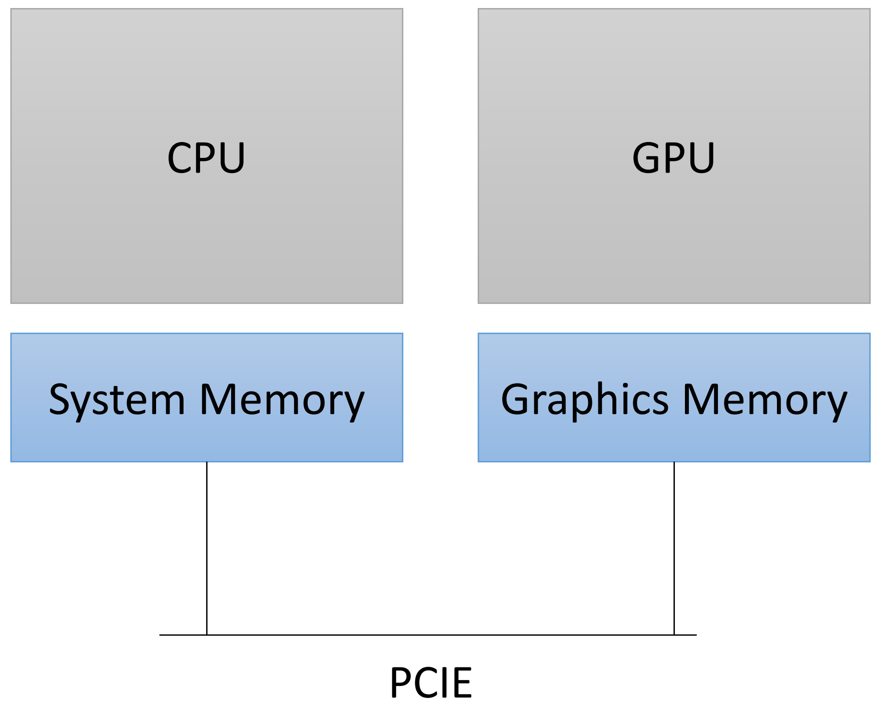


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

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

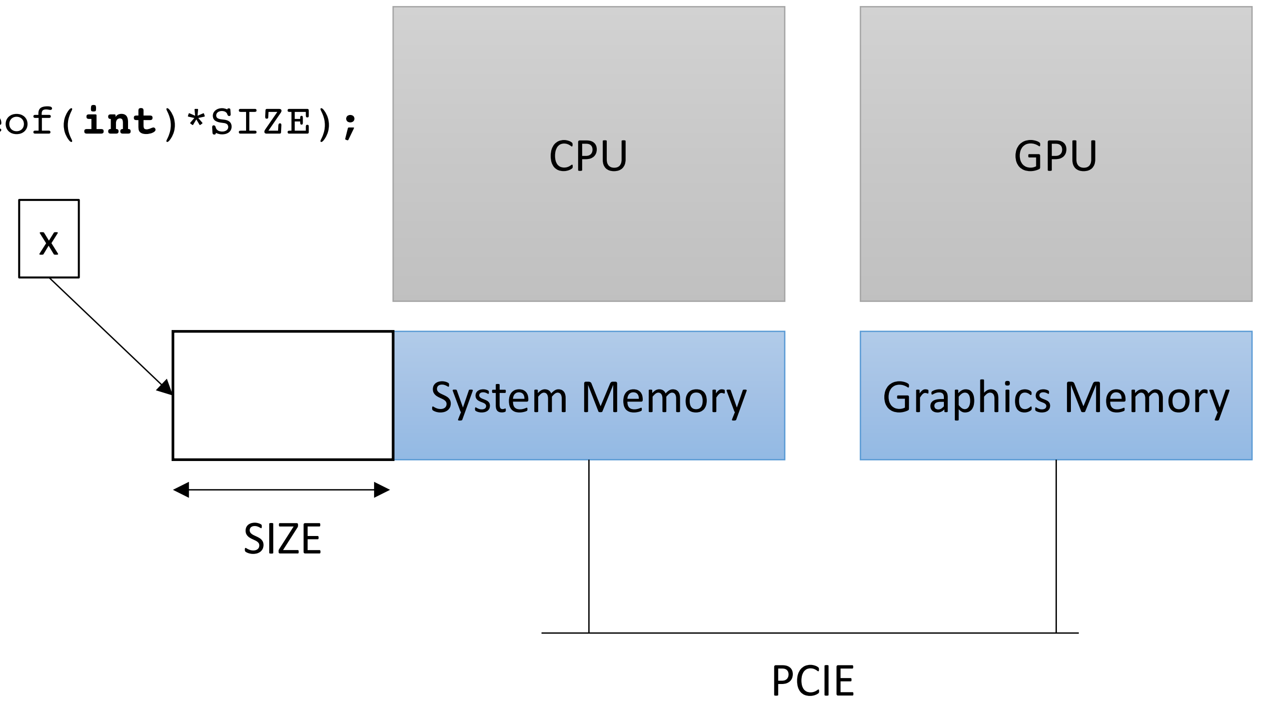


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

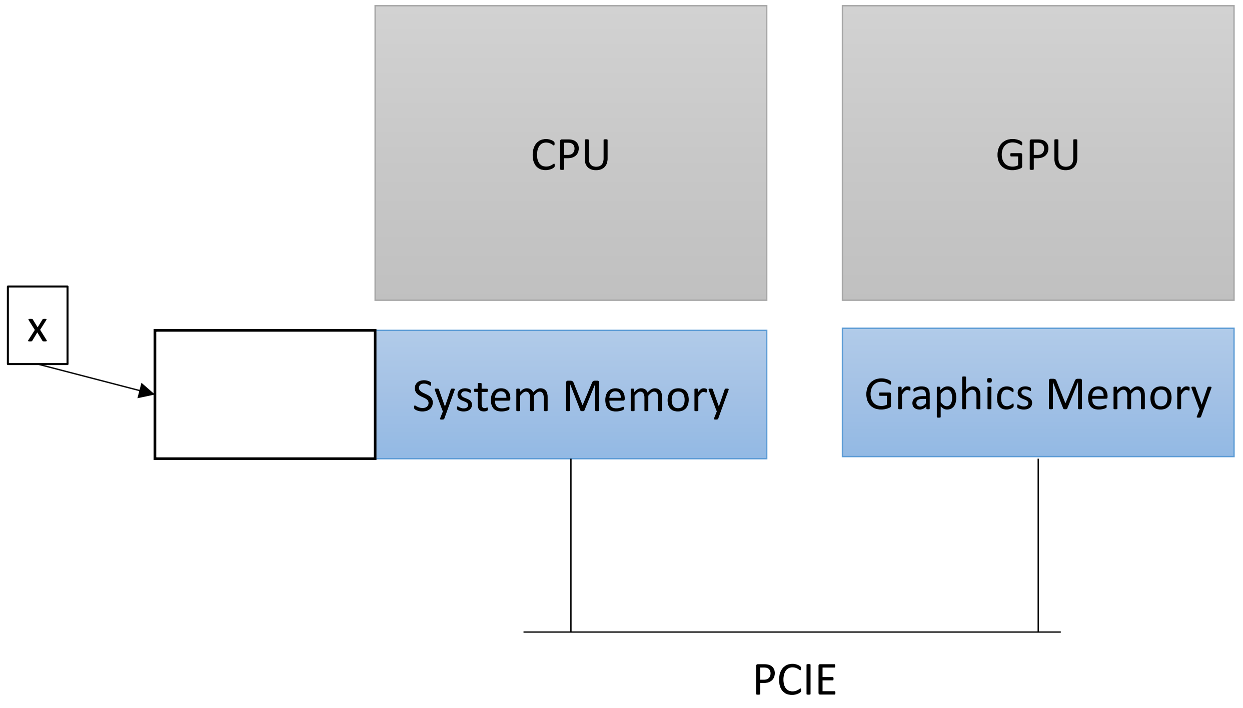
```
int *x = (int*) malloc(sizeof(int)*SIZE);
```



GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

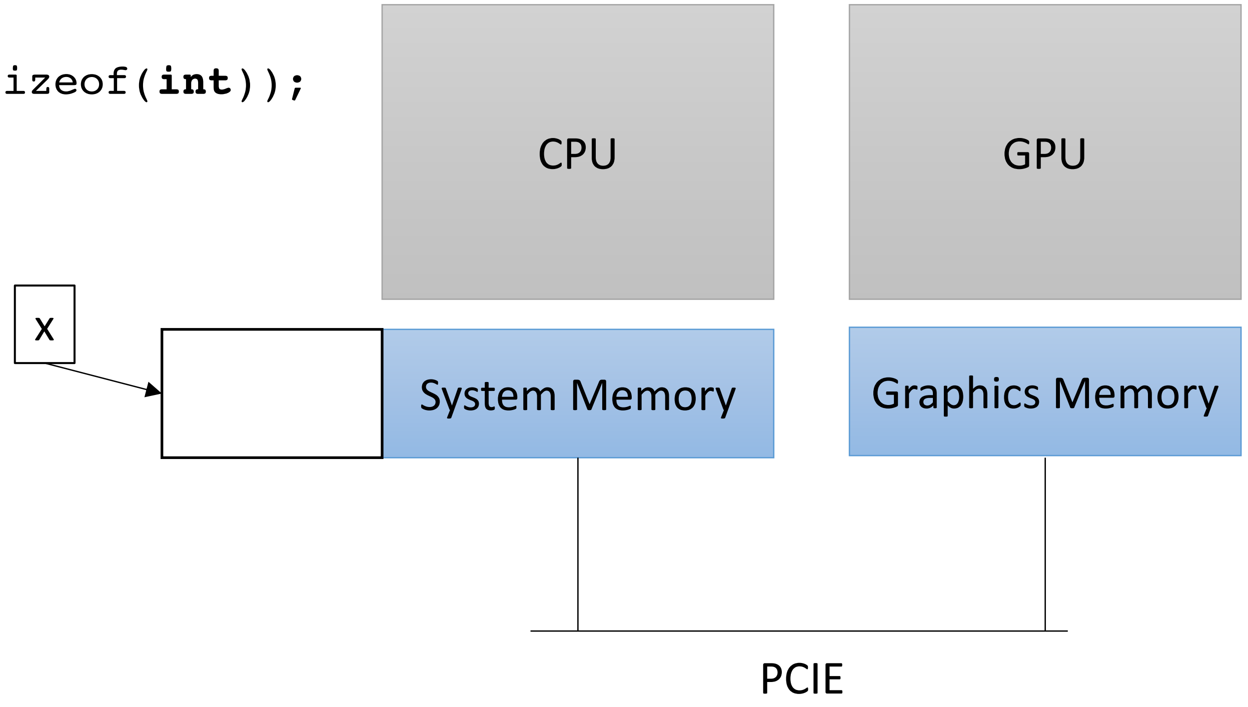


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

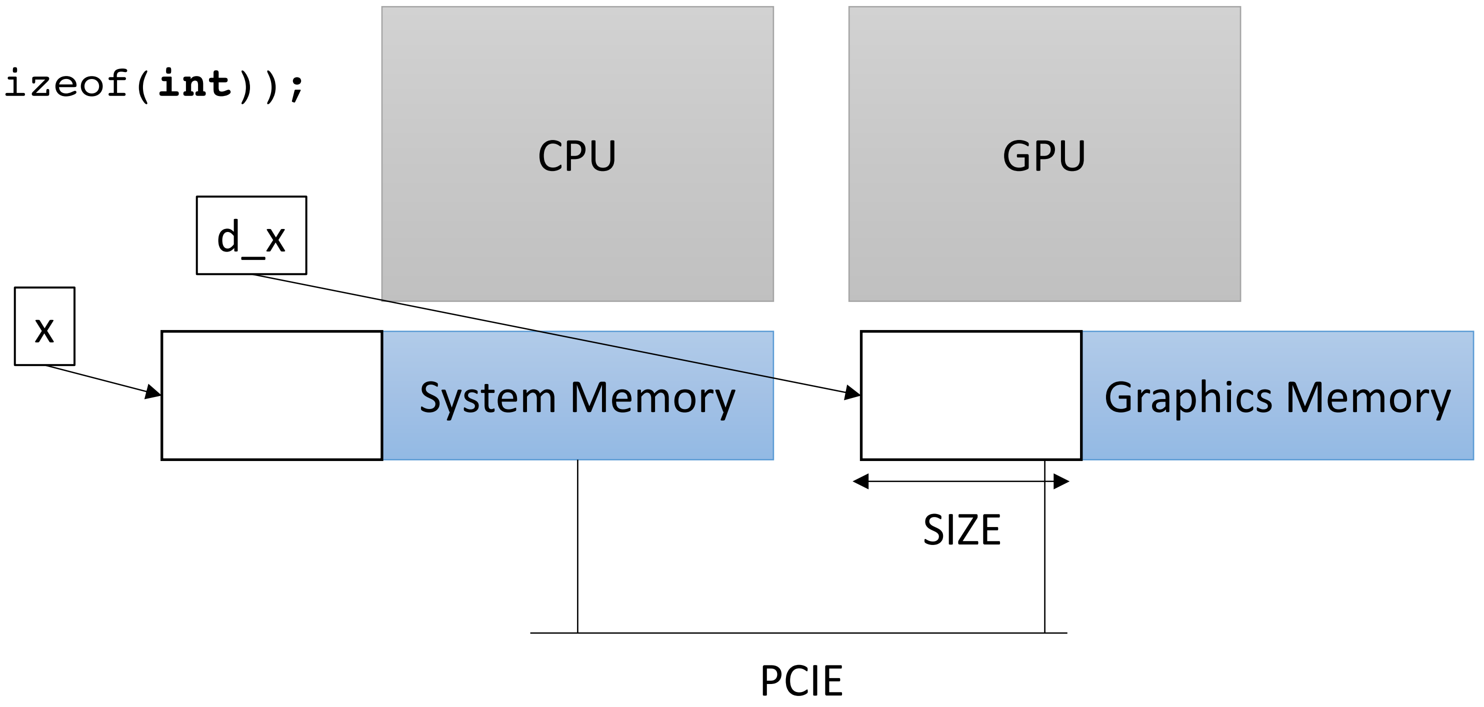


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

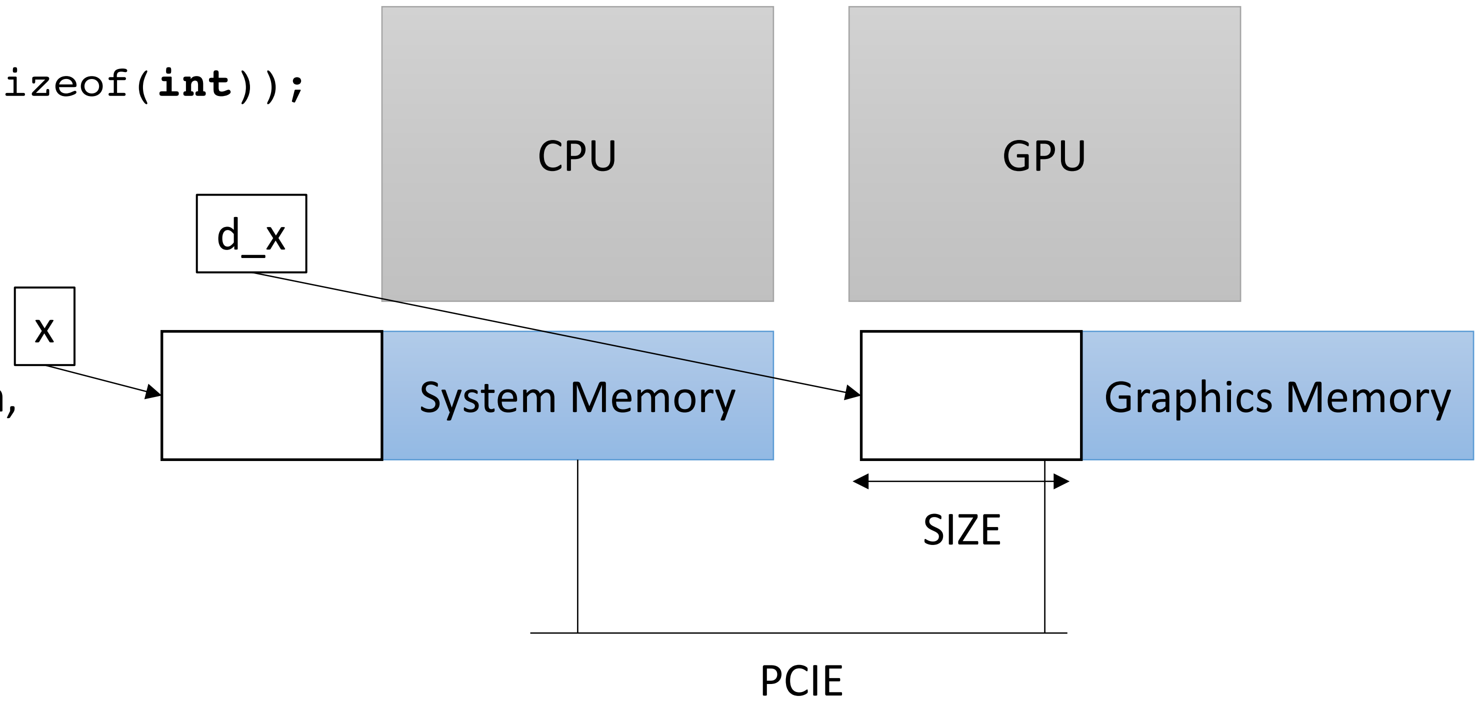


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

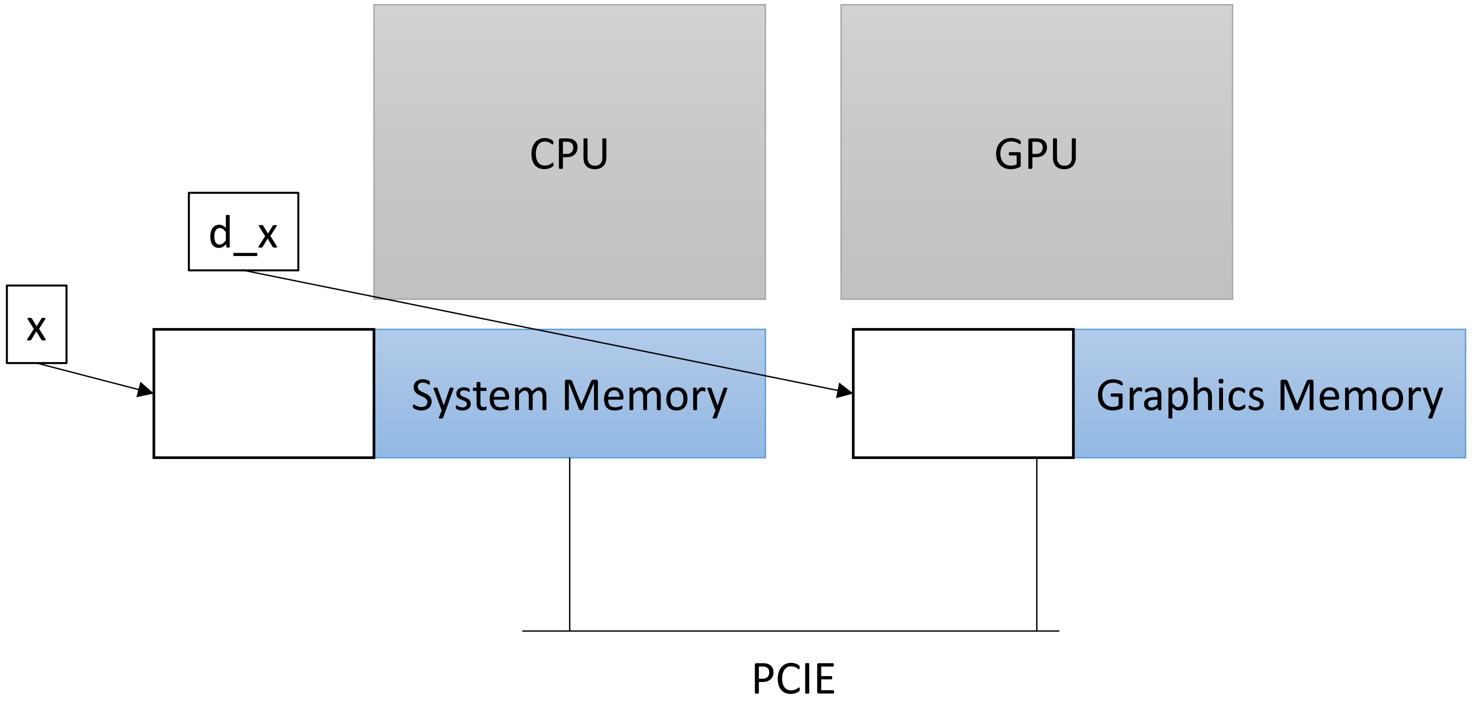


`d_x` is a pointer, in the CPU program, that points to memory on the GPU.

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

GPU set up

- Our heterogeneous, parallel, programming model

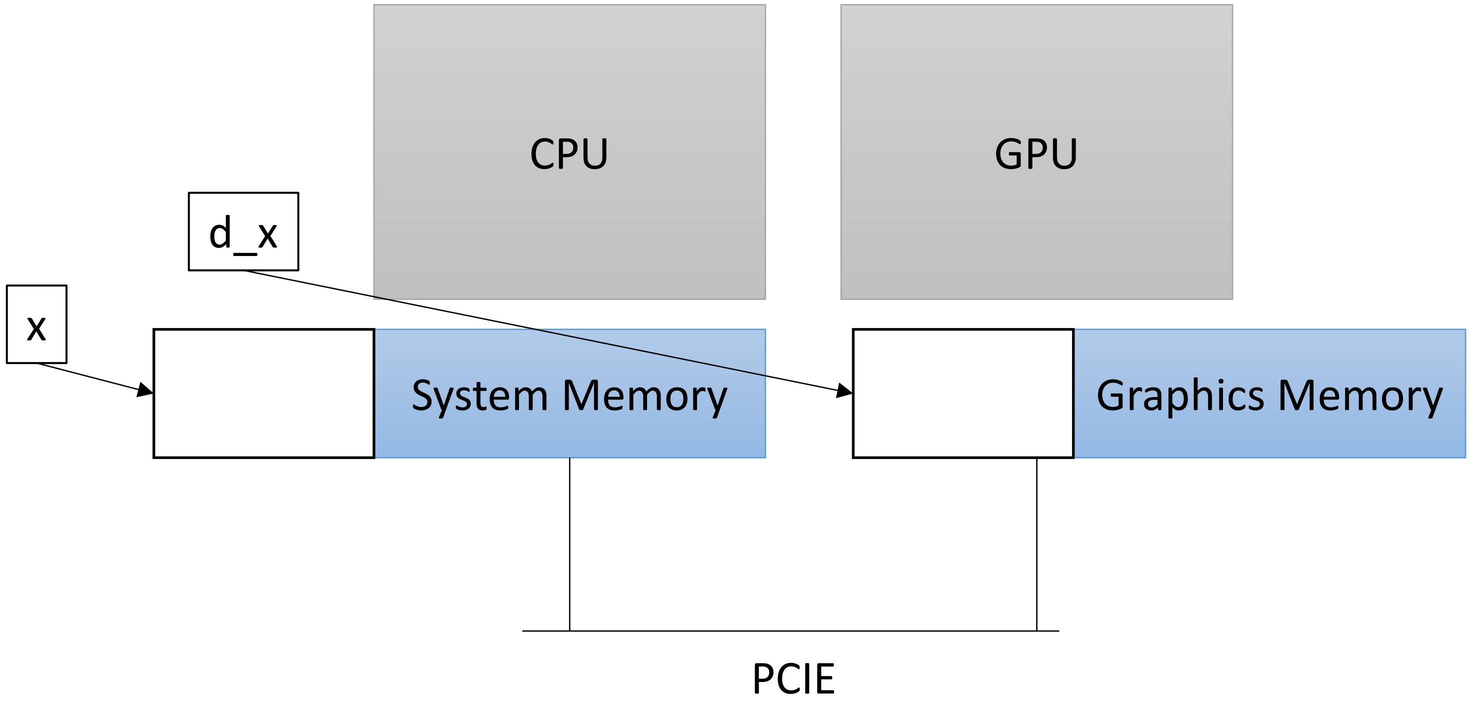


GPU set up

- 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



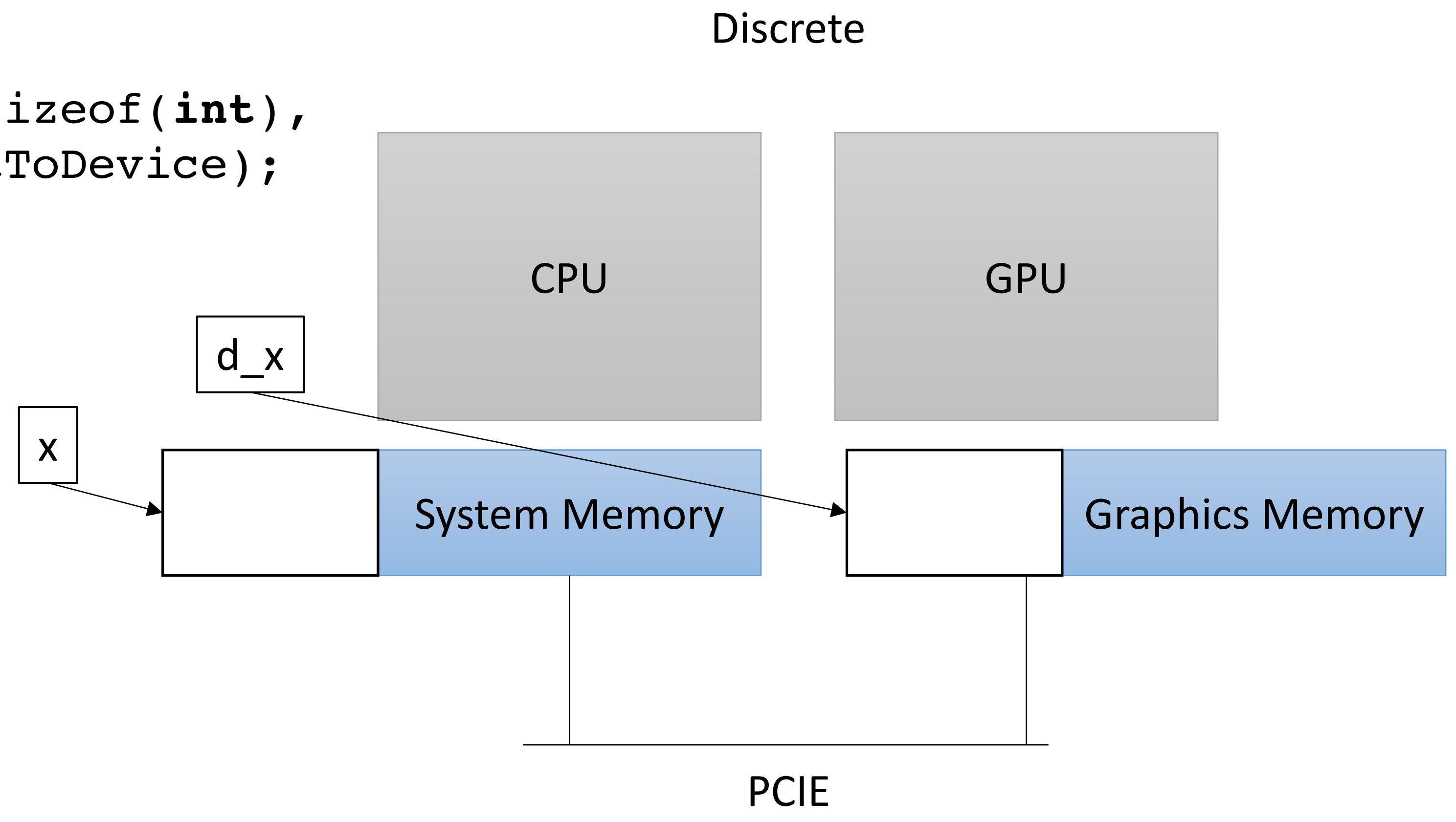
GPU set up

- 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);
```



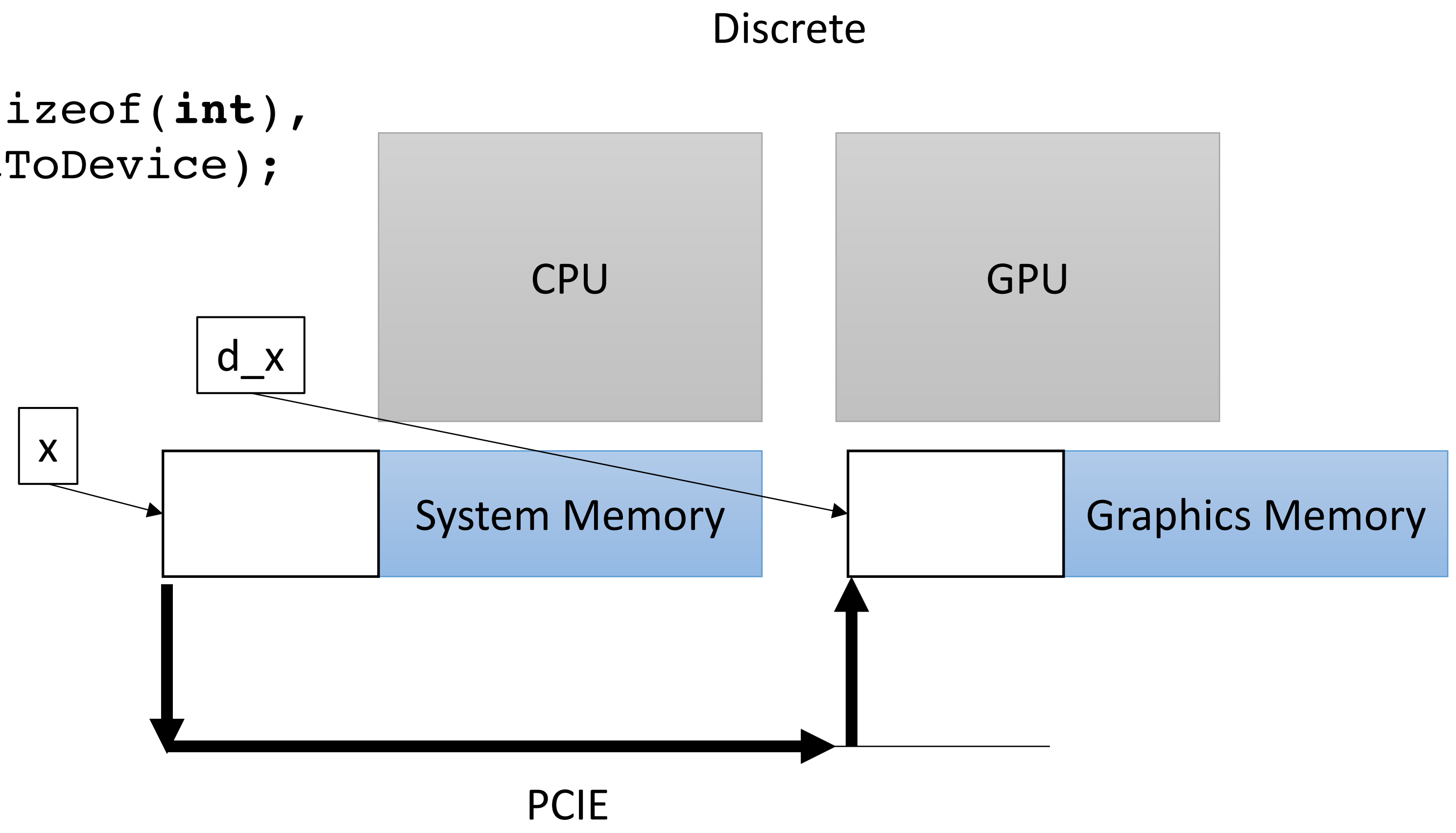
GPU set up

- Our heterogeneous, parallel, programming model

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);
```



How does this look in code?

How does this look in code?

Nothing too exciting yet.

The GPU Program

- Write a special function in your C++ code.
  - Called a Kernel
  - Use the new keyword `__global__`
  - Keywords in
    - OpenCL `__kernel`
    - Metal `kernel`

- Write it how you’d write any other function

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];
  }
}
```

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];
   }
}
```

calling the function

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

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];
  }
}
```

What in the world?
special new CUDA syntax. We will talk more soon

calling the function

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

The GPU Program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
   for (int i = 0; i < size; i++) {
     d_a[i] = d_b[i] + d_c[i];
   }
}
```

Pass in pointers to memory on the device

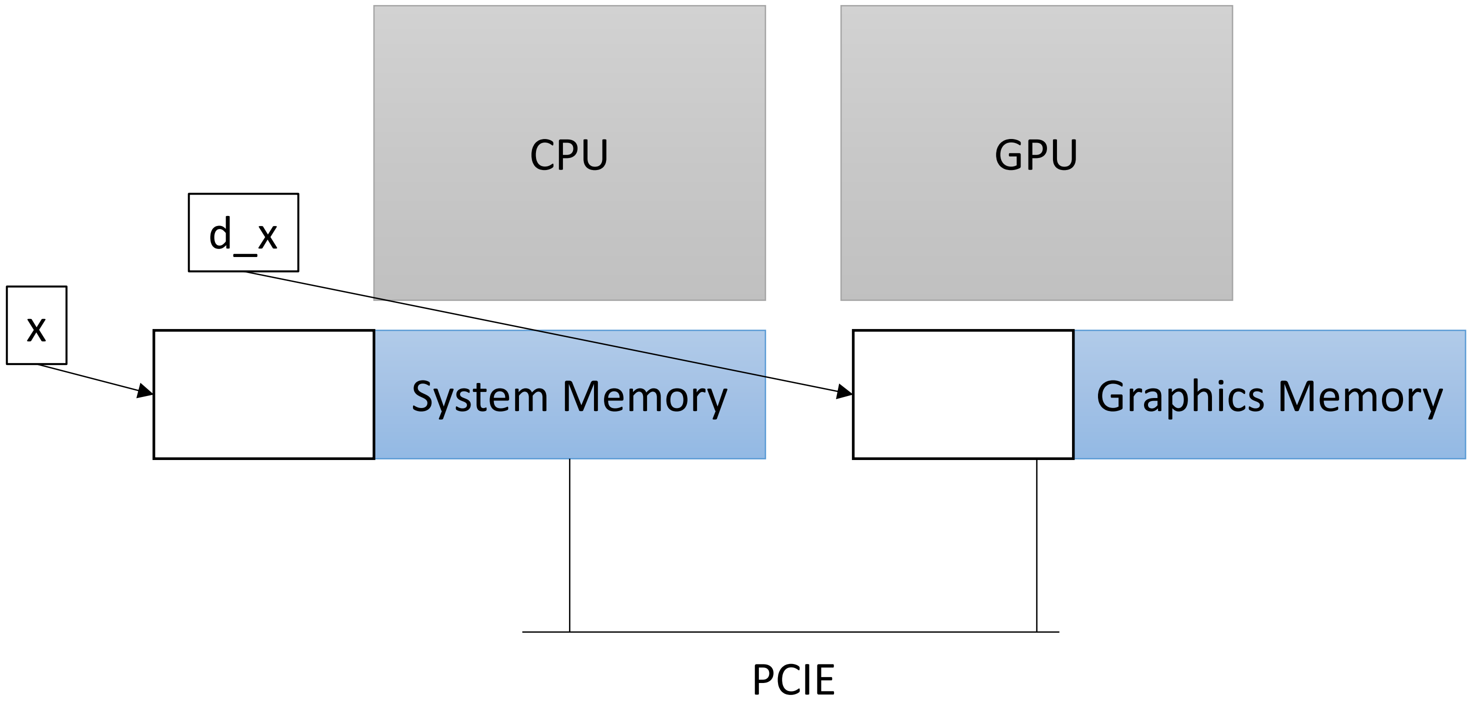
calling the function

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

GPU set up

- Our heterogeneous, parallel, programming model

Remember, GPU needs to access its own memory



The GPU Program

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
   for (int i = 0; i < size; i++) {
     d_a[i] = d_b[i] + d_c[i];
  }
}
```

Constants can be passed in regularly

calling the function

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

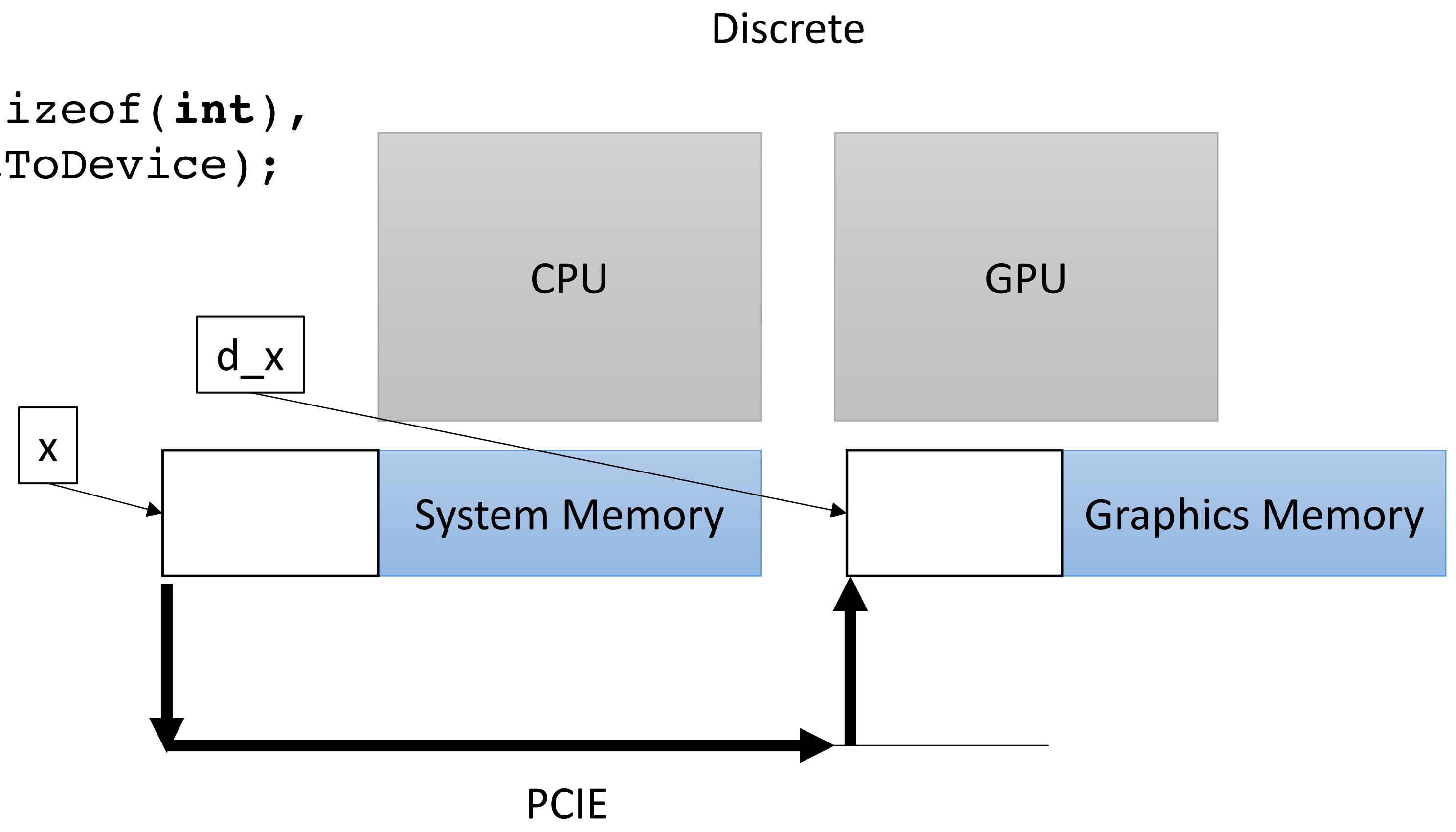
The GPU Program

Are we ready to run the program? What are we missing?

GPU set up

- Our heterogeneous, parallel, programming model

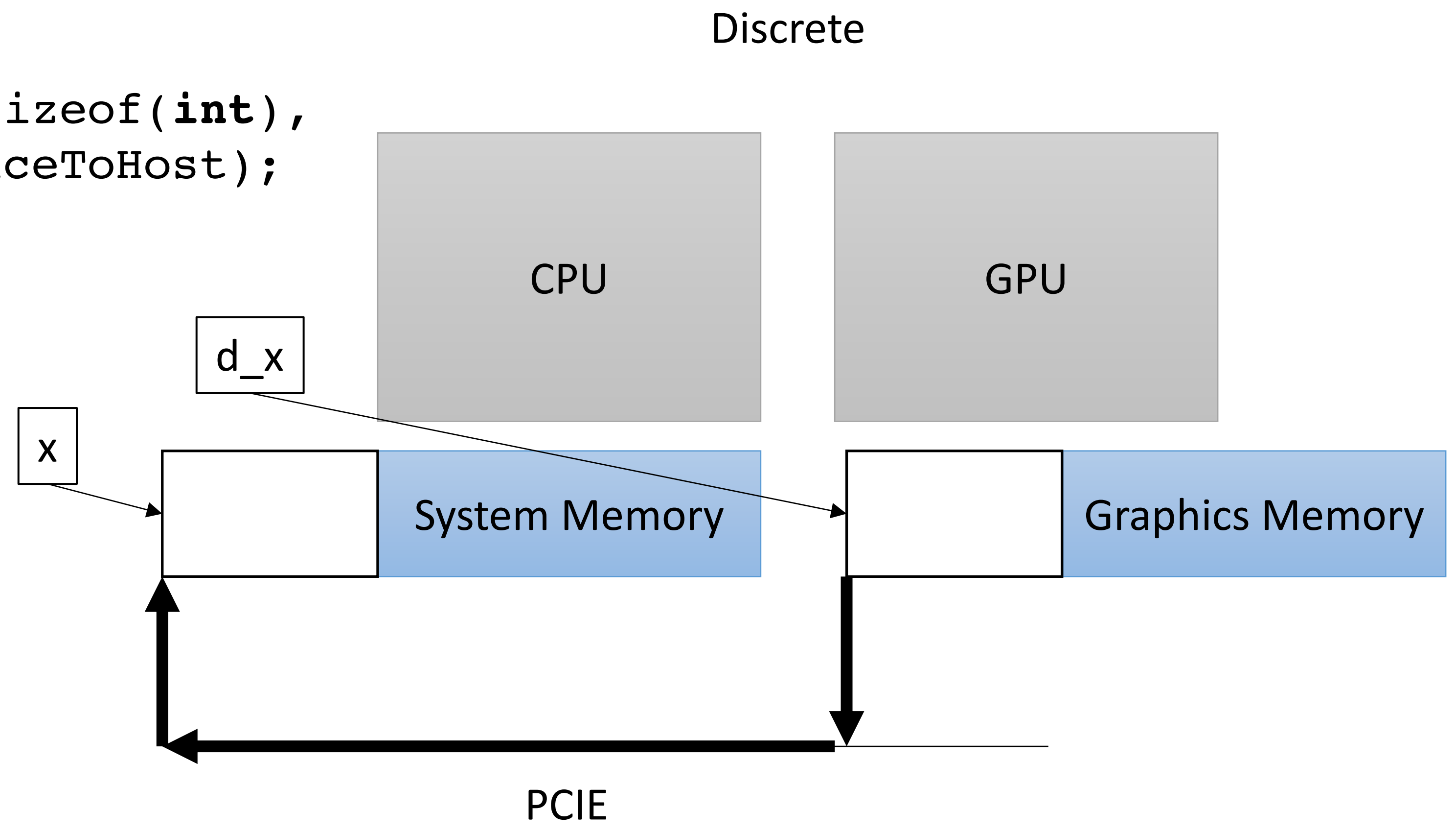
```
//initialize x on host
cudaMemcpy(d_x, x, SIZE*sizeof(int),
           cudaMemcpyHostToDevice);
```



GPU set up

- Our heterogeneous, parallel, programming model

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



The GPU Program

Finally, we can run the GPU program!

Lets see what all the hype is about

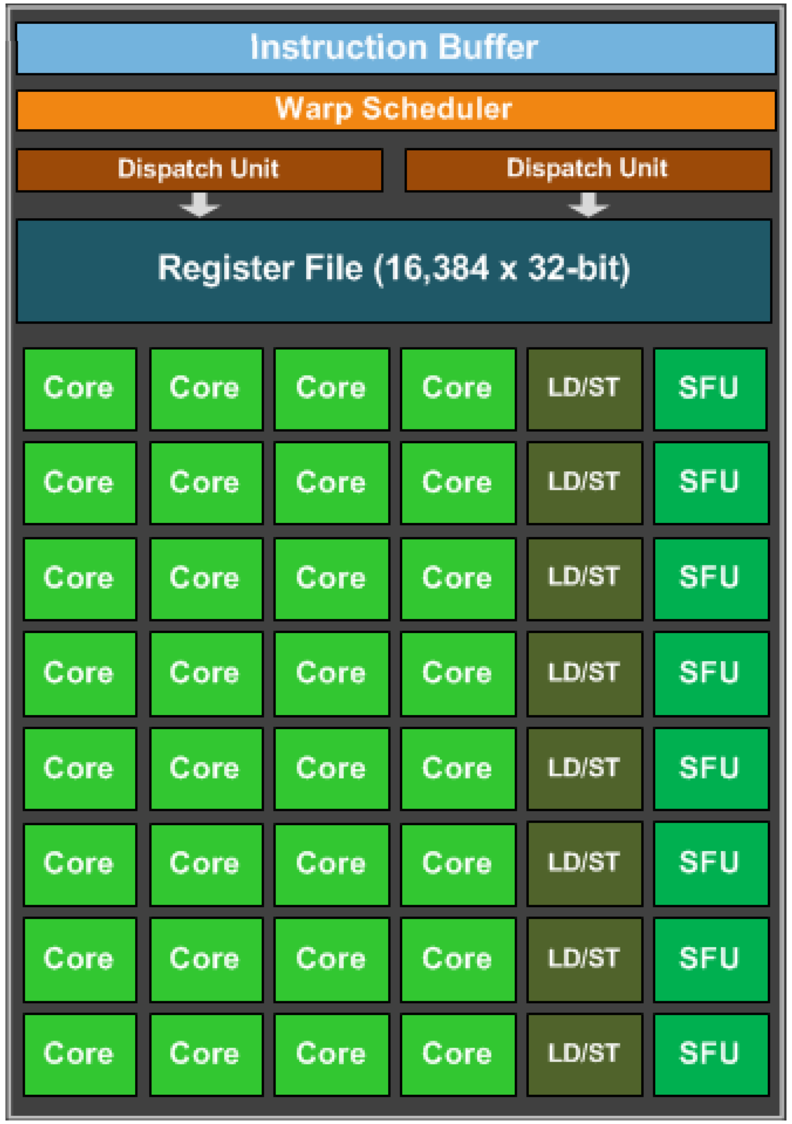
The GPU Program

😥 It didn’t do so well...

First parallelization attempt

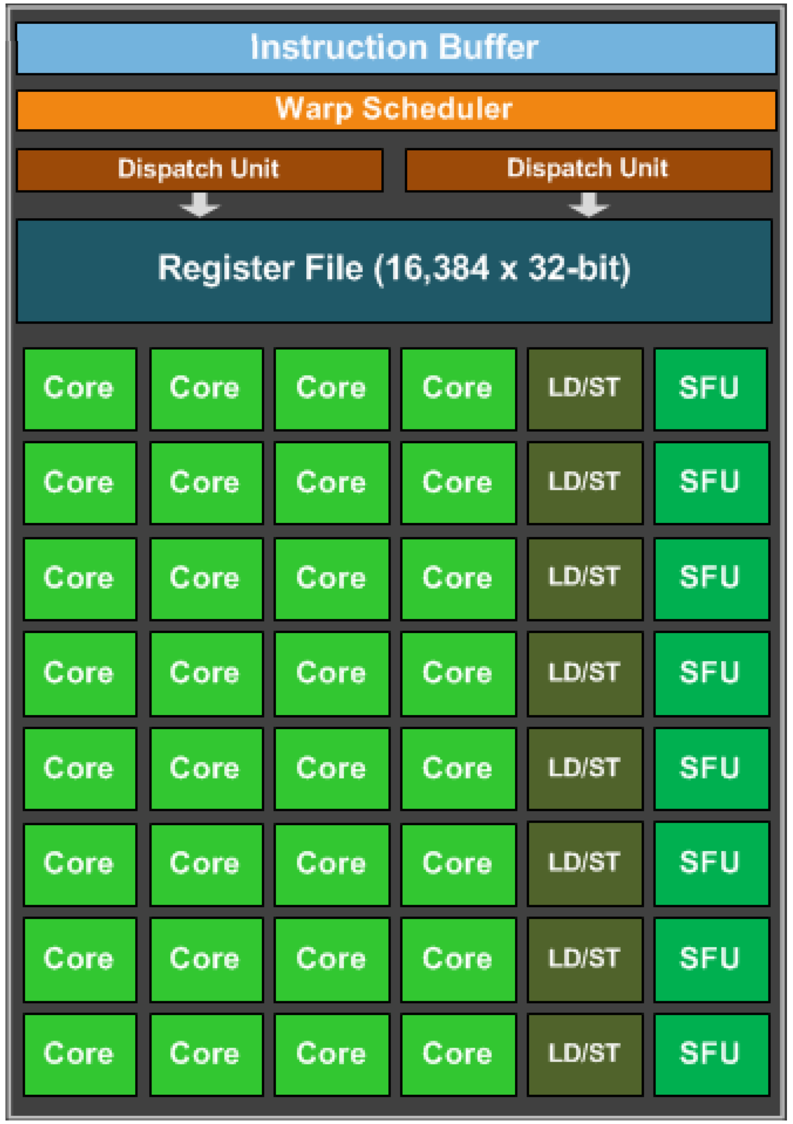
- Lets look at some GPU documentation.

- The Maxwell whitepaper shows a diagram of one of the GPU cores



https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf woah, 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) {
   for (int i = 0; i < size; i++) {
     d_a[i] = d_b[i] + d_c[i];
  }
}
```

calling the function

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

First parallelization attempt

```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
   for (int i = 0; i < size; 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 to launch the program with

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

number of threads

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

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

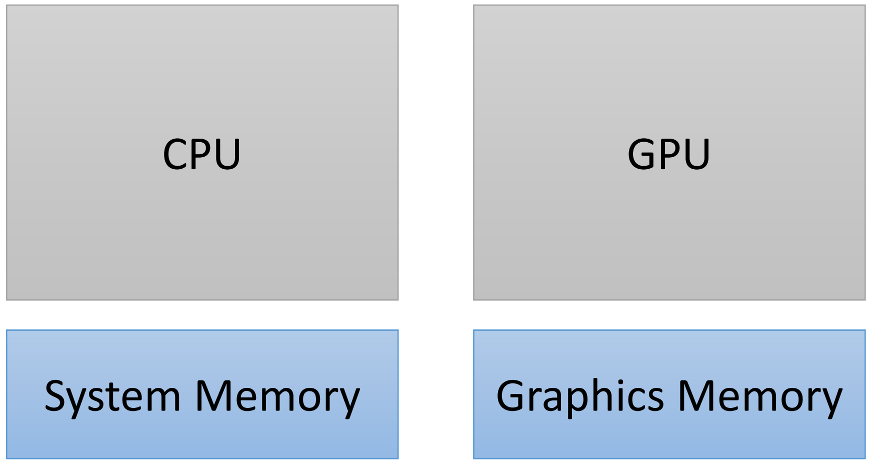
First parallelization attempt

Lets try it! What do we think?

First parallelization attempt

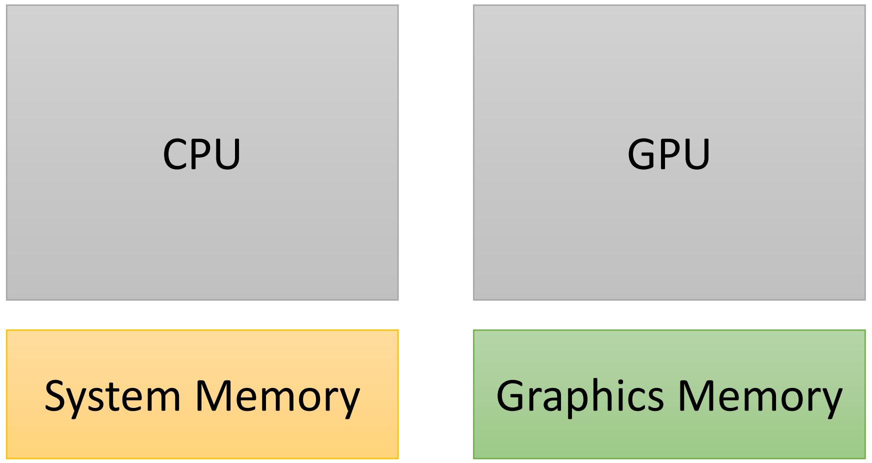
Getting better but we have a long ways to go!

GPU Memory



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

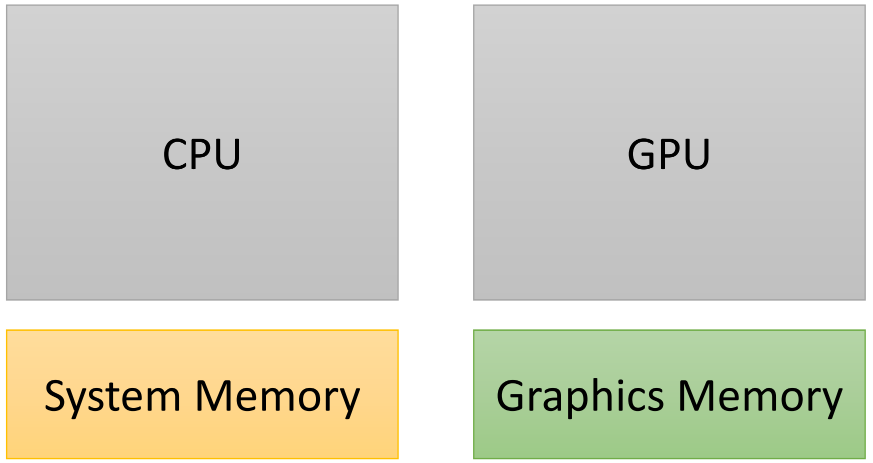
16-lane highway on a windy road driven by semi trucks

GPU Memory

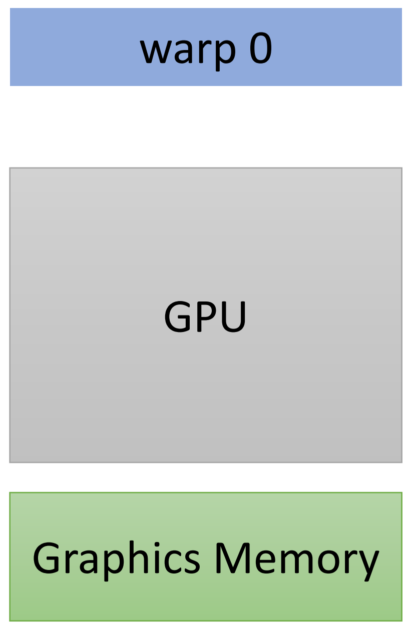
bandwidth:
~**700 GB/s** for GPU
~**50 GB/s** for CPUs

memory Latency:
~**600** cycles for GPU memory
~**200** cycles for CPU memory

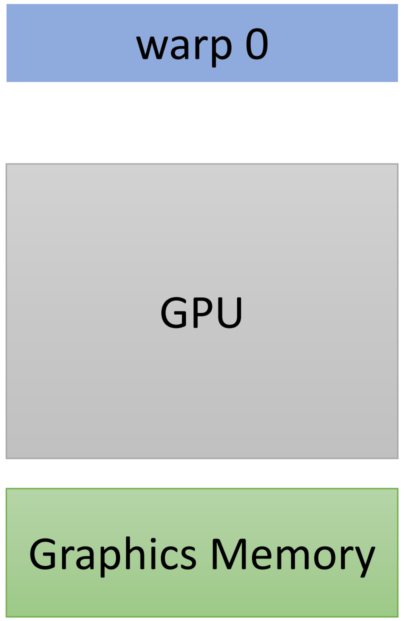
Cache Latency:
~**28** cycles for L1 hit for GPU
~**4** cycles for L1 hit on CPUs



Preemption and concurrency?

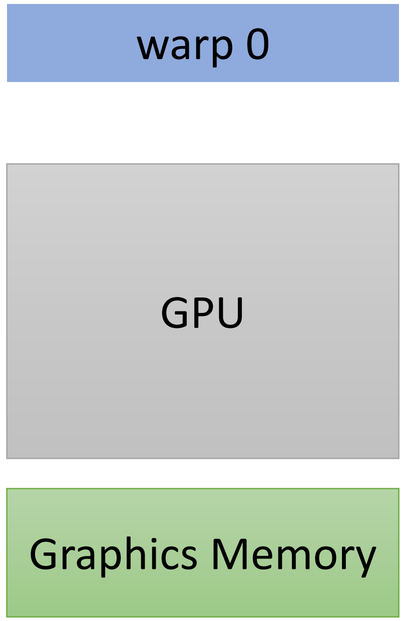


Preemption and concurrency?



all threads load from memory.

Preemption and concurrency?



all threads load from memory.

600 cycles!

Preemption and concurrency?

warp 0

warp 1

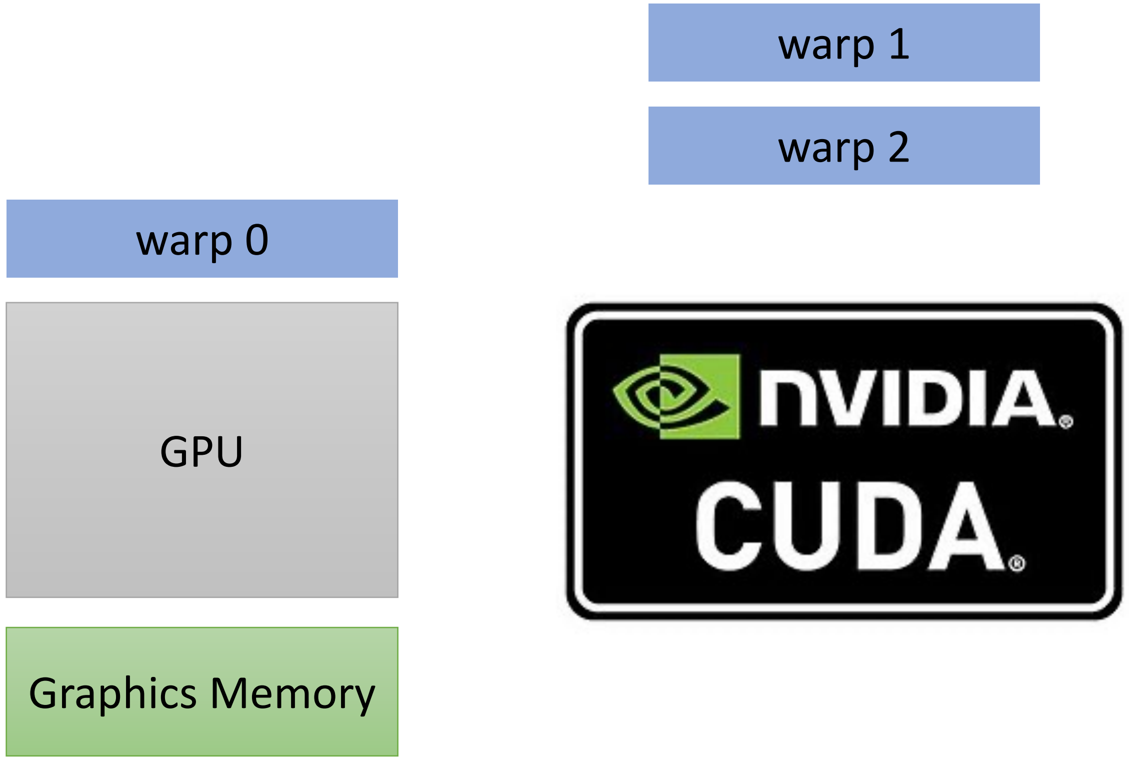
warp 2

We can hide latency through preemption and concurrency!

GPU

Graphics Memory

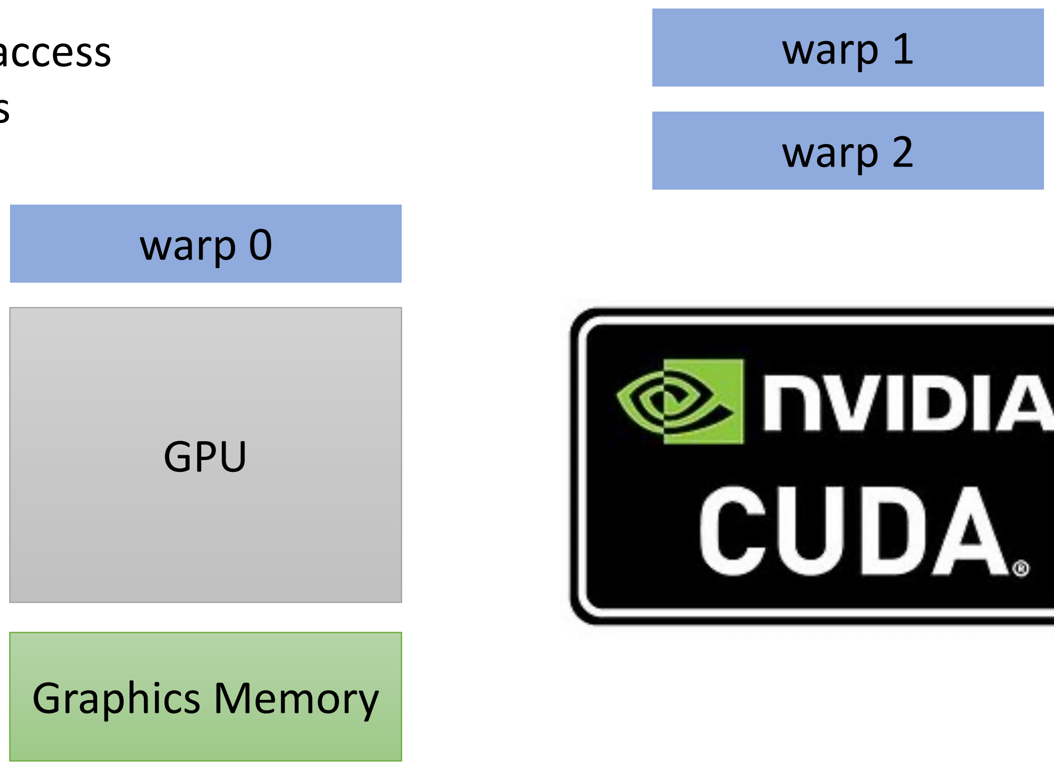
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

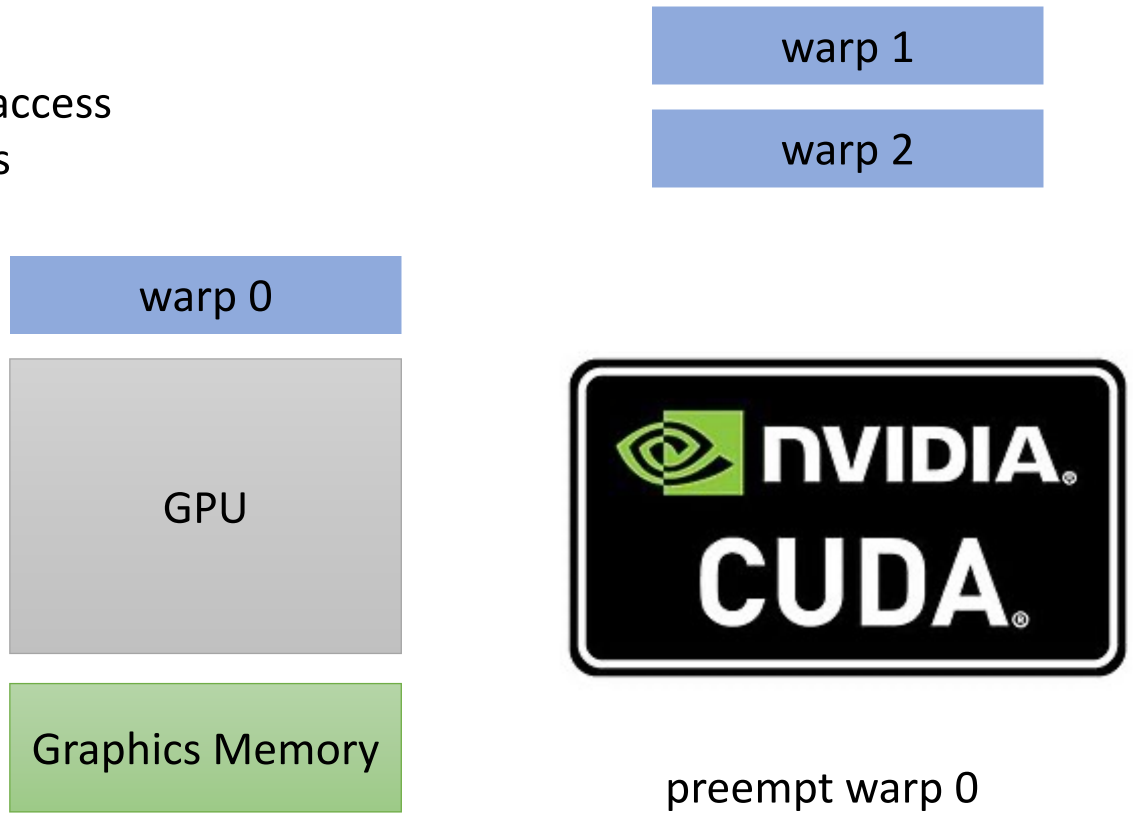
memory access
600 cycles



We can hide latency through preemption and concurrency!

Preemption and concurrency?

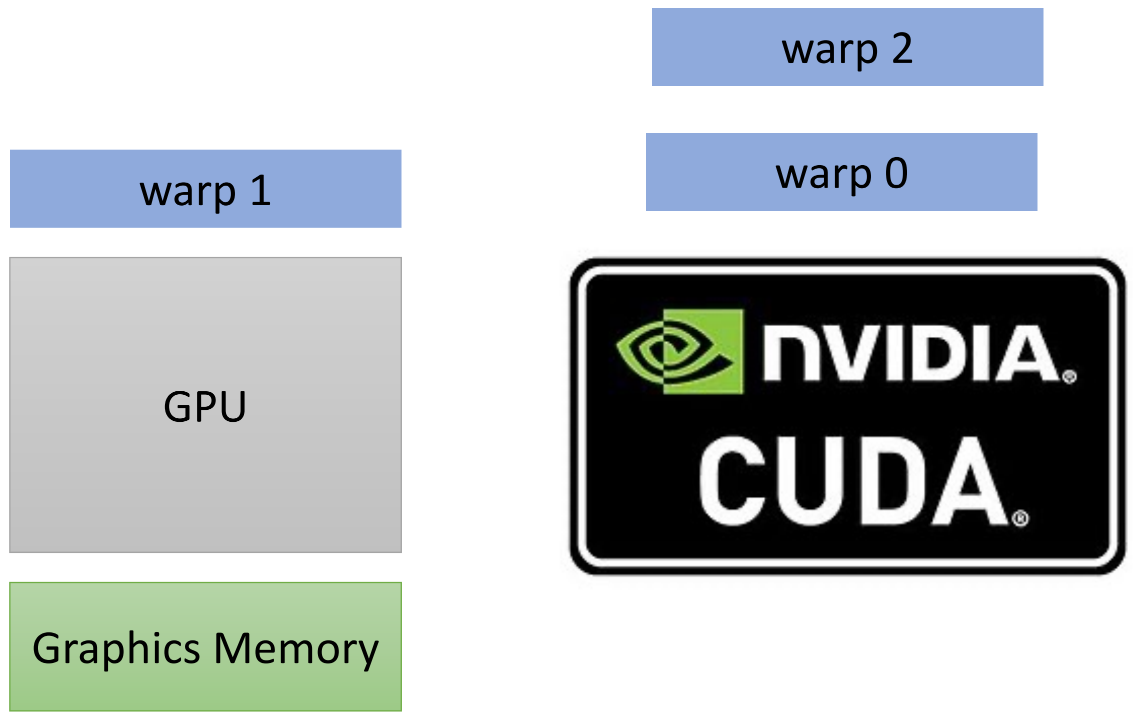
memory access
600 cycles



preempt warp 0
and put warp 1 on

We can hide latency through
preemption and concurrency!

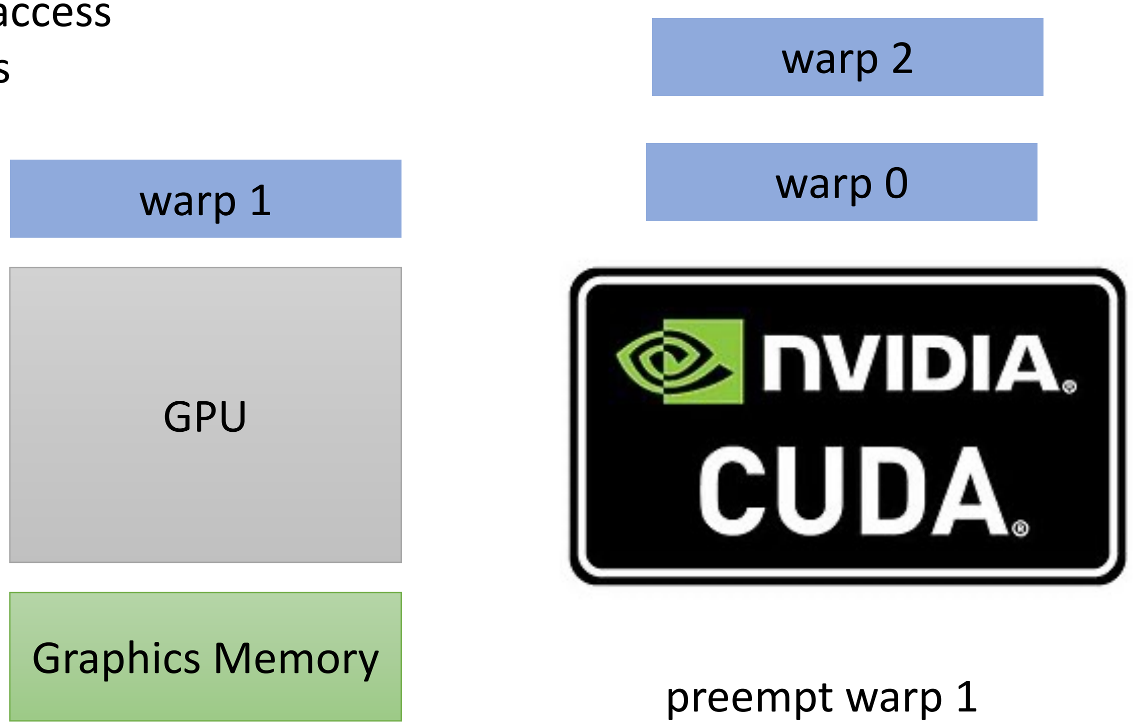
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

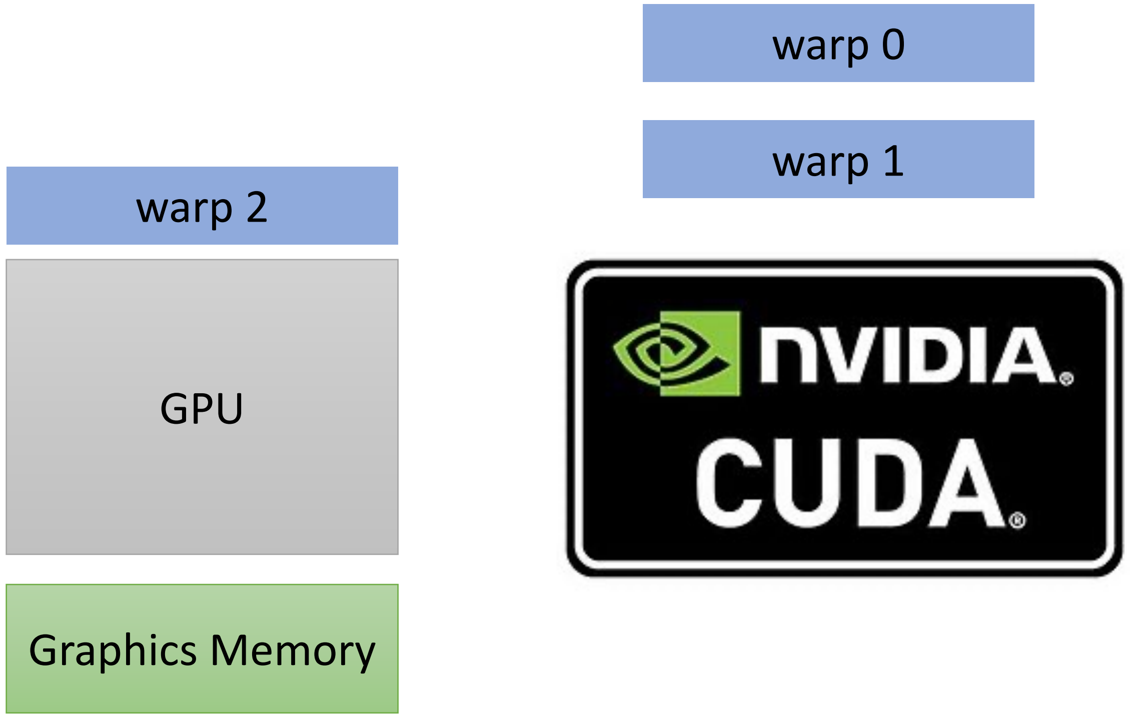
memory access
600 cycles



preempt warp 1
and put warp 2 on

We can hide latency through
preemption and concurrency!

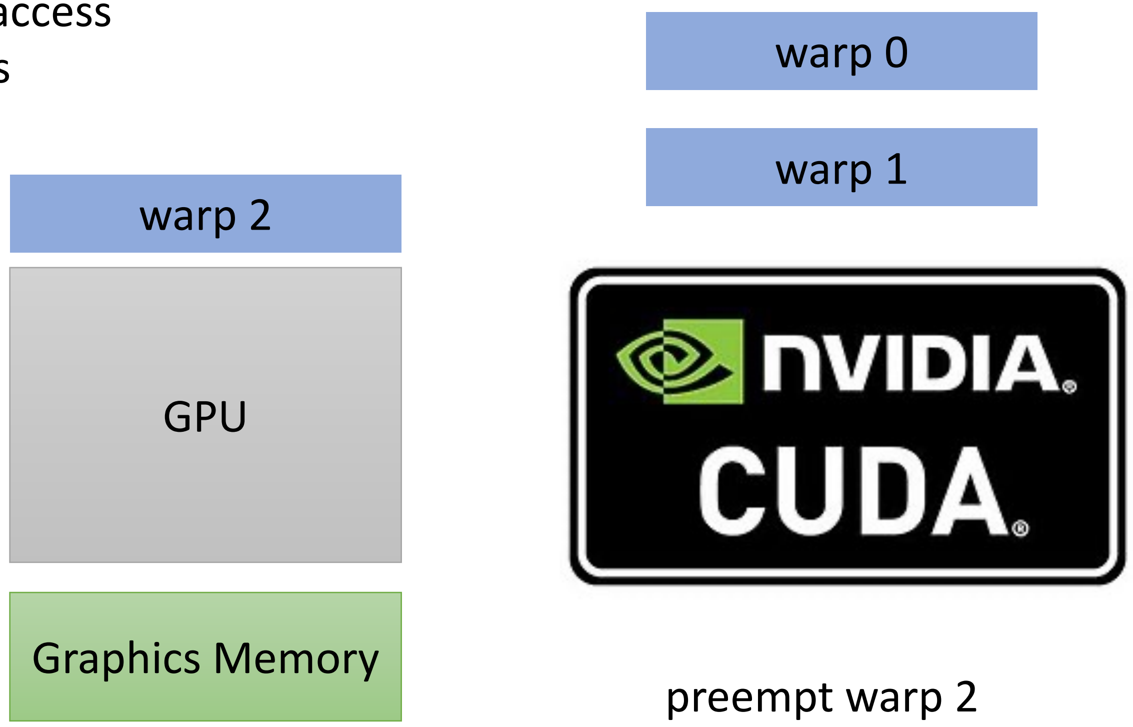
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

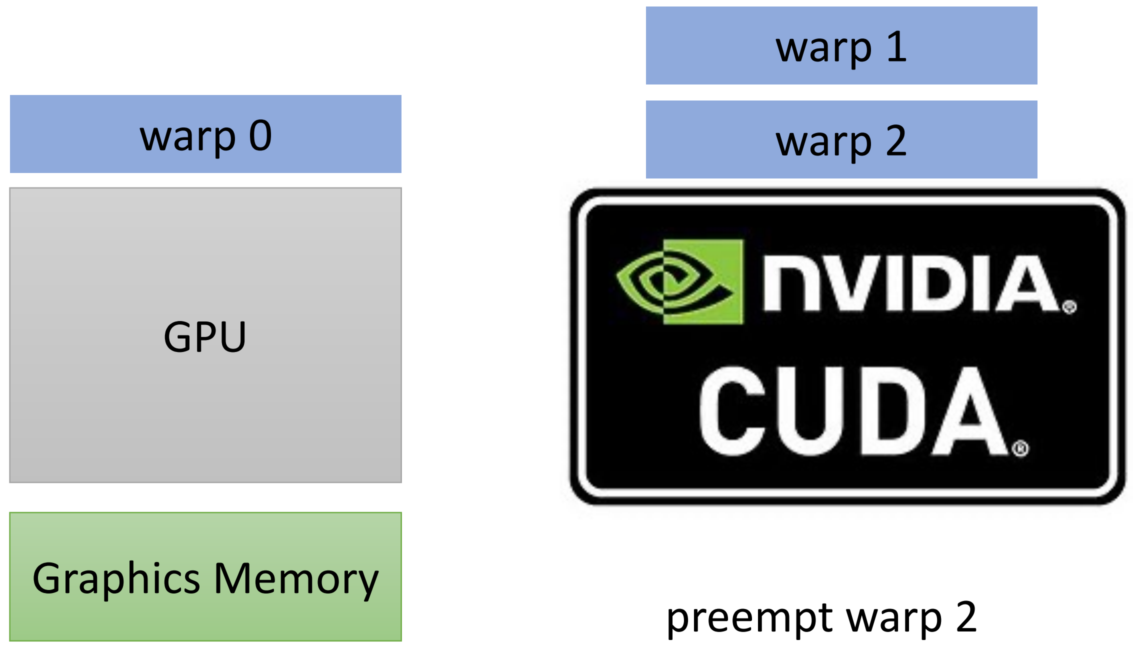


preempt warp 2
and put warp 0 on

We can hide latency through
preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!



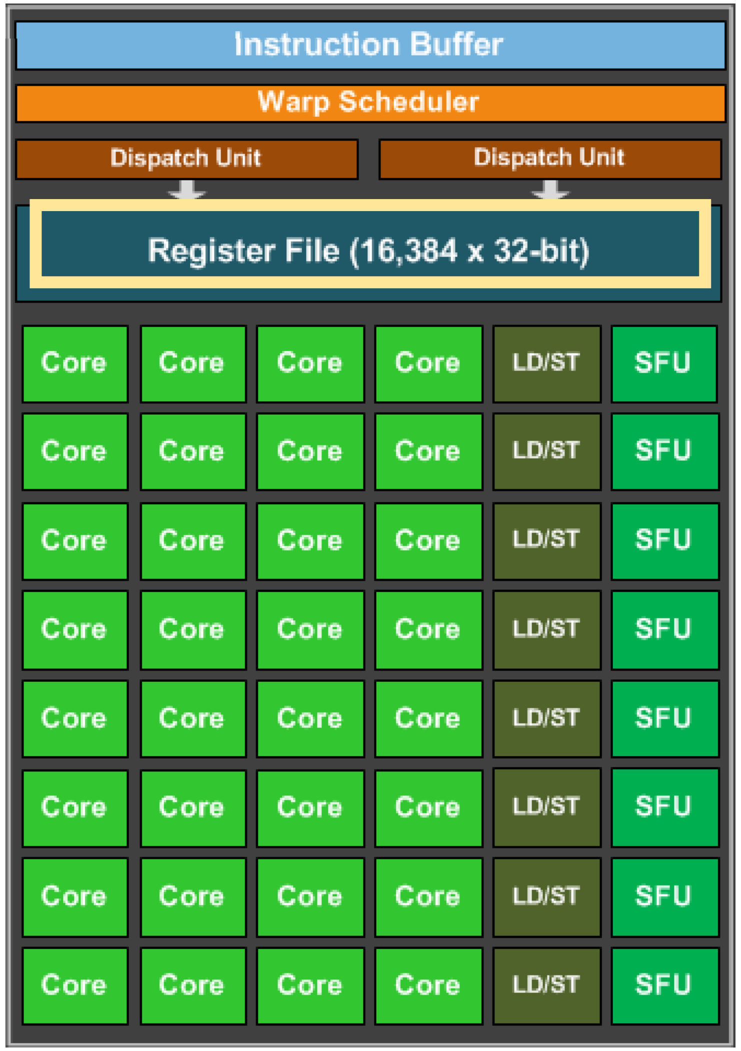
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

But wait, I thought preemption was expensive?

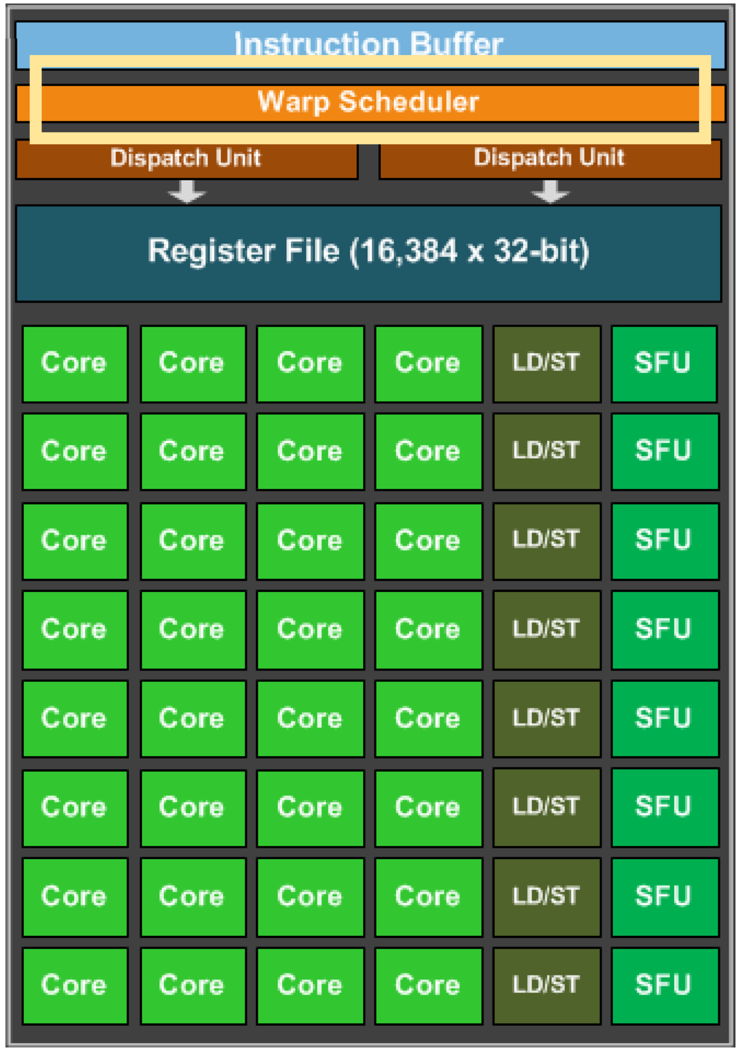
Preemption and concurrency?



But wait, I thought preemption was expensive?

Registers all stay on chip

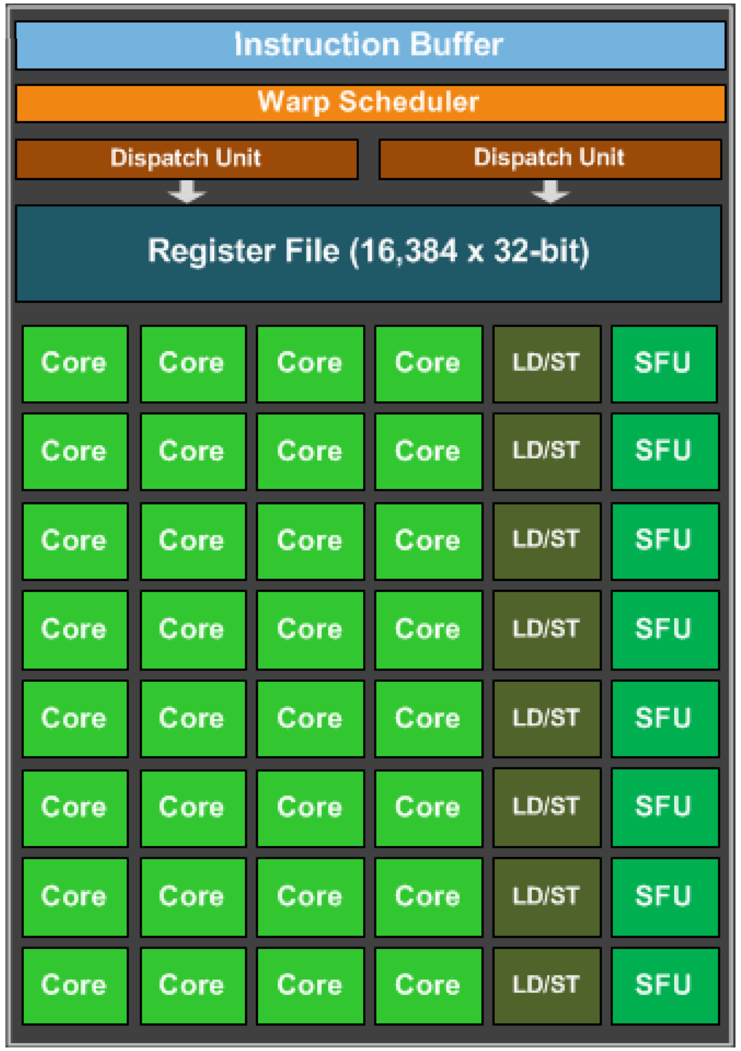
Preemption and concurrency?



But wait, I thought preemption was expensive?

dedicated scheduler logic

Preemption and concurrency?



But wait, I thought preemption was expensive?

bound on number of warps: 32

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<<<1,32>>>(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 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

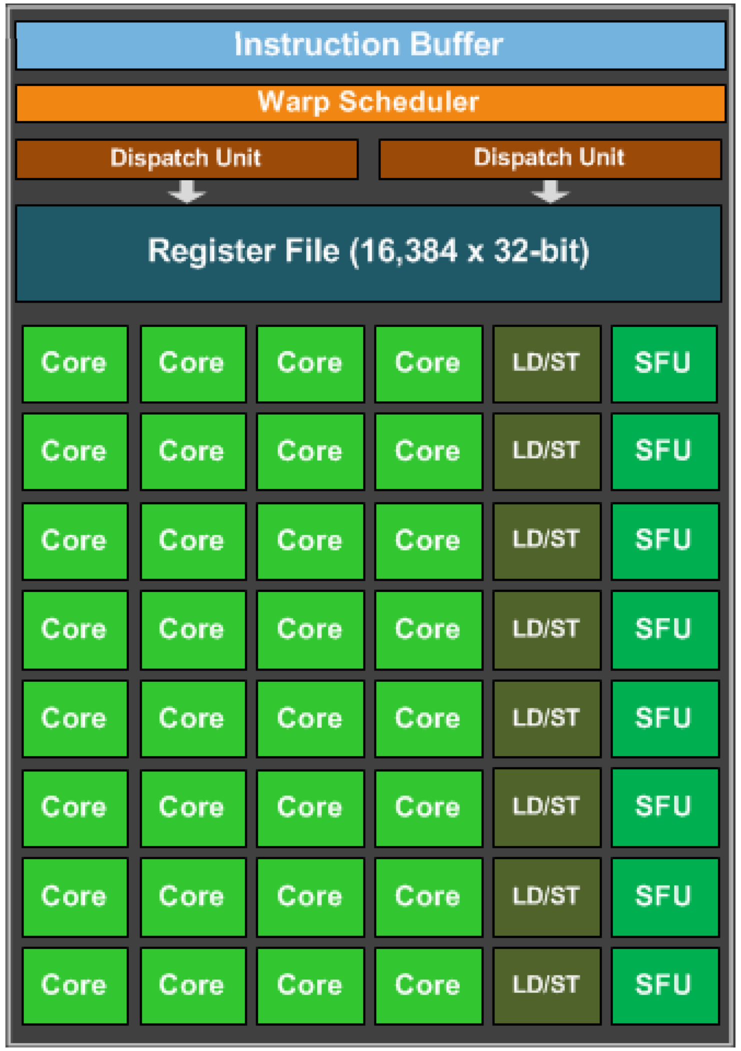
Lets try it! What do we think?

Concurrent warps

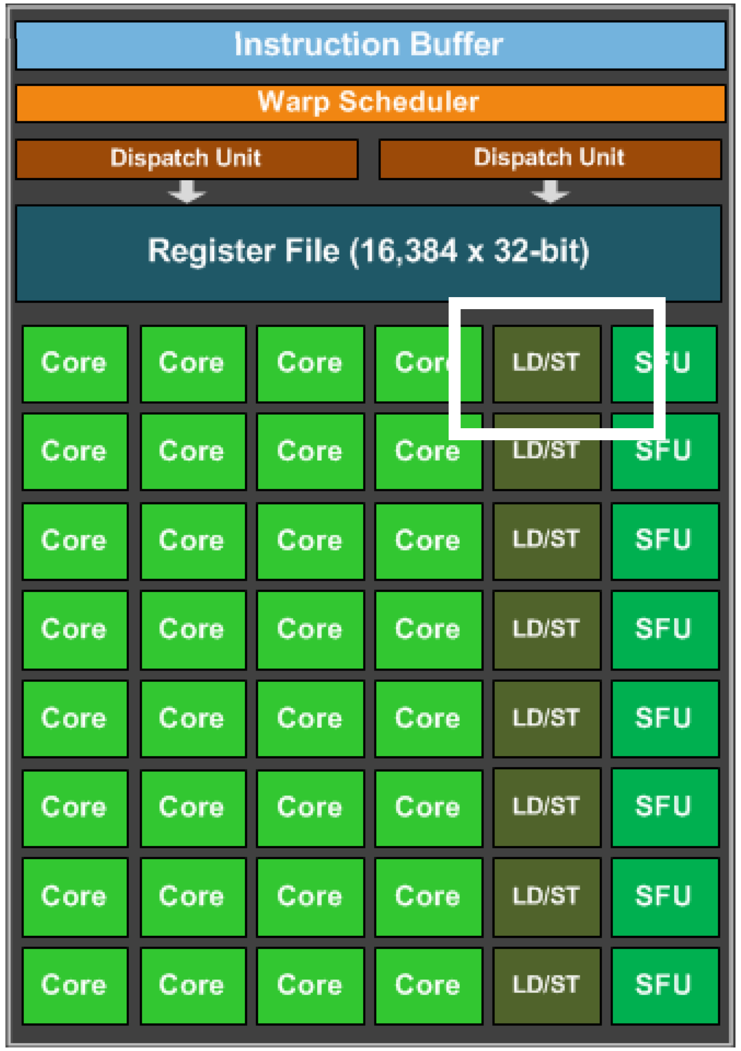
Lets try it! What do we think? 😀

Getting better!

Optimizing memory accesses



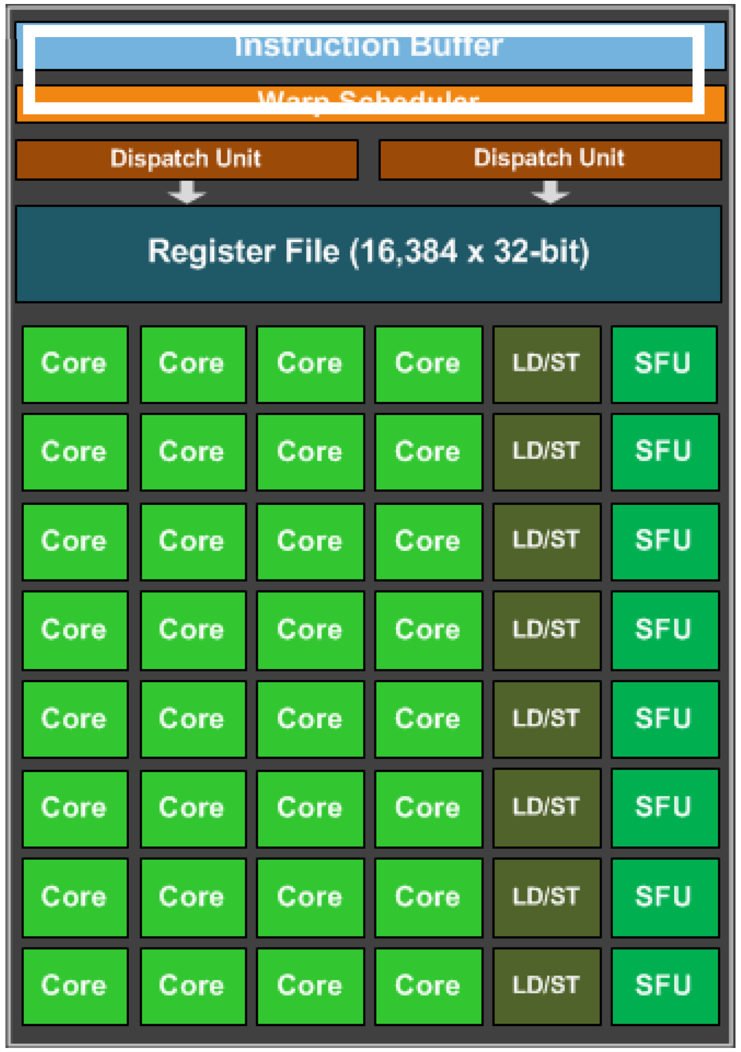
Optimizing memory accesses



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

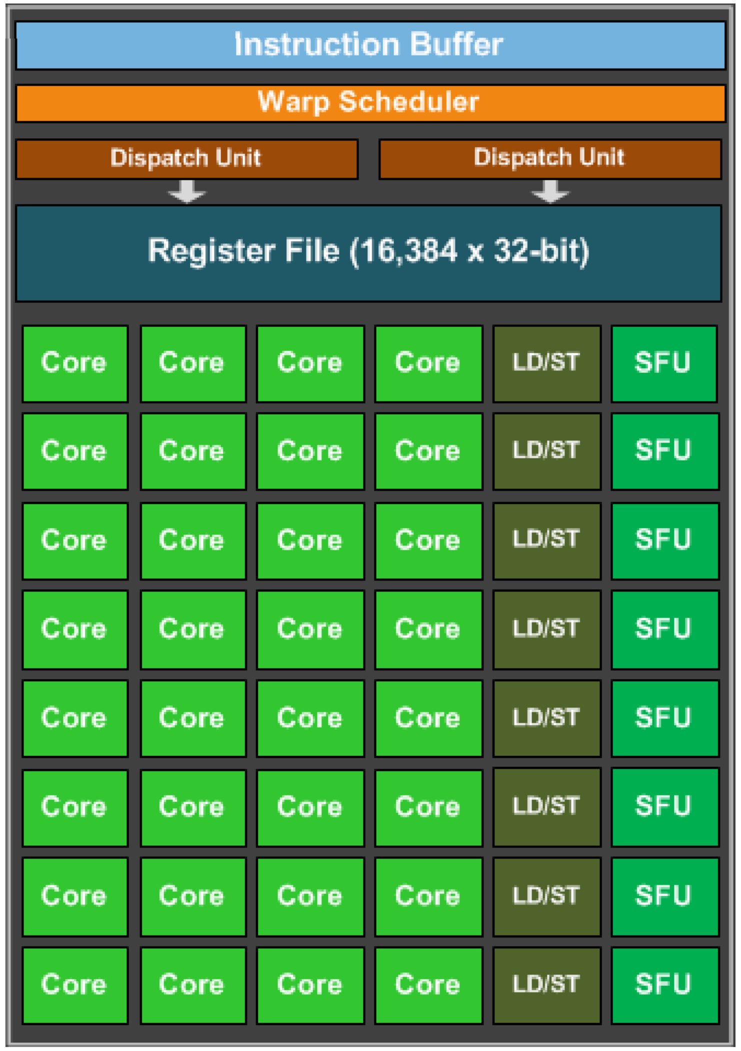


This is the instruction cache... Why doesn't every core have a instruction buffer to keep track of its program?

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

Why doesn't every core have one?

Warp execution



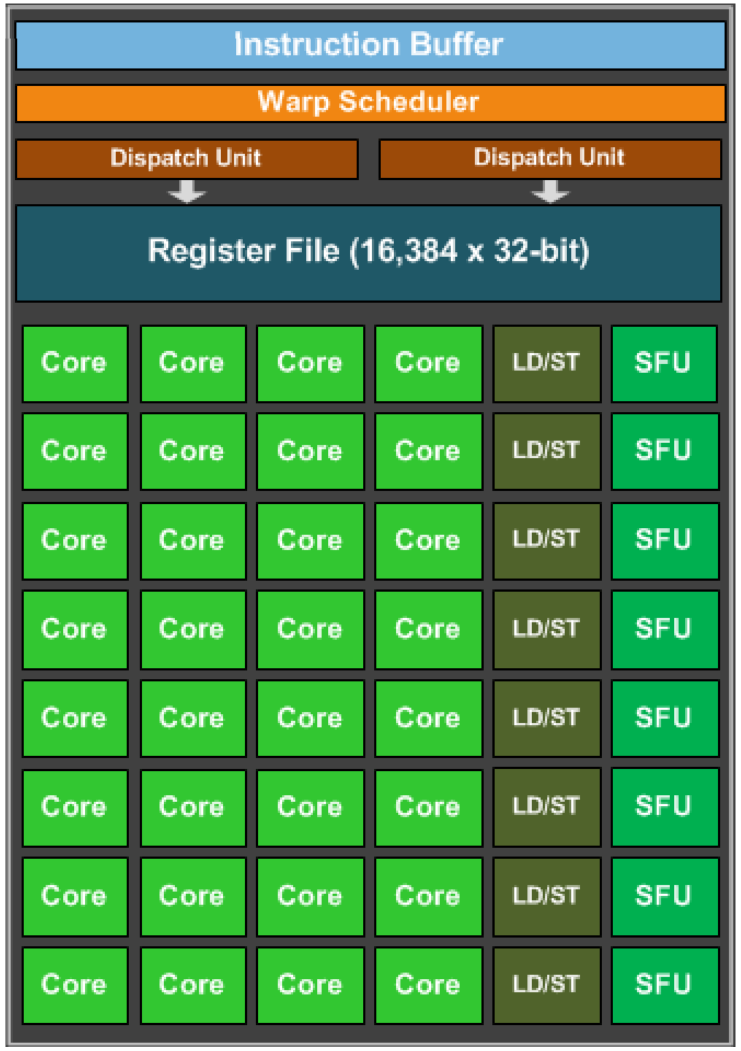
Groups of 32 threads are called a “warp”

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”

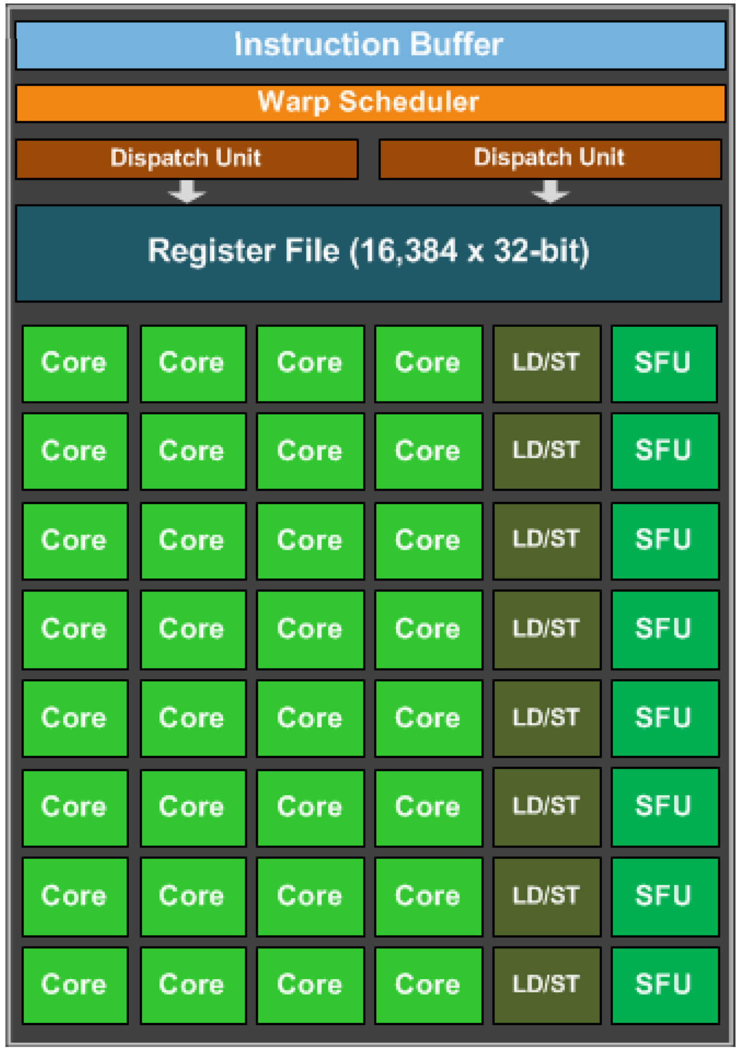
They are executed in lock-step, i.e. they all execute the same instruction at the same time



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

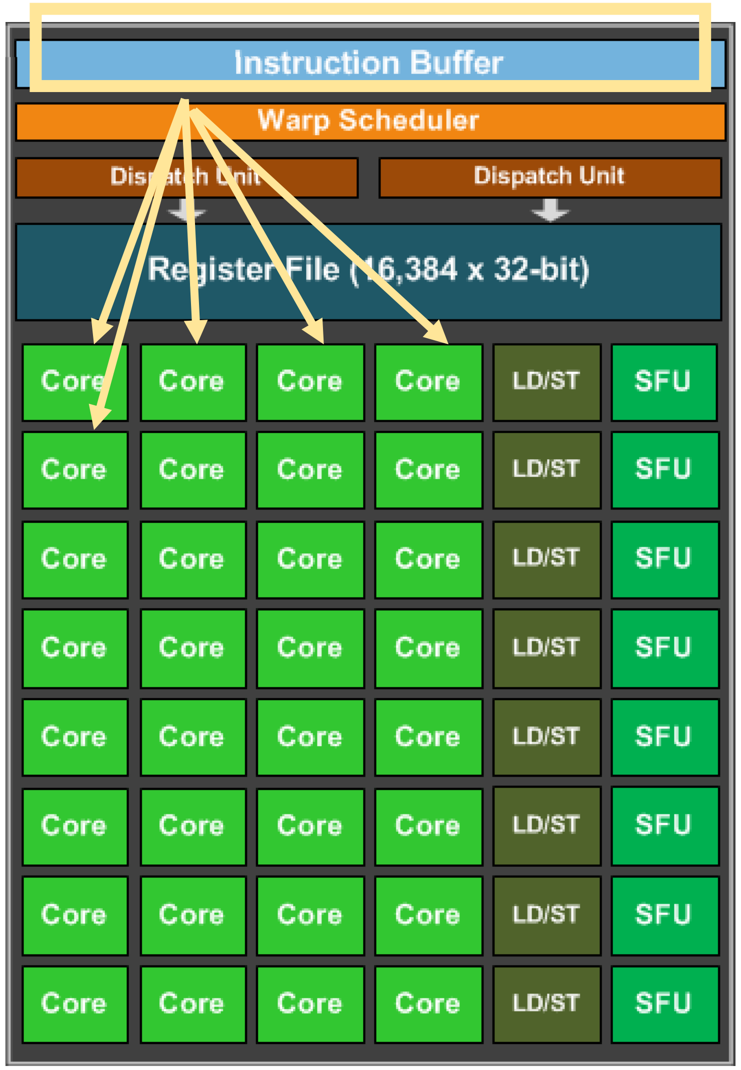
Program:

```
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.

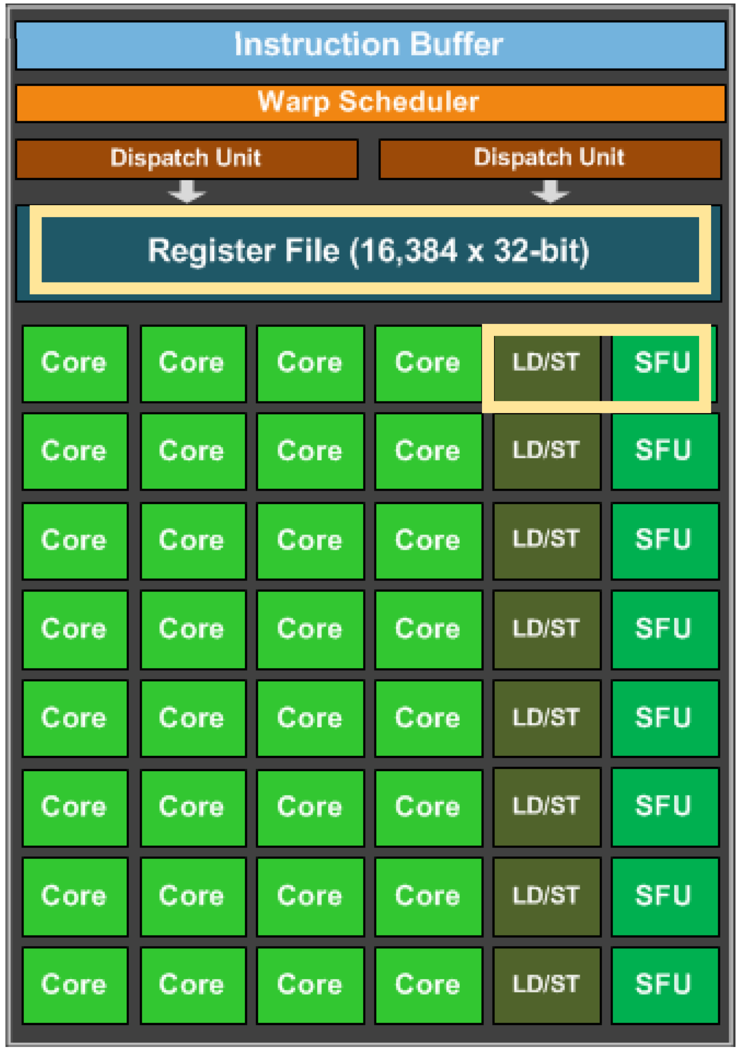
Program:

```
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)

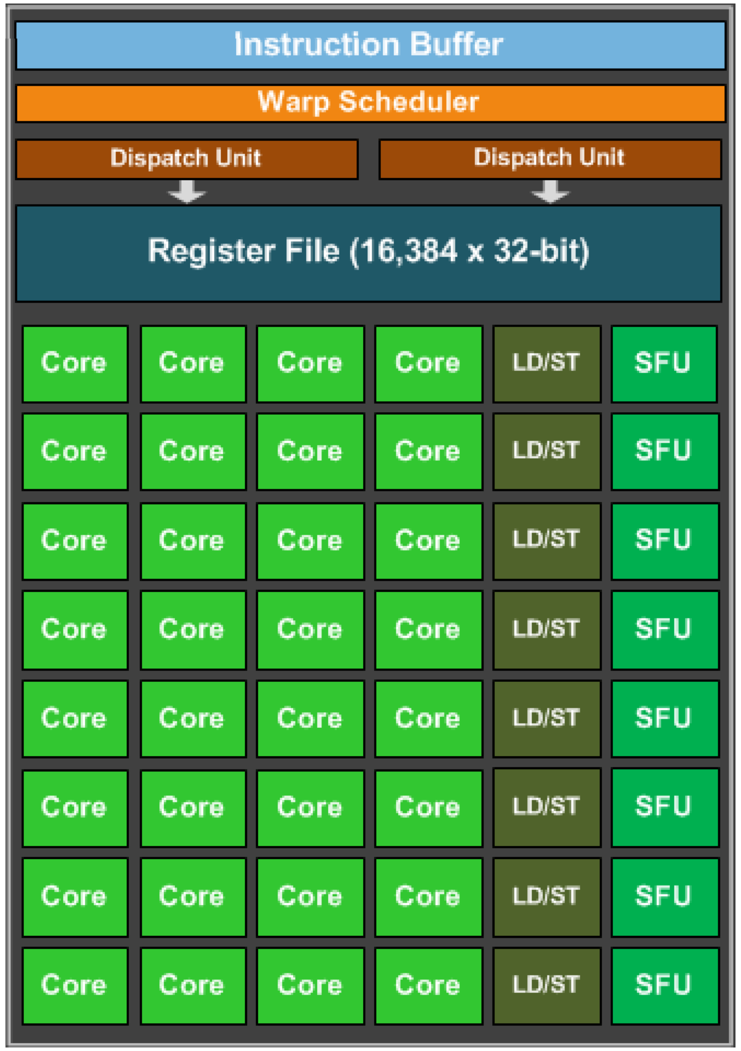
Program:

```
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

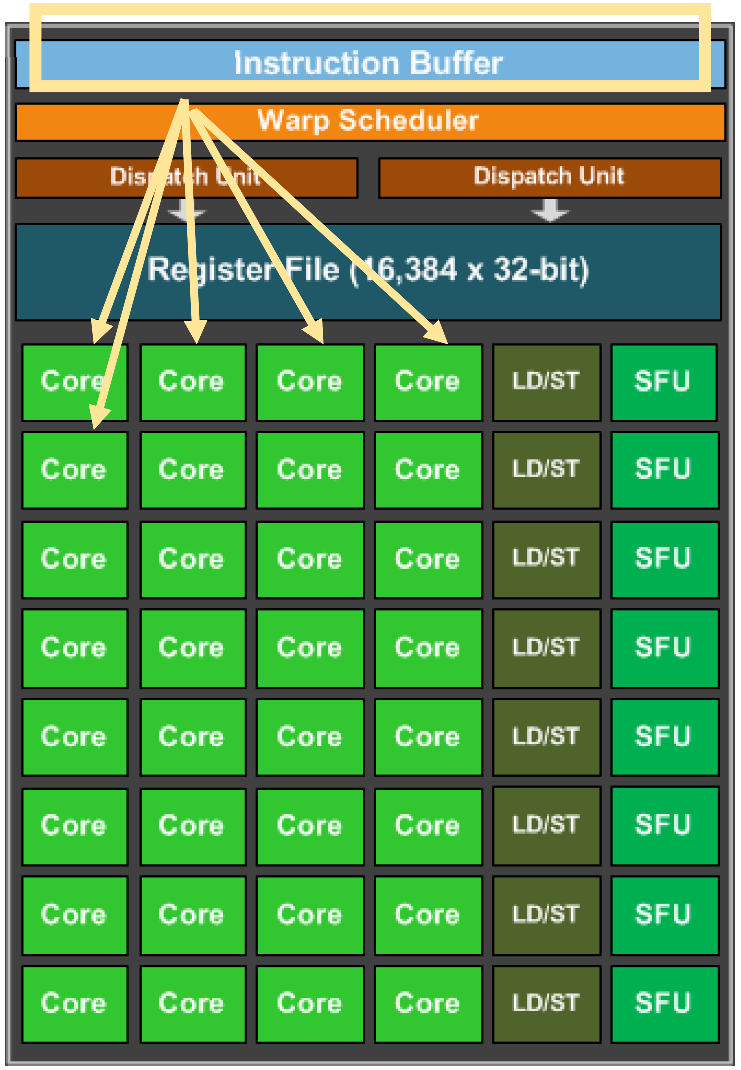
Program:

```
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:

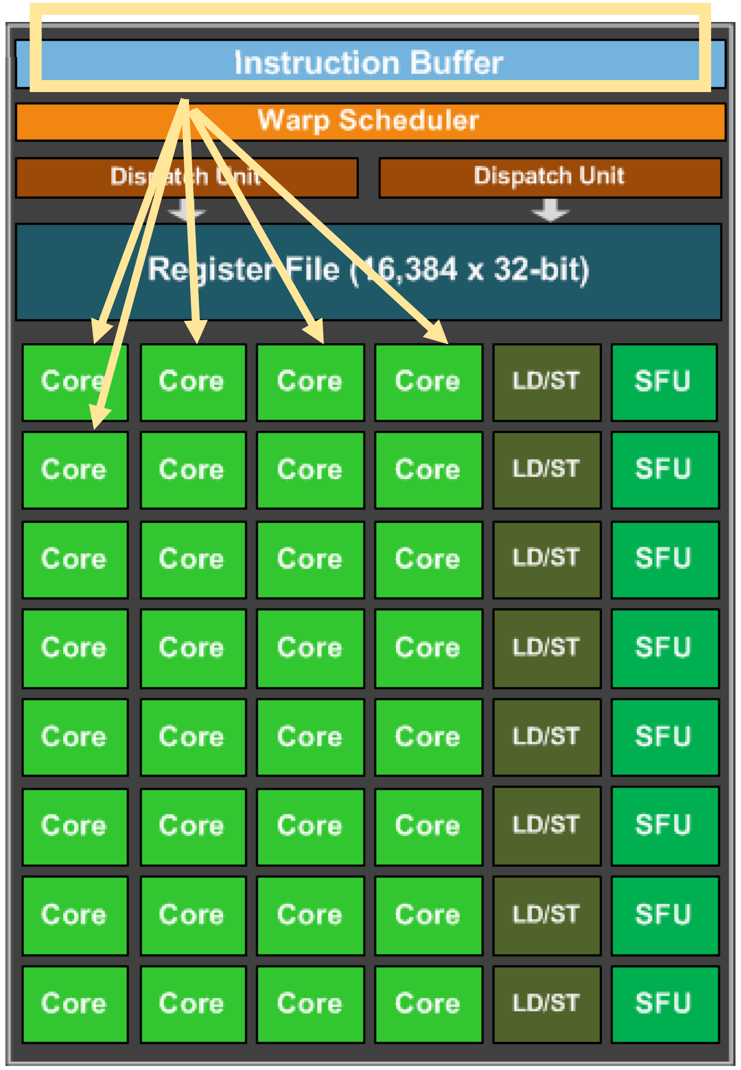
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

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.

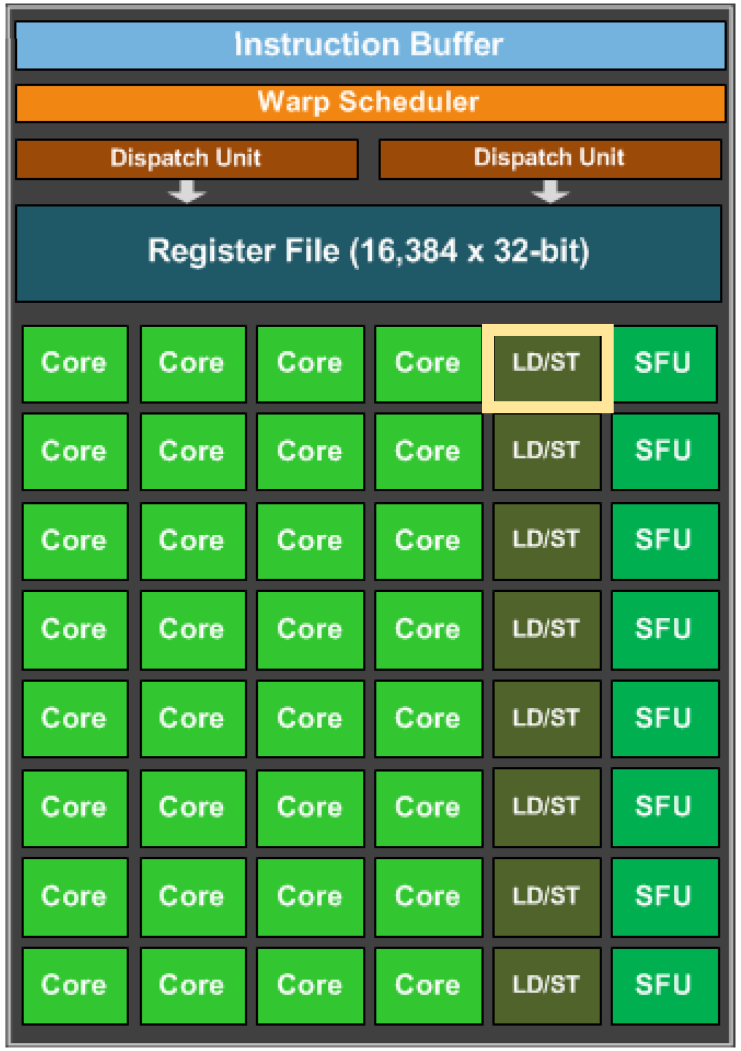
Program:

```
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

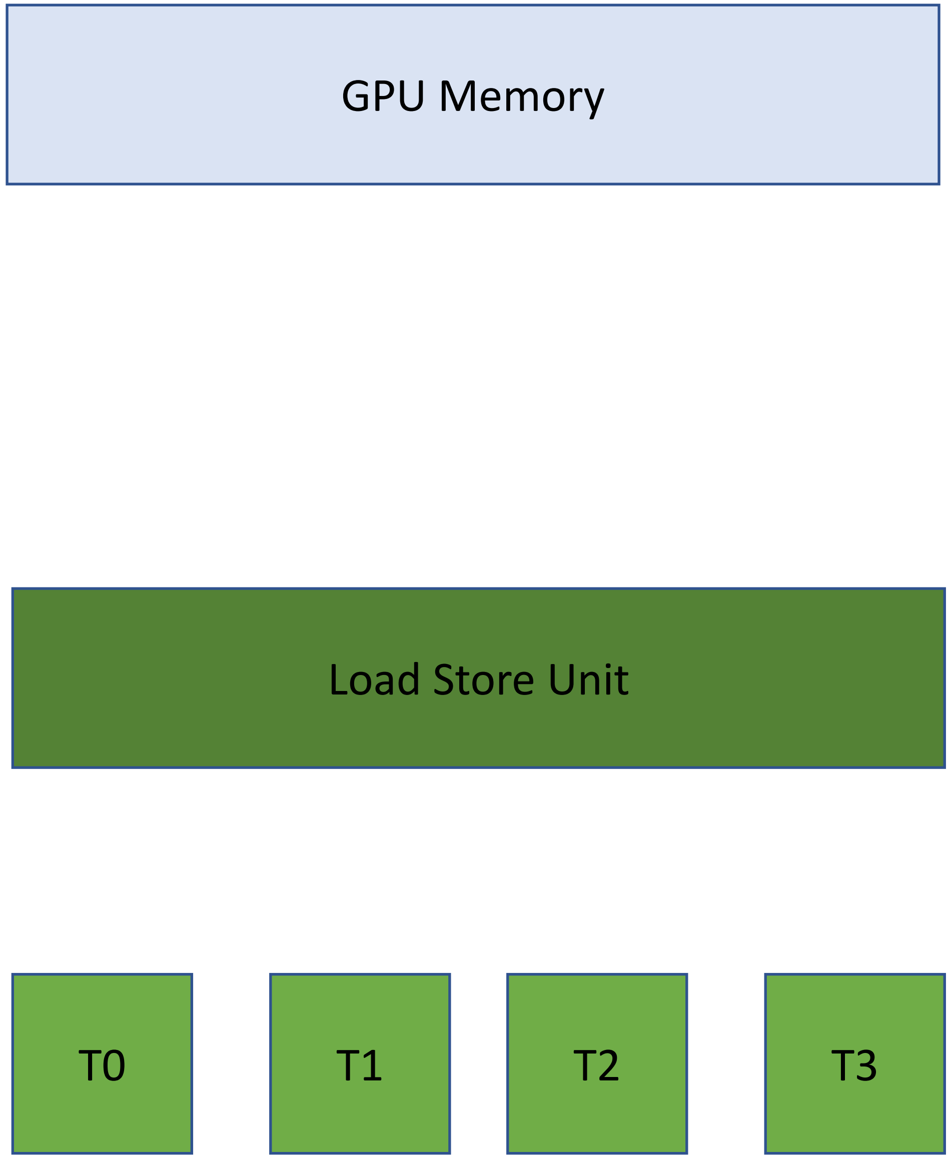


Program:

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

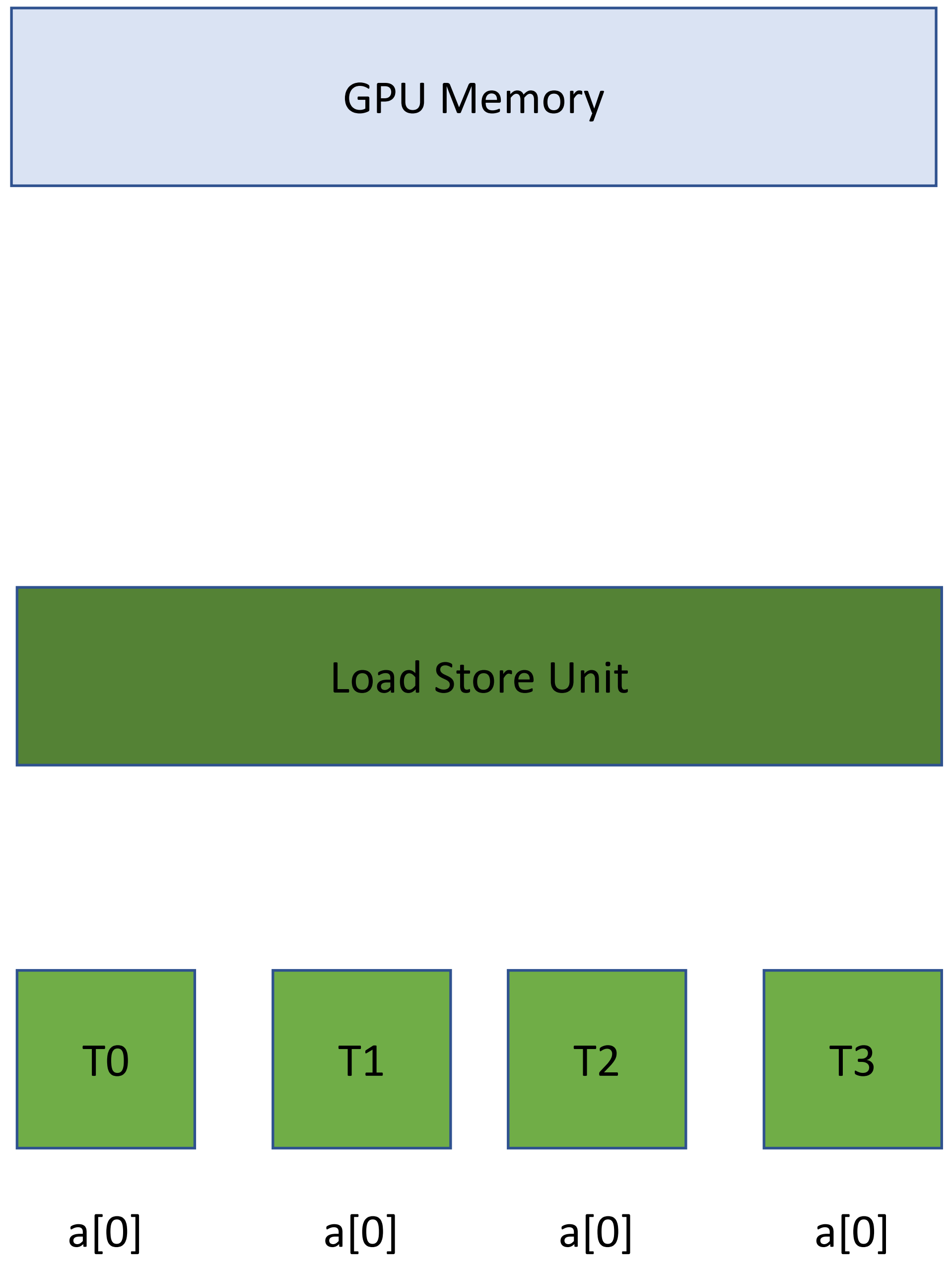
4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen



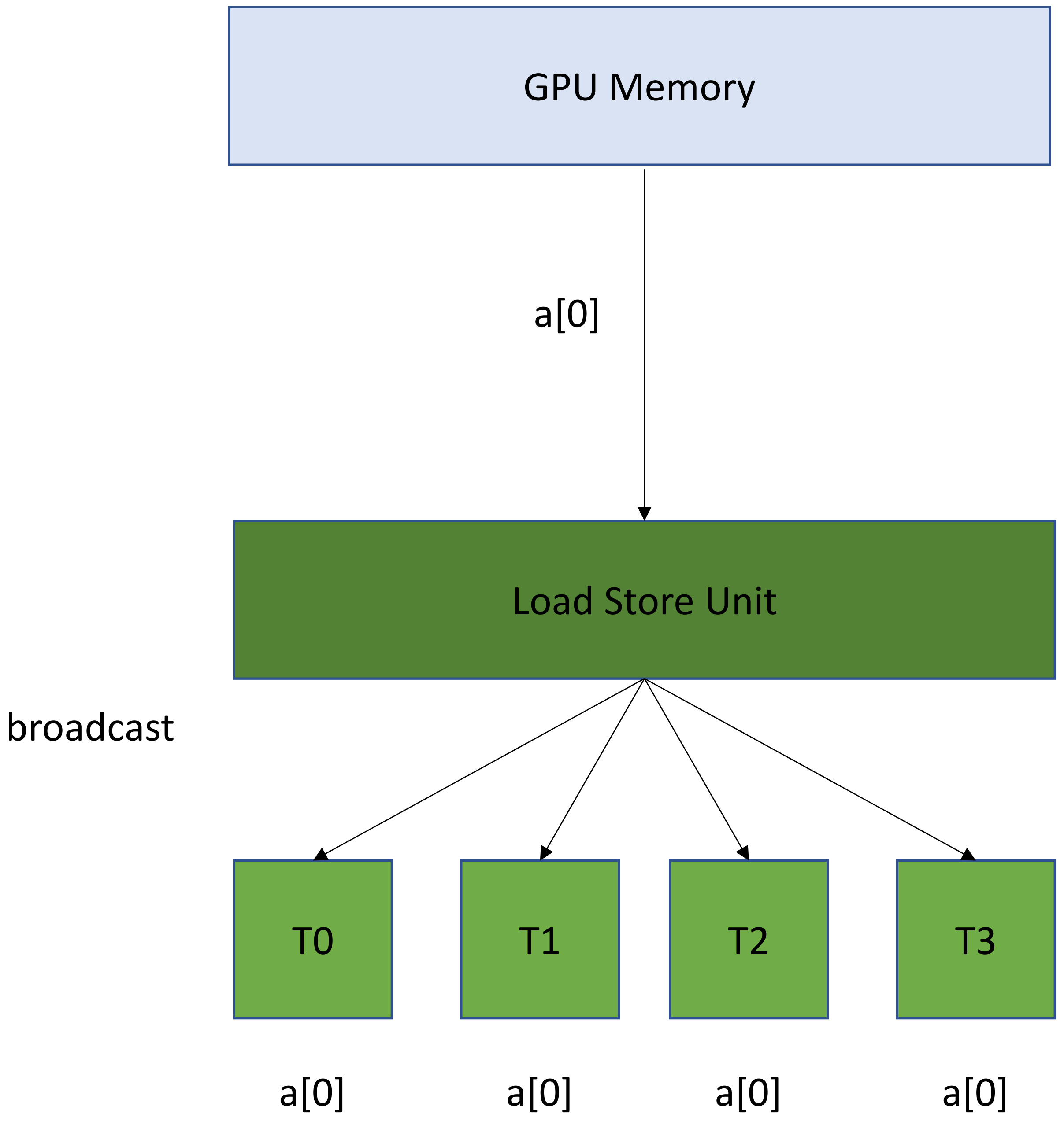
4 cores are accessing memory. What can happen

All read the same value



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.

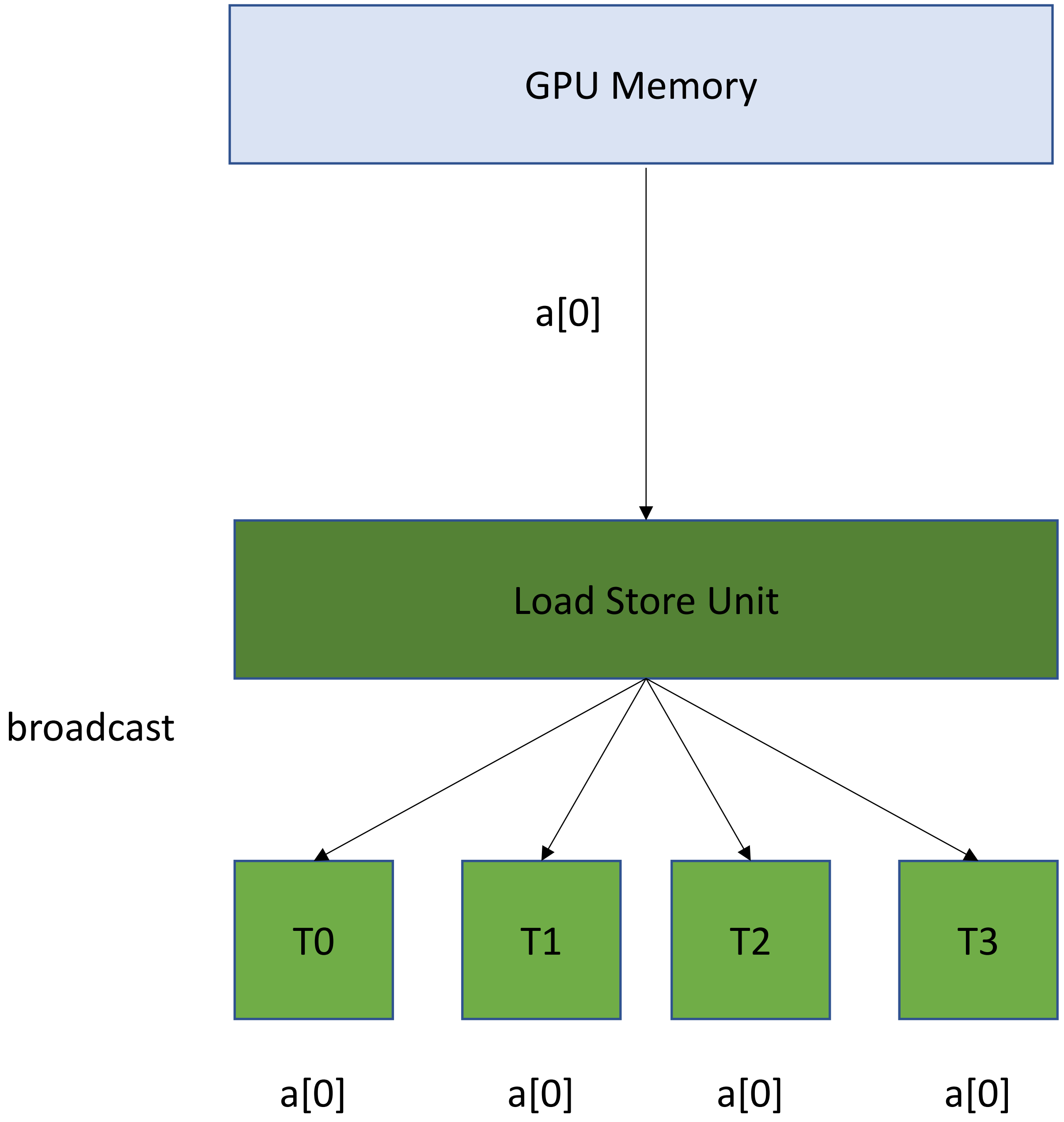


4 cores are accessing memory. What can happen

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

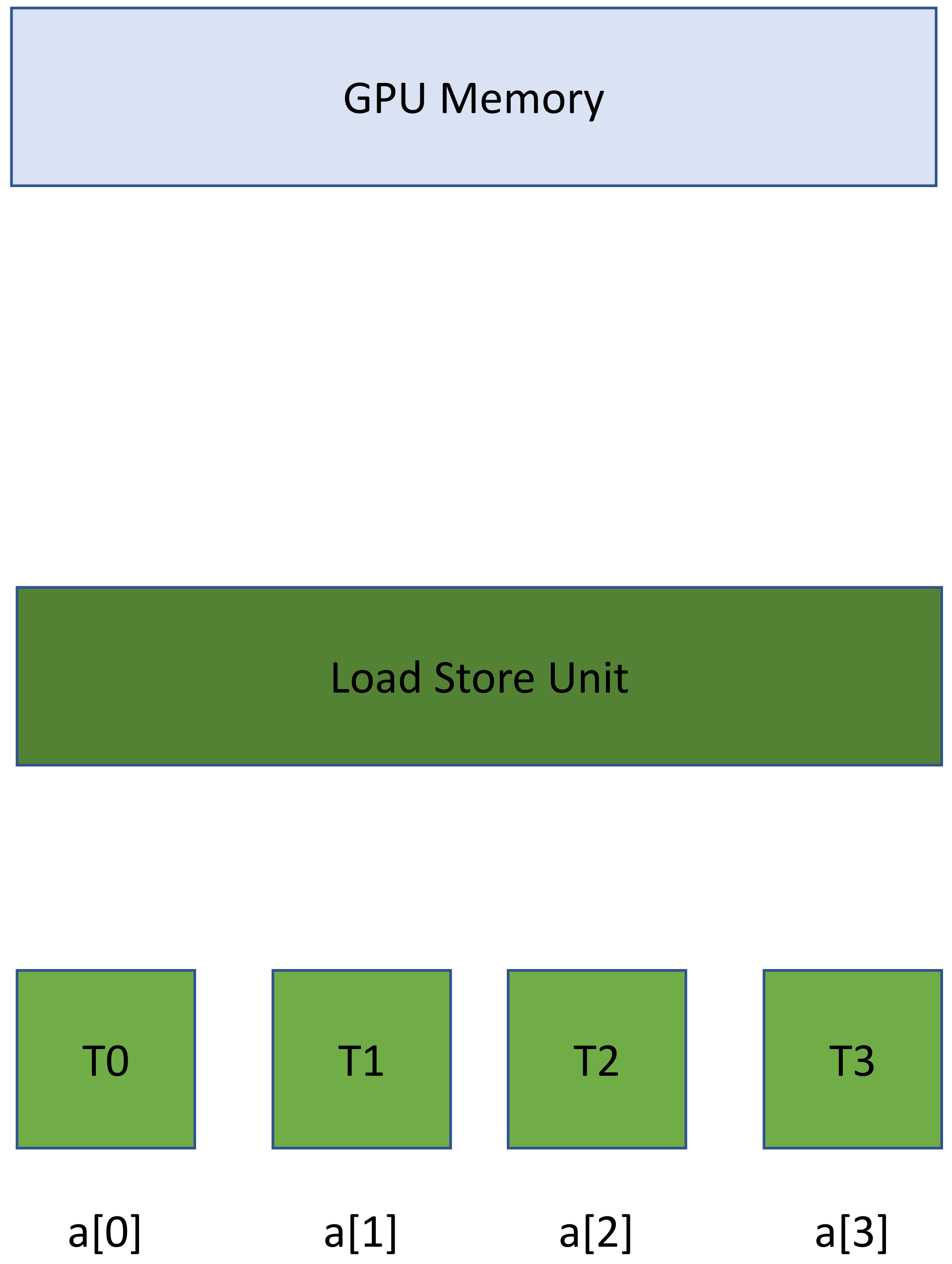
1 request to GPU memory

Efficient, but probably not too common.



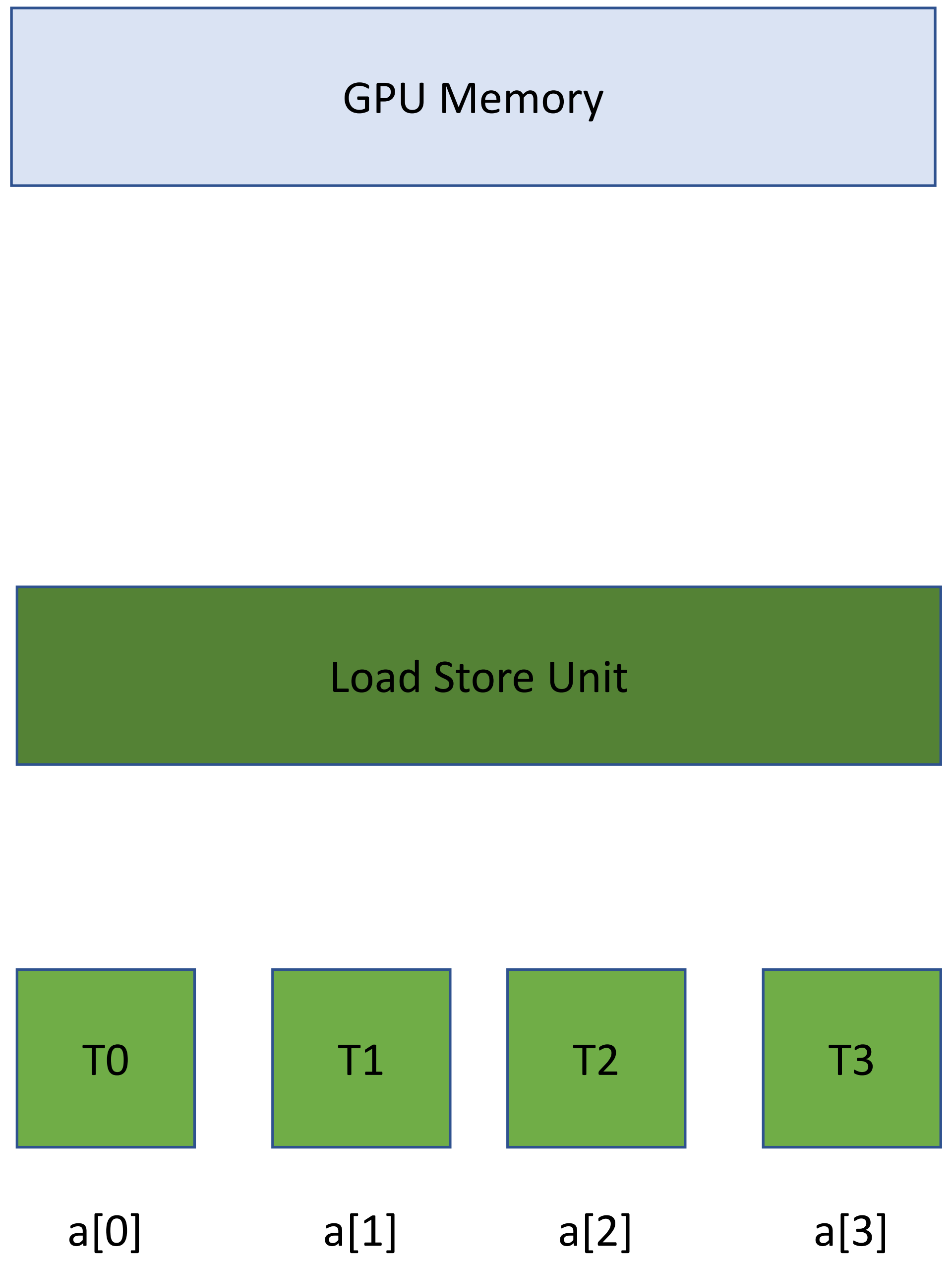
4 cores are accessing memory. What can happen

Read contiguous values



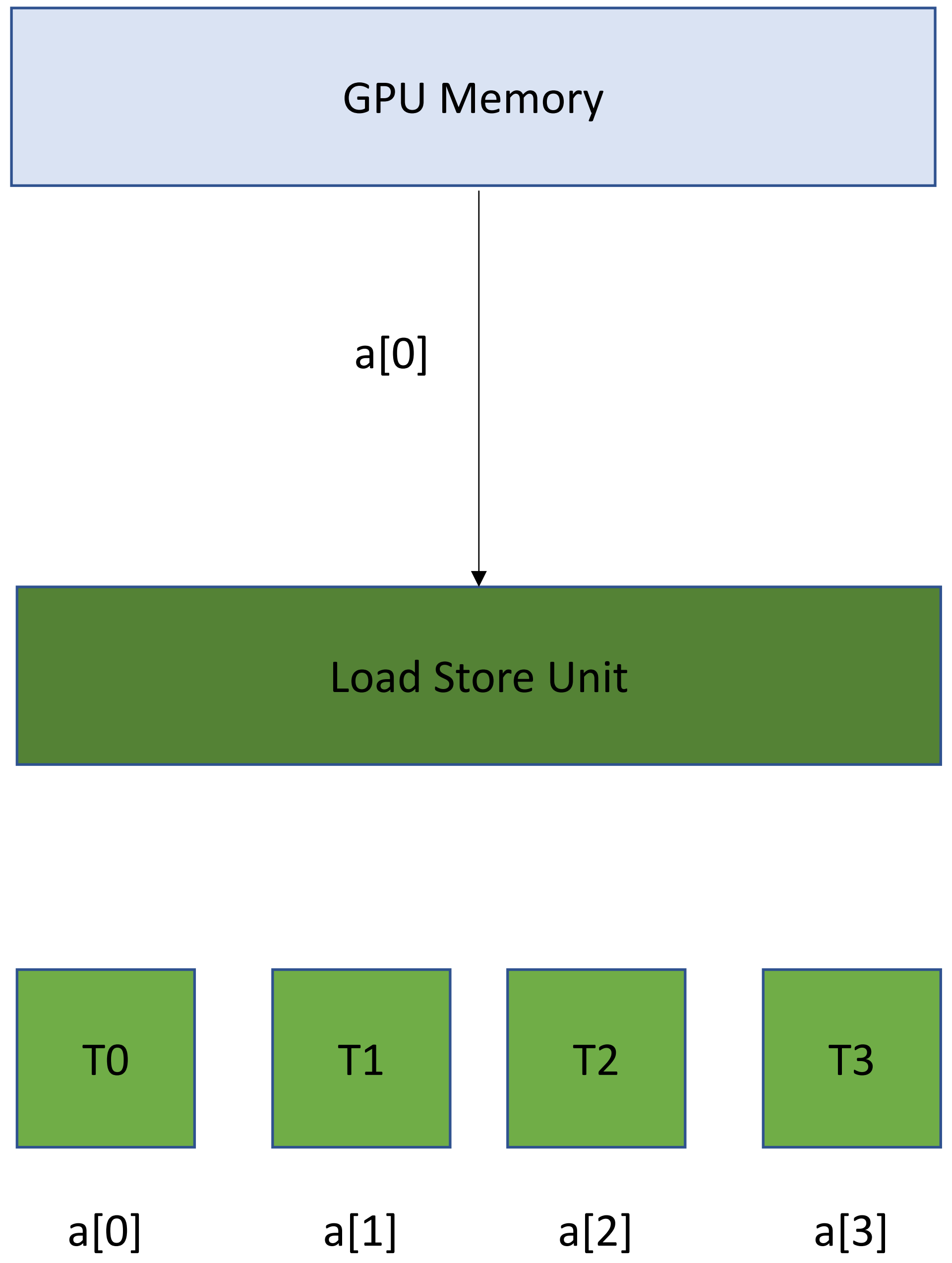
4 cores are accessing memory. What can happen

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



4 cores are accessing memory. What can happen

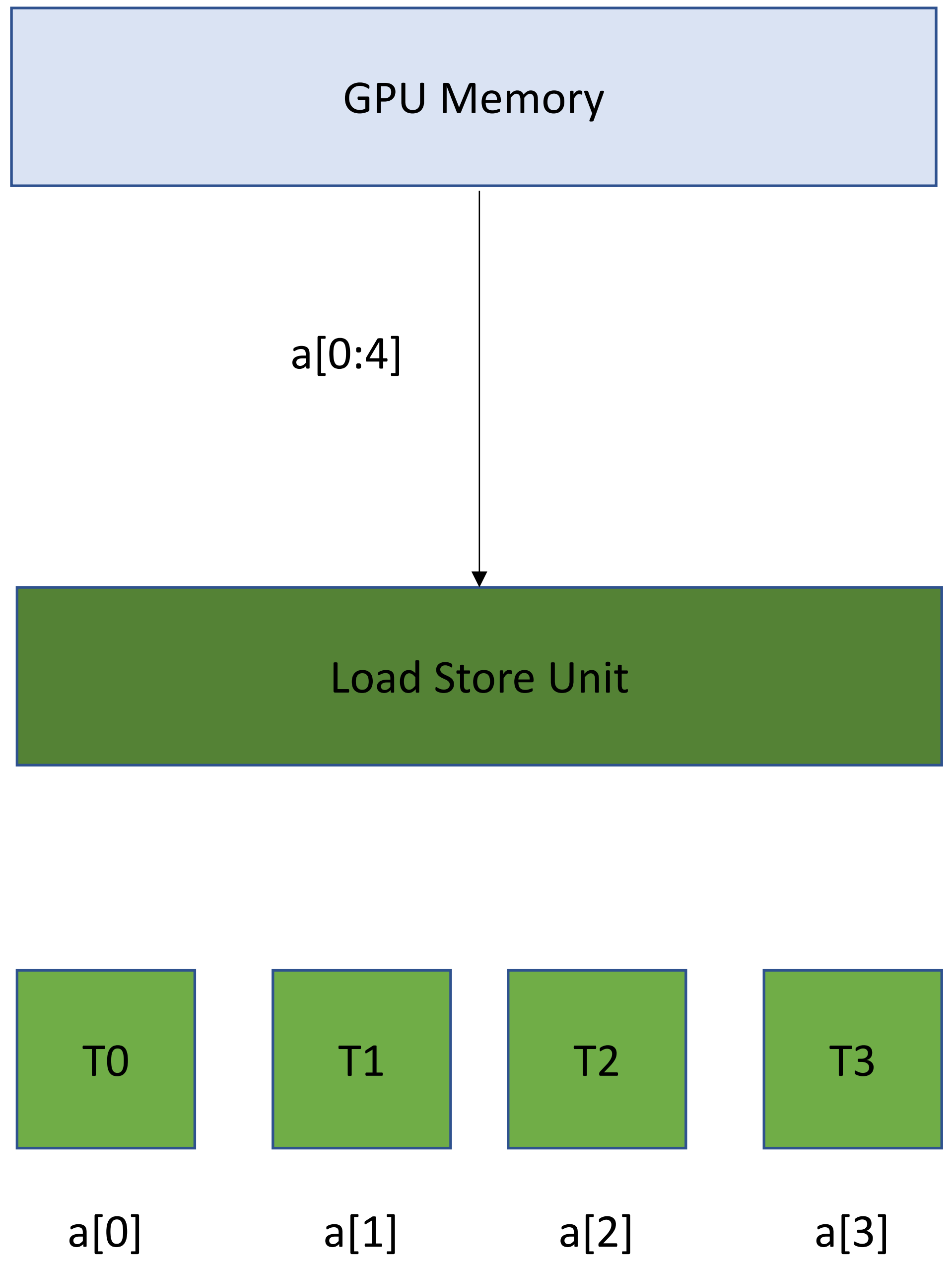
Read contiguous values
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes



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

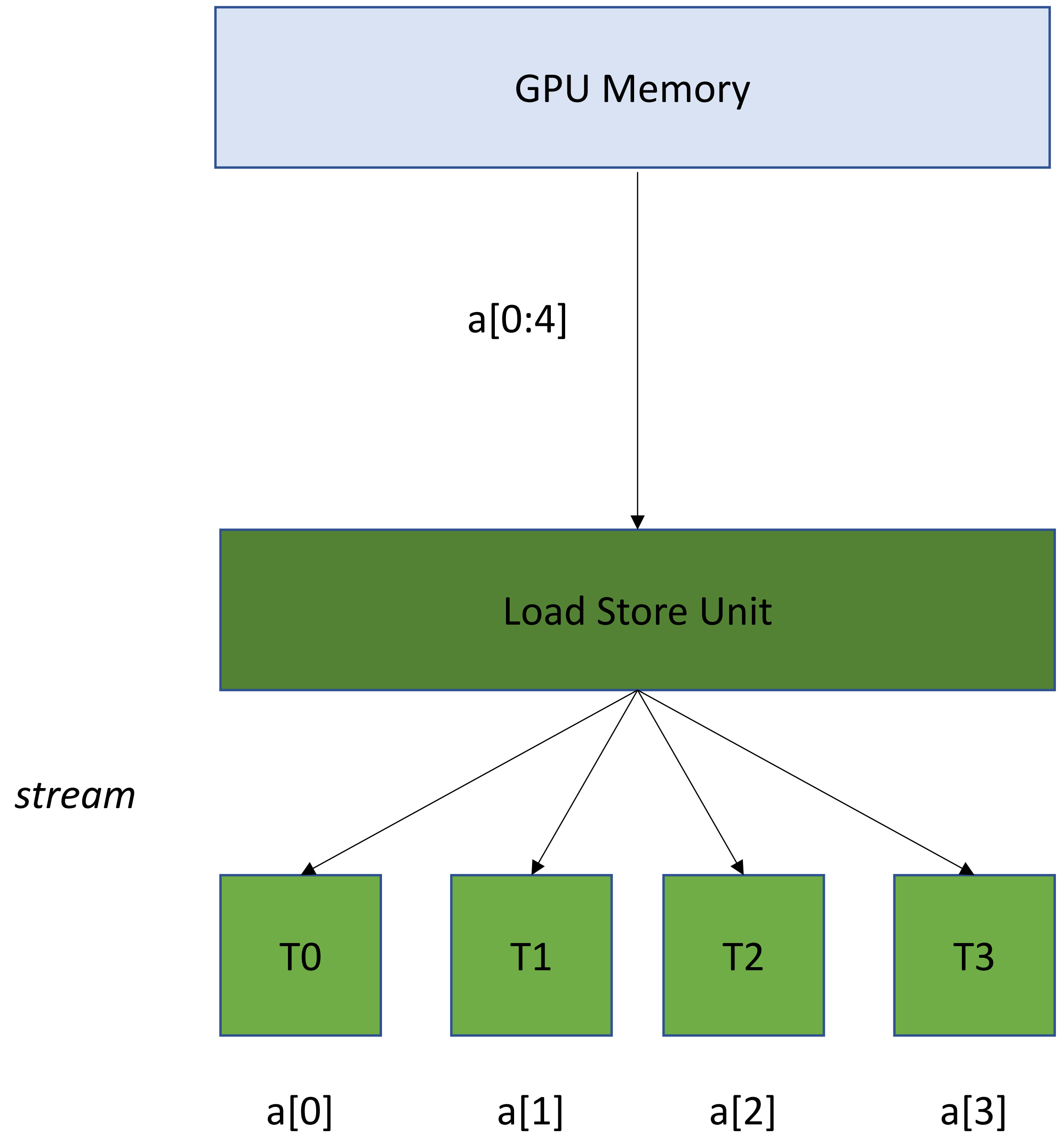


4 cores are accessing memory. What can happen

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

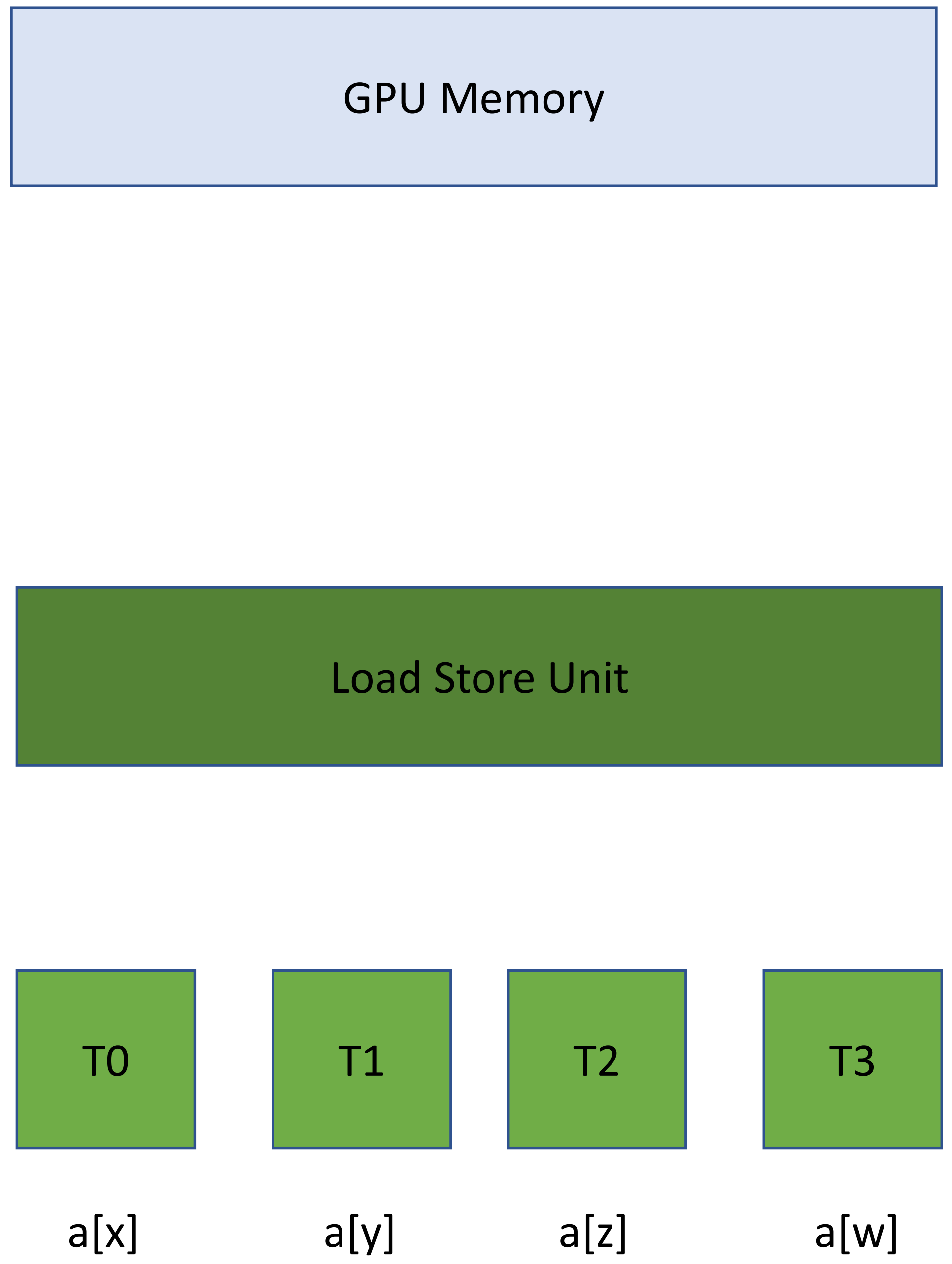


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

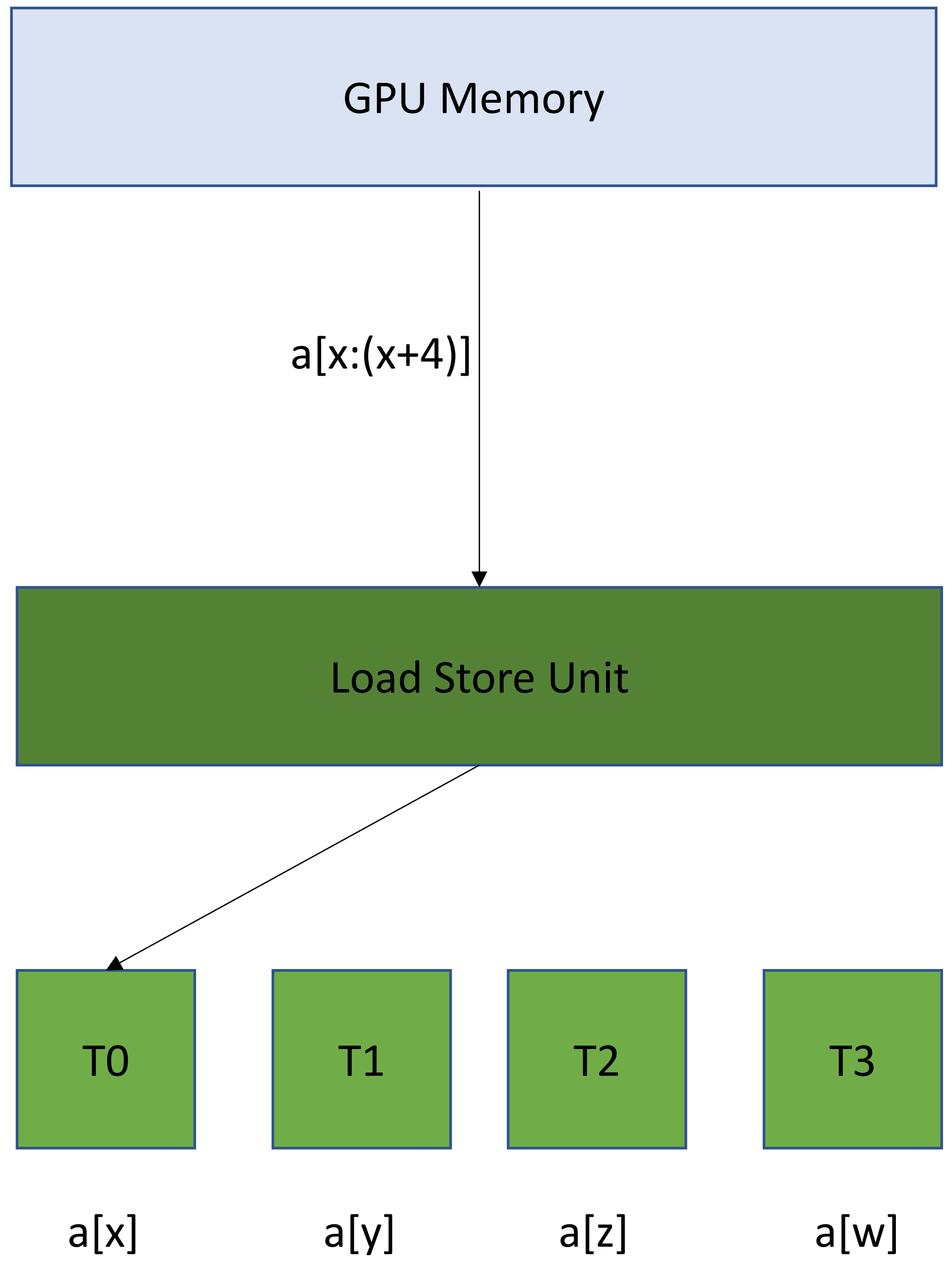


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

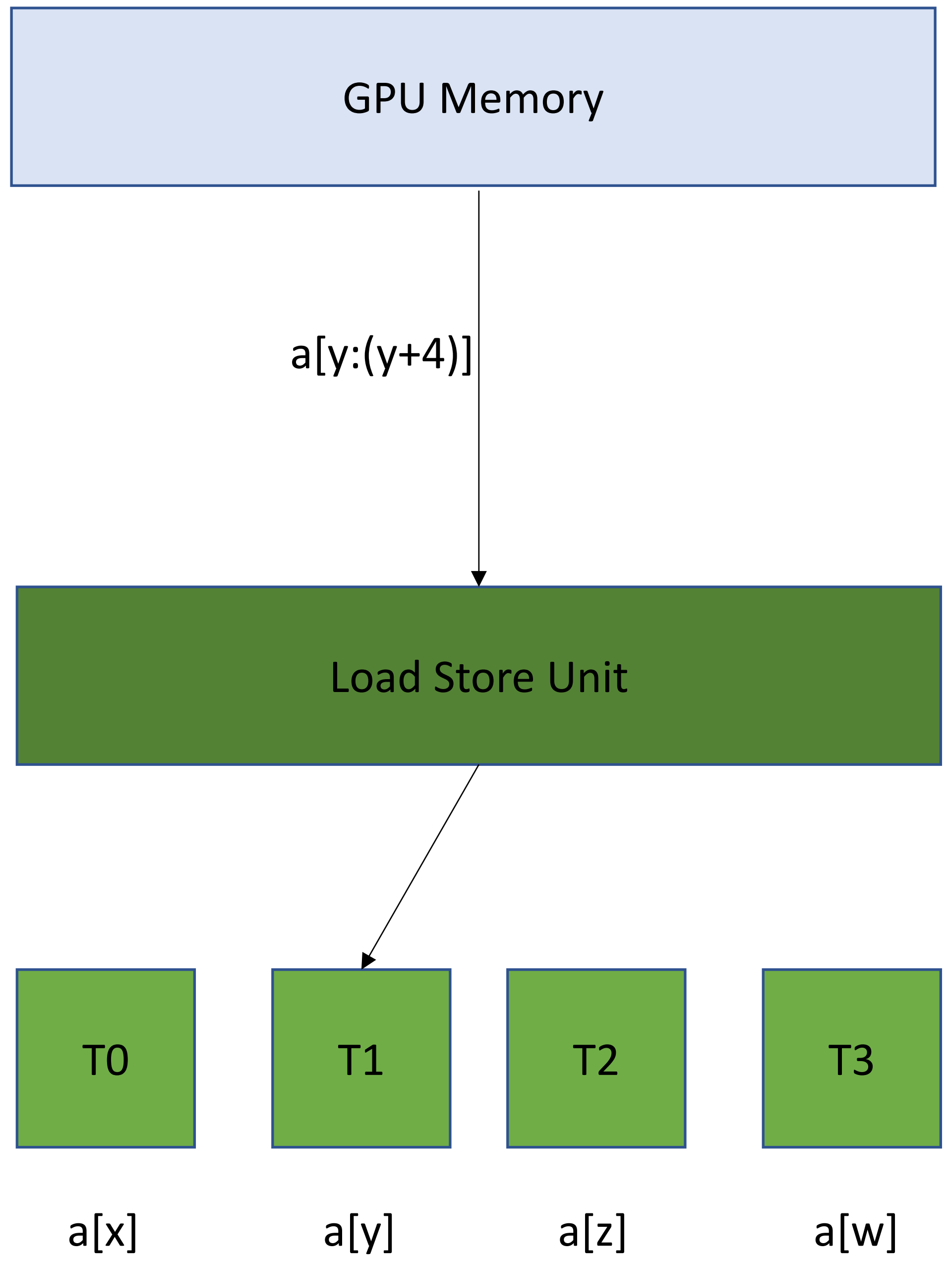


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

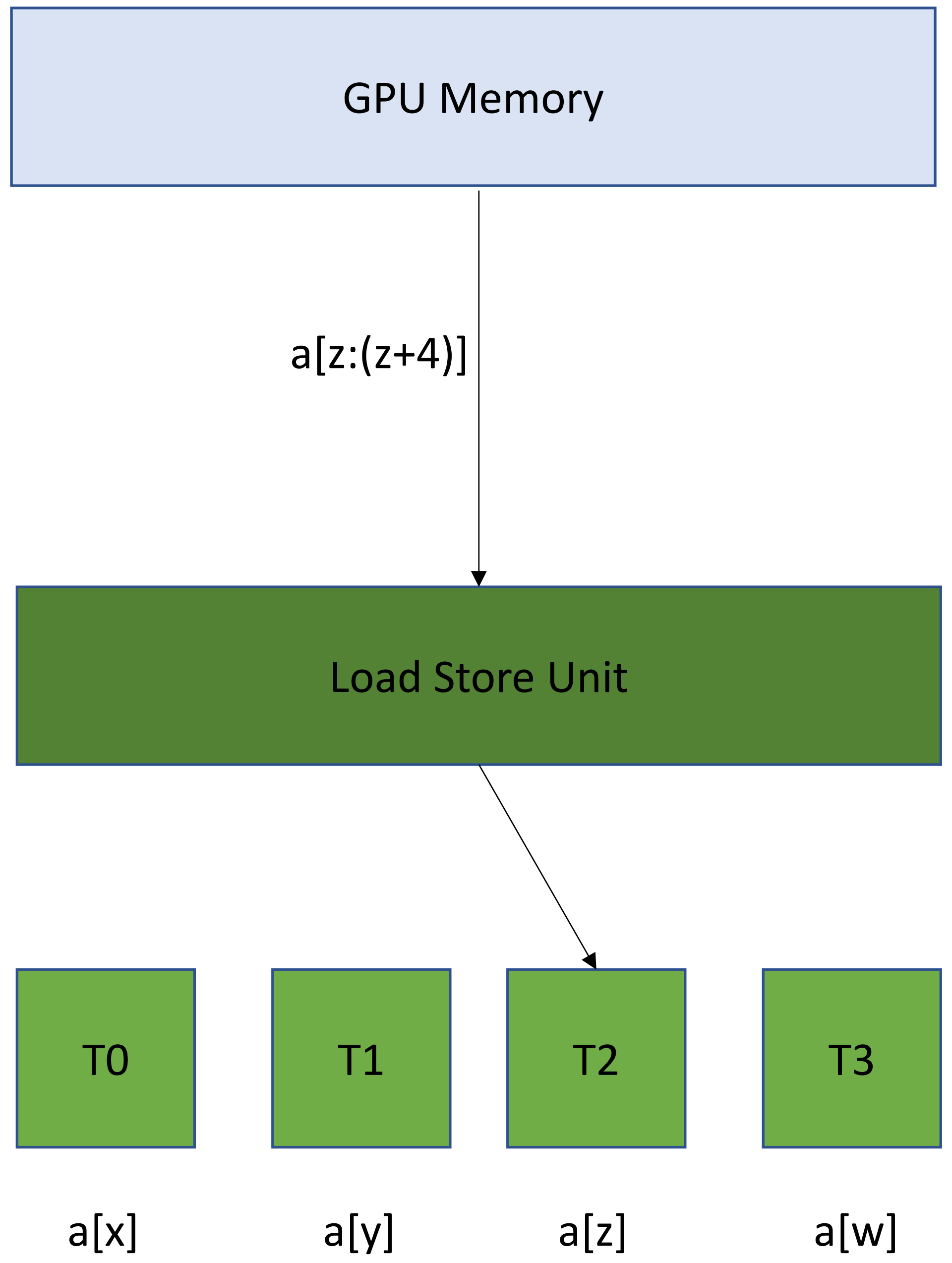


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

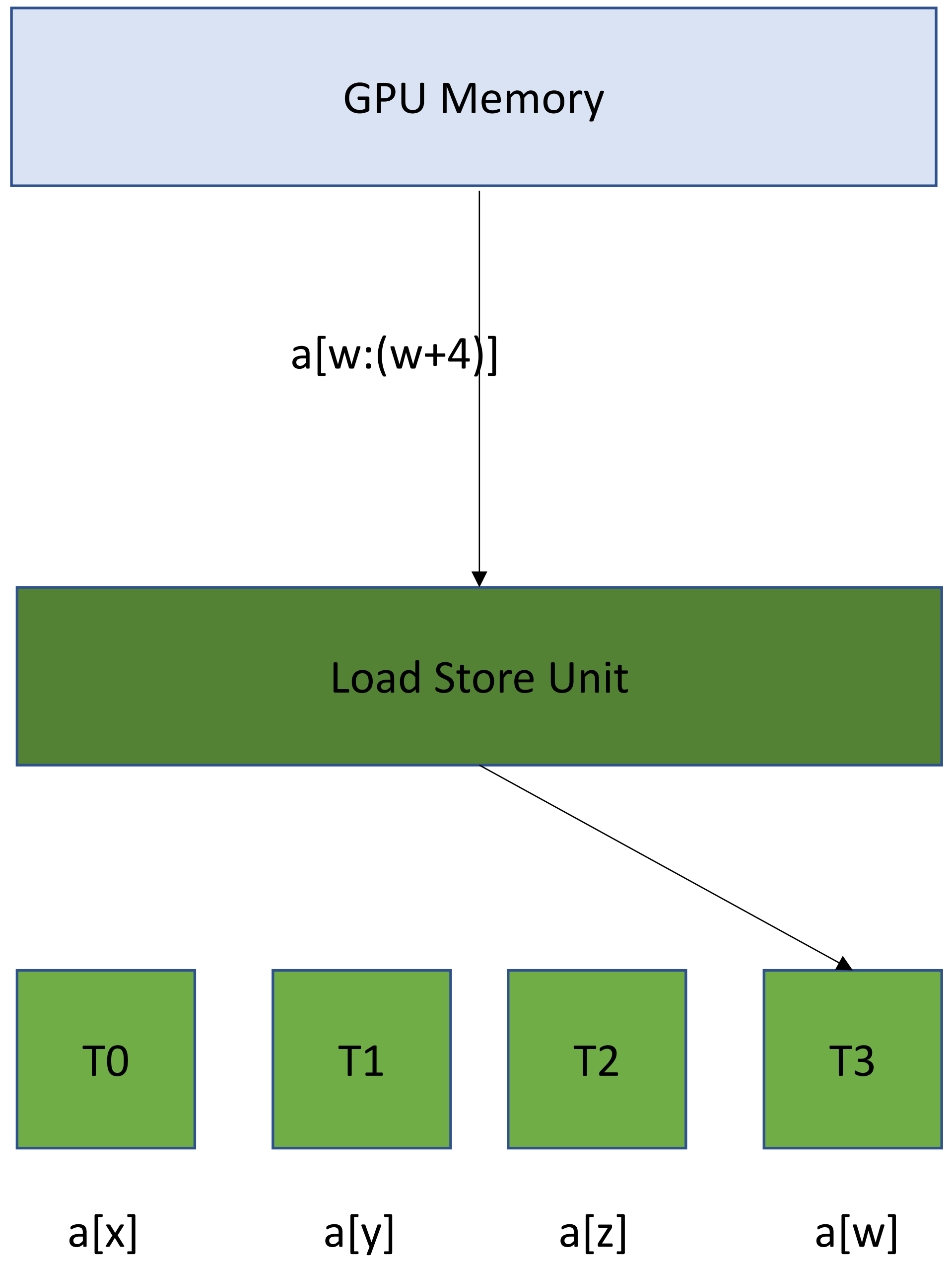


4 cores are accessing memory. What can happen

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];
   }
}
```

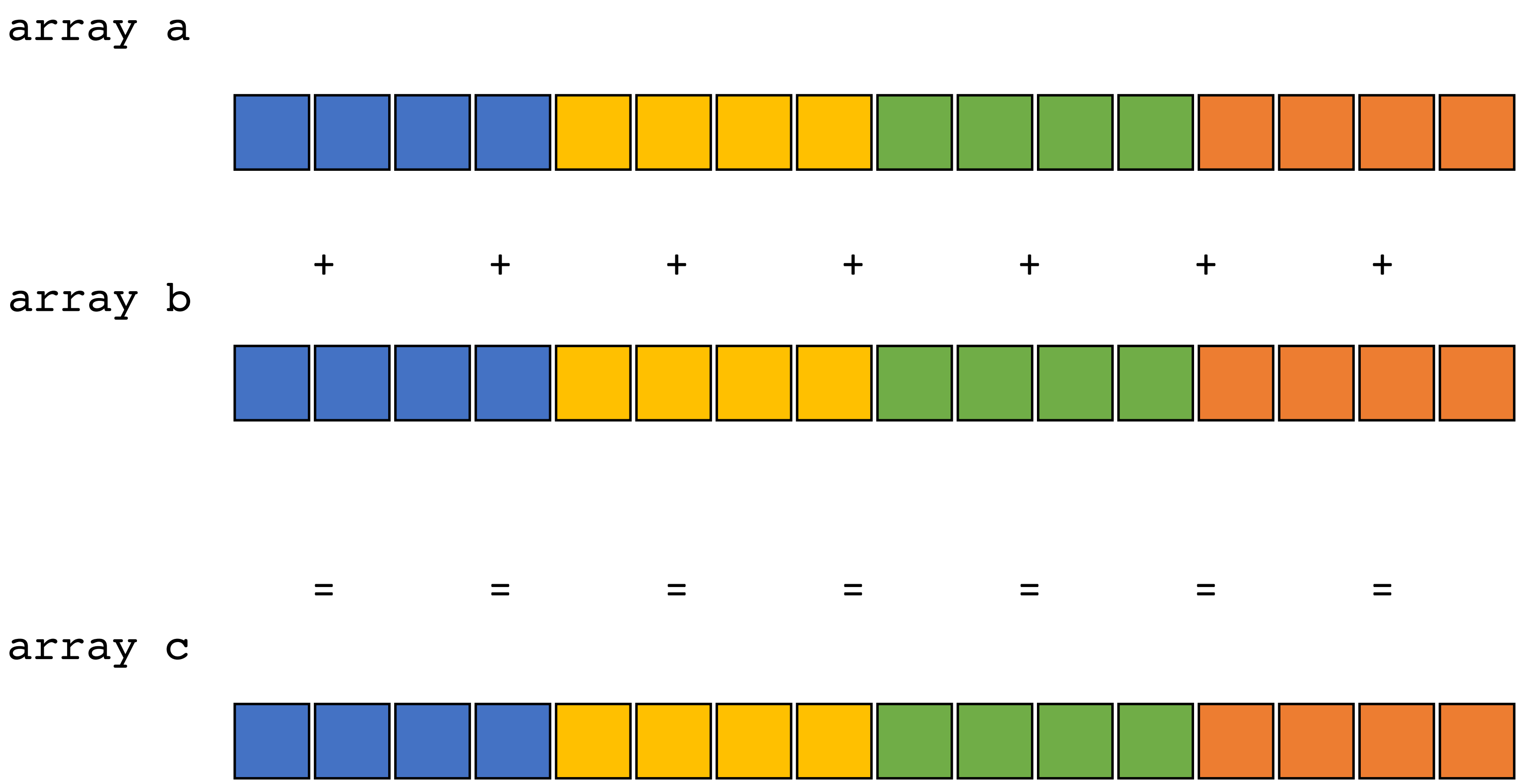
calling the function

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

Chunked Pattern

Computation can easily be divided into threads

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

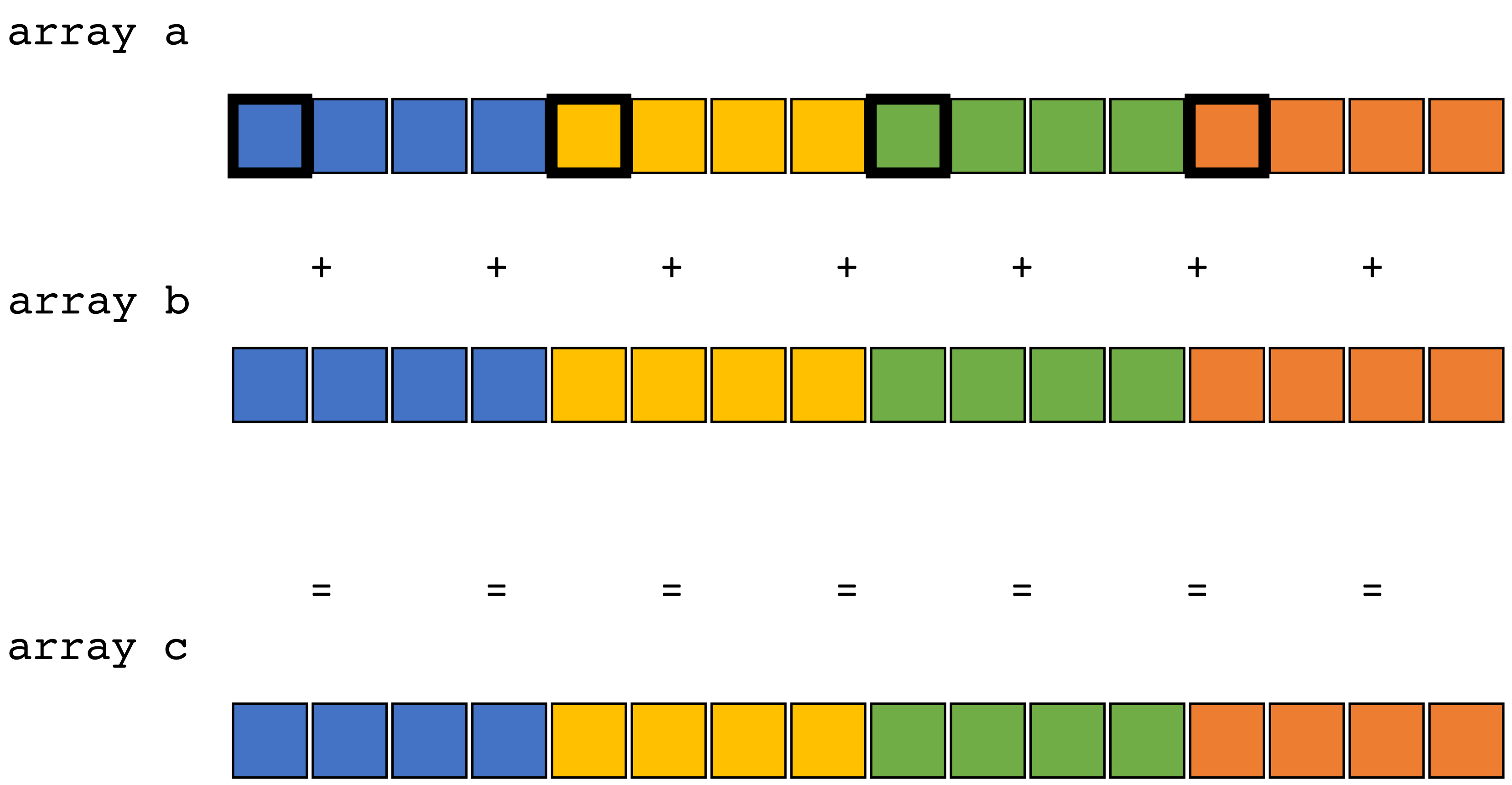


Chunked Pattern

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

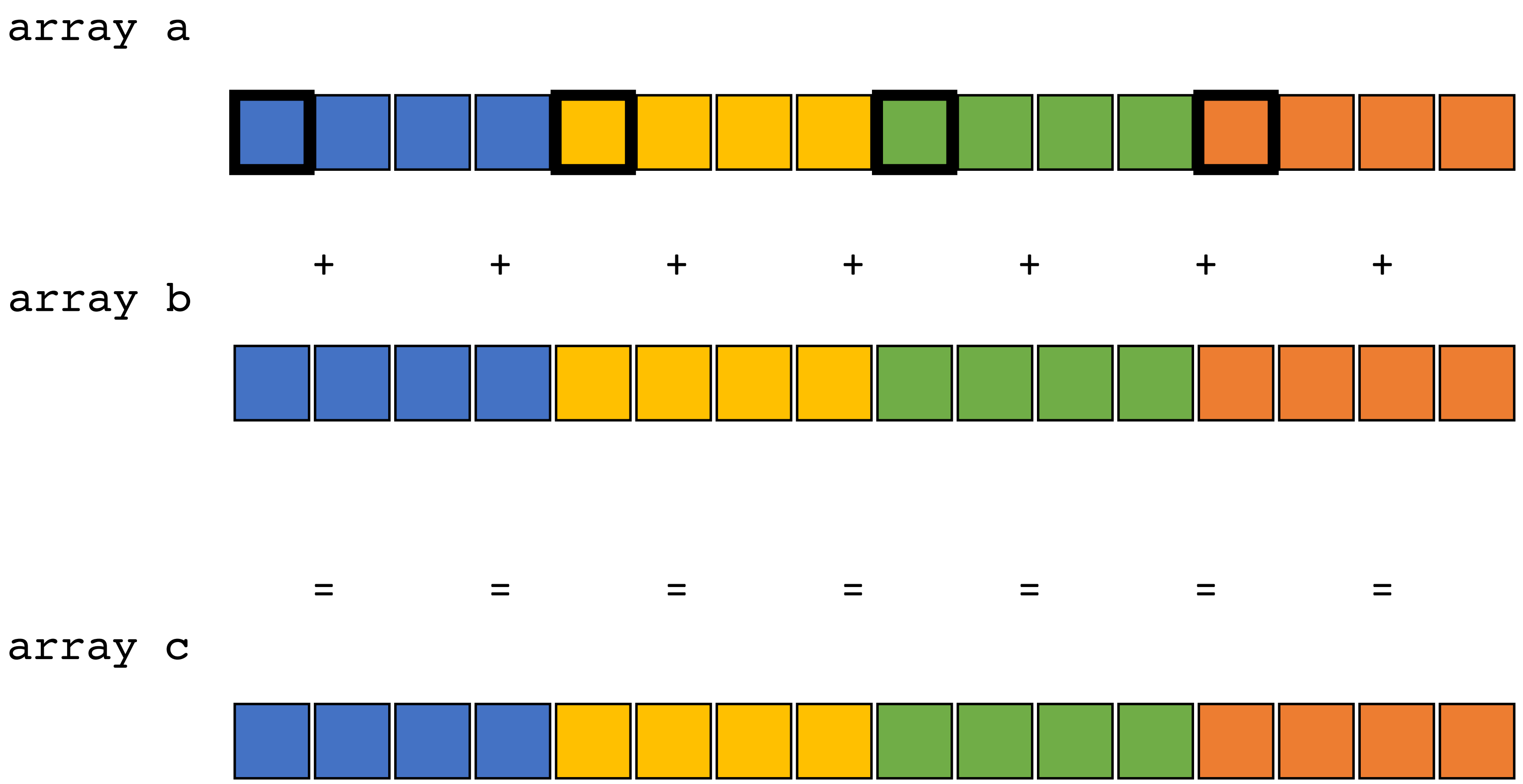


Chunked Pattern

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

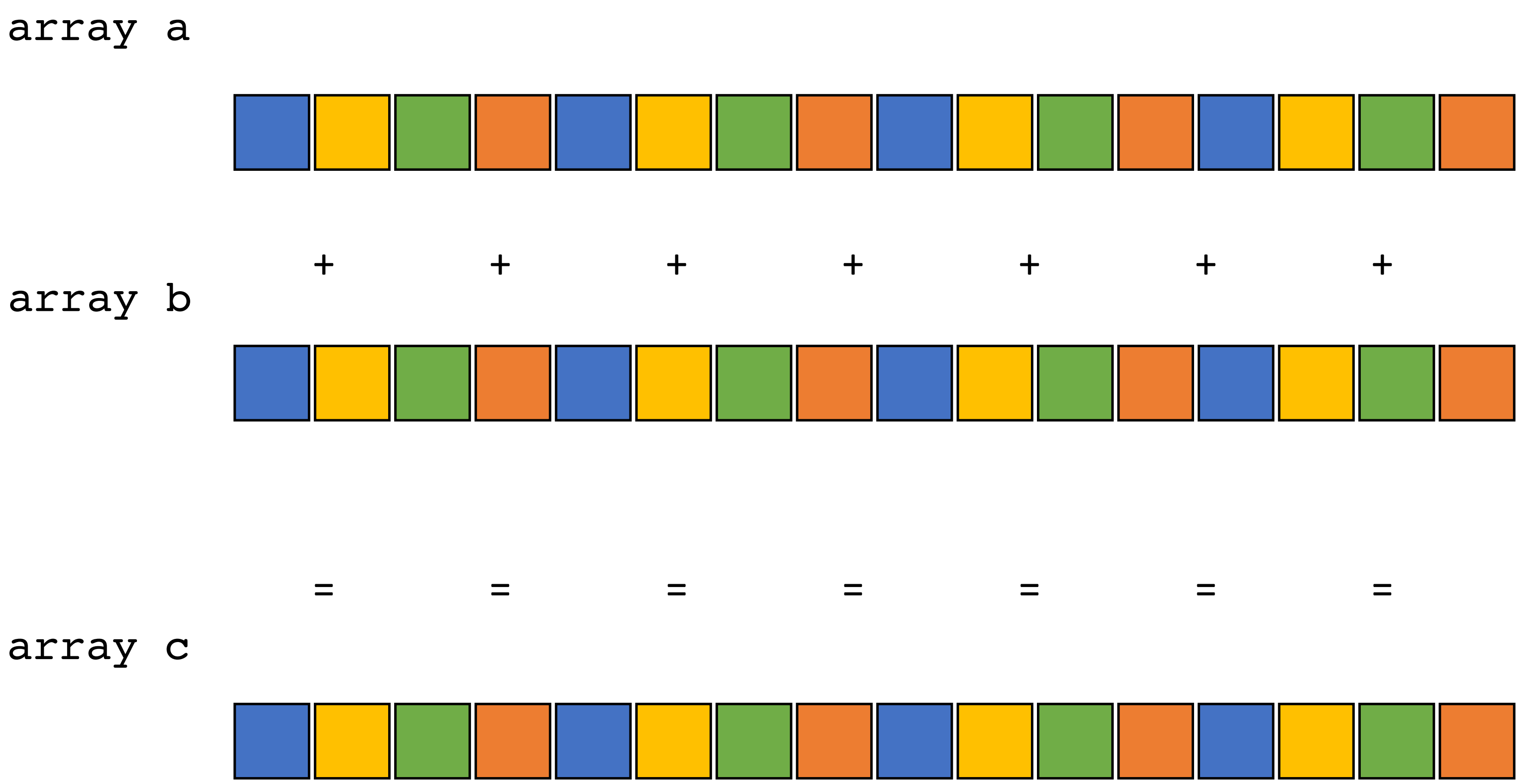
Computation can easily be divided into threads

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

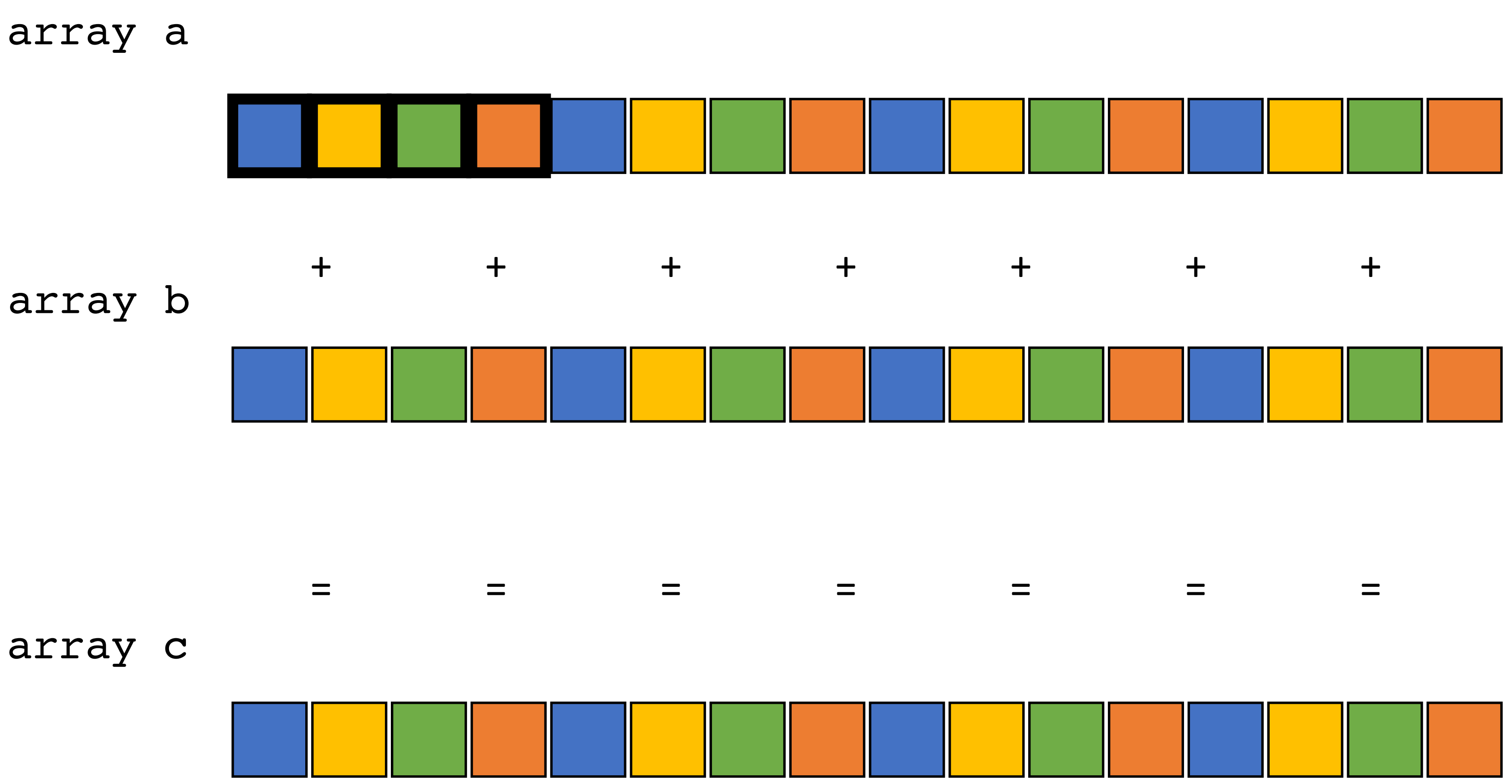
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

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?

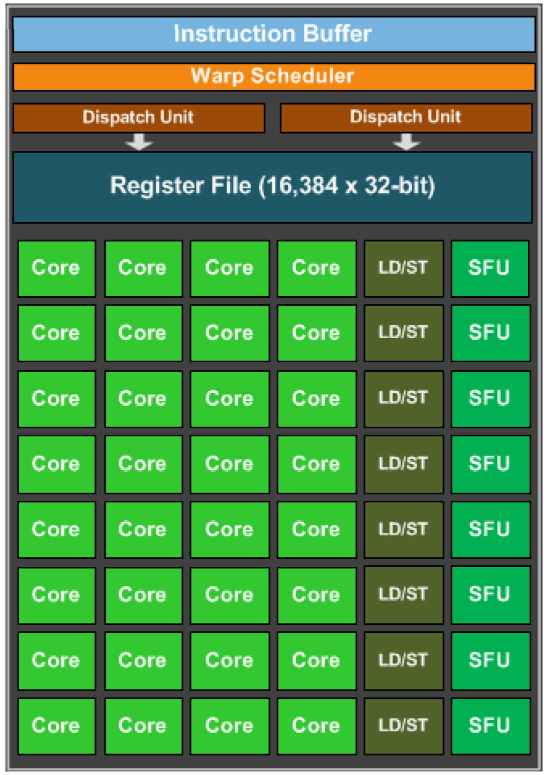
Coalesced memory accesses

Lets try it! What do we think? 😀

What else can we do?

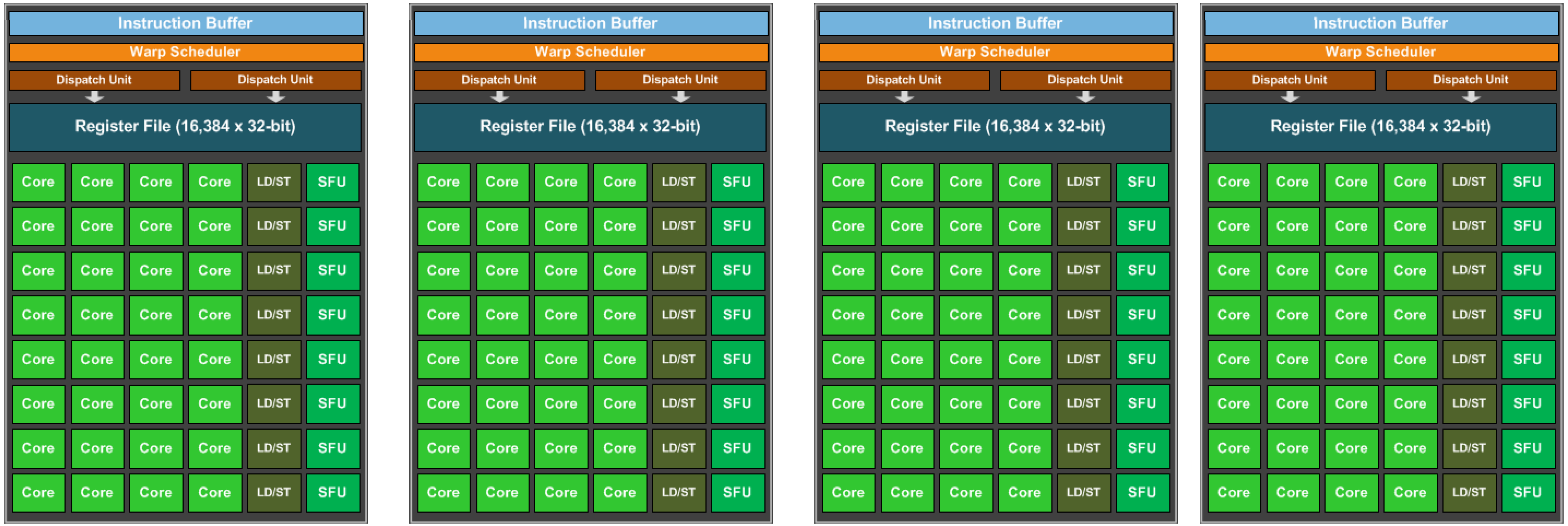
Multiple streaming multiprocessors

We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs big ML GPUs have 32. My GPU has 4



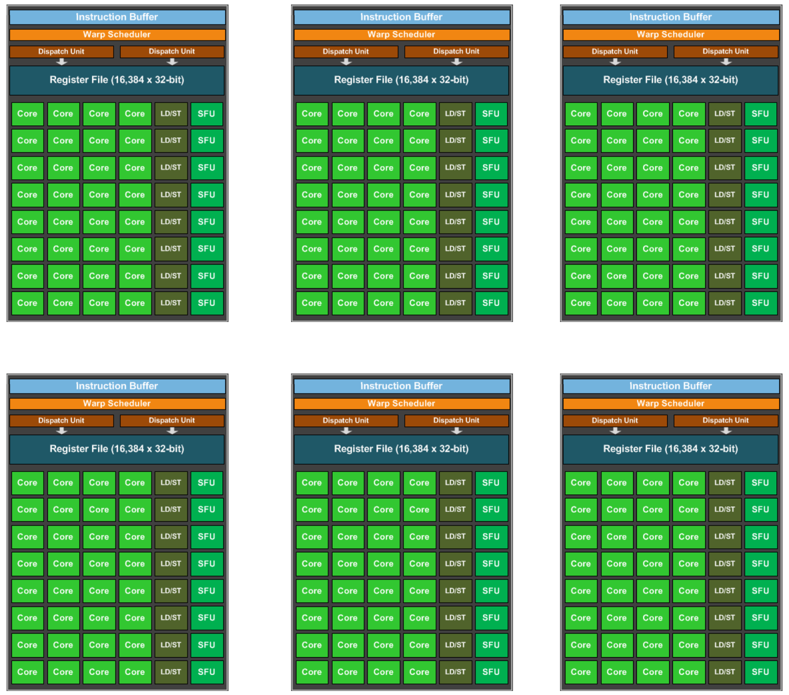
Multiple streaming multiprocessors

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

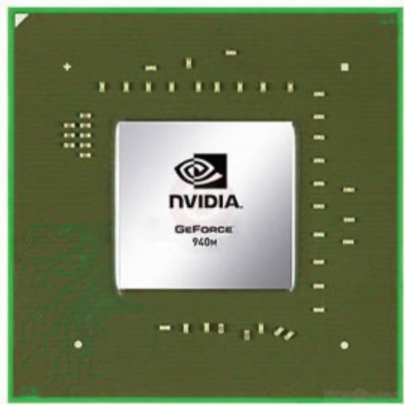
```
vector_add<<<1024,1024>>>(d_a, d_b, d_c, size);
```

Now we have 1 thread for each element

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

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/

See you on Friday

- Turn in HW 4 if you haven’t already
- Working on GPU programming