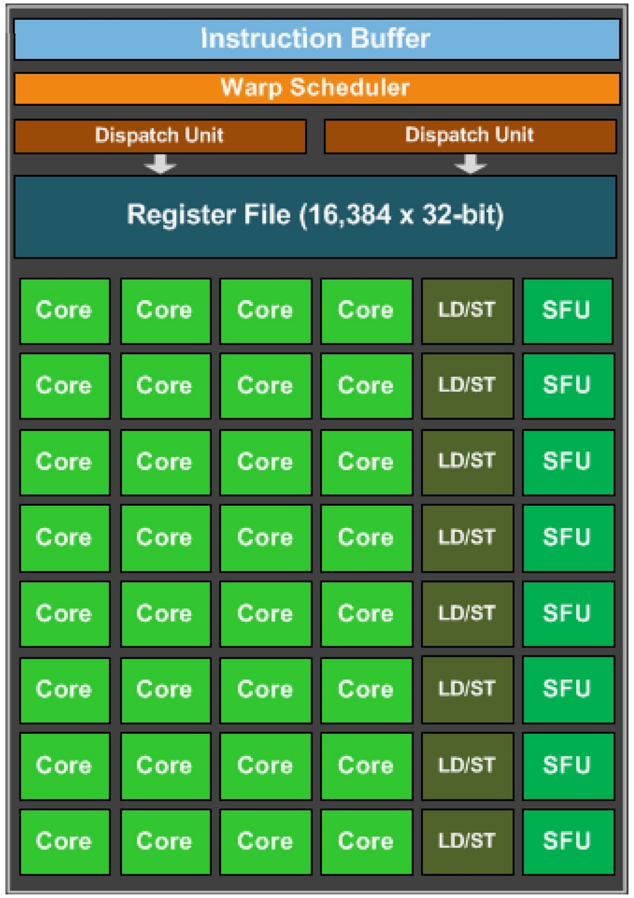
CSE113: Parallel Programming

May 27, 2021

- **Topic**: GPUs 1
  - GPU history
  - Optimizing a GPU program



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

Announcements

- HW2 grades posted
  - Talk to us in 1 week if you have questions/issues
- We plan to have HW3 done in ~1 week.
- HW4 is out
  - Please try not to be late on this one!
  - Due on Monday, June 7
  - There is no guarantee that we will check Piazza on the weekend
  - There is no Tyler office hours immediately before the deadline (we will do a joint office hour this next wedesday).

Announcements

- SETs are out
  - Probably don’t fill them out until after the final so you can have the full view of the class.
  - I will continue to bug you about these

- Final:
  - Wendesday June 9.
  - You have 1 day (Released midnight June 8, due midnight June 9)
  - If you want to budget time: 4pm - 7pm is our allotted time
  - Plan on duration similar to midterm
  - We will be monitoring private piazza posts and emails for clarification questions
  - Late finals will not be accepted!

Announcements

- The rest of the quarter:
  - 2 lectures about GPUs
  - 1 lecture about distributed computing

- If you are interested in GPU programming:
  - CUDA by example is a great book!
  - Linked to in the course material
  - IF you are interested and IF you do not have an Nvidia GPU, message the teach mailing list and we can try to find (limited) resources on campus

Quiz

Quiz

- Go over answer

Schedule

- GPU History
- Optimizing a GPU program

Schedule

- **GPU History**
- Optimizing a GPU program

GPUs: a brief history

- Hard to track everything down
  - First chapter of CUDA by Example
  - https://www.techspot.com/article/650-history-of-the-gpu/
- Please send me any other references you might find!

The very beginning

- Specialized hardware to accelerate graphics rendering
- One of the first real-time computers: Whirlwind 1 at MIT (1951)
  - Flight simulator for bombers
  - vector graphics

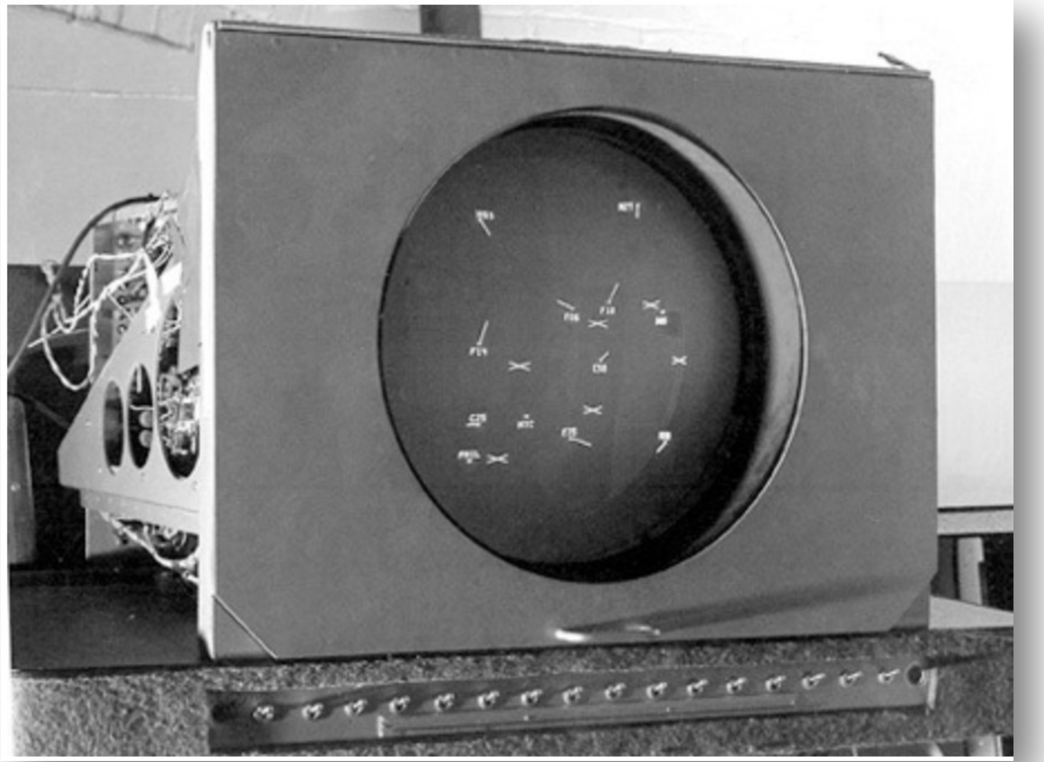


Image from: https://ohiostate.pressbooks.pub/graphicshistory/chapter/2-1-whirlwind-and-sage/

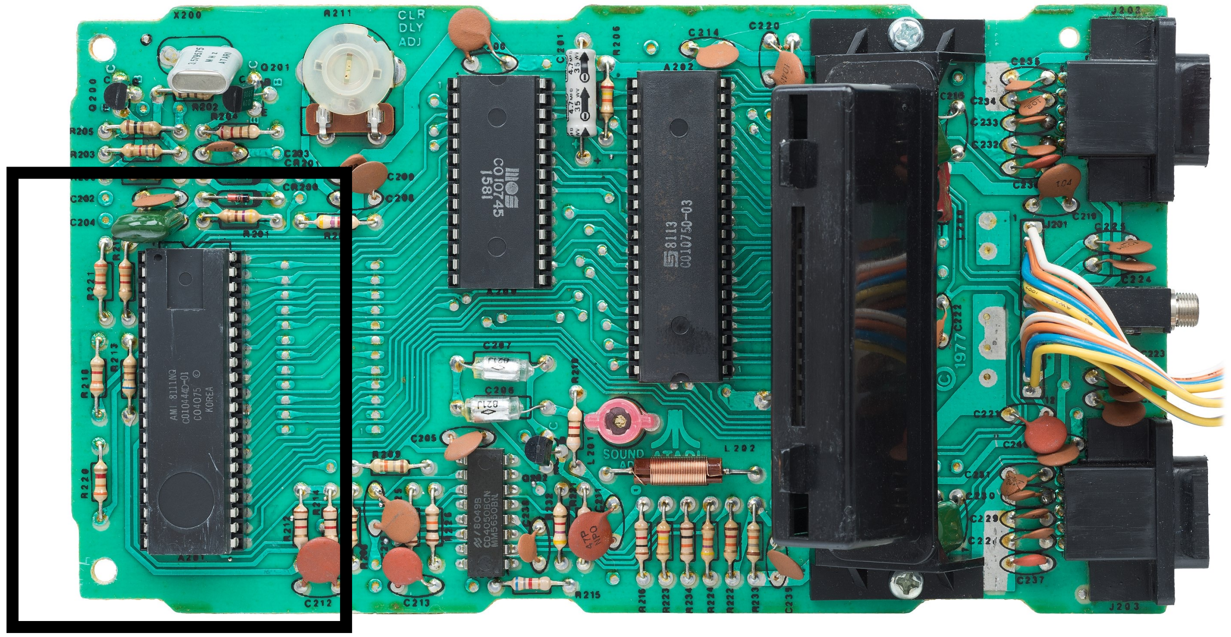
Specialization

- Next 30 years, specialized hardware for specialized software to display 2D graphics

- Specialized
  - Typically ran specific programs
  - portability was not a top priority
  - Even the idea of portable ISAs were not mainstream

Multi-program devices

- 1977: Television Interface Adapter
  - One of the first (and widely produced) portable (i.e. multiple program) GPUs



from: https://en.wikipedia.org/wiki/Television_Interface_Adaptor https://en.wikipedia.org/wiki/DirectX

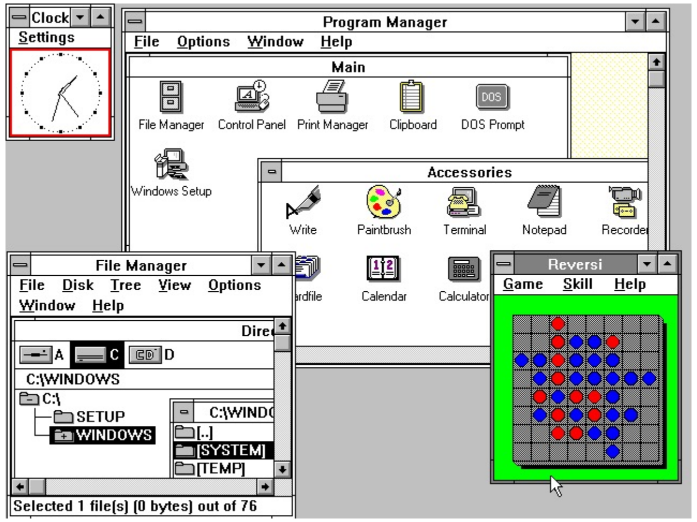
https://en.wikipedia.org/wiki/Microsoft_Windows

https://en.wikipedia.org/wiki/OpenGL

OS integration

- 1990s: Windows: a graphical operating systems, required chips to support 2D graphics.

- New APIs starting appearing, so write GUI programs



Windows 3 (1990)

1992

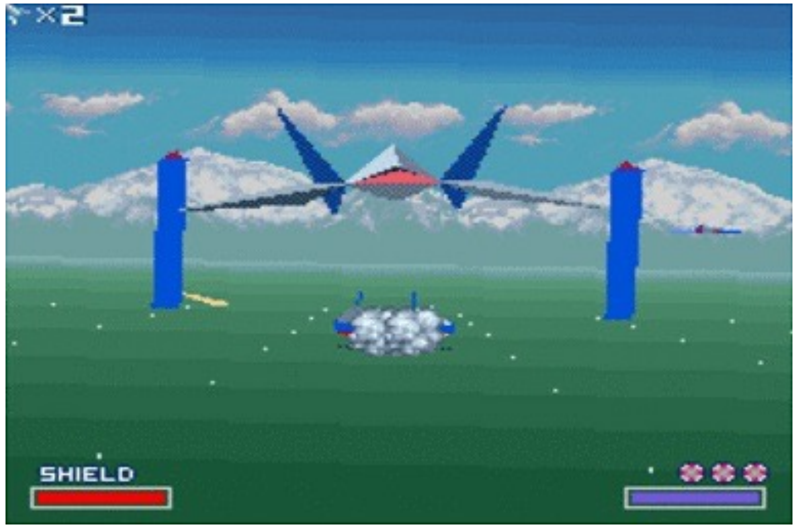
1995

3D graphics in consoles (1993)

- Super Nintendo was not powerful enough to draw 3D graphics
- Shigeru Miyamoto really wanted a 3D flight simulator though
- Worked with a British software company to develop...

3D graphics in consoles (1993)

- Super Nintendo was not powerful enough to draw 3D graphics
- Shigeru Miyamoto really wanted a 3D flight simulator though
- Worked with a British software company to develop...



STAR FOX
Nintendo
Revolutionary Super FX Micro Chip Creates Special Effects Like Never Before!
SUPER NINTENDO
ENTERTAINMENT SYSTEM
SUPER FX

https://en.wikipedia.org/wiki/Star_Fox_(1993_video_game)

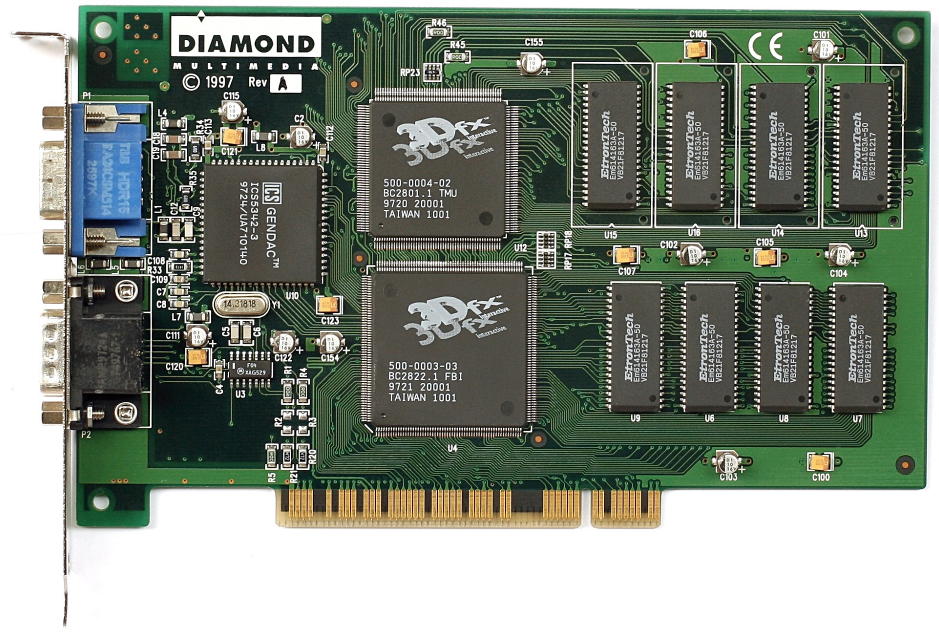
3D graphics in consoles (1993)

- Game cartridges shipped with a “mini GPU” on them:
  - the Super FX

https://twitter.com/gameminesocials/status/1322946537077526528?s=20

3D graphics acceleration

- 1996 : First 3D graphics accelerator: 3Dfx Vodoo
  - Discrete GPU
  - Early 3D games: e.g. tomb raider
  - Acquired by Nvidia in 2002



https://en.wikipedia.org/wiki/3dfx_Interactive

3D graphics acceleration

- 3D accelerators continued, many companies competing:
  - Nvidia
  - ATI
  - 3Dfx
  - and more...
- Next milestone in 1999:
  - Nvidia coins the term “GPU”
  - Compare with modern website

https://web.archive.org/web/20030814003456/www.nvidia.com/object/gpu.html

Programmable 3D accelerators

- 2001: Microsoft DirectX 8 required programmable vertex and pixel shaders.

- 2001: First GPU to satisfy the requirement was Nvidia GeForce 3
  - we are now on 17
  - Used on the original Xbox

- Programmers started writing general programs for these GPUs:
  - Present your data as a graphical input (e.g. Textures and Triangles)
  - Read the output after a series of “graphics” API calls

GPGPU Programming

- 2006: Nvidia releases CUDA: programming language for their GPUs
  - Supported by 8th generation CUDA devices.
  - Integrated vertex and pixel cores into “shader cores”
  - Support for IEEE floating point

- Soon after...

GPGPU Programming

- 2006: Nvidia releases CUDA: programming language for their GPUs
  - Supported by 8th generation CUDA devices.
  - Integrated vertex and pixel cores into “shader cores”
  - Support for IEEE floating point
- Soon after...
- 2008: The Khronos Group launches OpenCL for cross vendor GPGPU:
  - including AMD, Intel, Qualcomm

Khronos Group

- Started in 2000 by Apple as a standards body for graphics API:
  - A way to unify APIs across many different vendors
  - at the time: ATI, Nvidia, Intel, Sun Microsystems (and a few others)
  - now: Many companies, including AMD, Nvidia, Intel, Qualcomm, ARM, Google

  - OpenGL is maybe the biggest standard they maintain (for graphics)
  - OpenCL is biggest for compute
  - Vulkan is their new standard (will it catch on??)
  - (disclosure: I am an individual contributor ☺)

- Apple deprecated Khronos group standards to support Metal in 2018

https://en.wikipedia.org/wiki/Khronos_Group

Where are we now?

- Nvidia CUDA is widely used, driving many HPC and ML applications
- OpenCL is used to program other GPUs (although it is not as widely used)
- Metal is used for Apple devices
- Vulkan has momentum

- New GPGPU programming languages are on the horizon:
  - WebGPU - a javascript interface to unite Metal, Vulkan and DirectX
  - Its ambitious! Will it work?!
  - Available in canary builds of Chrome

GPU Shortages?

- Cryptocurrency:
    - 2018 reported tripling of GPU prices and shortages due to increase demand from miners.
    - Still happening will lots of market fulgurations.
    - Still plenty of GPUs in your phone, laptop, etc. ☺

Today’s lecture

- Will use CUDA!
    - It is widely used
    - The programming model is straightforward

- In the future I would want to use WebGPU (more available for those who do not have Nvidia GPUs)

Schedule

- GPU History

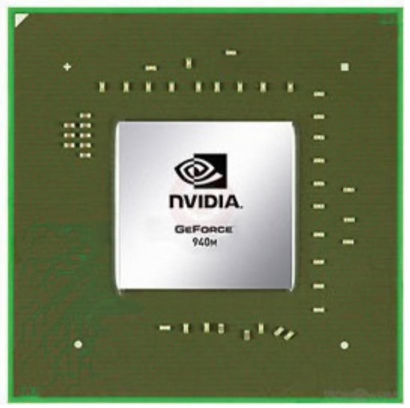
- Optimizing a GPU program

Schedule

- GPU History
- **Optimizing a GPU program**

Programming a GPU

The GPU in
my PhD laptop



Nvidia 940m
1.8 Billion transistors
75 TDP
Est. $130

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

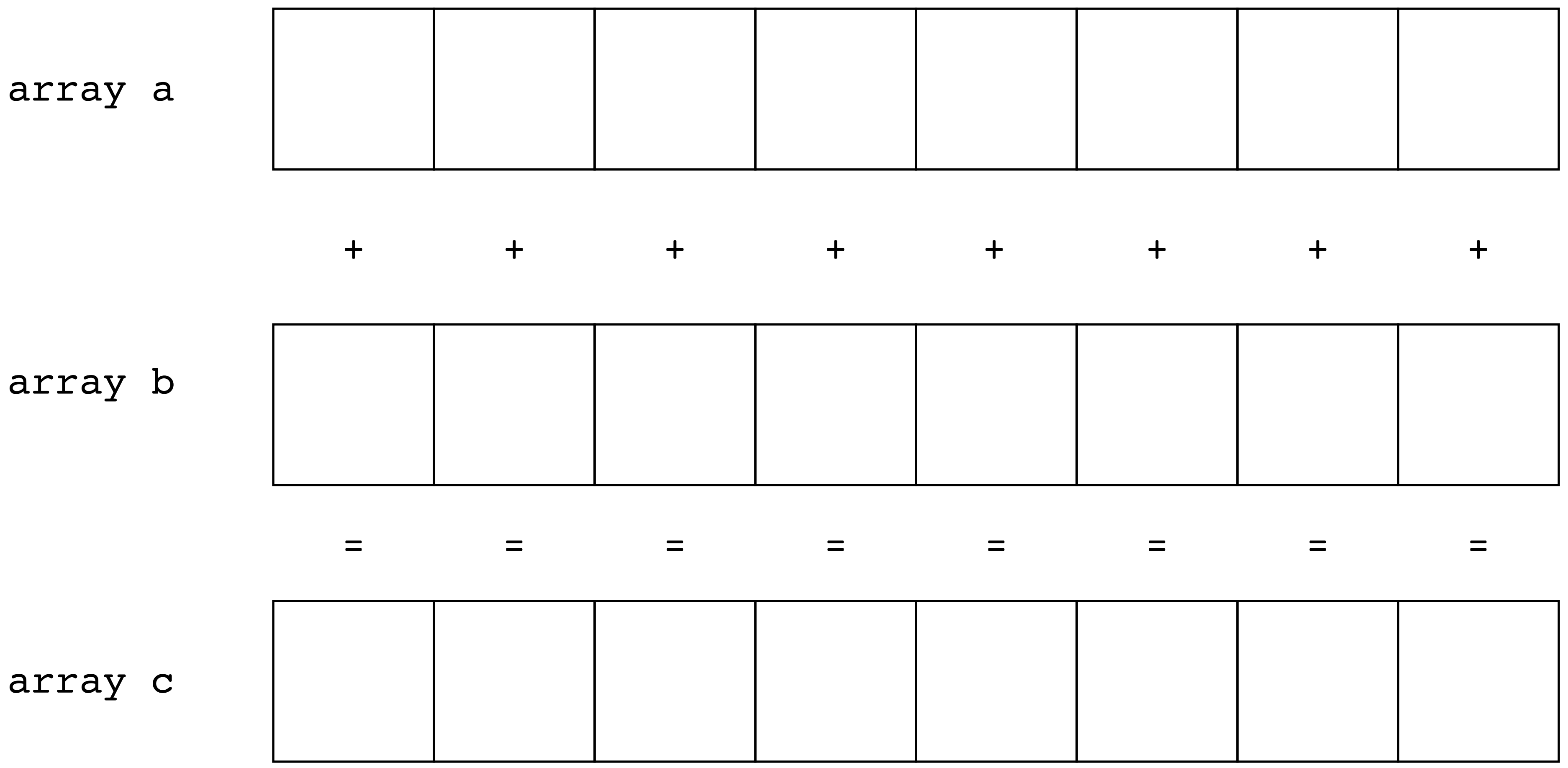
Programming a GPU

- The problem: Vector addition

Embarrassingly parallel

Computation can easily be divided into threads

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



Programming a GPU

- The problem: Vector addition
- Who can do it faster?

Lets set up the CPU

- CPU code

Now for the GPU

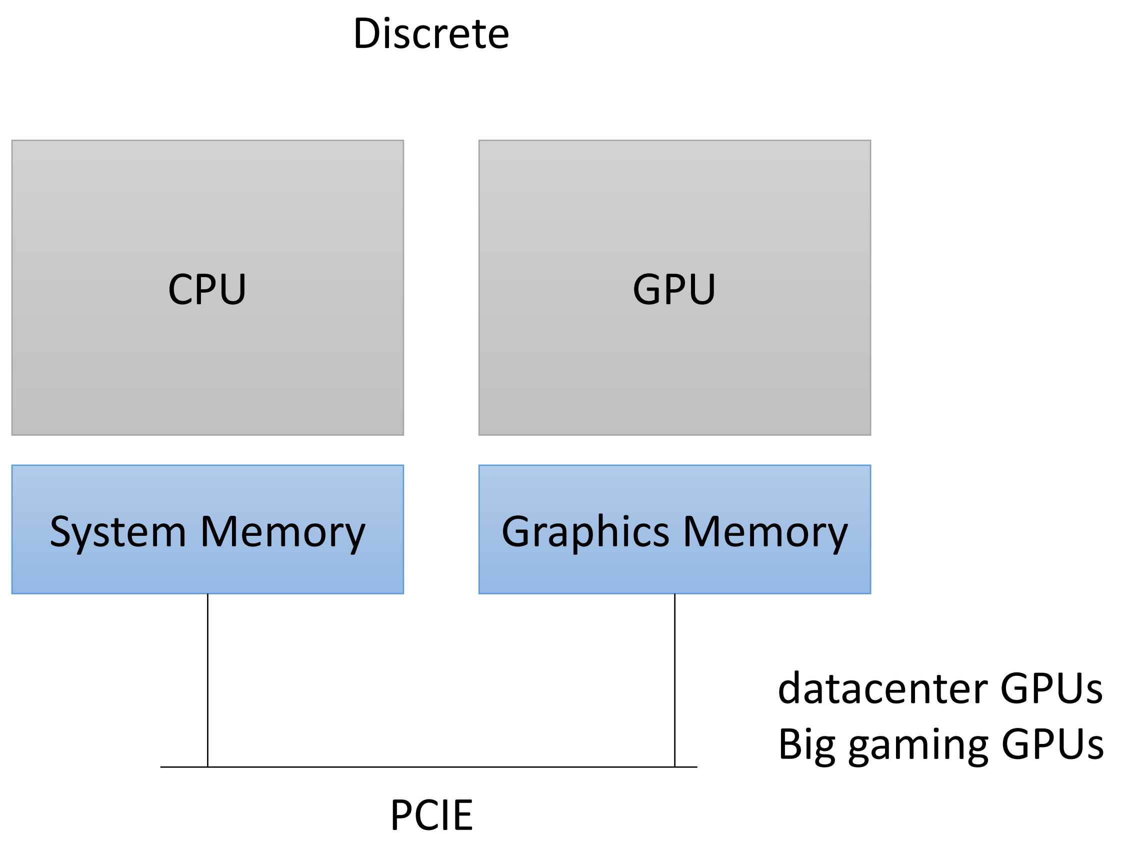
- Its going to take a bit of work....

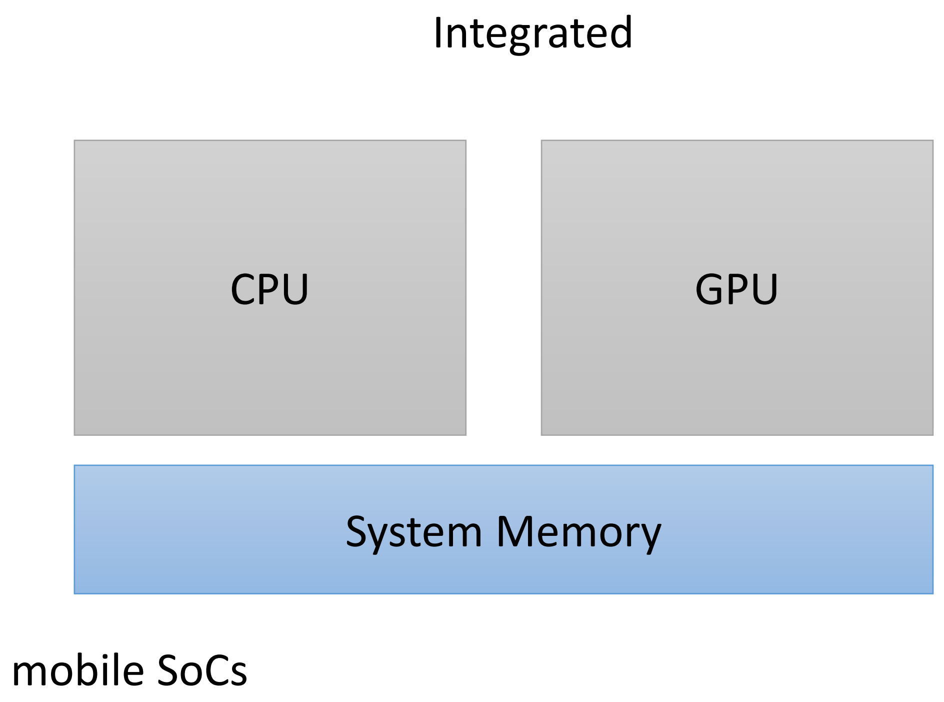
GPU set up

- We need to allocate and initialize memory

GPU set up

- GPUs come in two flavors

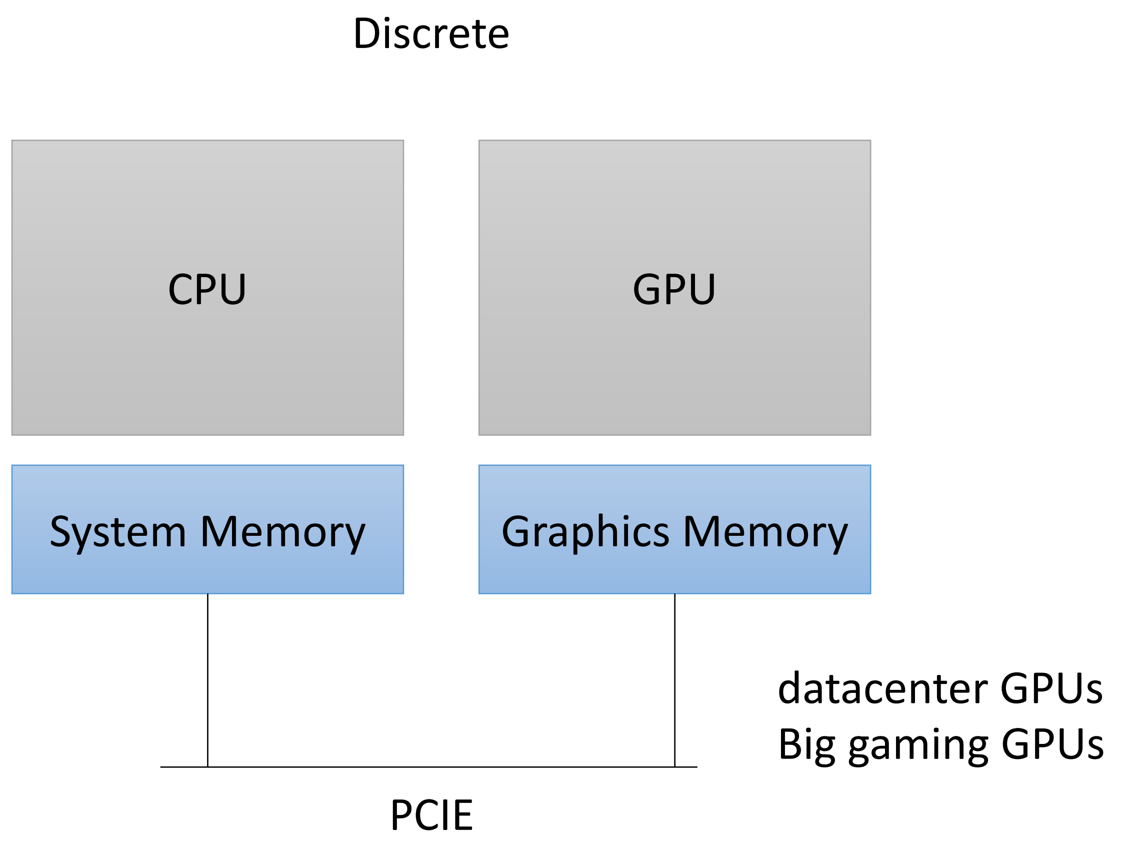




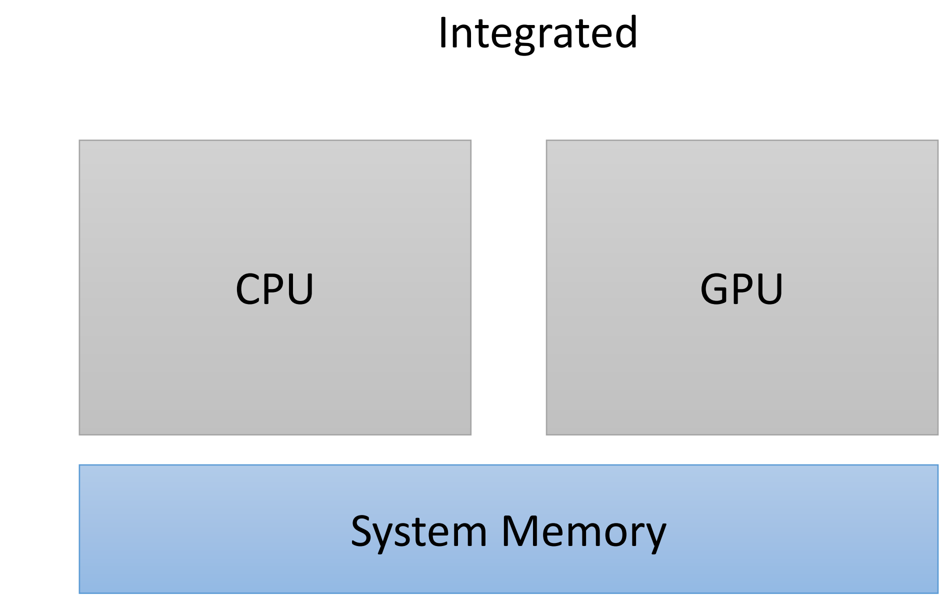
GPU set up

Pros and cons of each?

- GPUs come in two flavors



datacenter GPUs
Big gaming GPUs

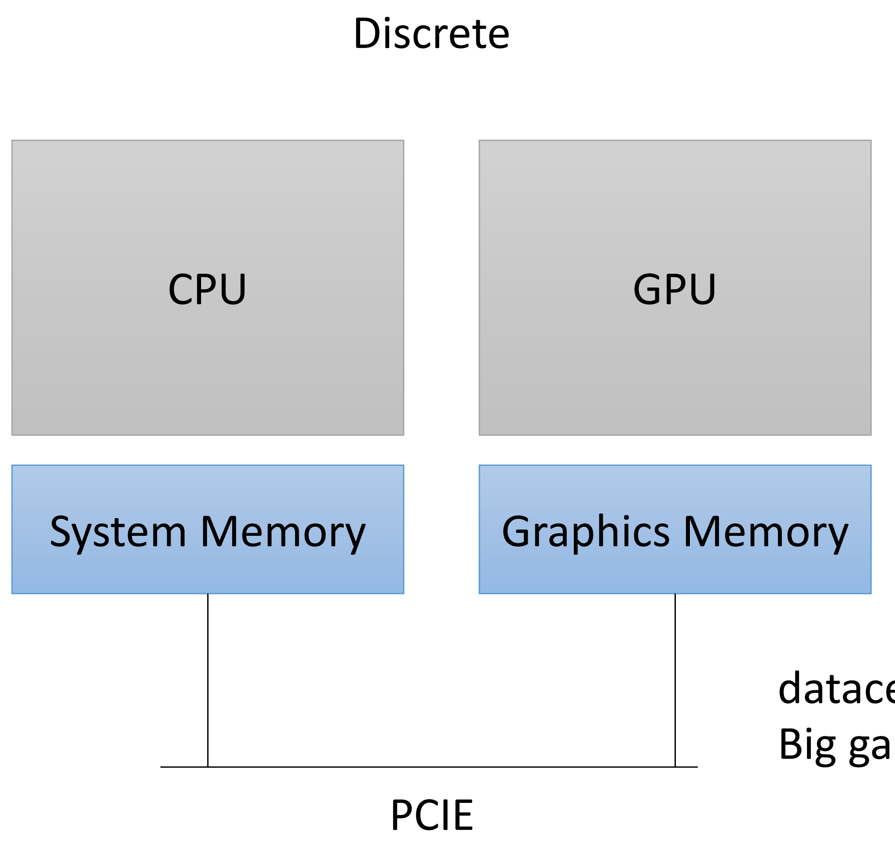


mobile SoCs

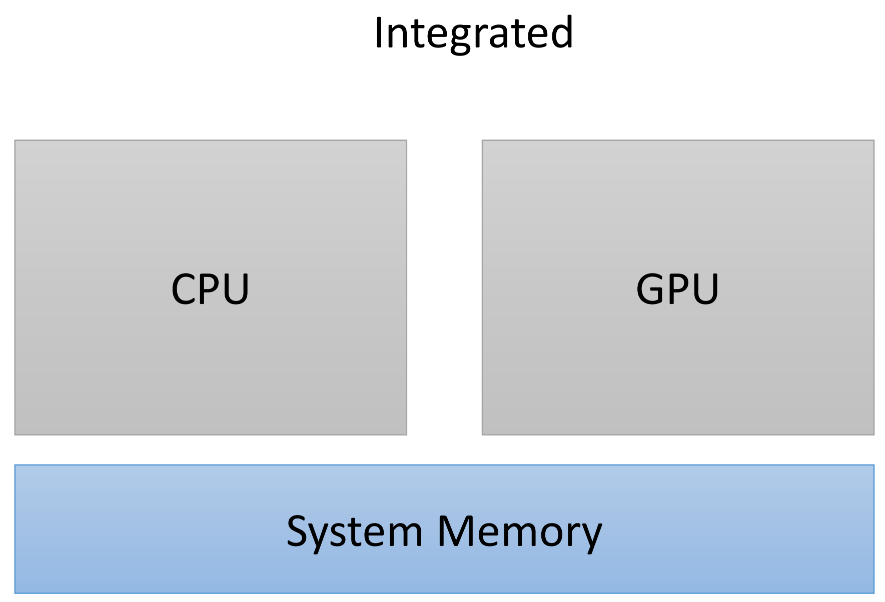
GPU set up

- GPUs come in two flavors

Pros and cons of each?
*Different types of memory for discrete
*Swappable for discrete
*More energy efficient for integrated
*Better memory utilization for integrated



datacenter GPUs
Big gaming GPUs



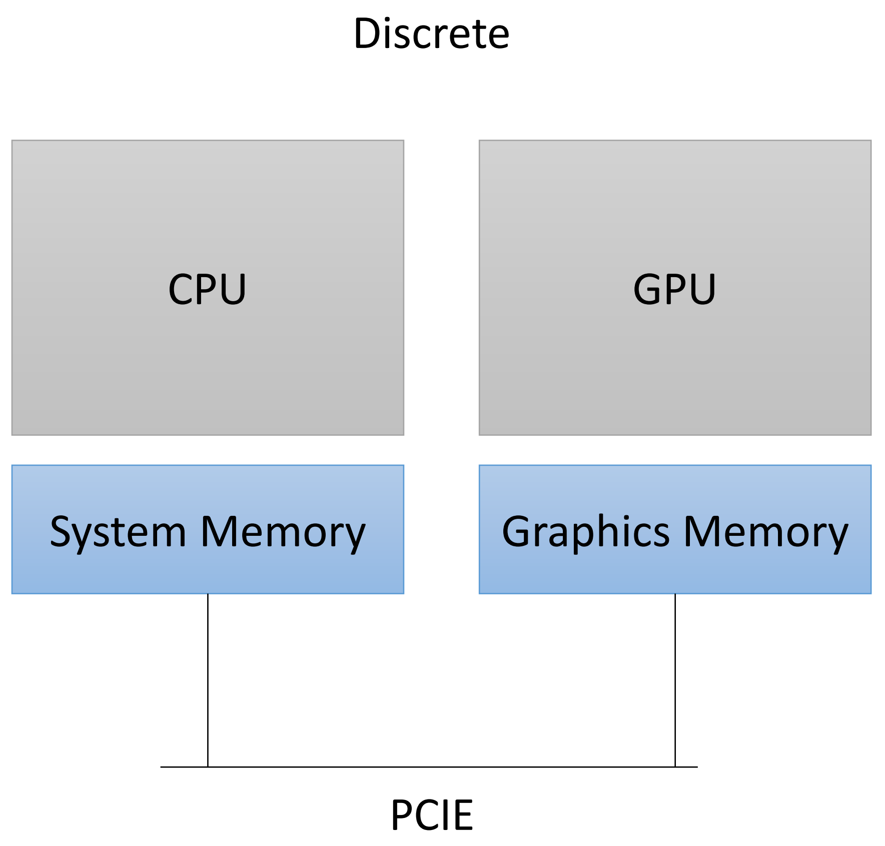
mobile SoCs

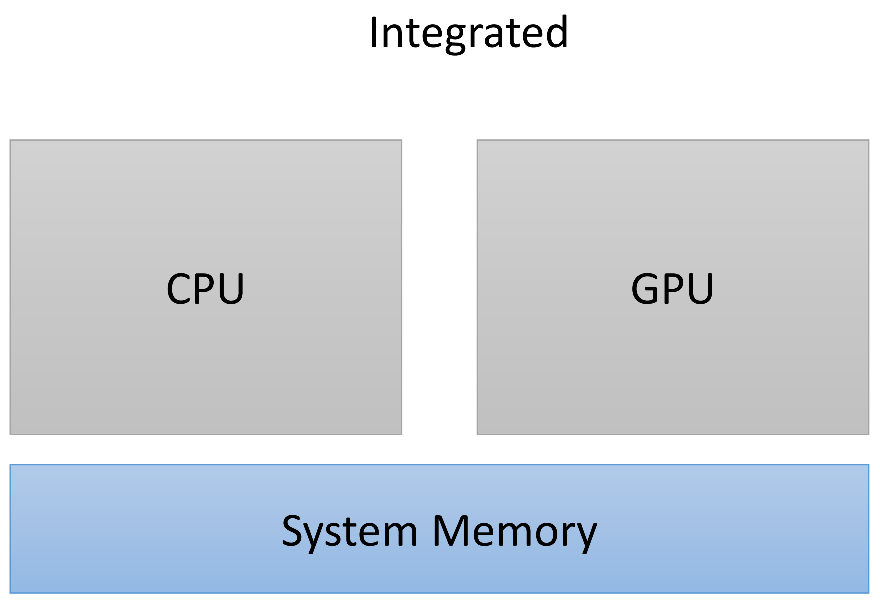
GPU set up

- GPUs come in two flavors

Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?



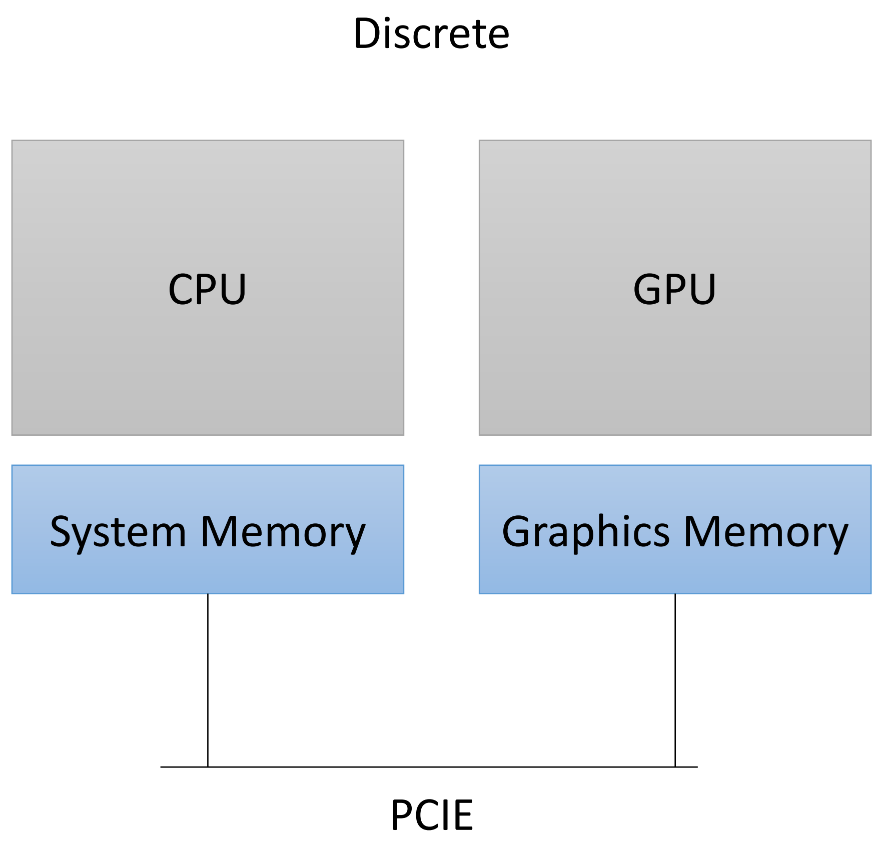


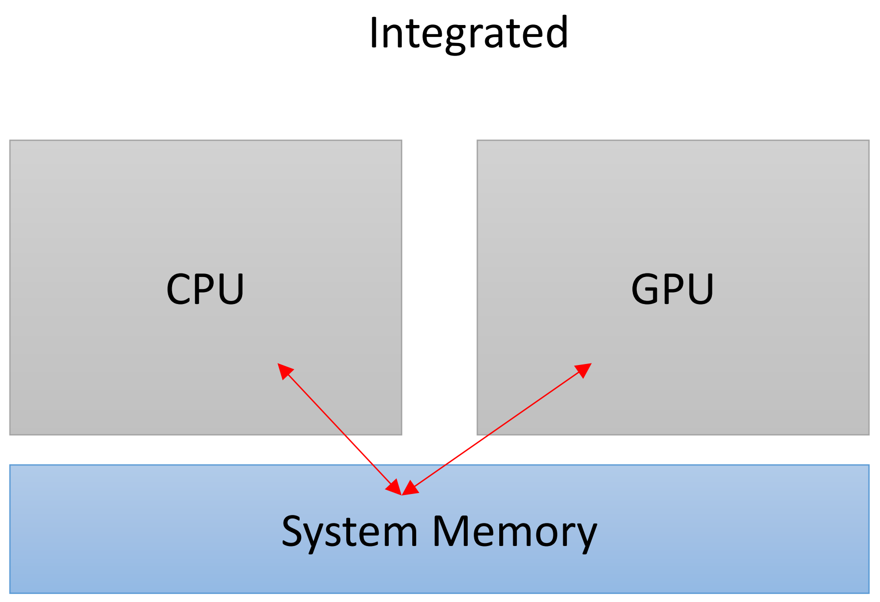
GPU set up

- GPUs come in two flavors

Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?



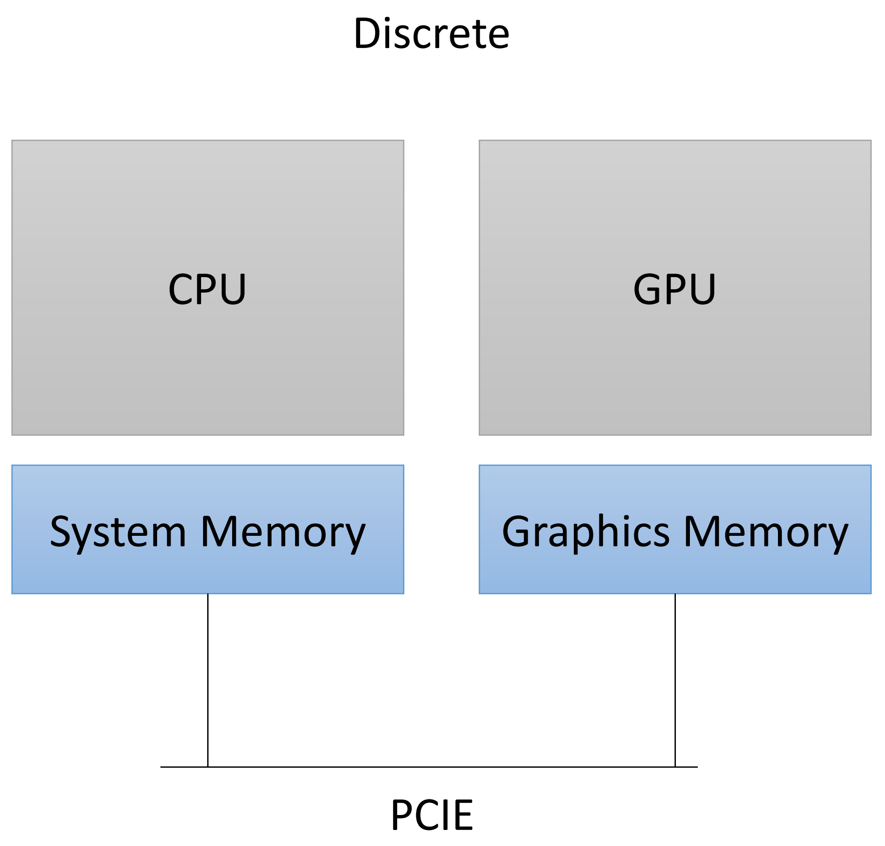


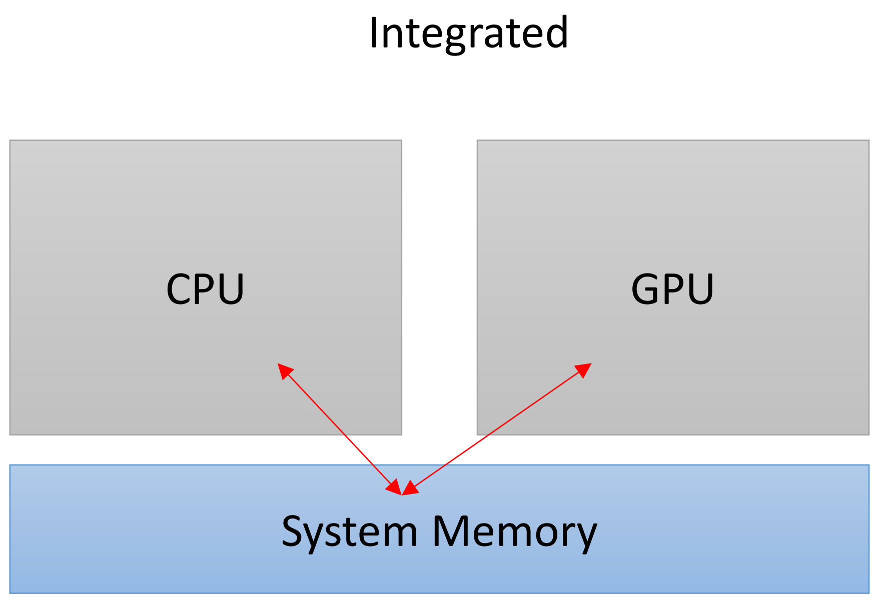
GPU set up

- GPUs come in two flavors

Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

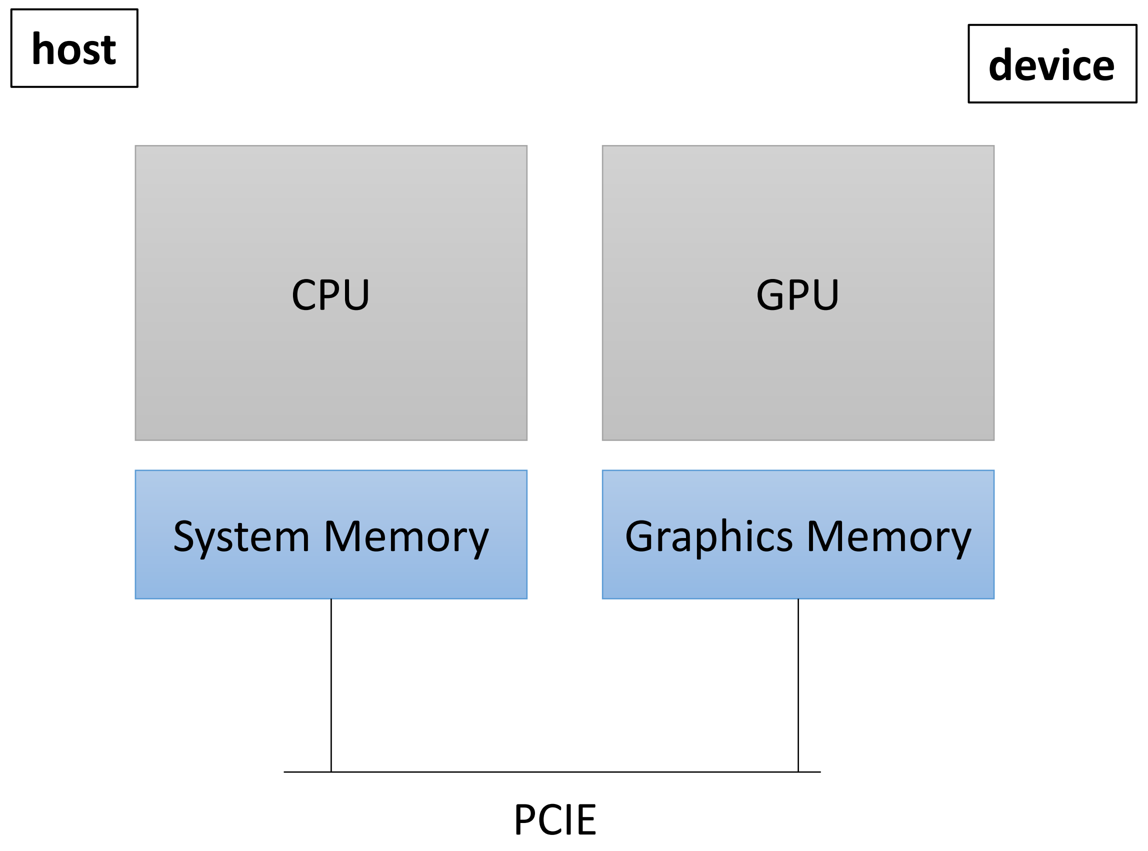




In many cases, CPU-GPU communication is not fully supported coherence, fences, and RMWs might now be supported.

GPU set up

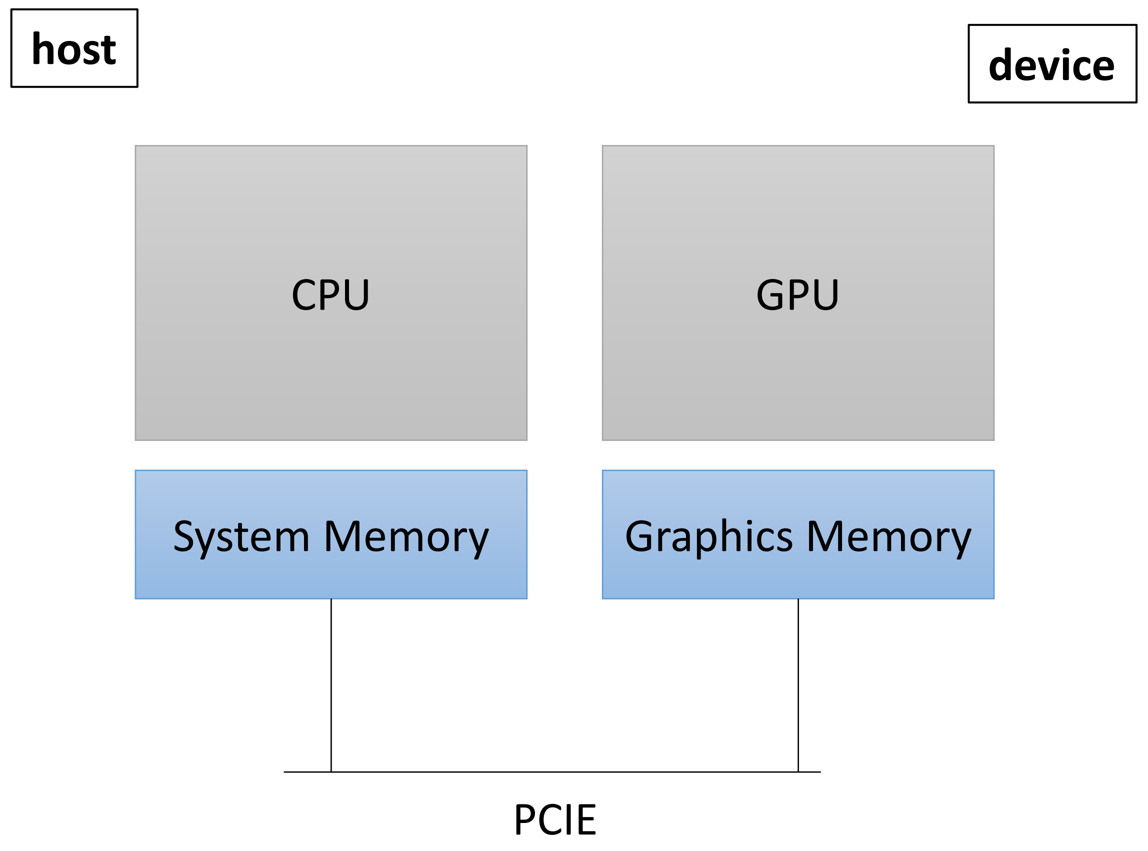
- Our heterogeneous, parallel, programming model



GPU set up

- Our heterogeneous, parallel, programming model

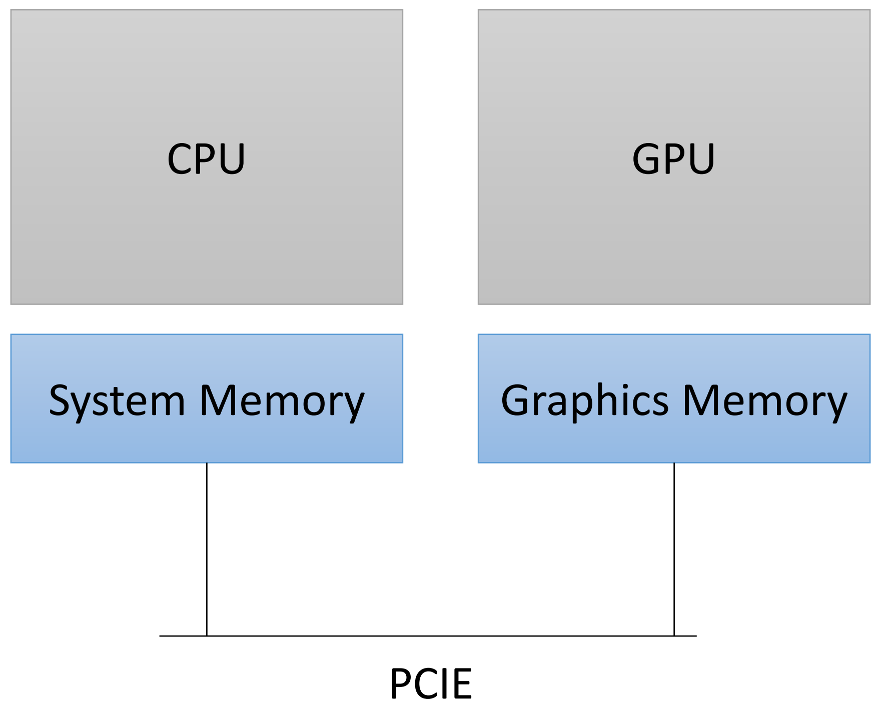
The host (CPU) will write a C++-like program that allocates and sets up memory on the GPU. The host will then call a GPU program (called) a kernel)



GPU set up

How do we allocate memory on a CPU?

- Our heterogeneous, parallel, programming model

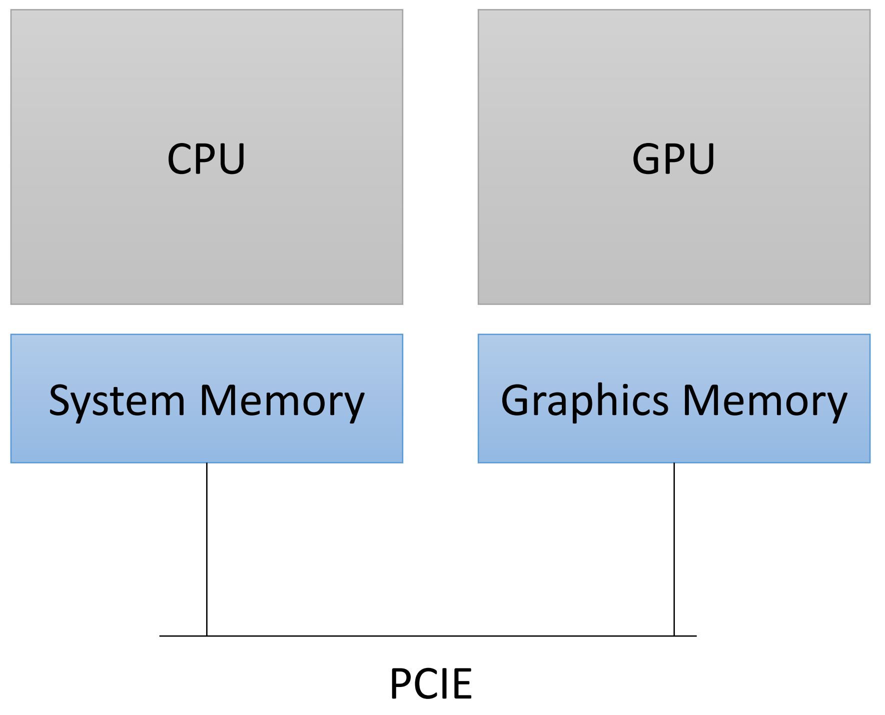


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

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

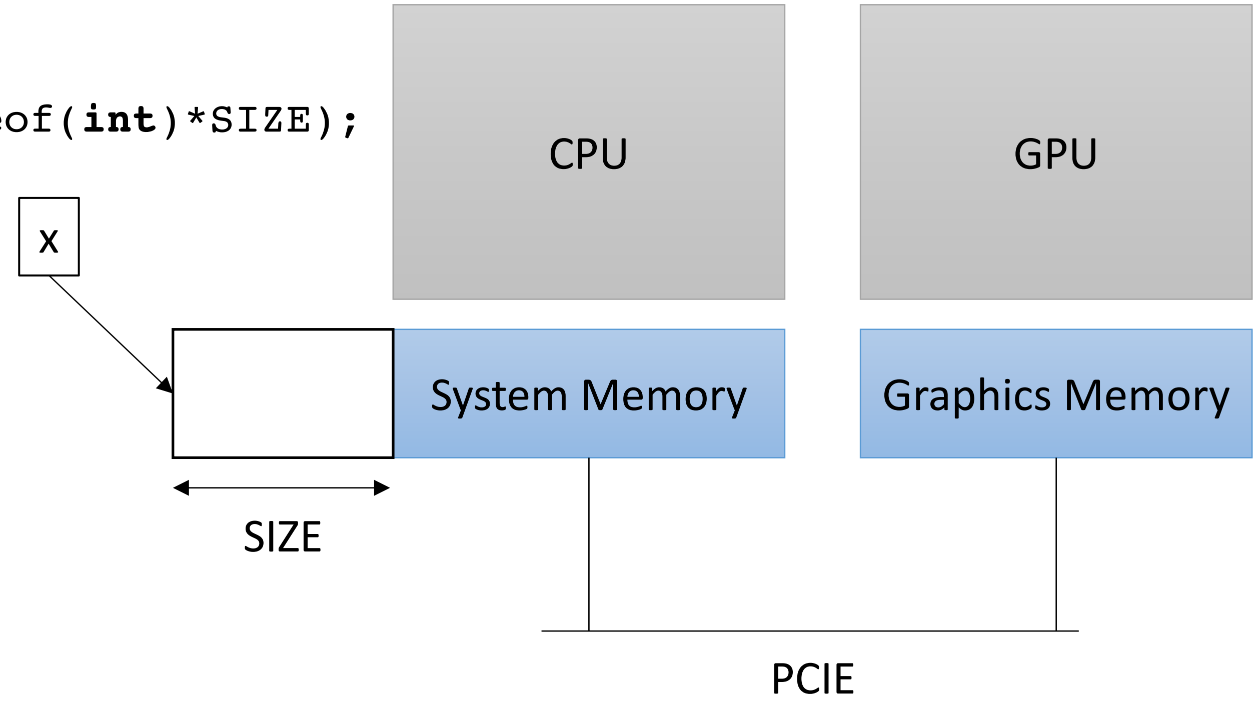


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

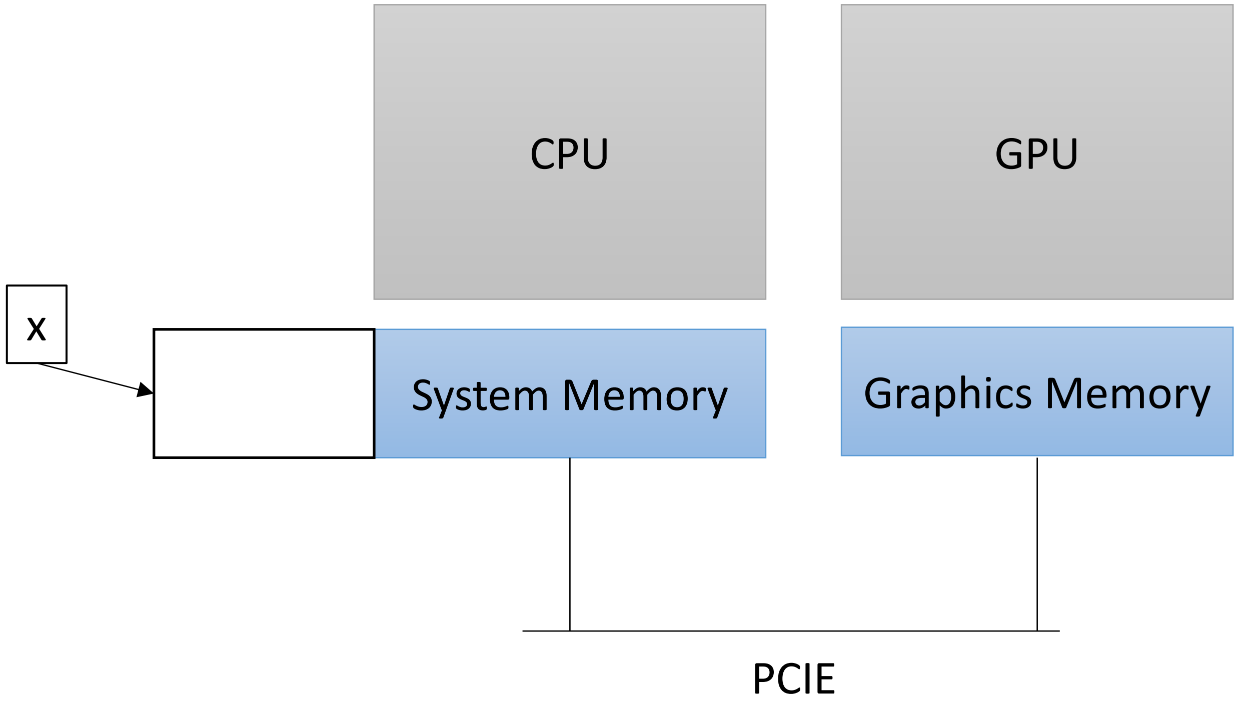
```
int *x = (int*) malloc(sizeof(int)*SIZE);
```



GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

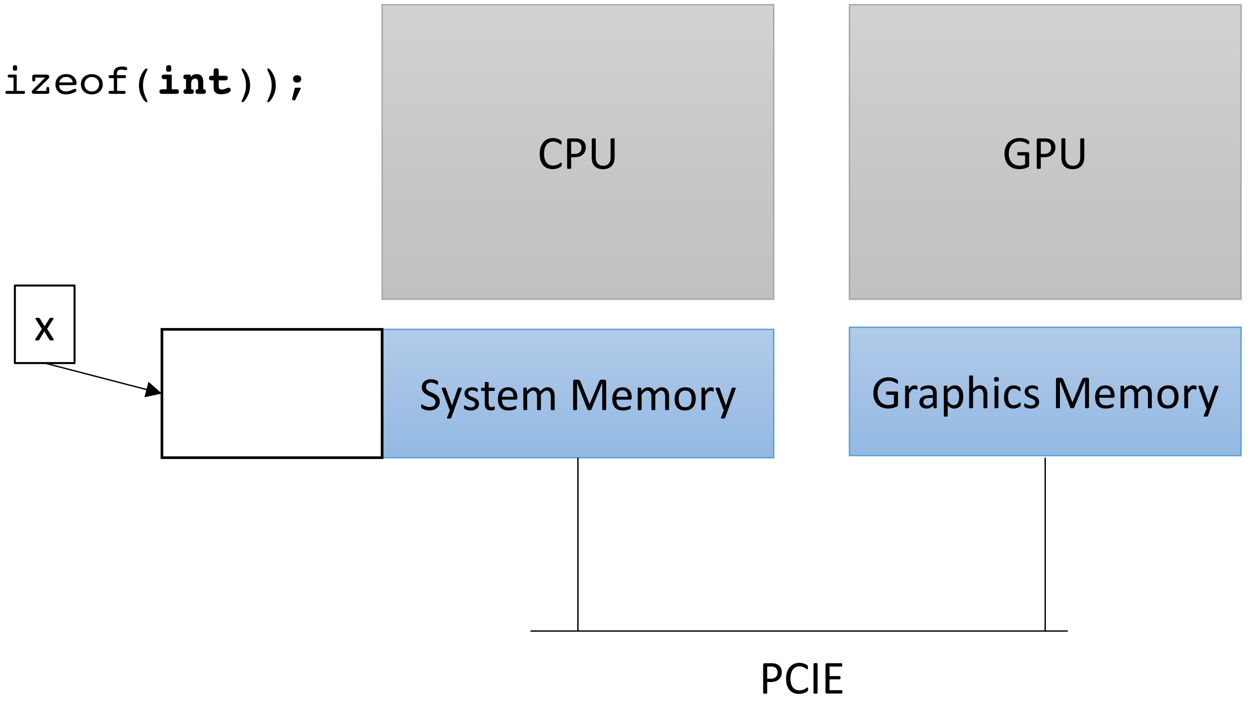


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

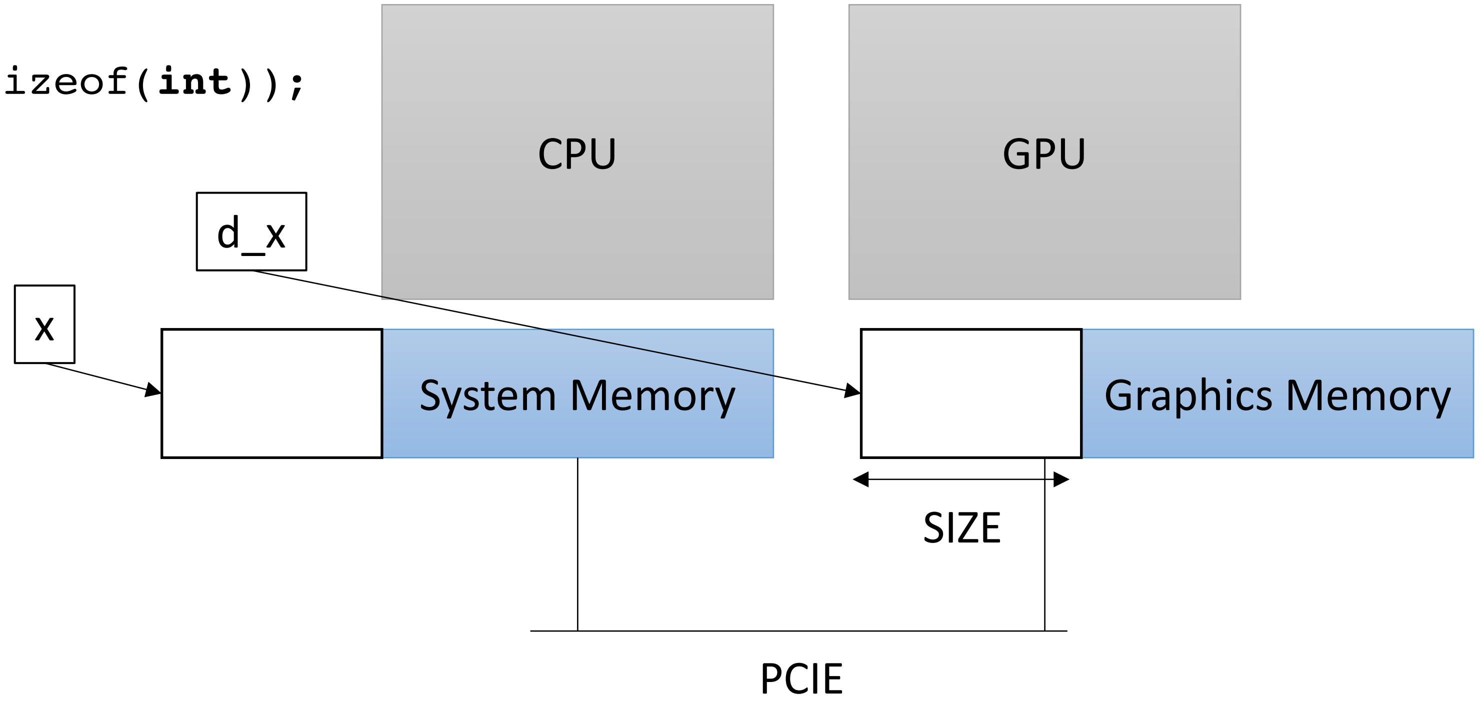


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

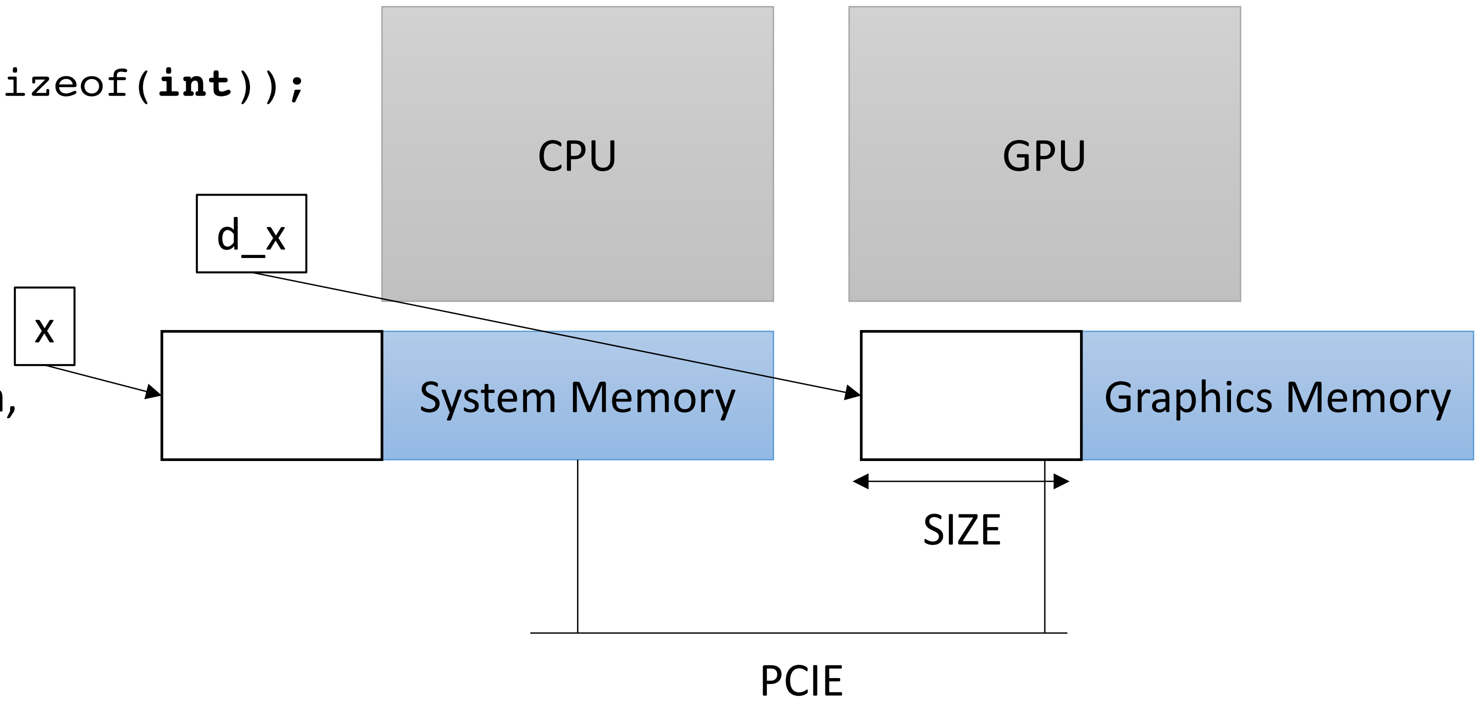


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

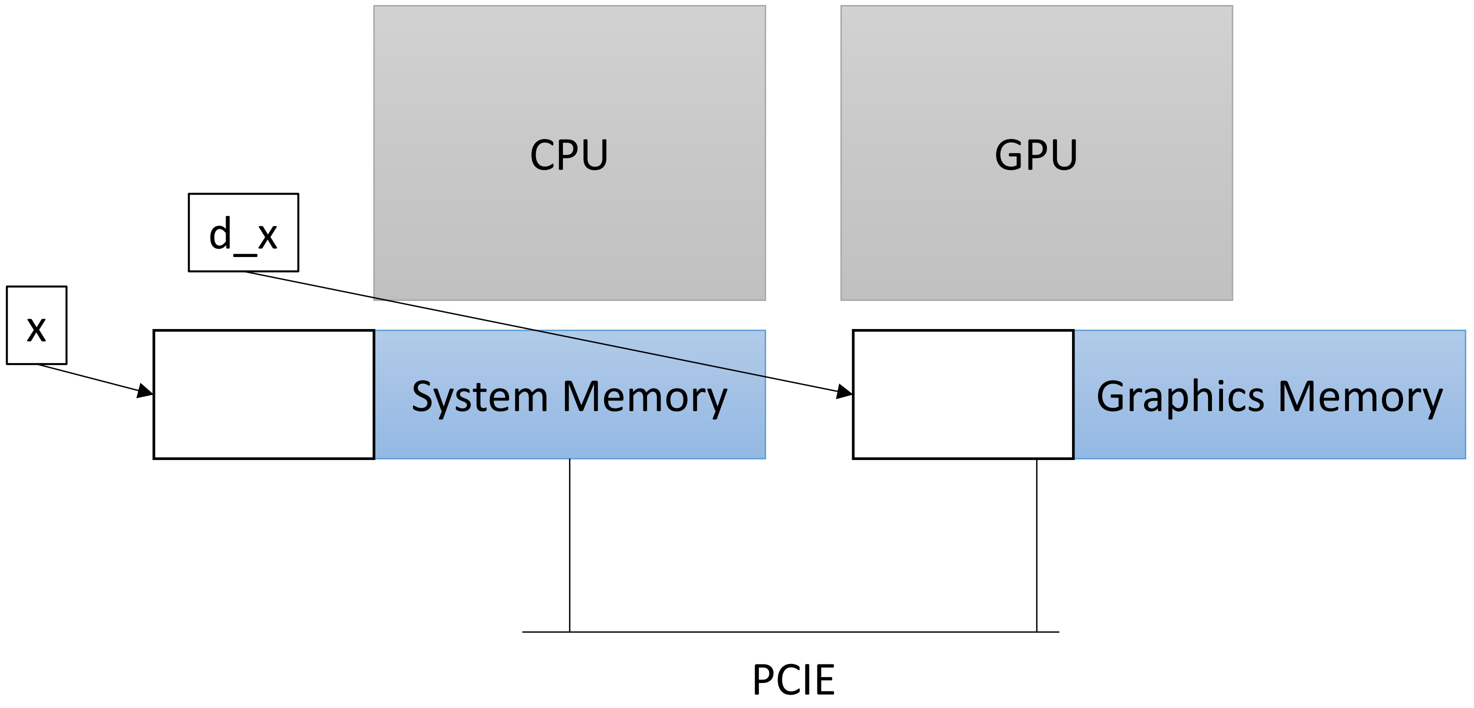


`d_x` is a pointer, in the CPU program, that points to memory on the GPU.

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

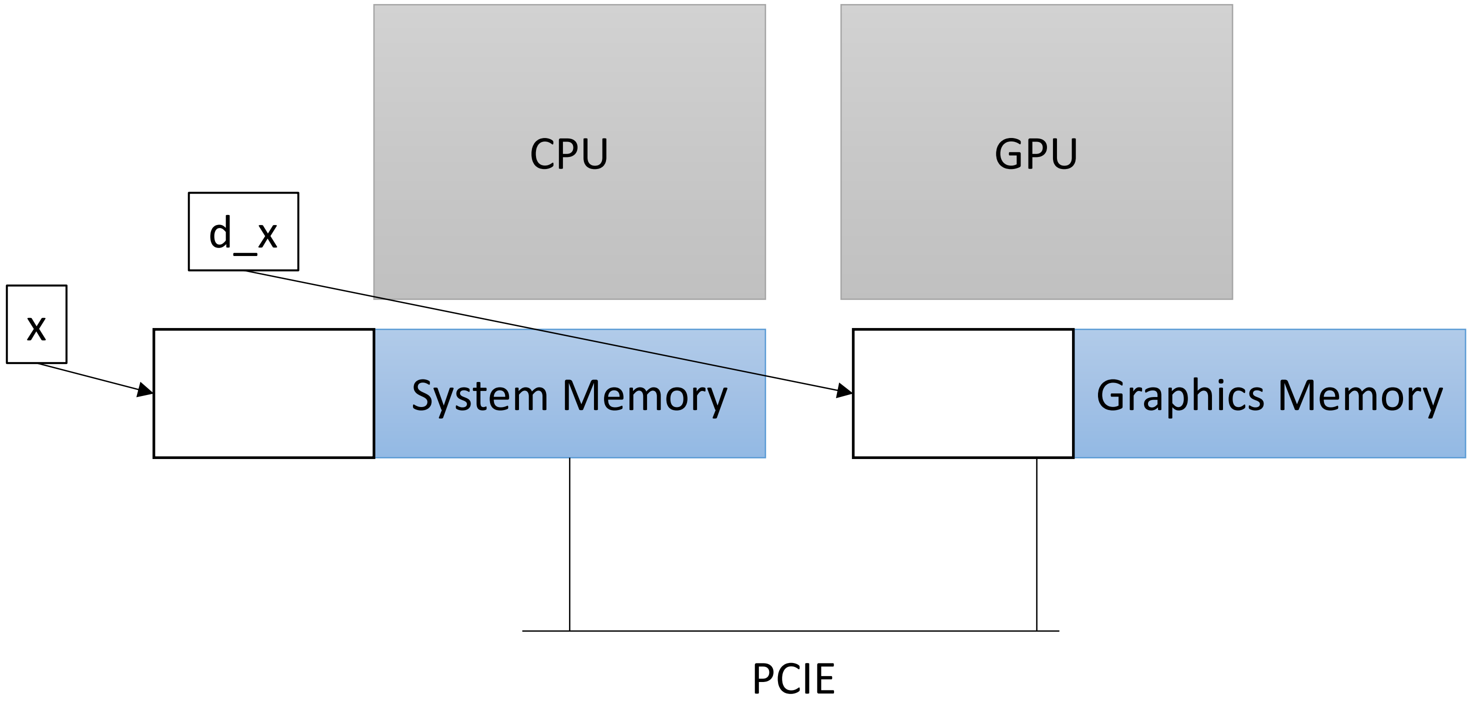


GPU set up

- 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



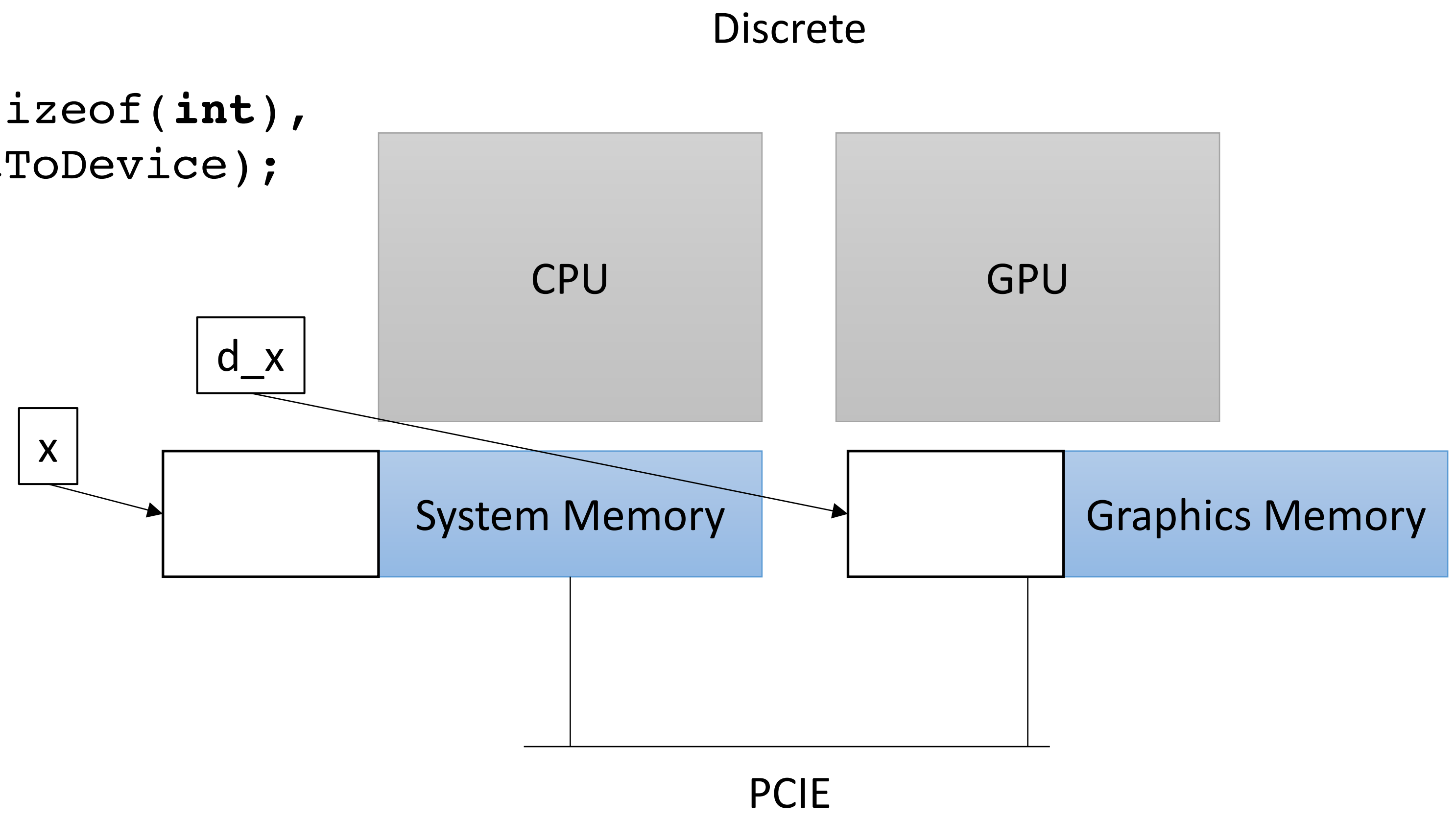
GPU set up

- 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);
```



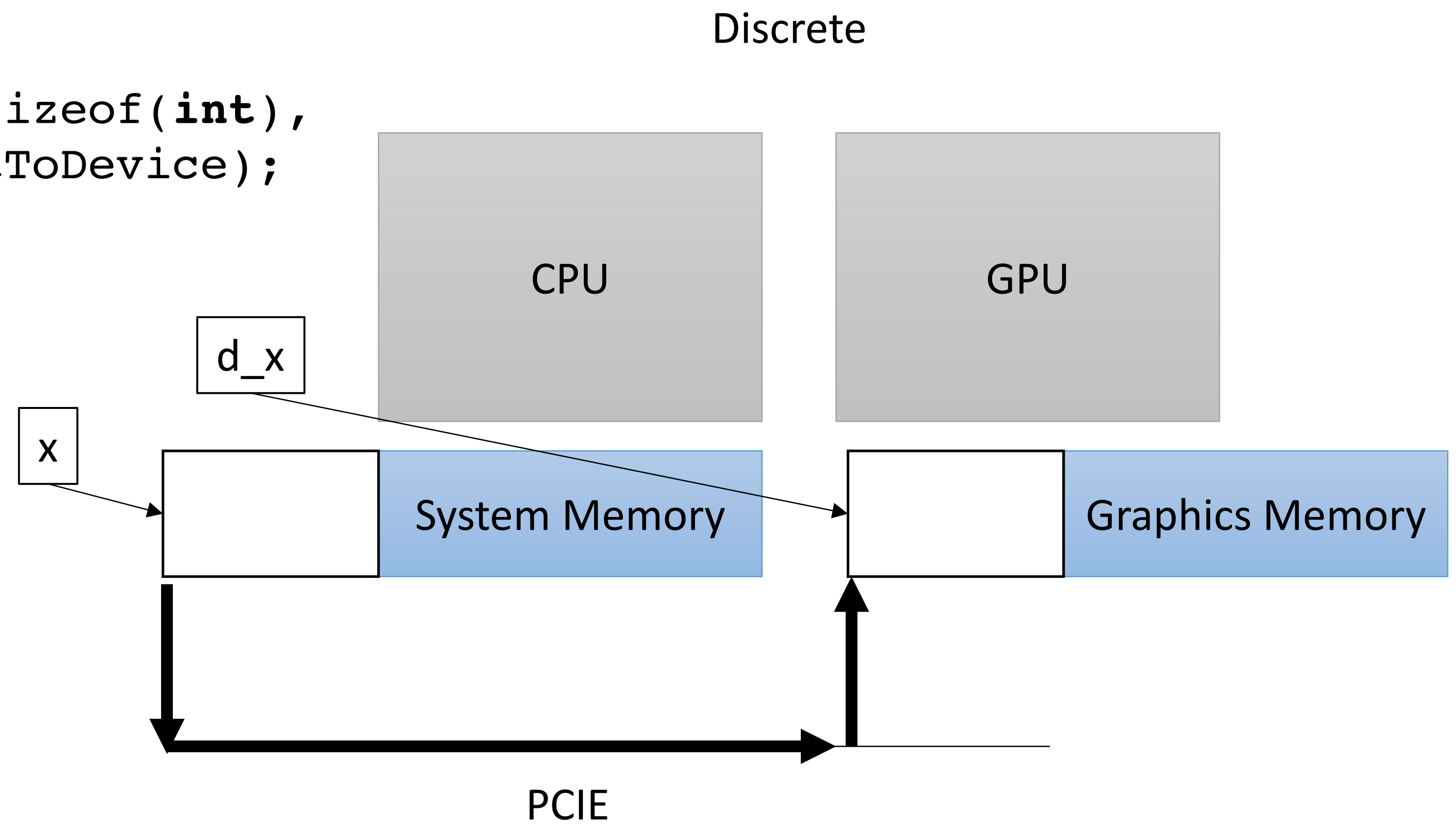
GPU set up

- 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

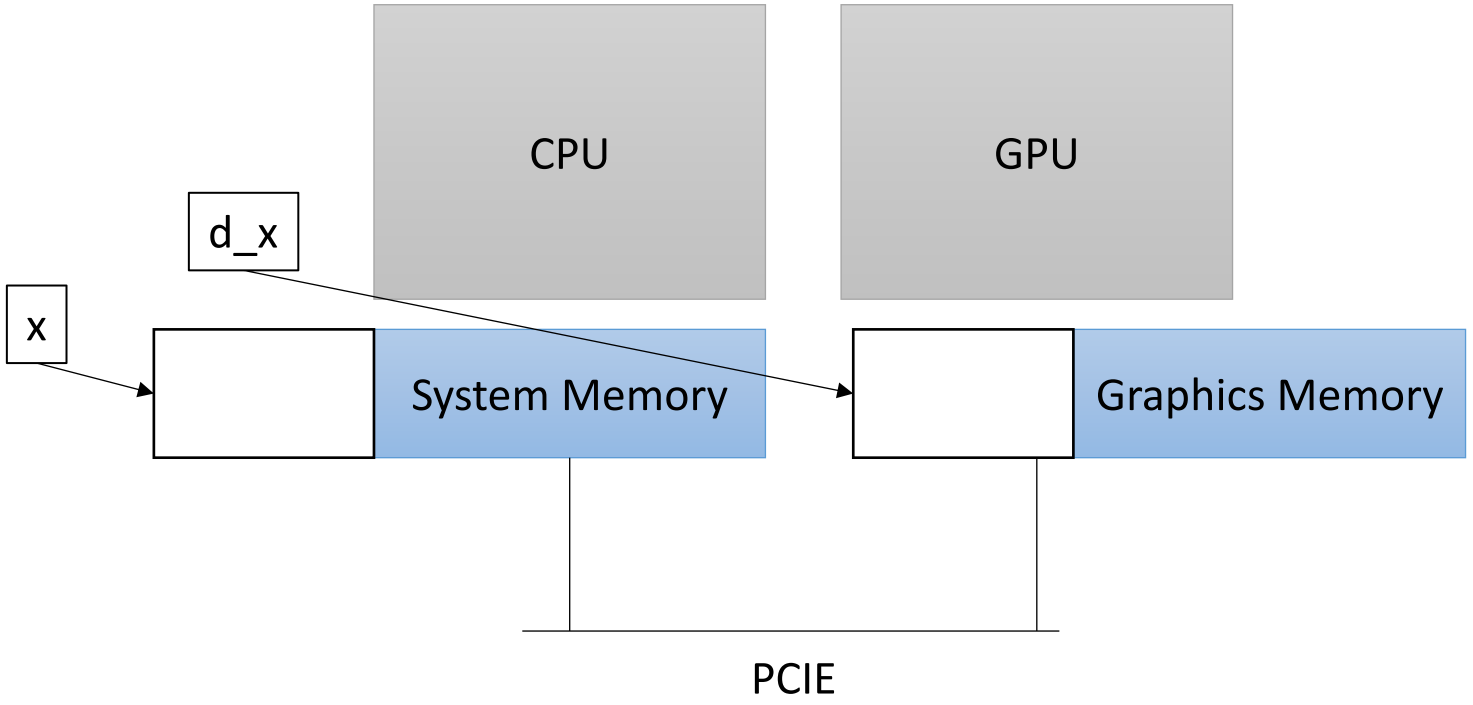
calling the function

```
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);
```

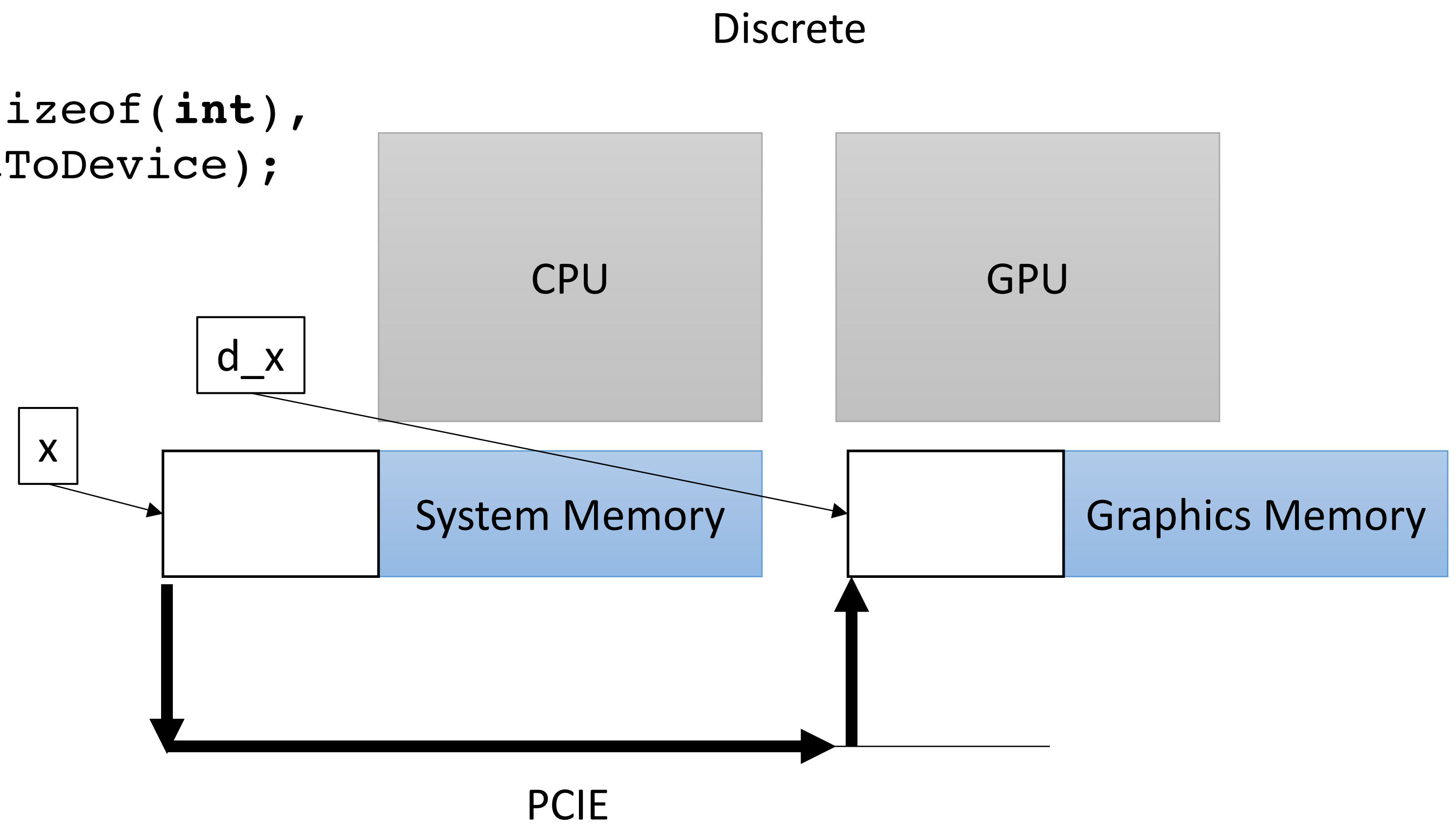
The GPU Program

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

GPU set up

- Our heterogeneous, parallel, programming model

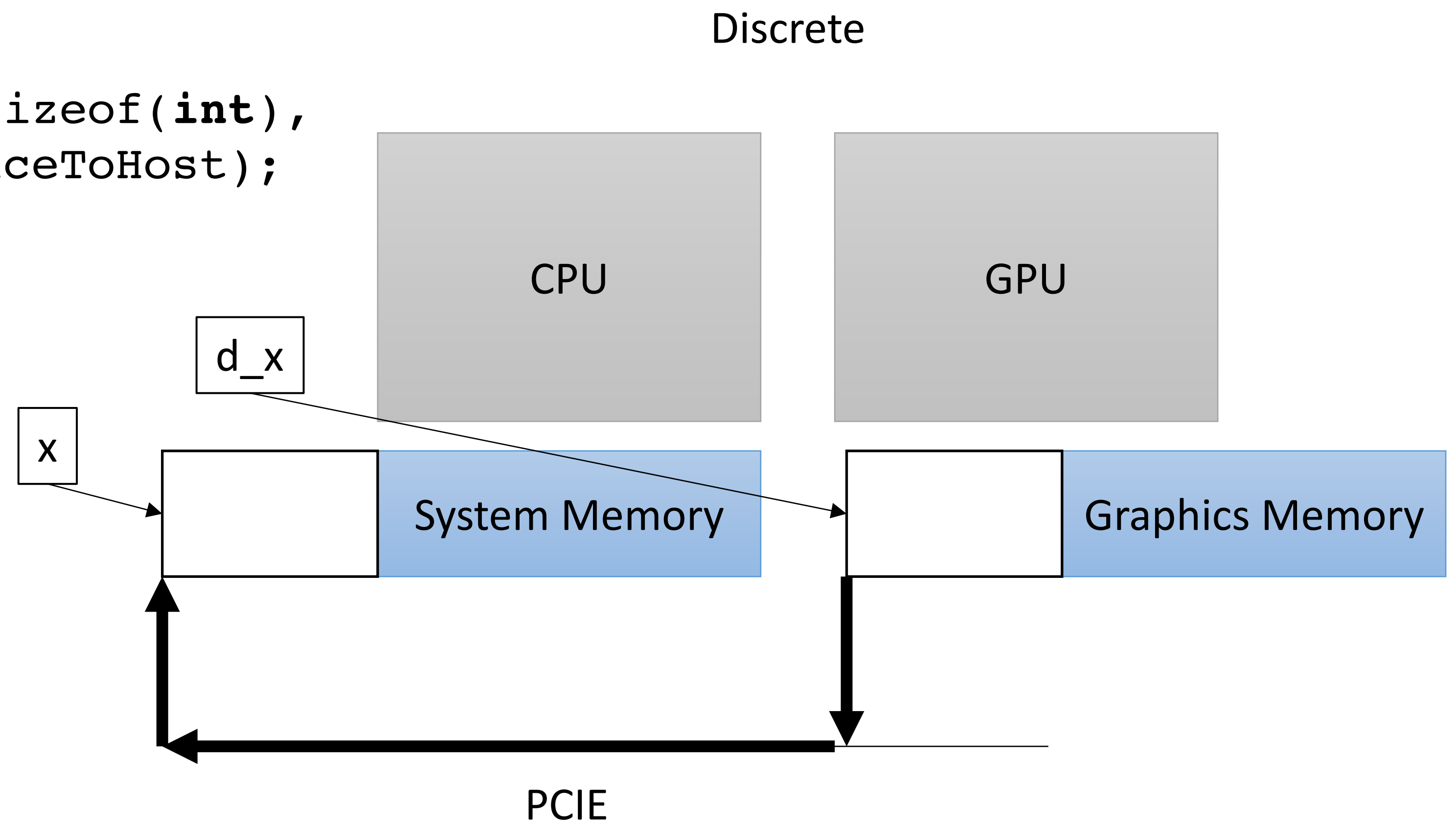
```
//initialize x on host
cudaMemcpy(d_x, x, SIZE*sizeof(int),
           cudaMemcpyHostToDevice);
```



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

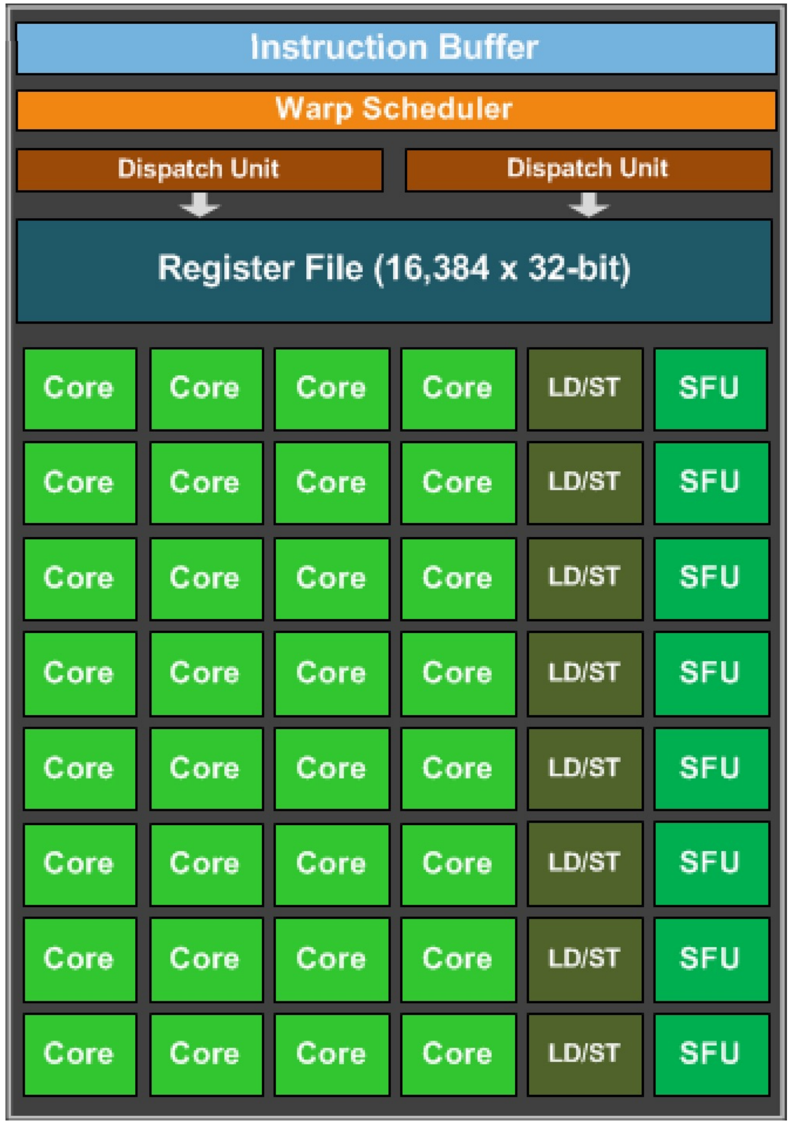
The GPU Program

It didn’t do so well...

First parallelization attempt

- Lets look at some GPU documentation.

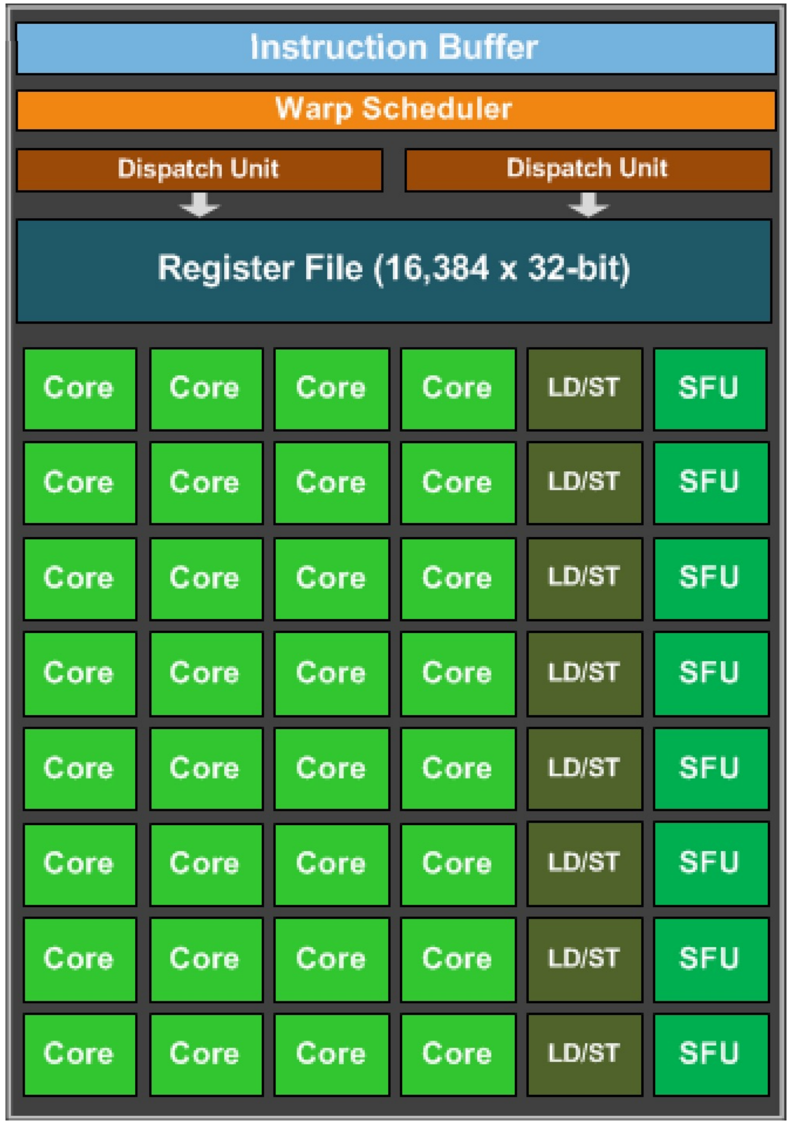
- The Maxwell whitepaper shows a diagram of one of the GPU cores



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

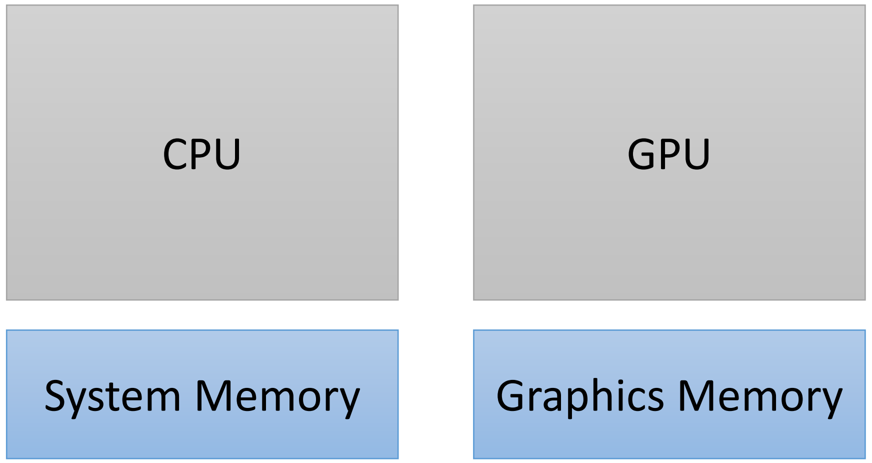
First parallelization attempt

Lets try it! What do we think?

First parallelization attempt

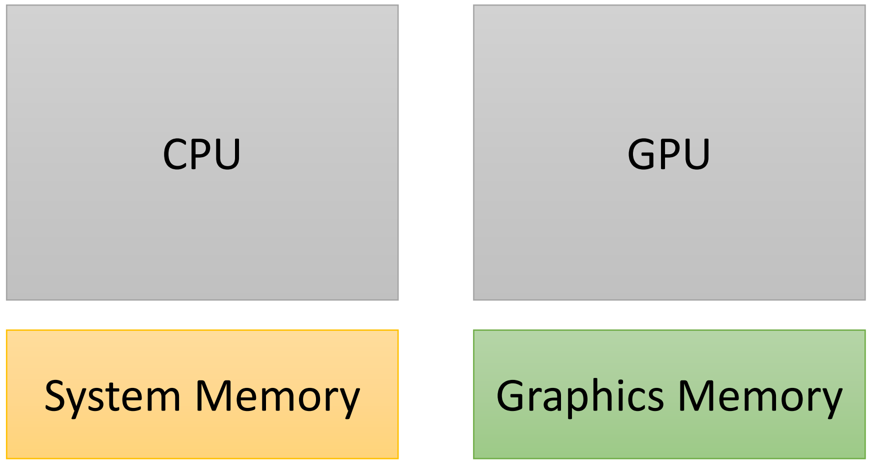
Getting better but we have a long ways to go!

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

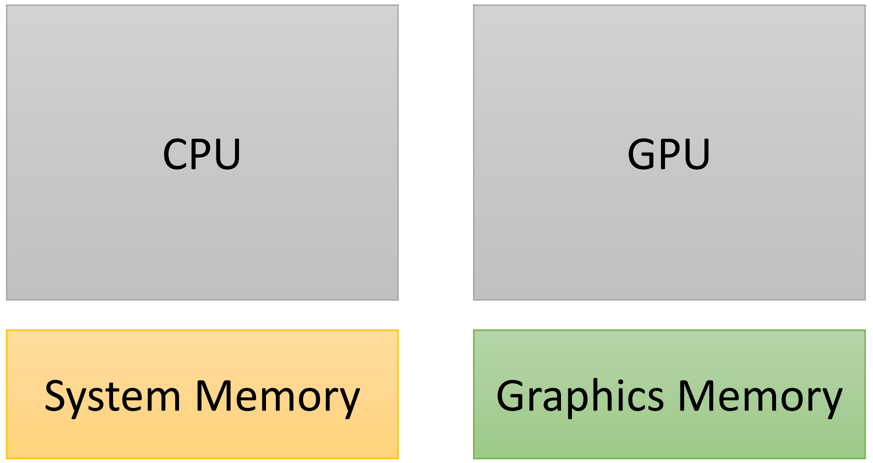
16-lane highway on a windy road driven by semi trucks

GPU Memory

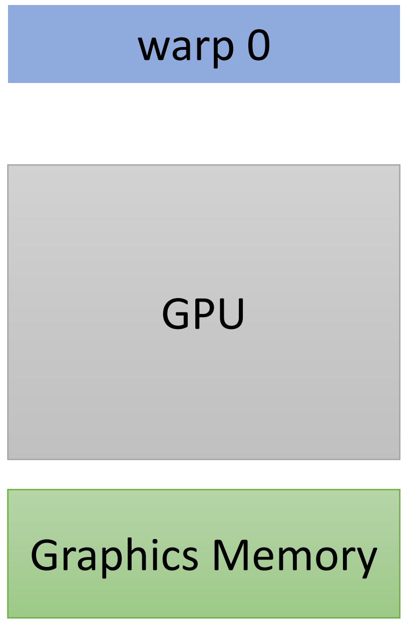
bandwidth:
~**700 GB/s** for GPU
~**50 GB/s** for CPUs

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

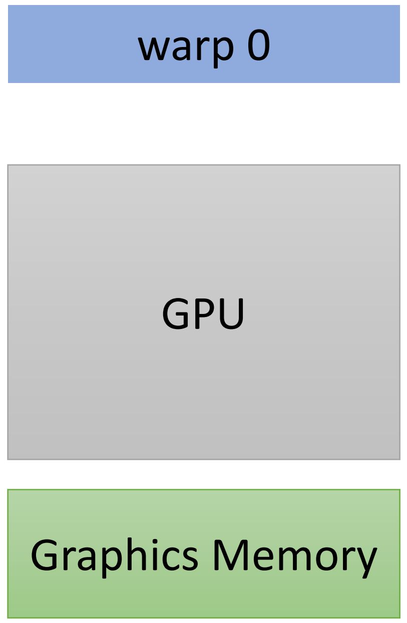
Cache Latency:
~**28** cycles for L1 hit for GPU
~**4** cycles for L1 hit on CPUs



Preemption and concurrency?

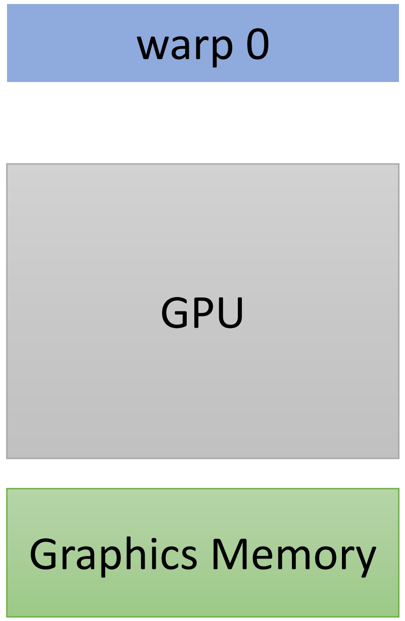


Preemption and concurrency?



all threads load from memory.

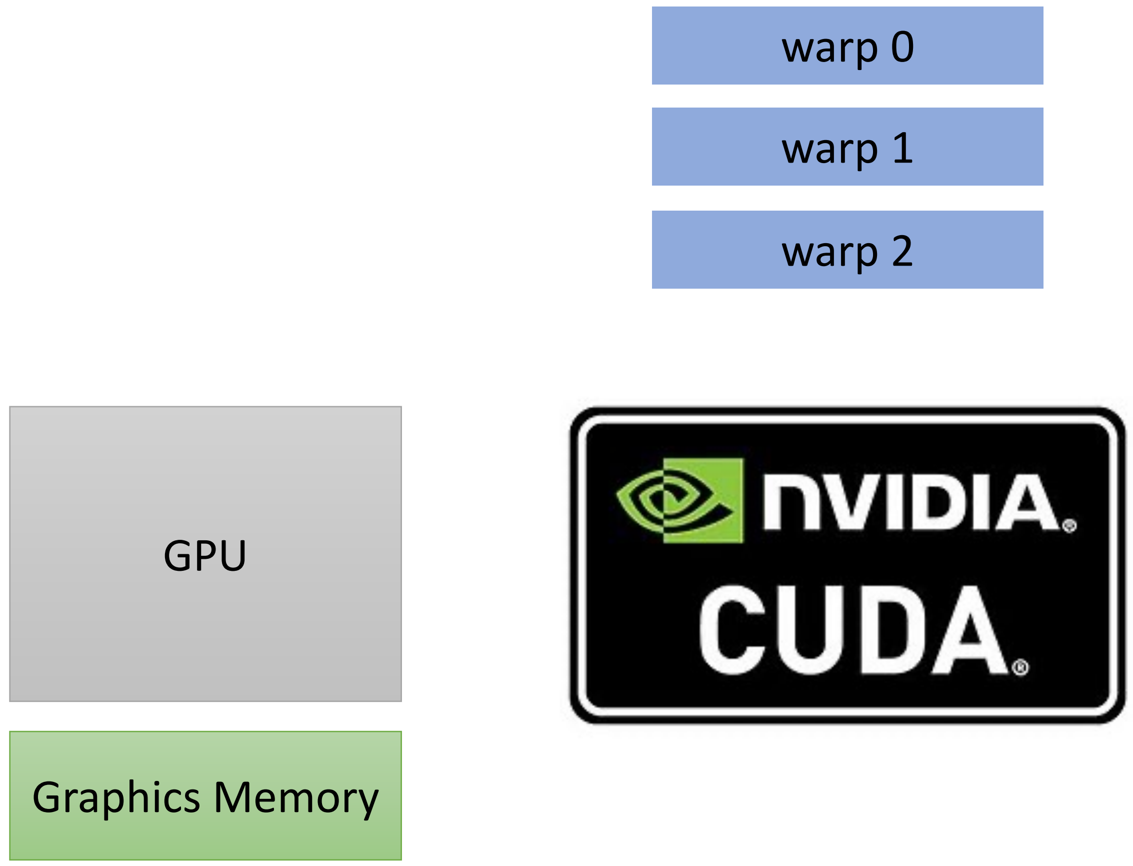
Preemption and concurrency?



all threads load from memory.

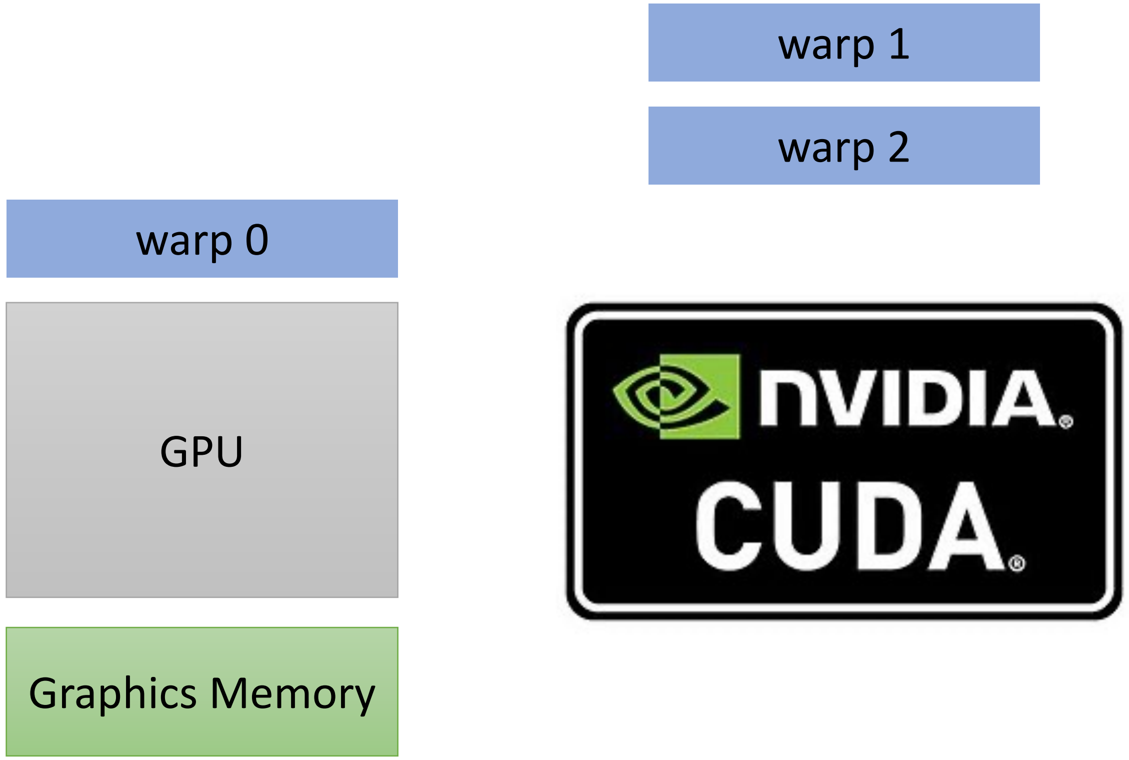
600 cycles!

Preemption and concurrency?



We can hide latency through preemption and concurrency!

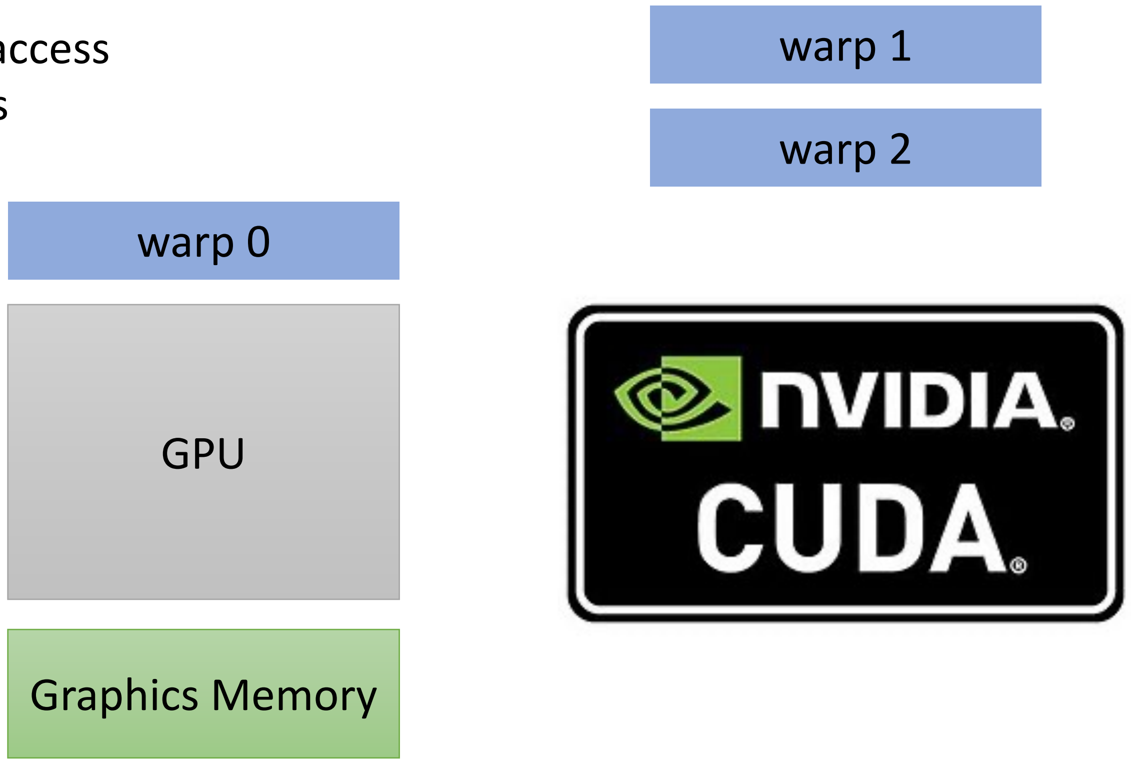
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

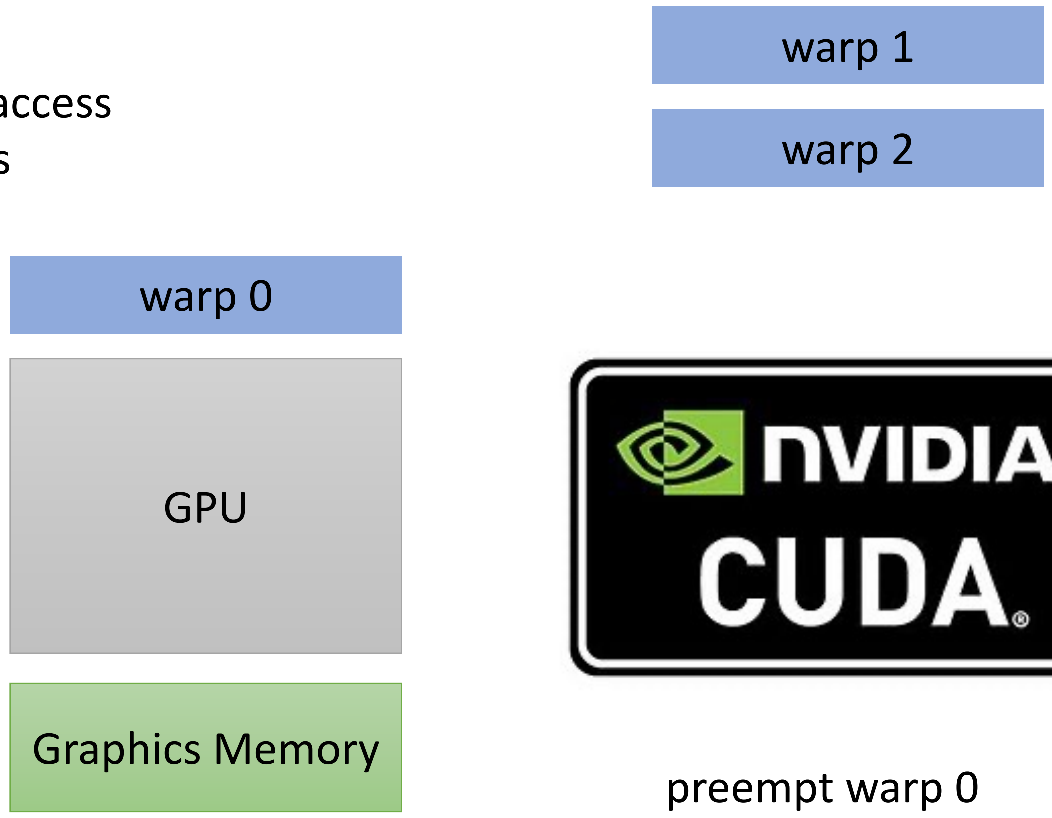
memory access
600 cycles



We can hide latency through preemption and concurrency!

Preemption and concurrency?

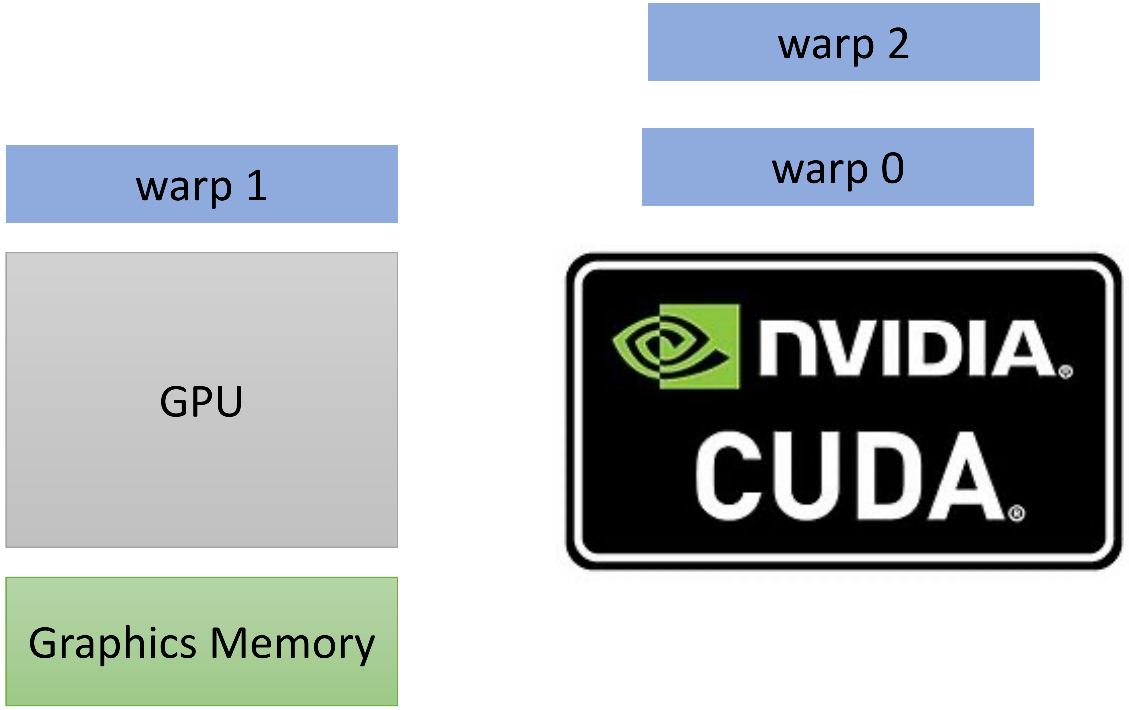
memory access
600 cycles



preempt warp 0
and put warp 1 on

We can hide latency through
preemption and concurrency!

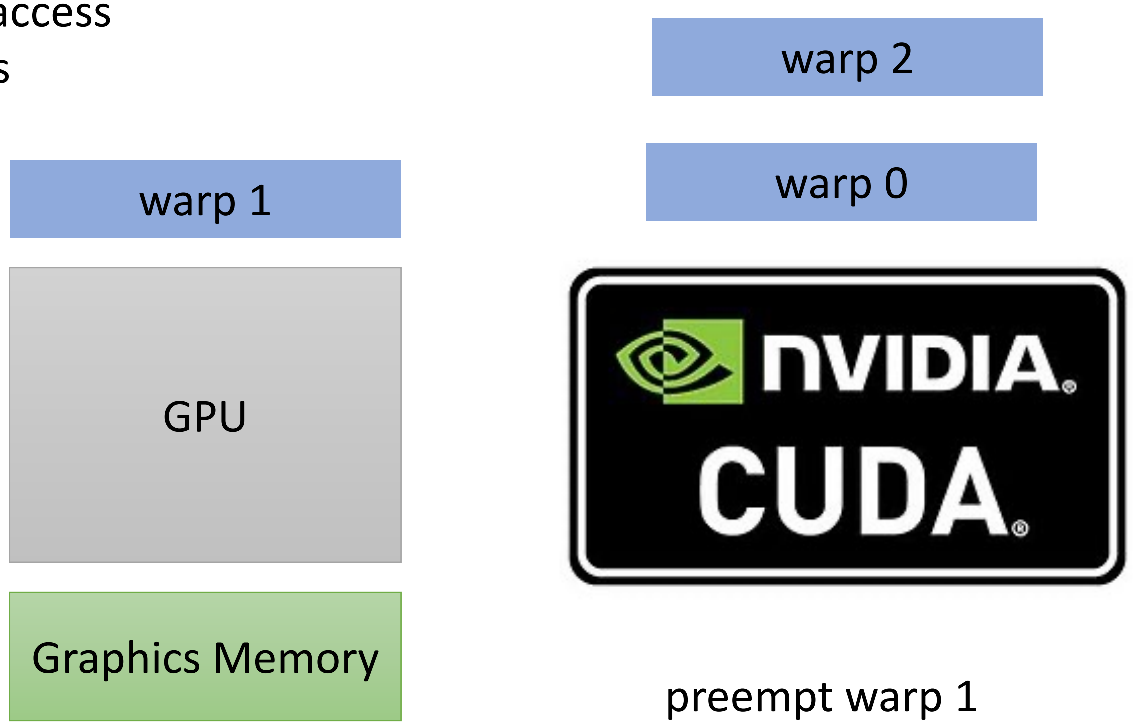
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

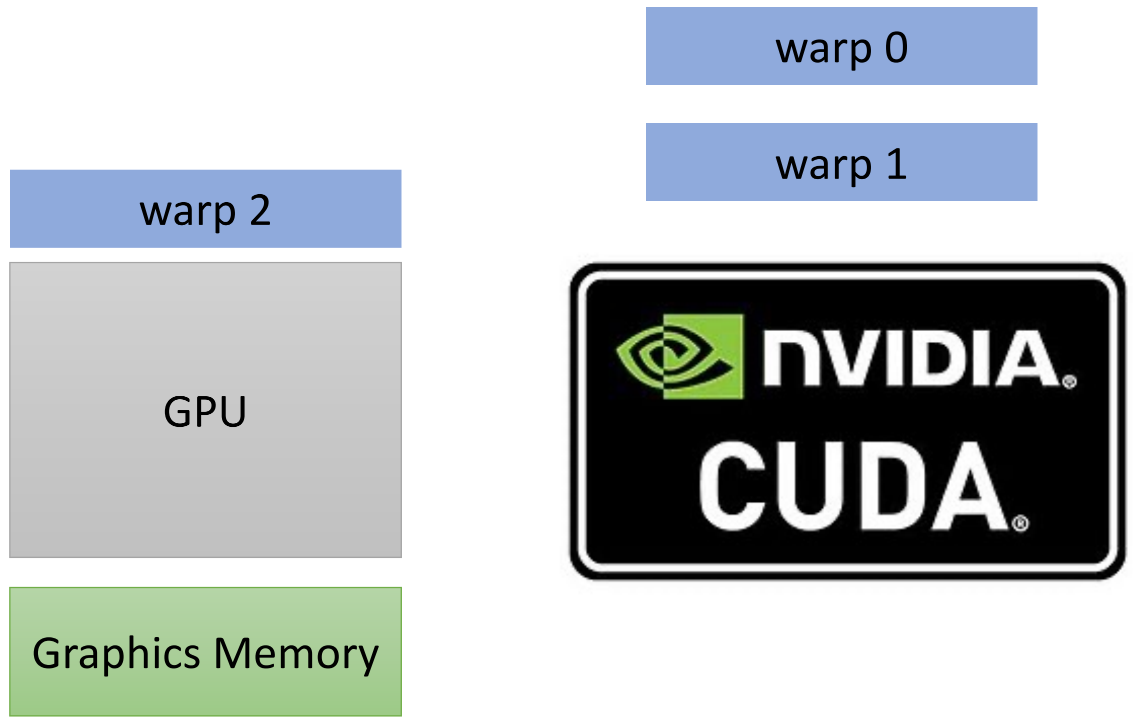
memory access
600 cycles



preempt warp 1
and put warp 2 on

We can hide latency through preemption and concurrency!

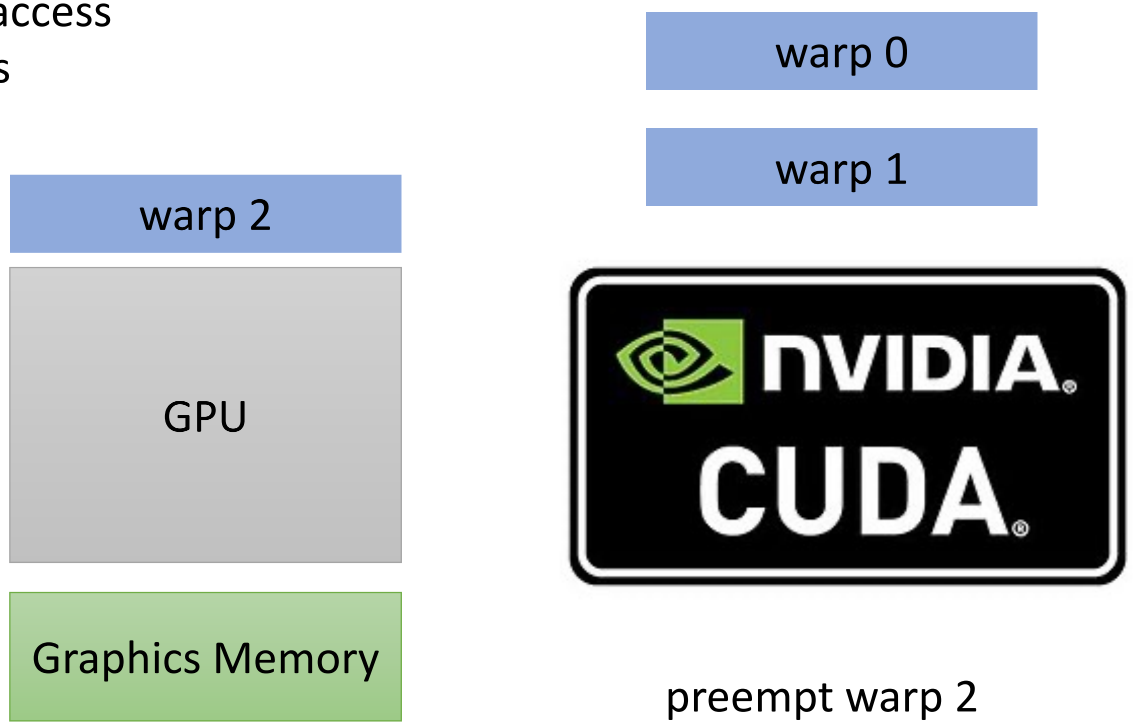
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

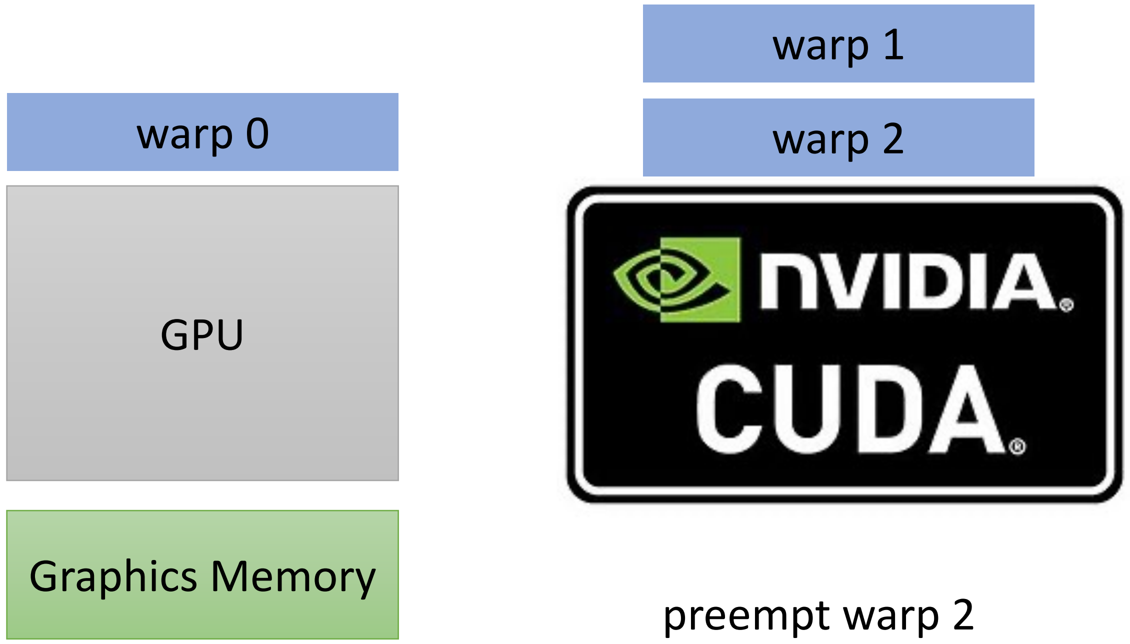


preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!



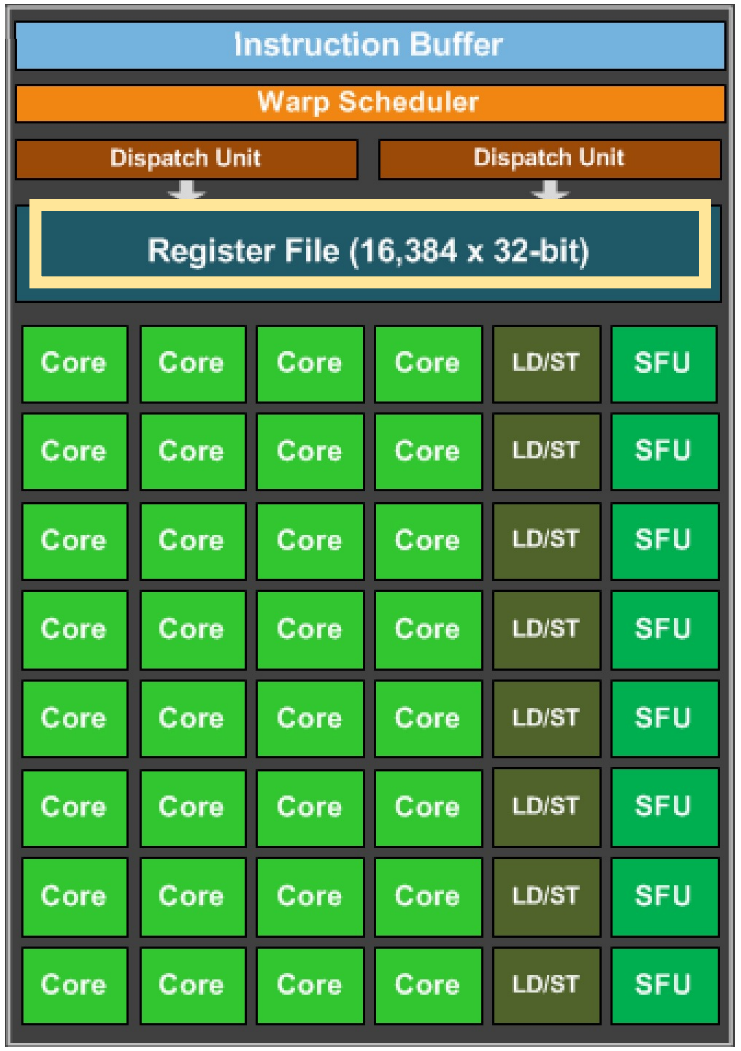
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

But wait, I thought preemption was expensive?

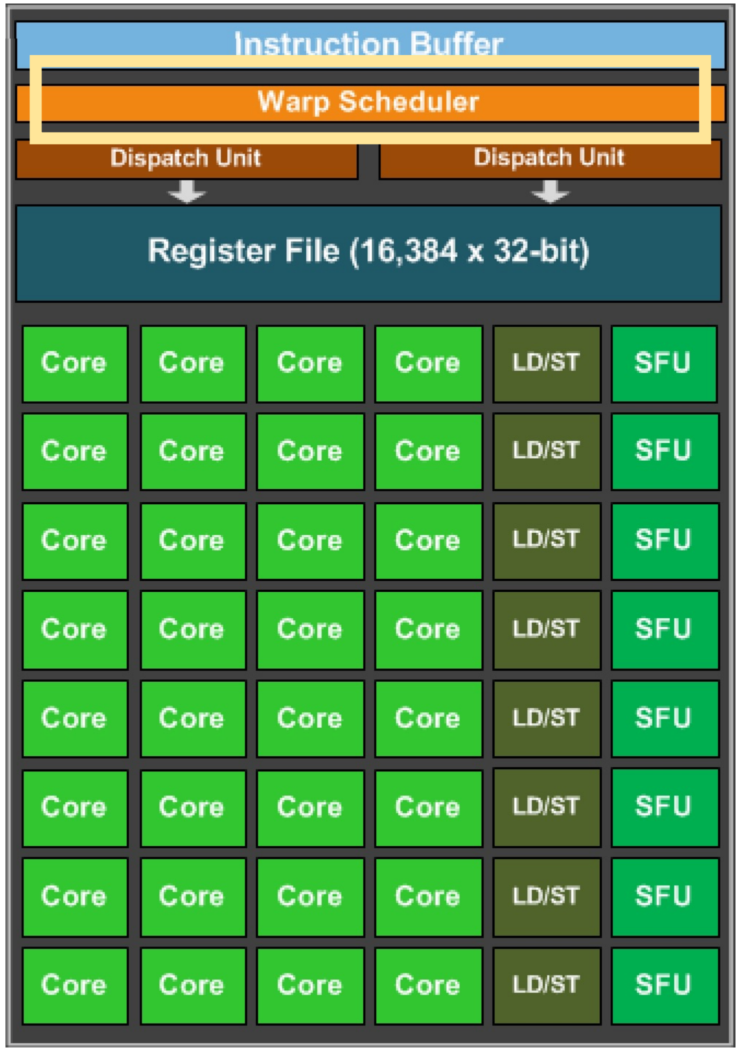
Preemption and concurrency?



But wait, I thought preemption was expensive?

Registers all stay on chip

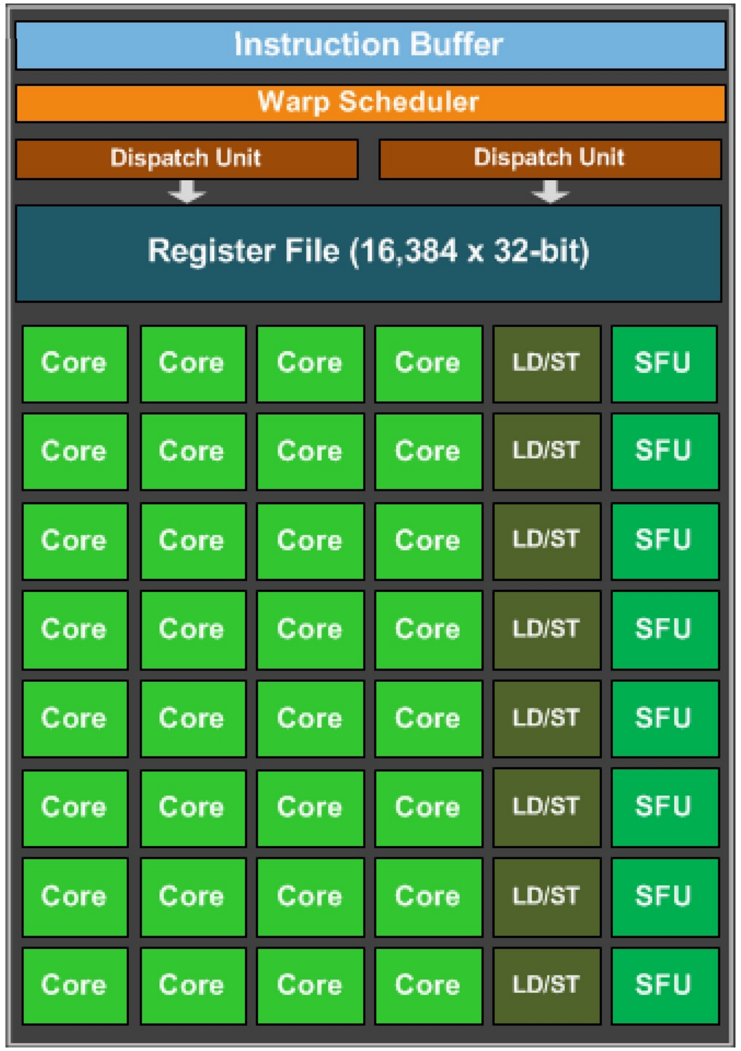
Preemption and concurrency?



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

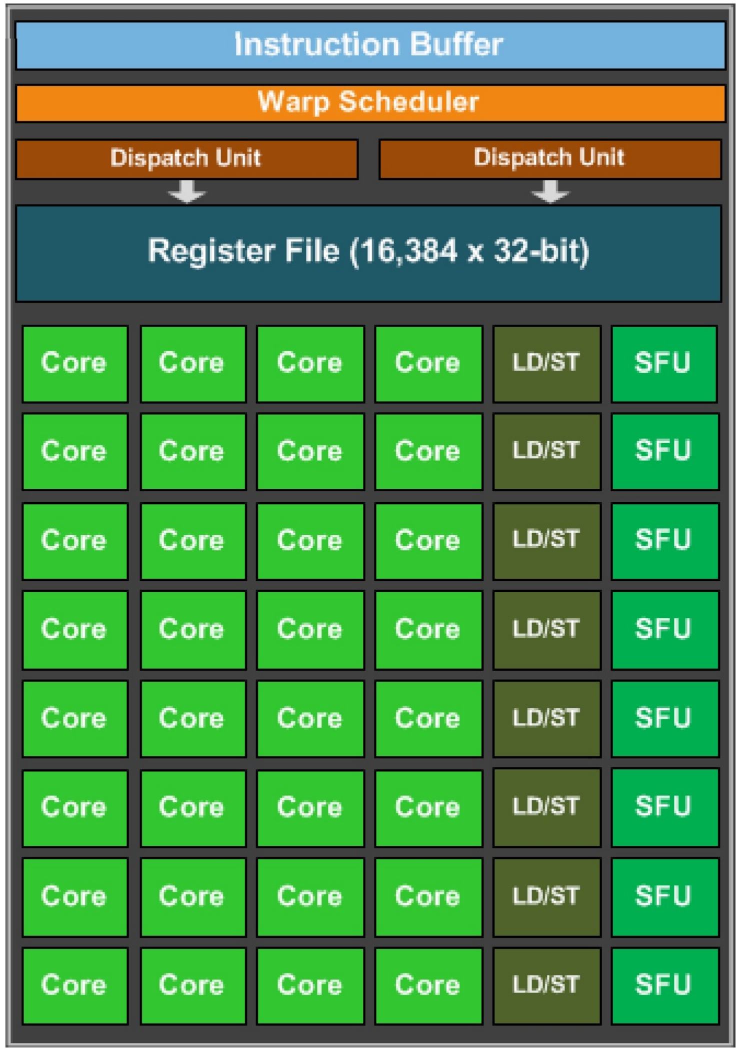
Lets try it! What do we think?

Concurrent warps

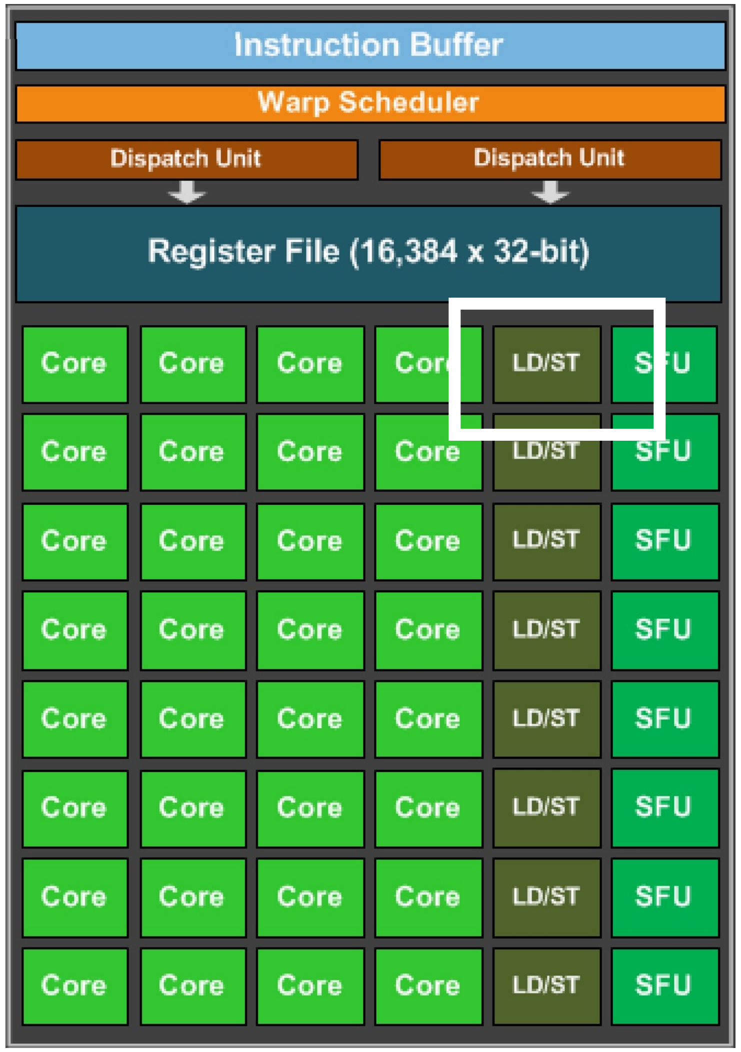
Lets try it! What do we think? 😀

Getting better!

Optimizing memory accesses



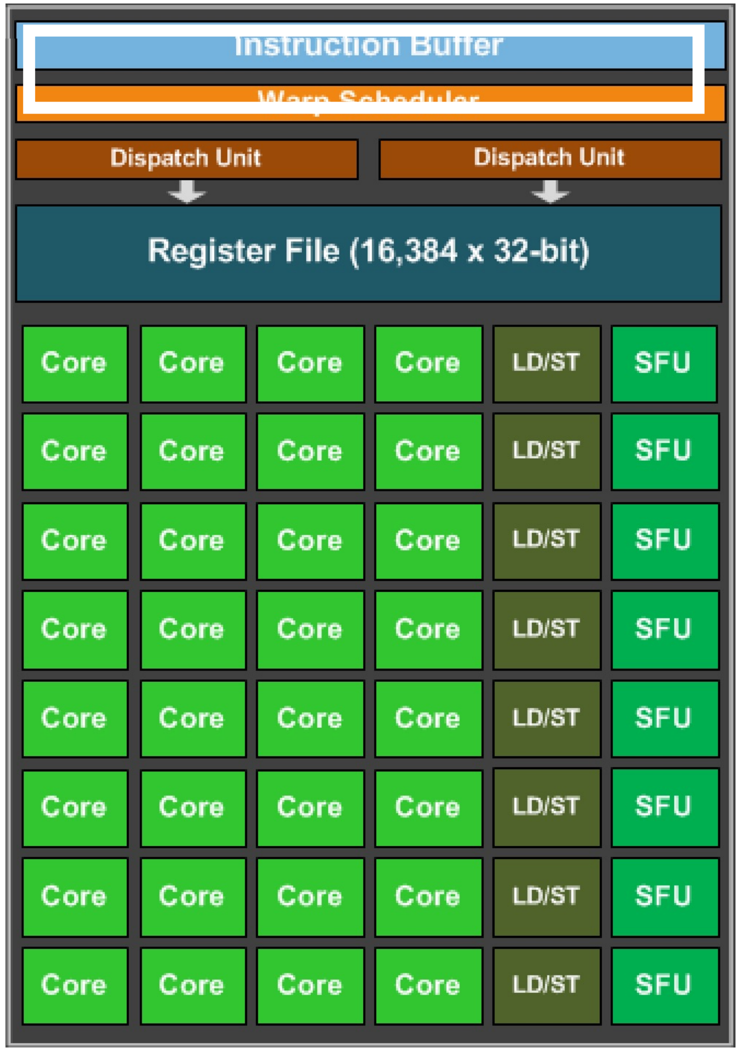
Optimizing memory accesses



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

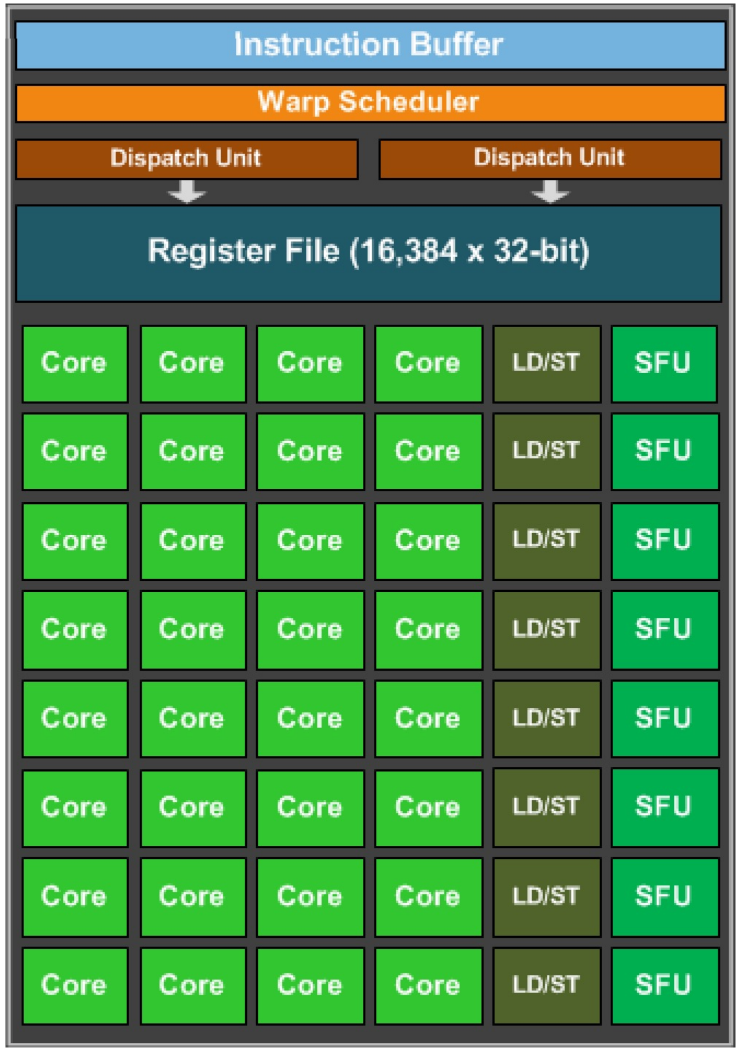


This is the instruction cache... Why doesn't every core have a instruction buffer to keep track of its program?

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

Why doesn't every core have one?

Warp execution



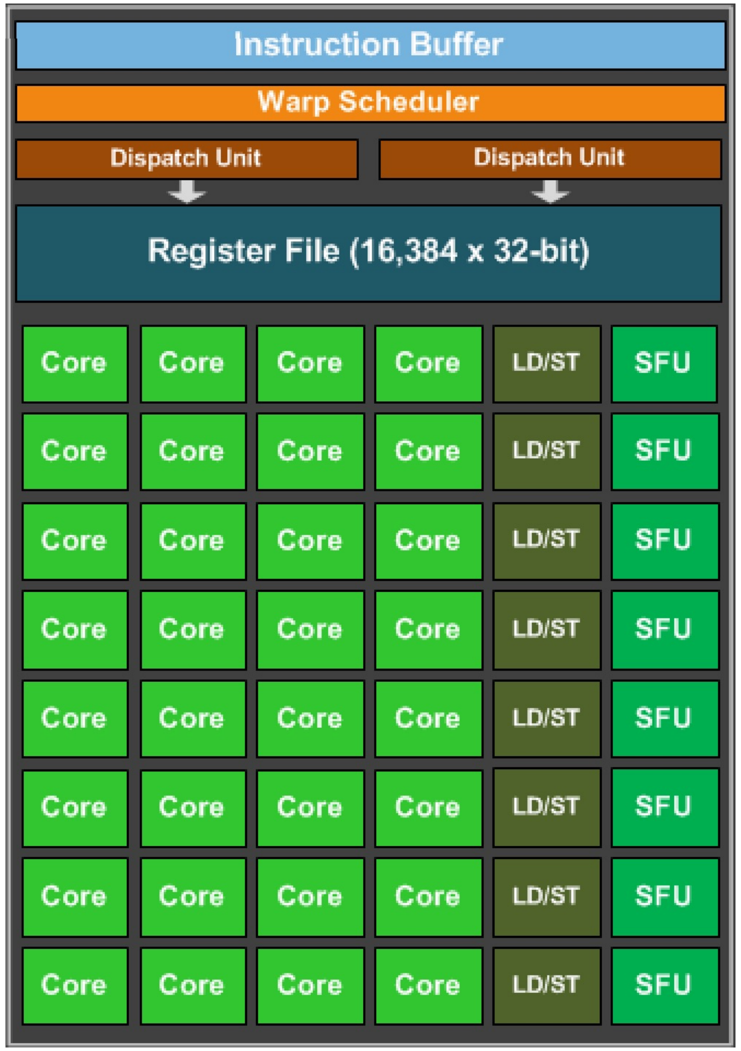
Groups of 32 threads are called a "warp"

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"

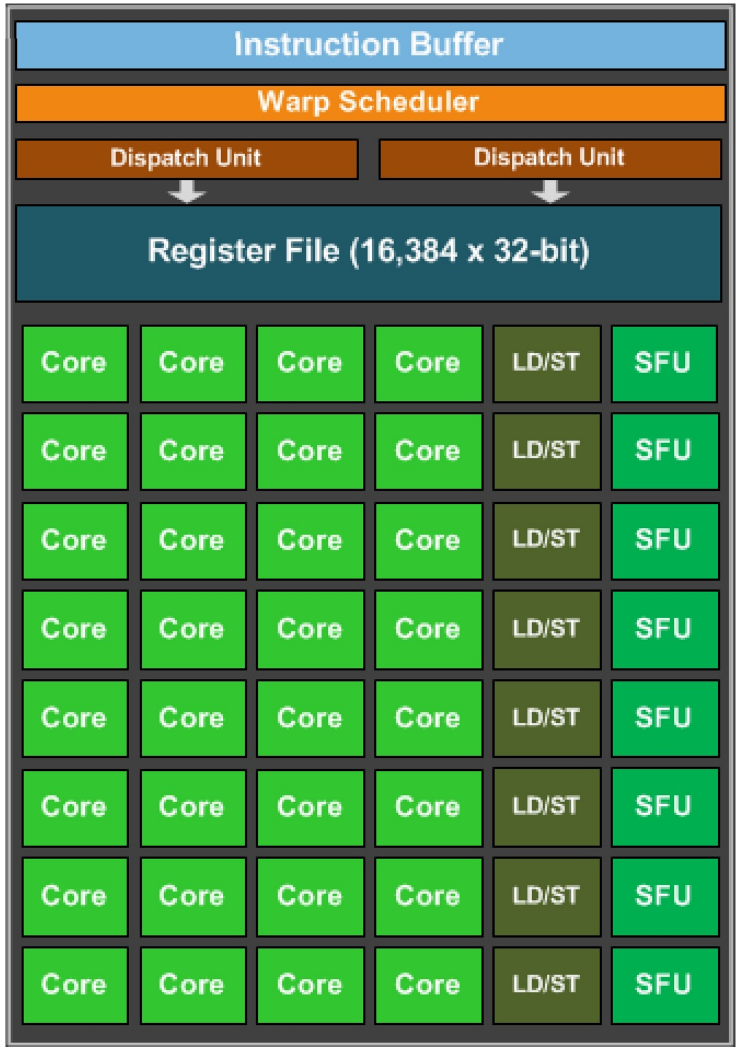
They are executed in lock-step, i.e. they all execute the same instruction at the same time



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

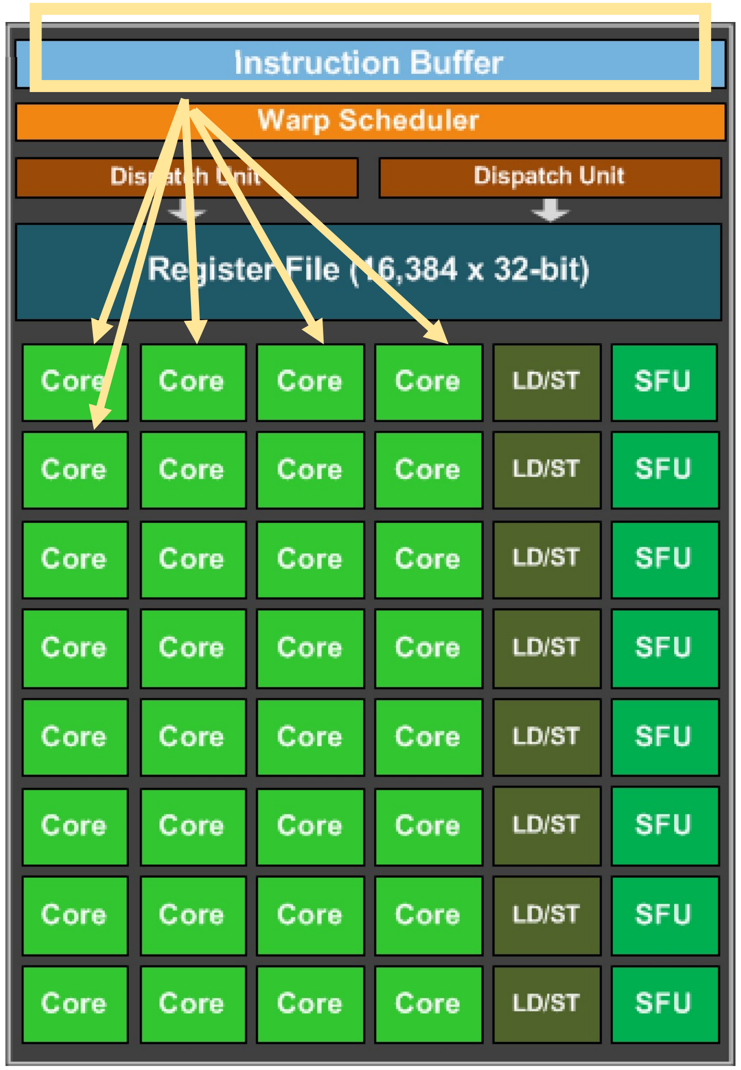
Program:

```
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.

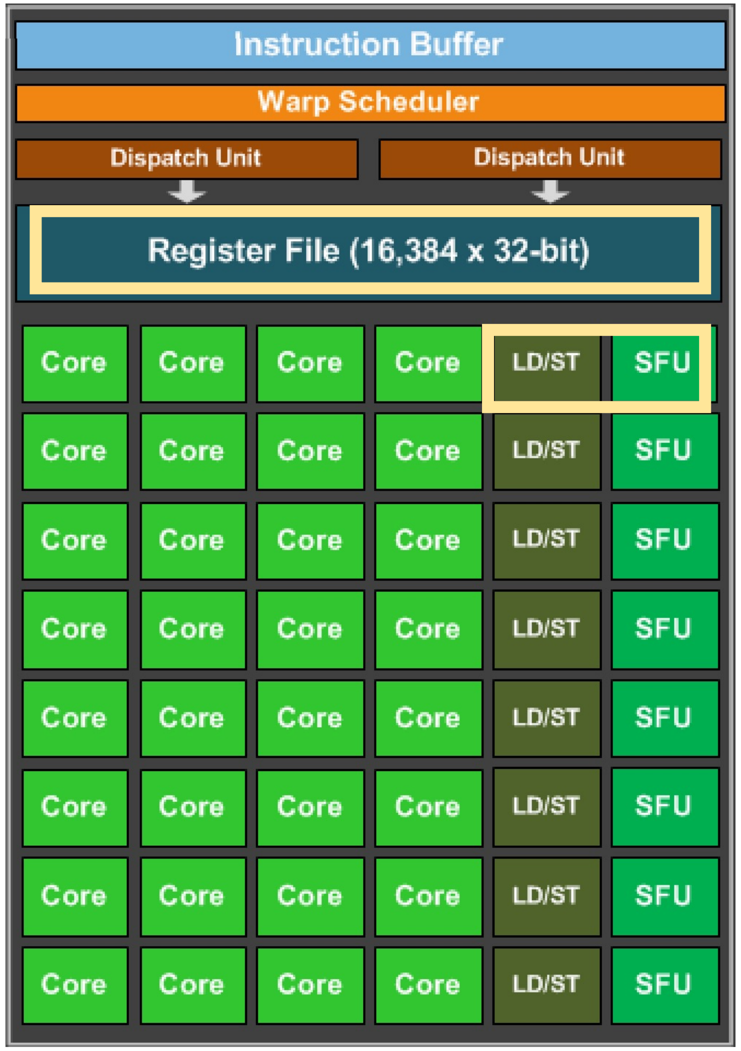
Program:

```
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)

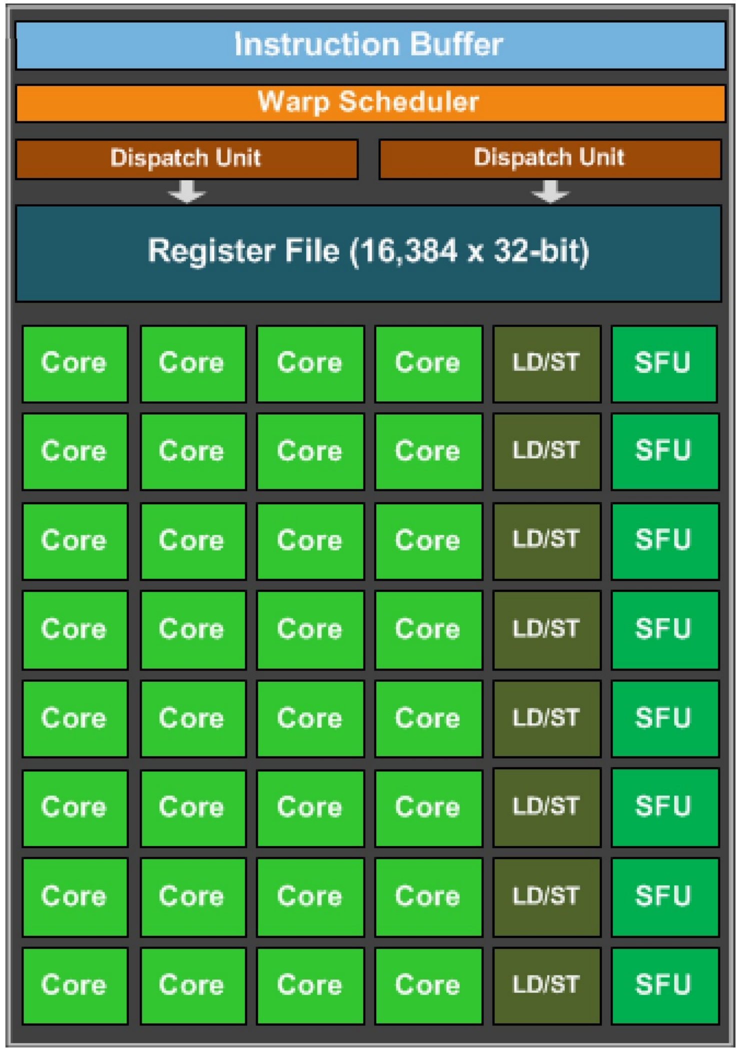
Program:

```
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

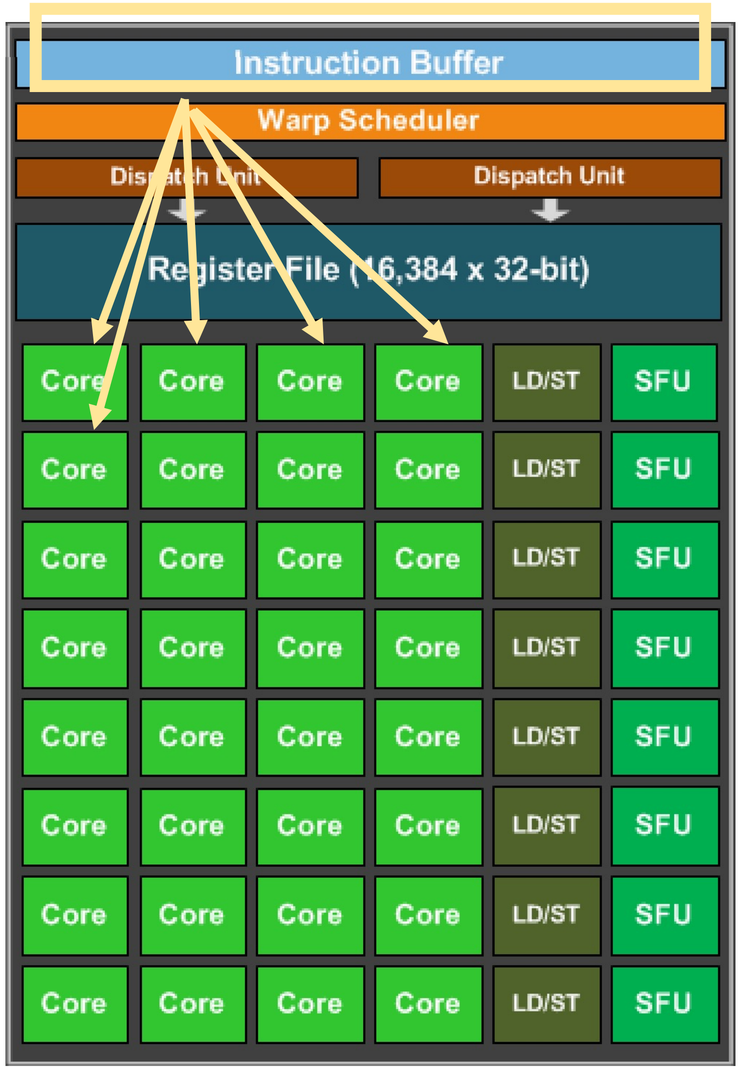
Program:

```
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:

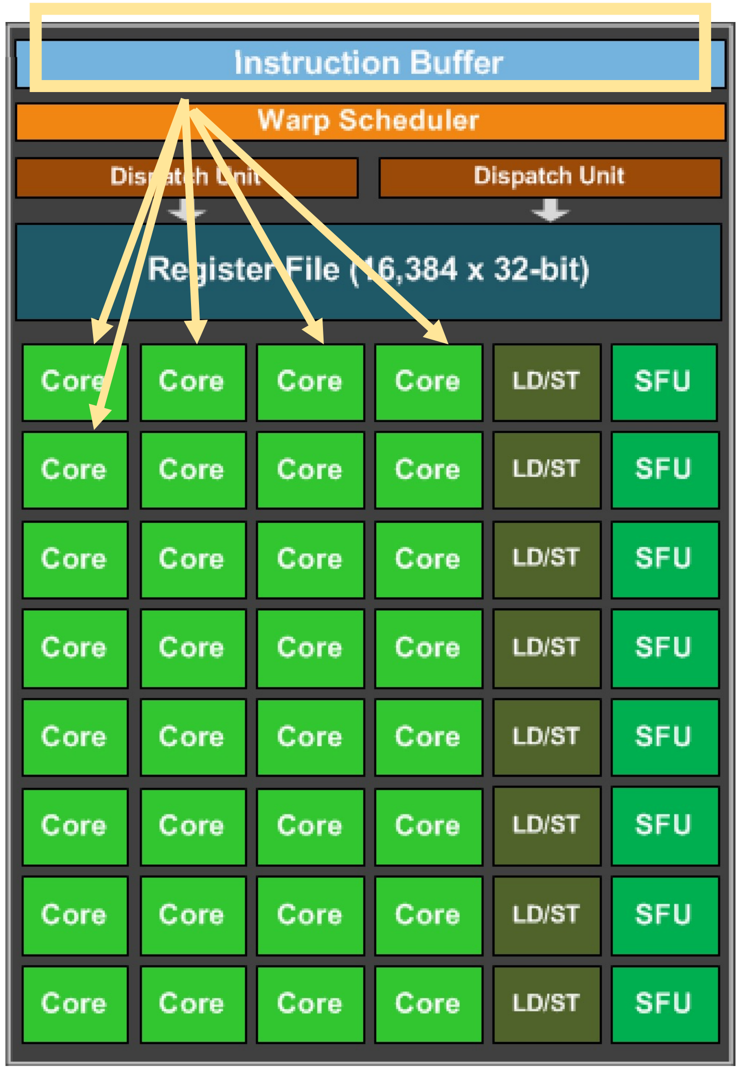
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

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.

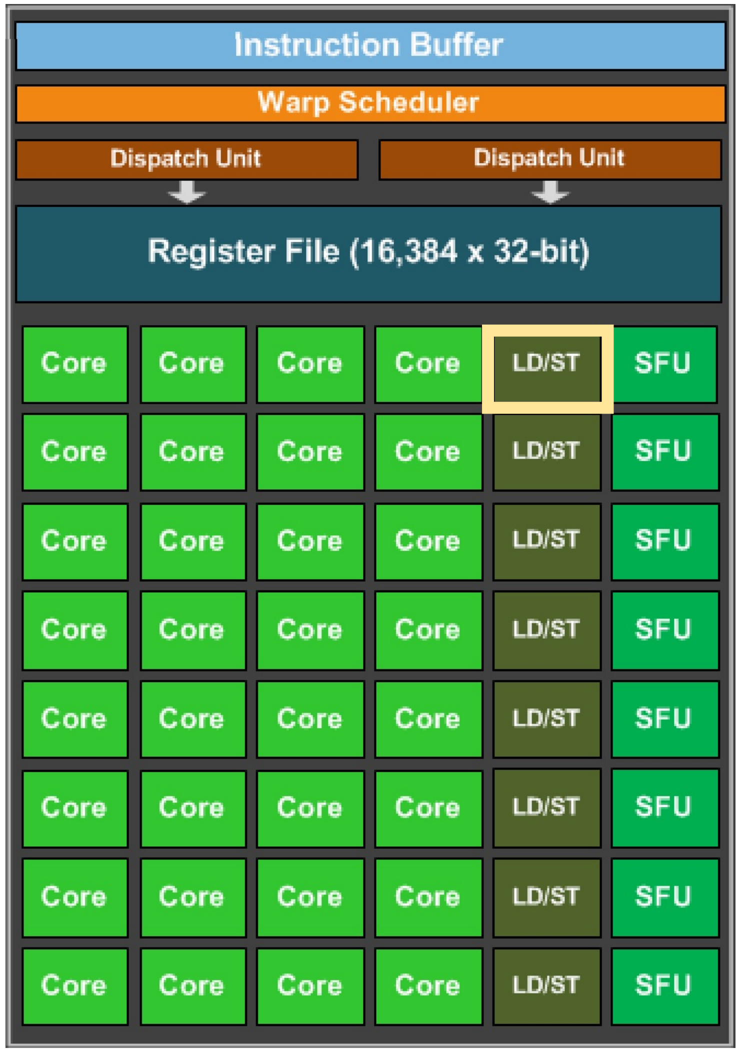
Program:

```
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

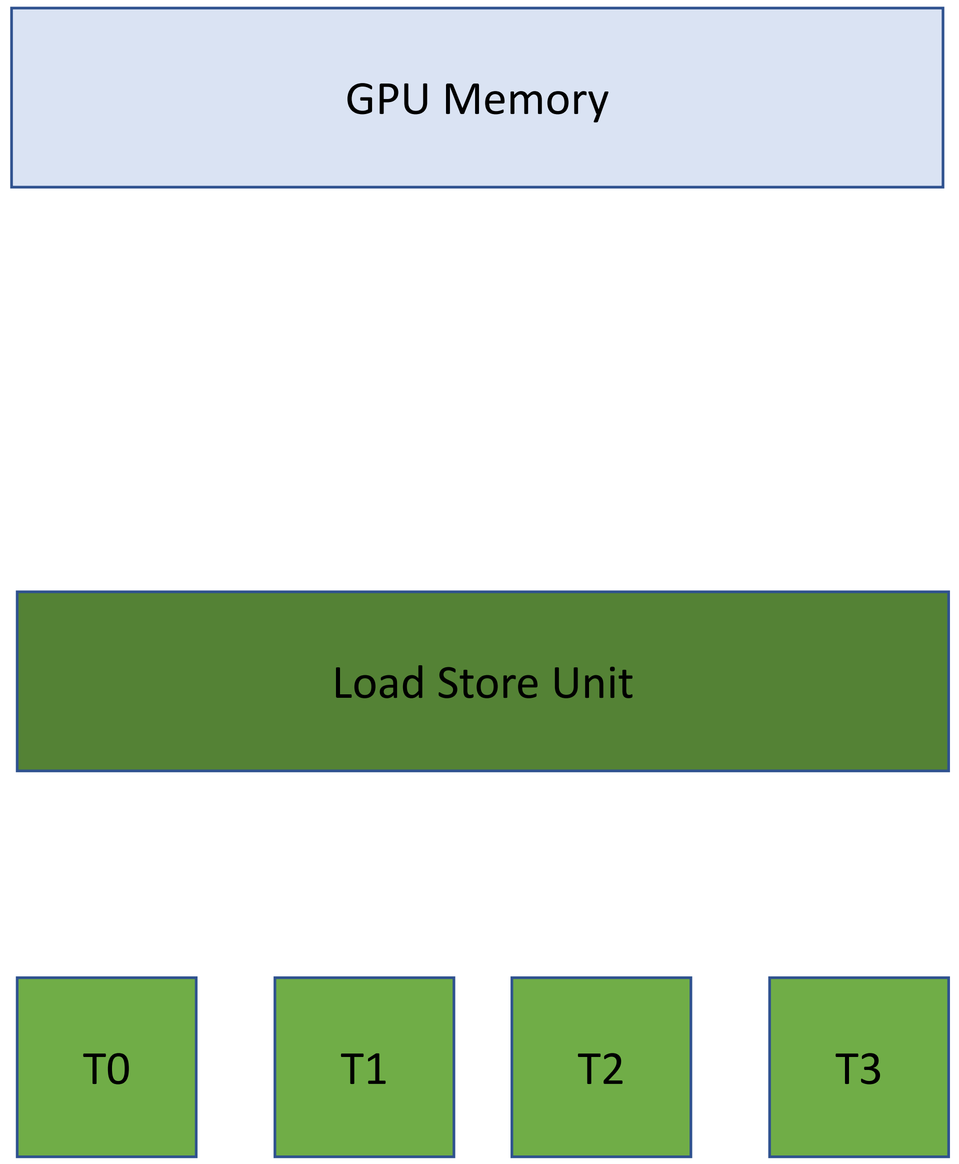


Program:

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

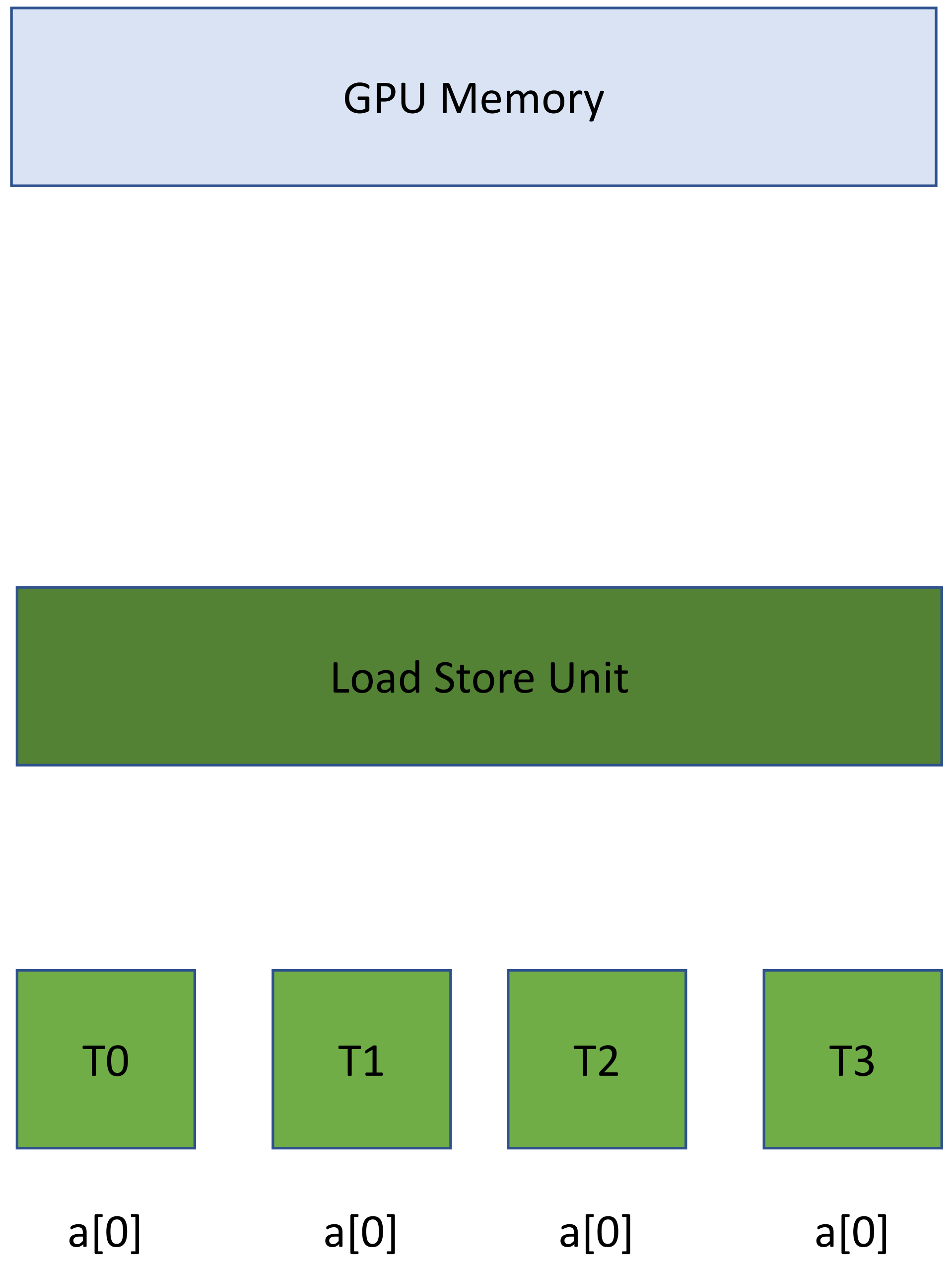
4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen



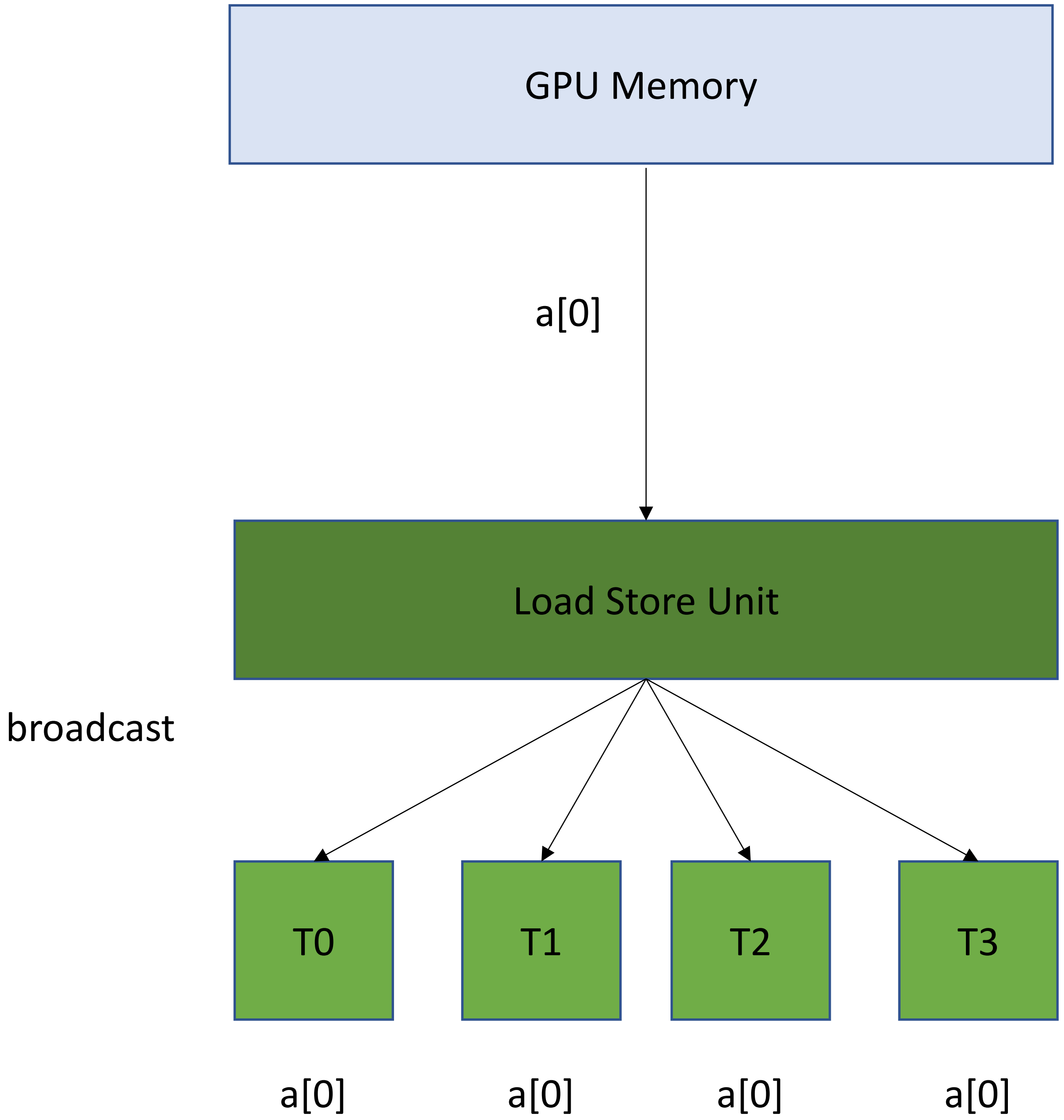
4 cores are accessing memory. What can happen

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.

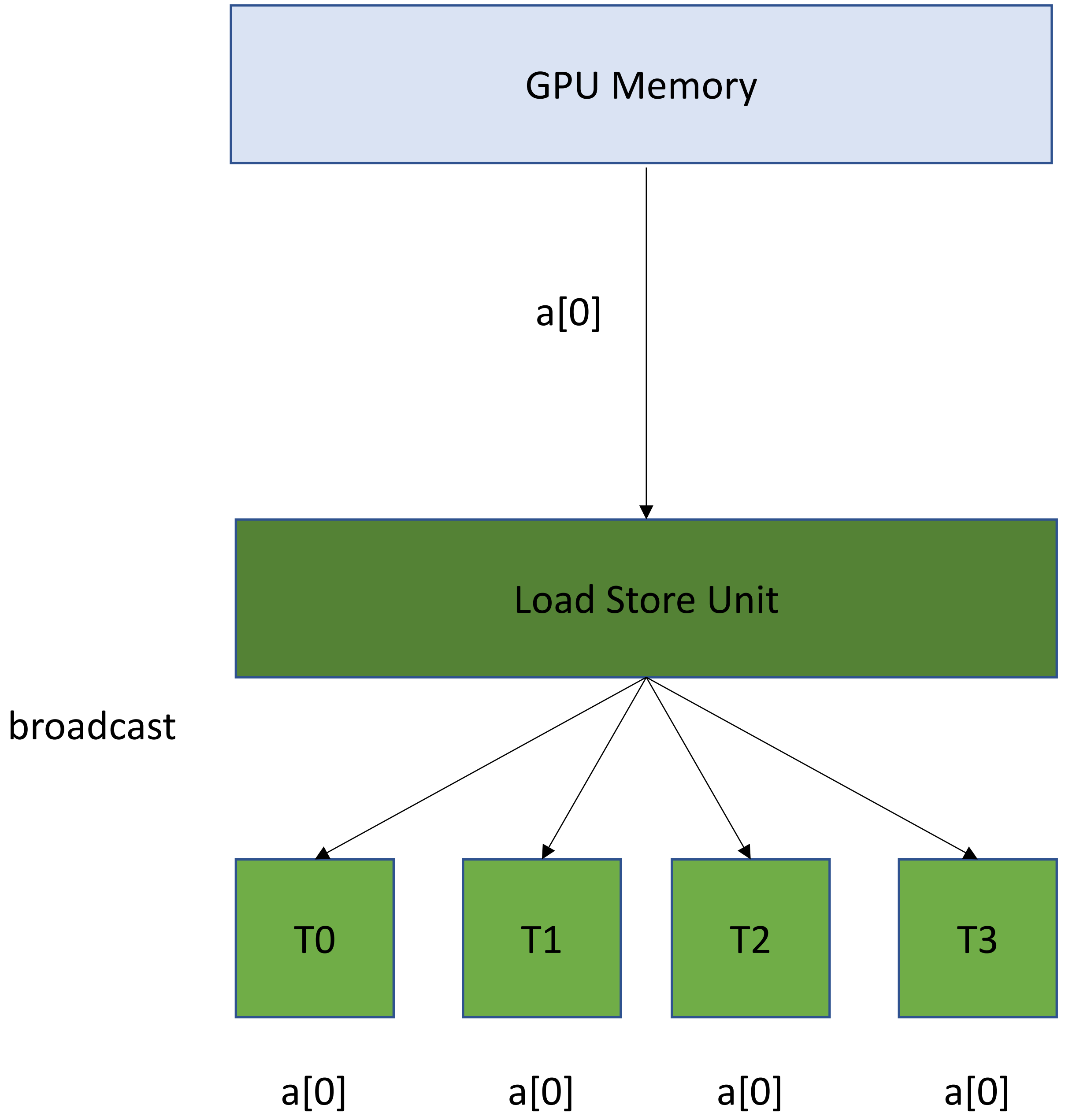


4 cores are accessing memory. What can happen

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

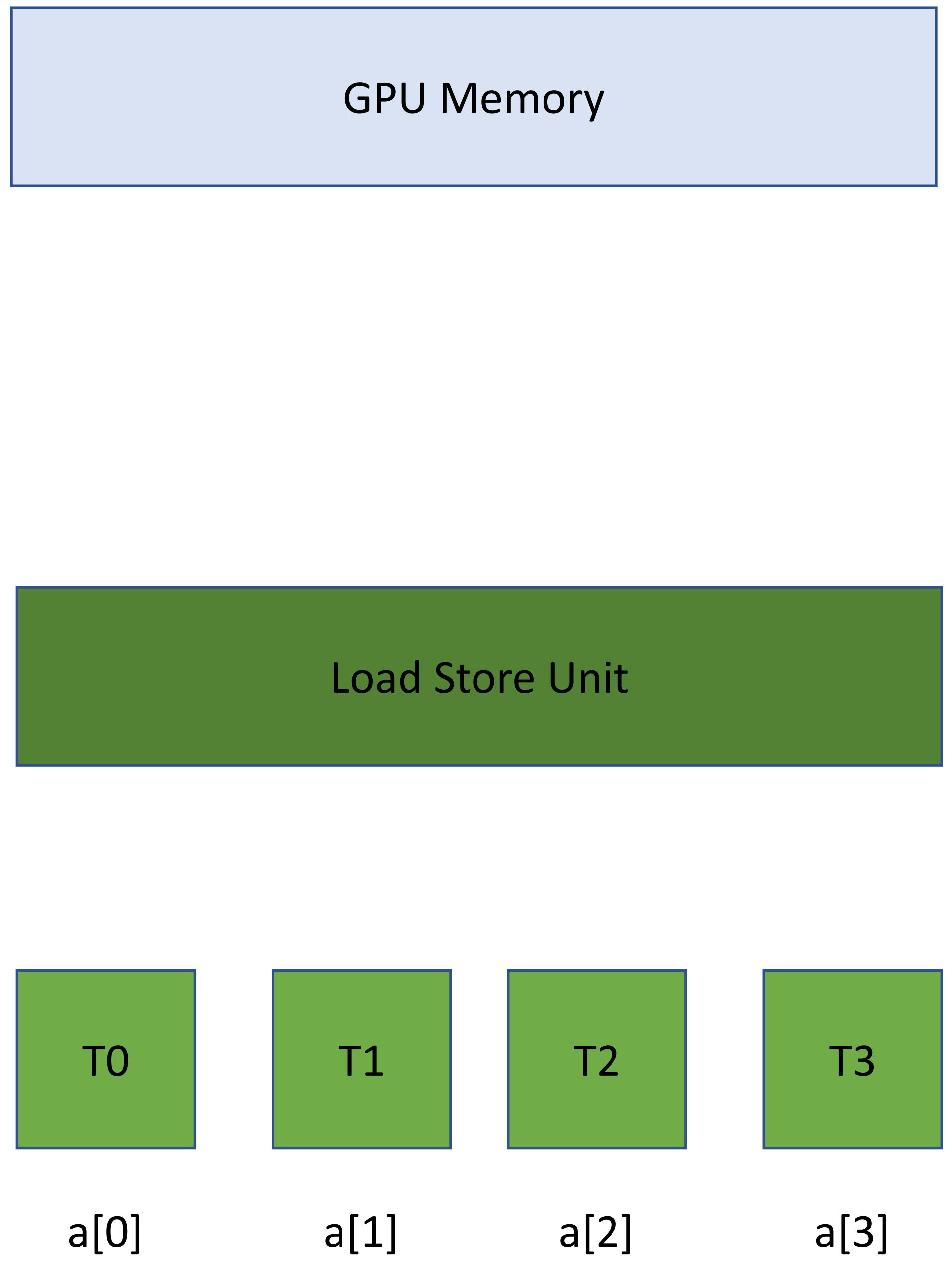
1 request to GPU memory

Efficient, but probably not too common.



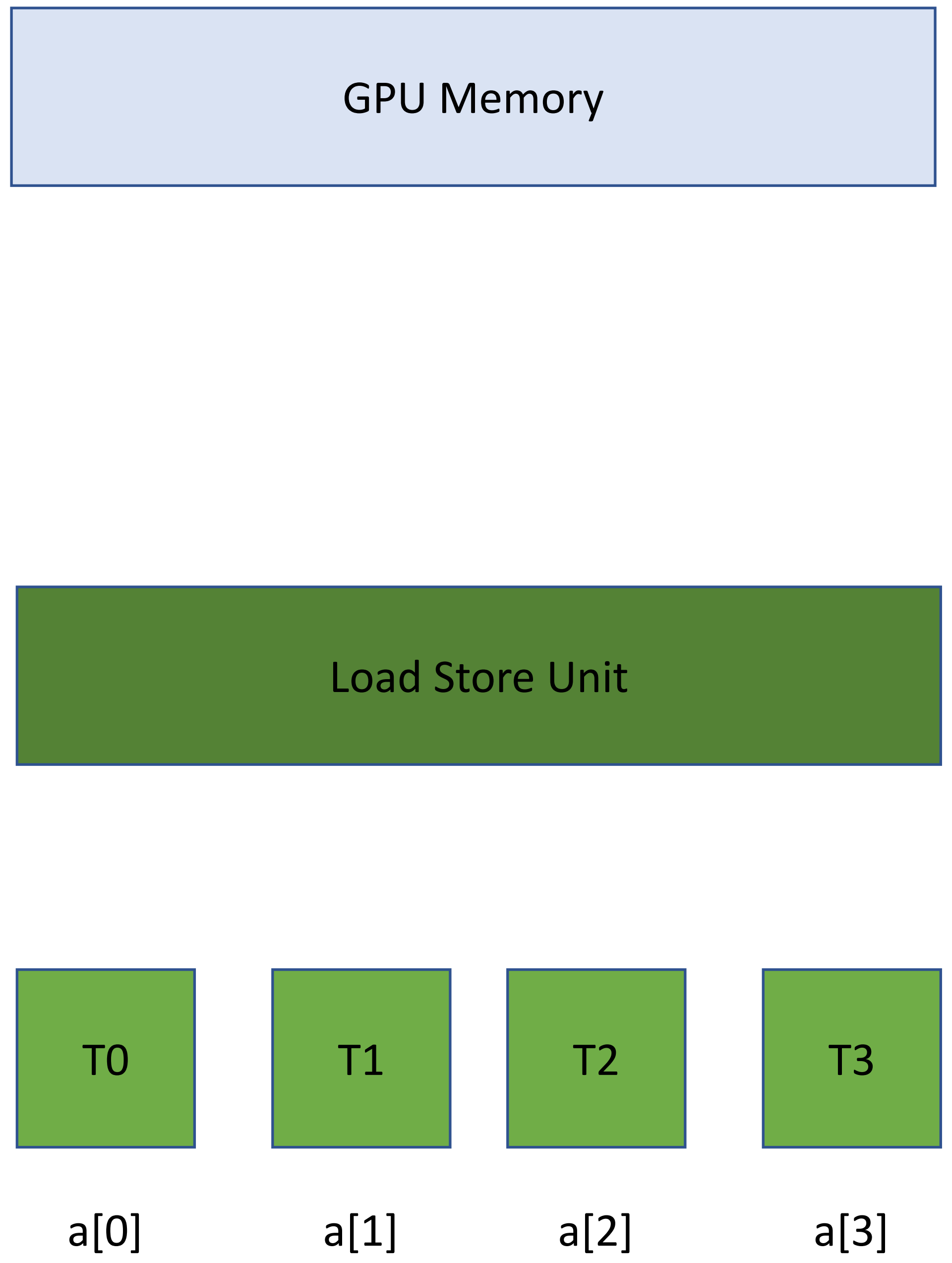
4 cores are accessing memory. What can happen

Read contiguous values



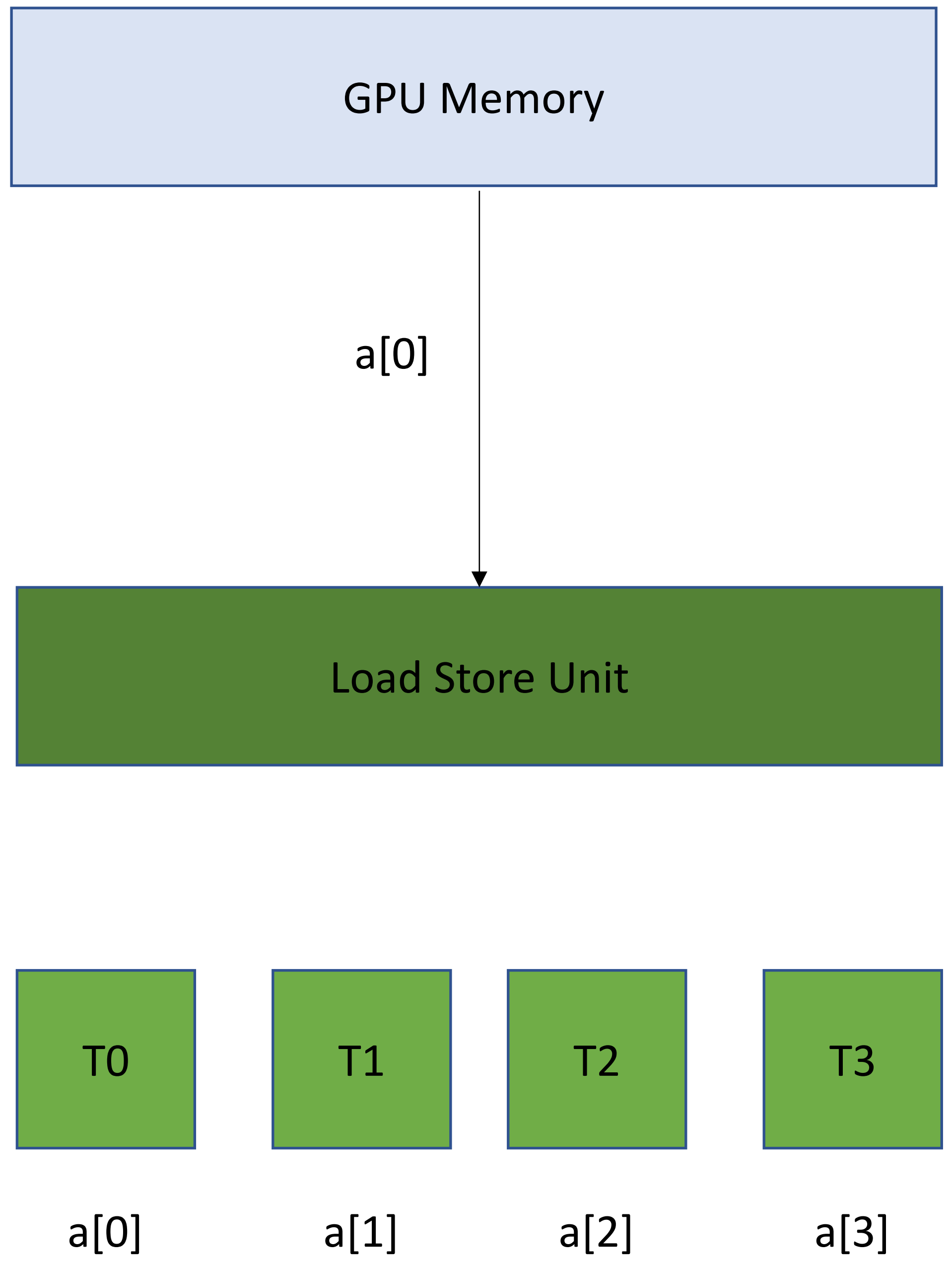
4 cores are accessing memory. What can happen

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

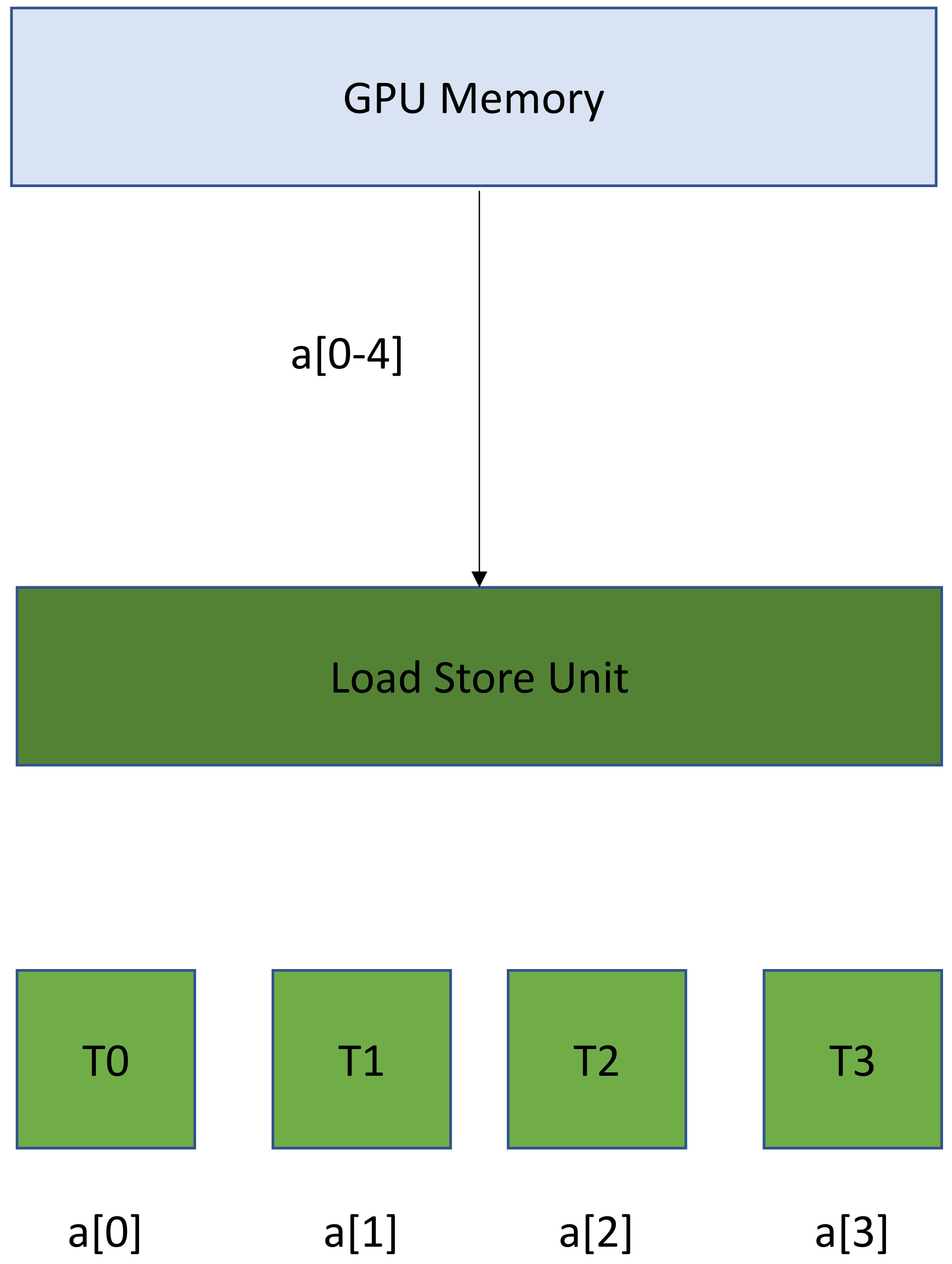
Read contiguous values
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes



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

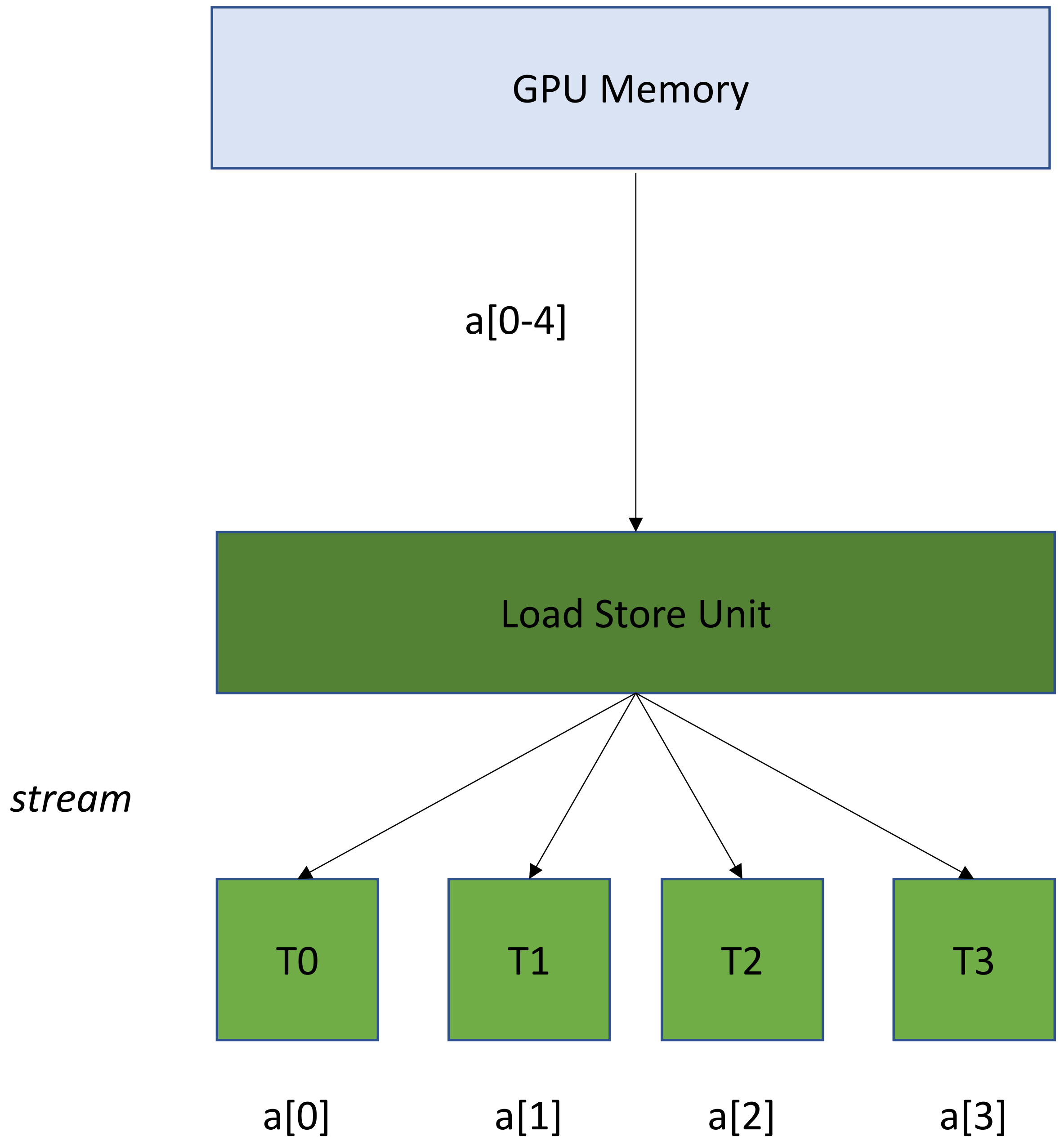


4 cores are accessing memory. What can happen

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

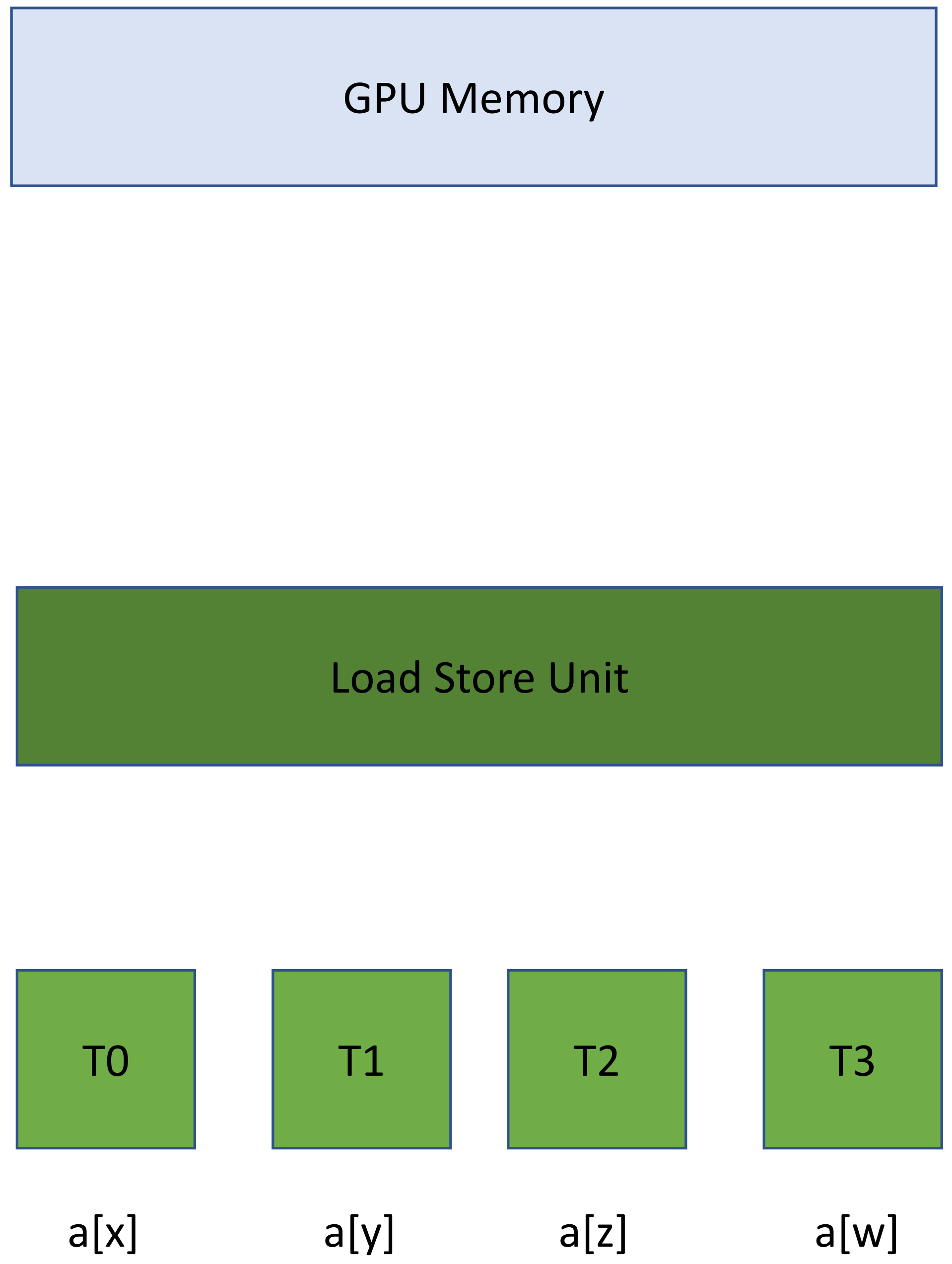


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

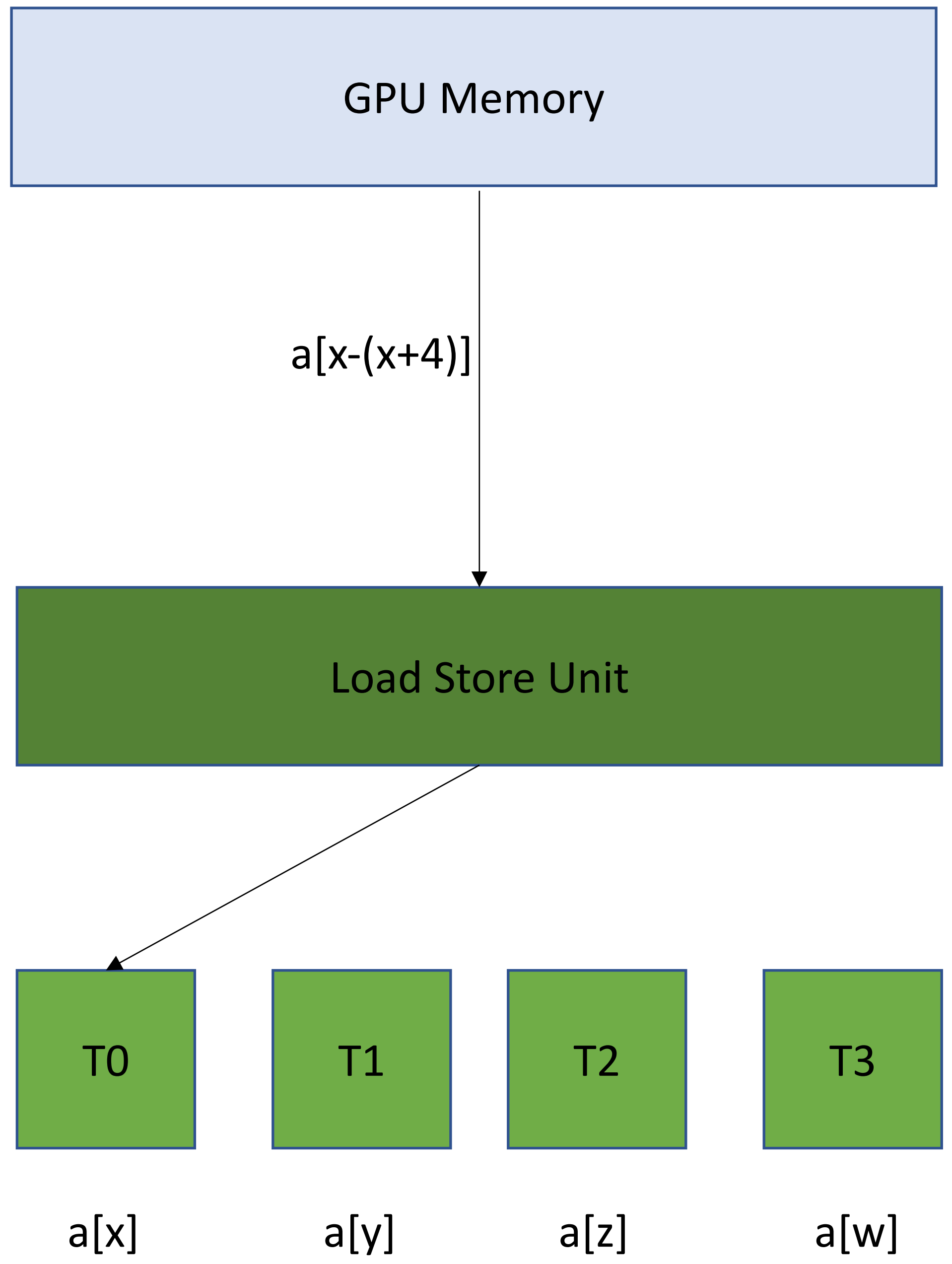


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

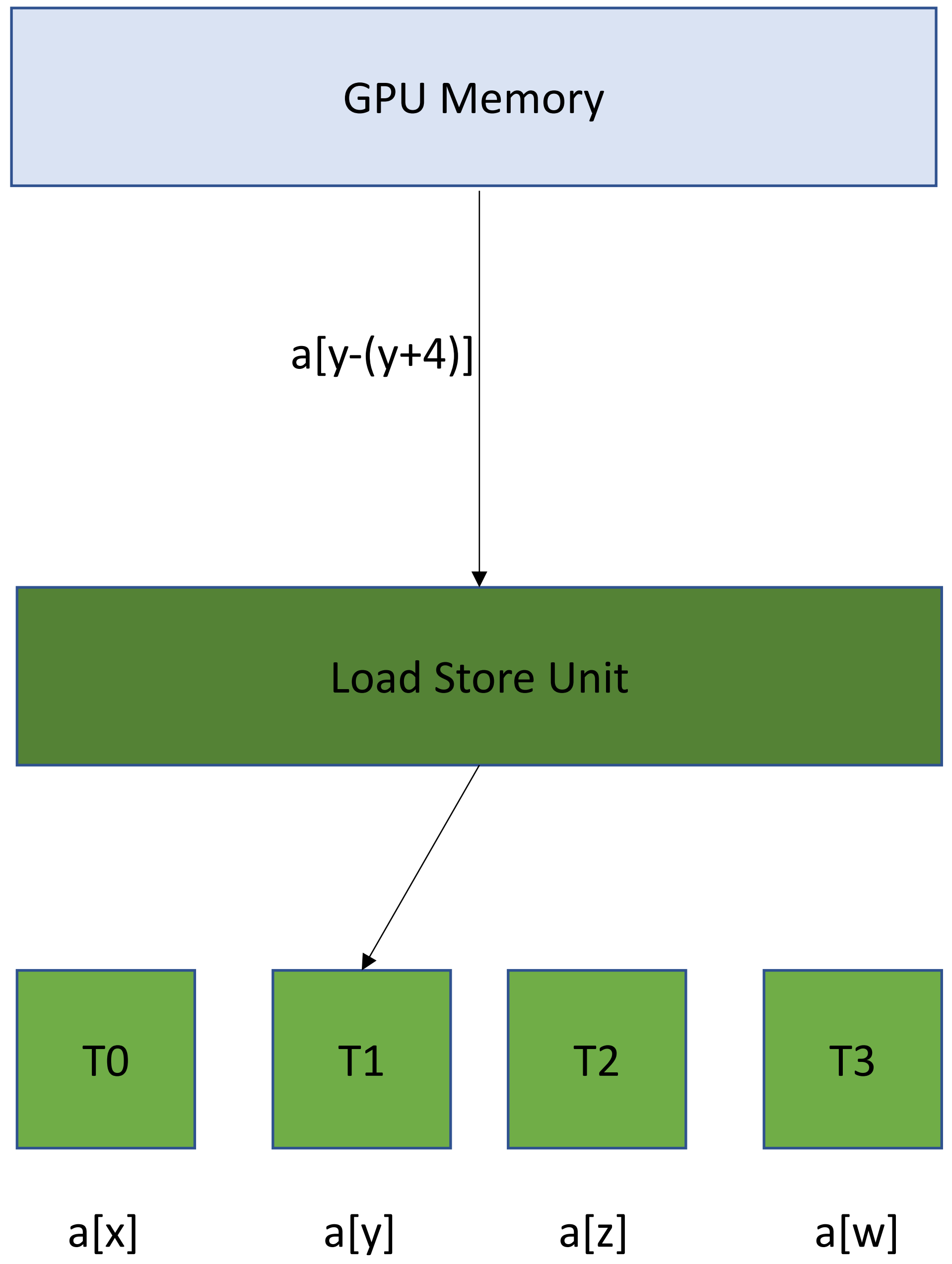


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

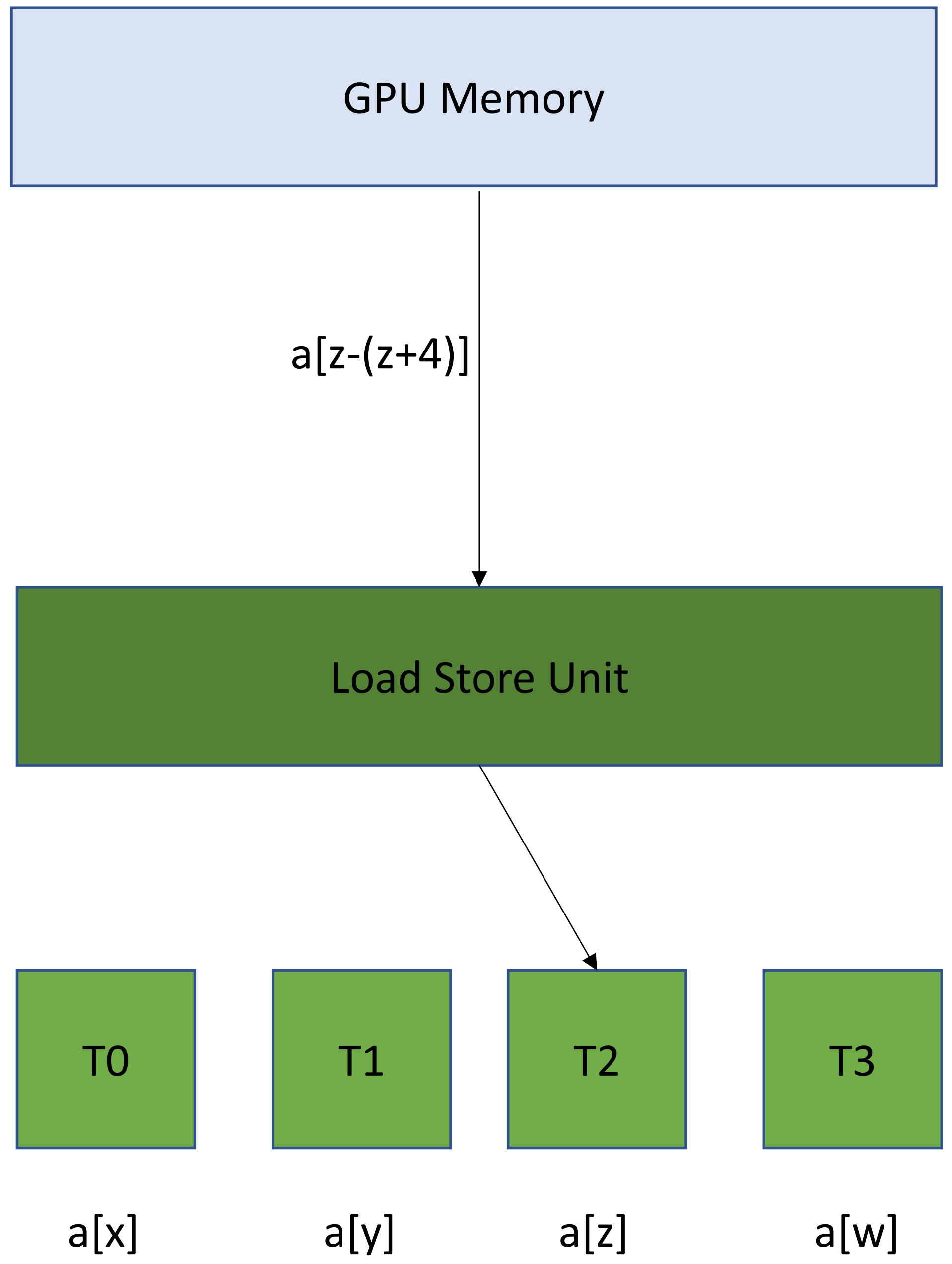


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

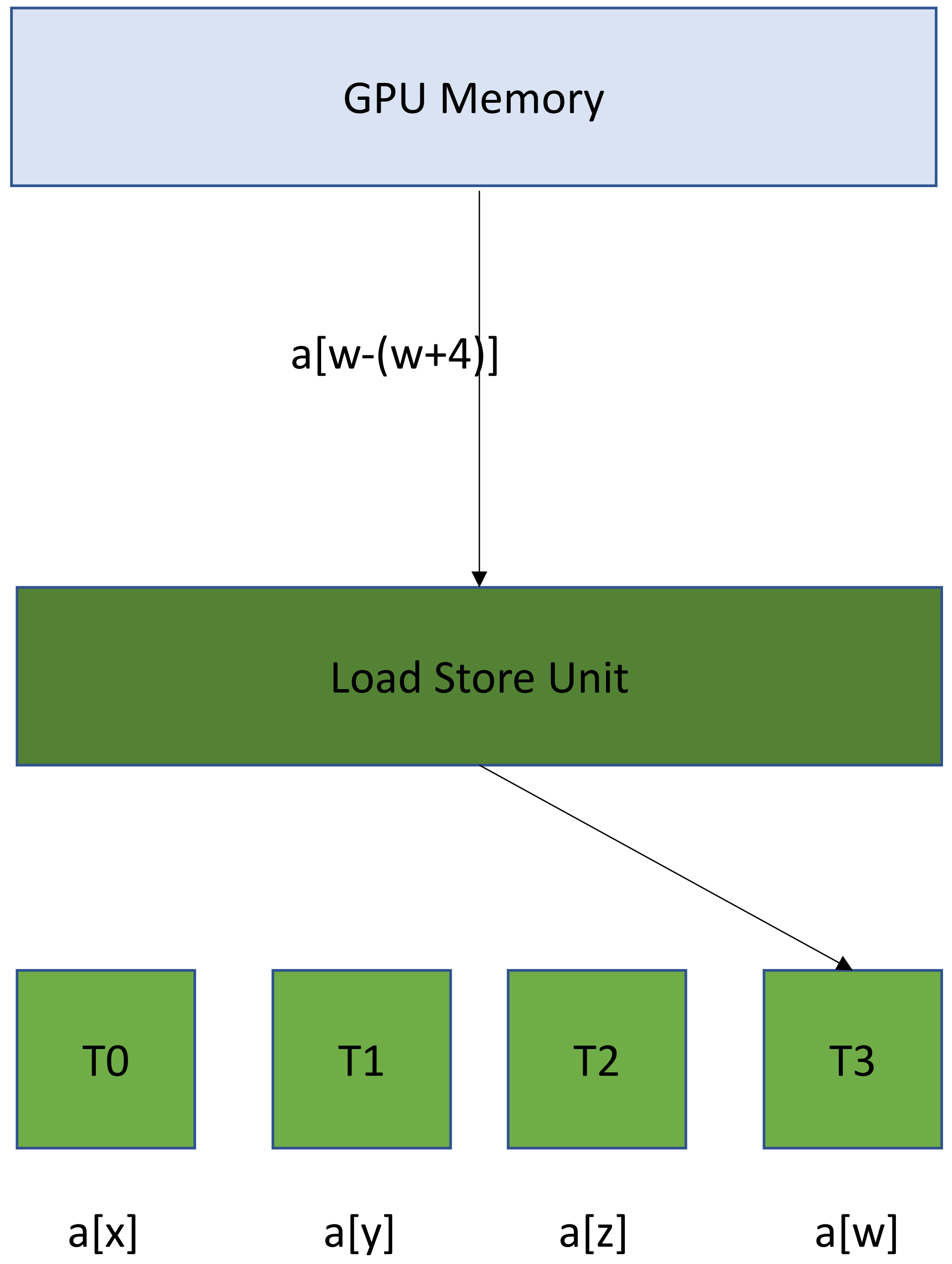


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory



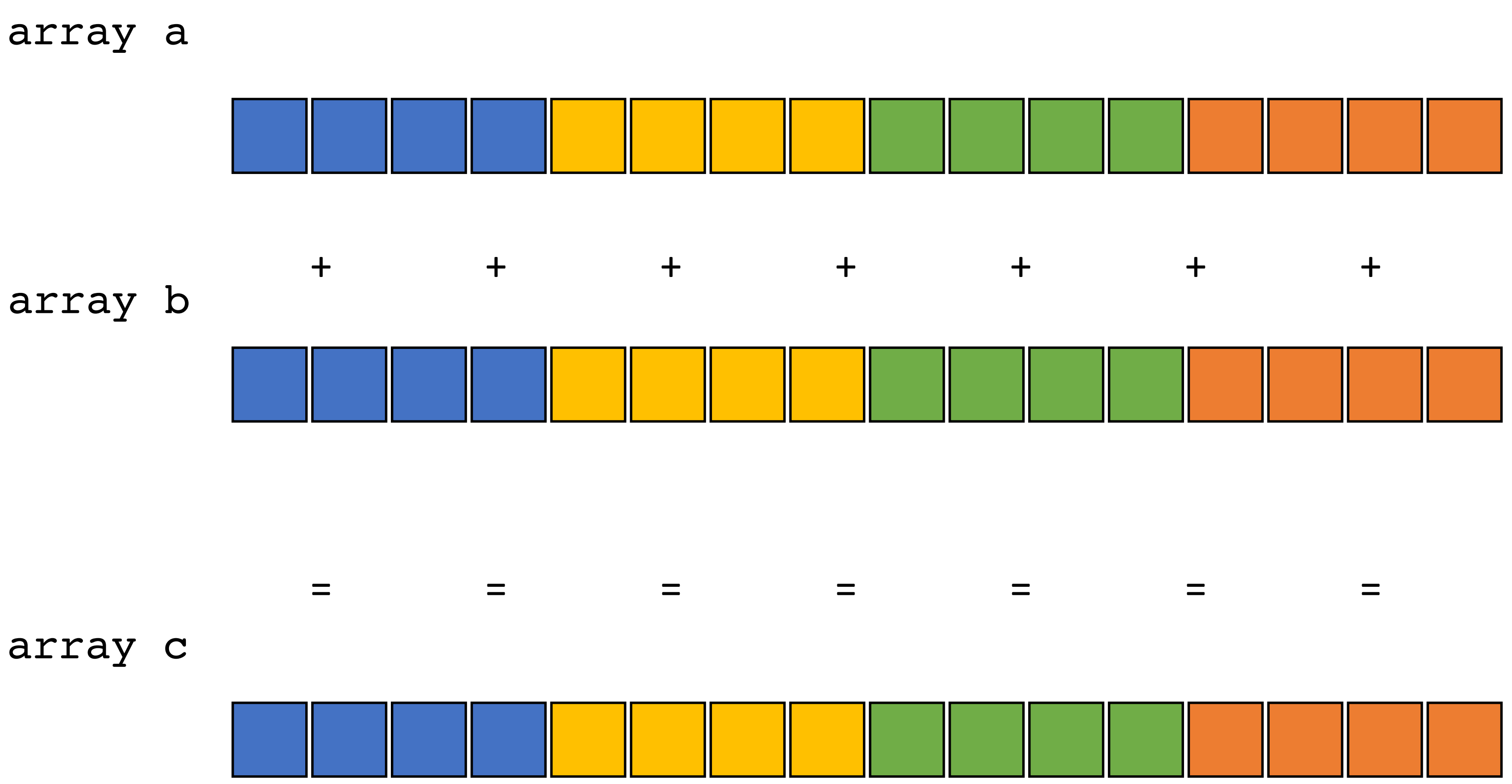
Go back to our program

```
__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

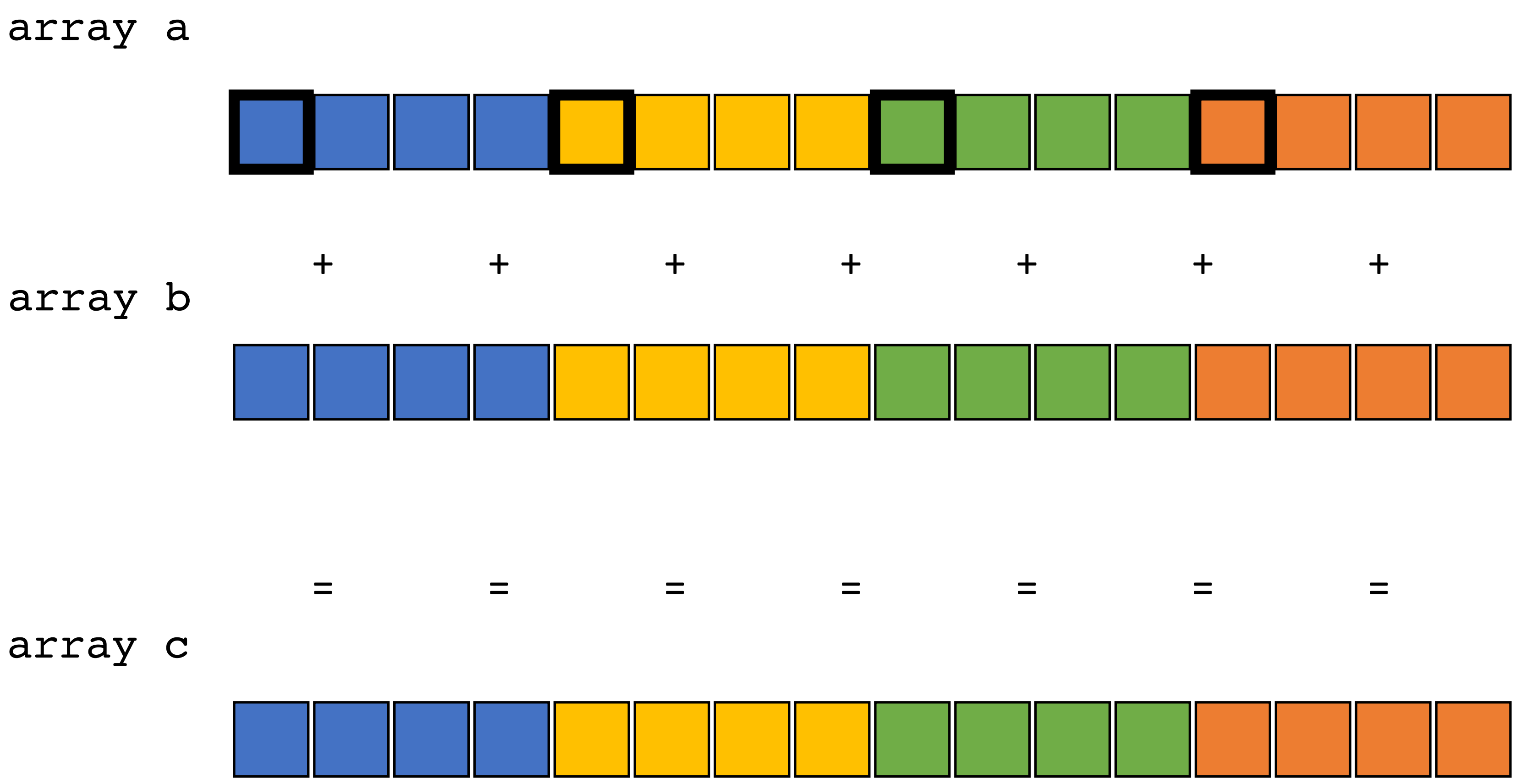


Computation can easily be divided into threads

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 queue. What sort of access is this?

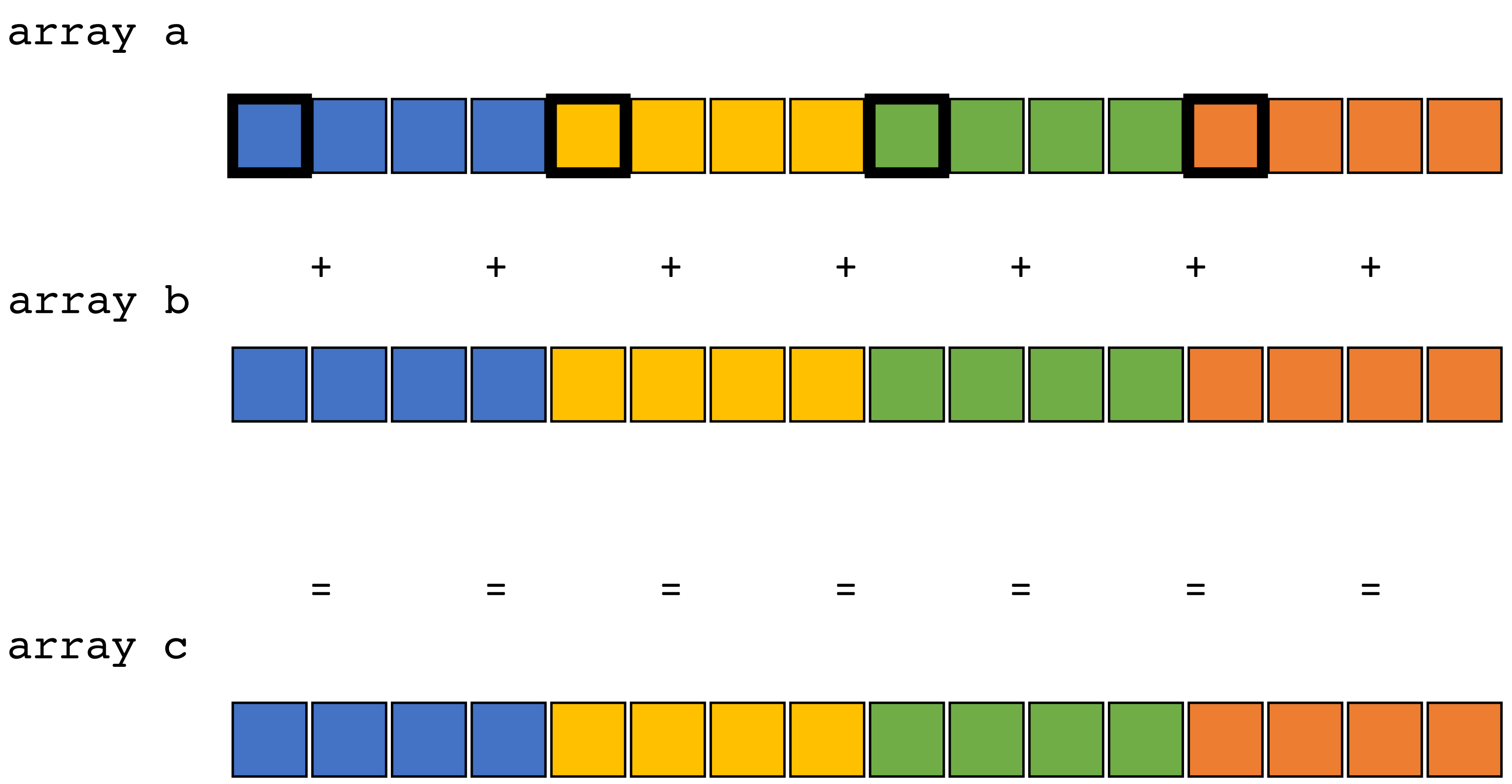


Computation can easily be divided into threads

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 queue. What sort of access is this?



Computation can easily be divided into threads

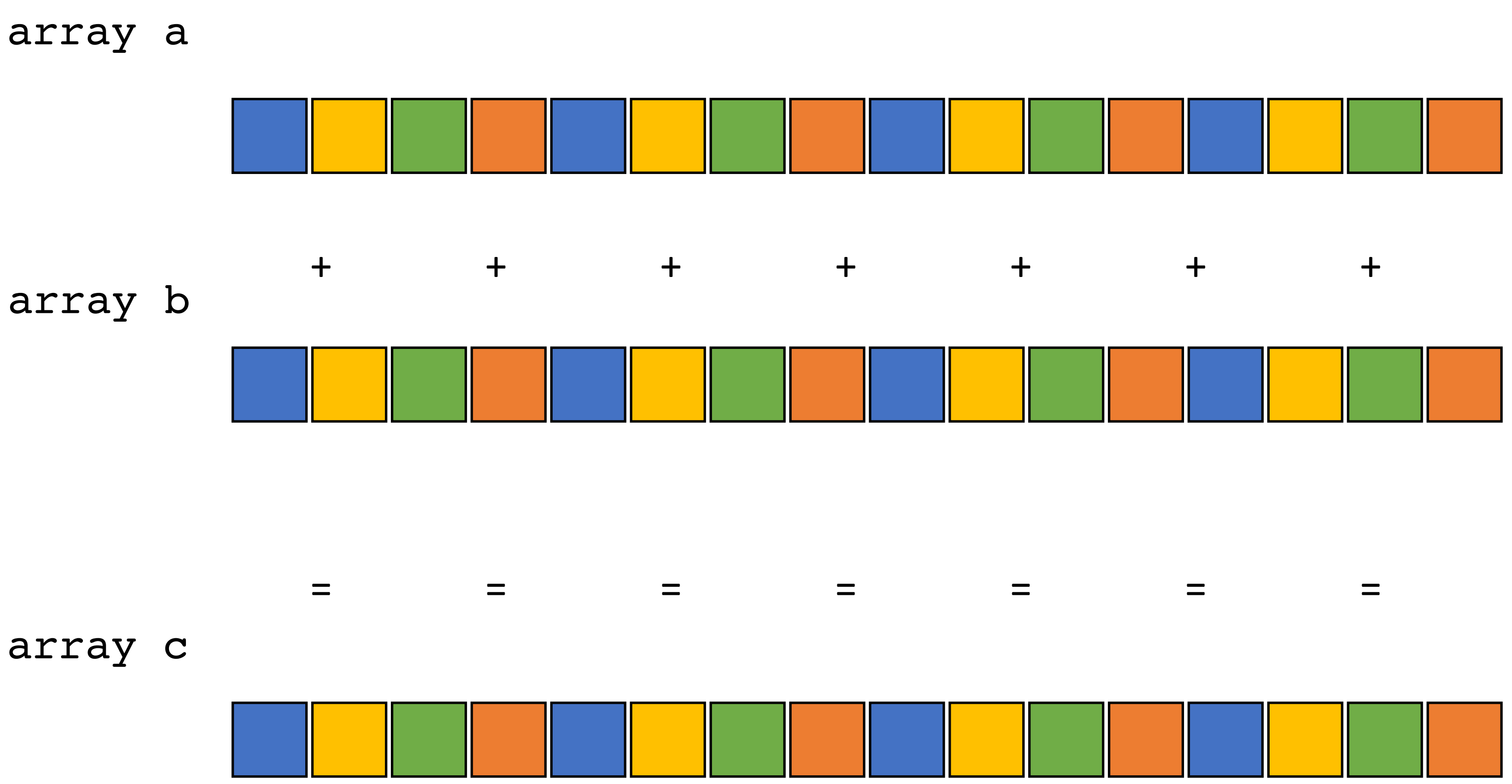
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

How can we fix this

Stride Pattern

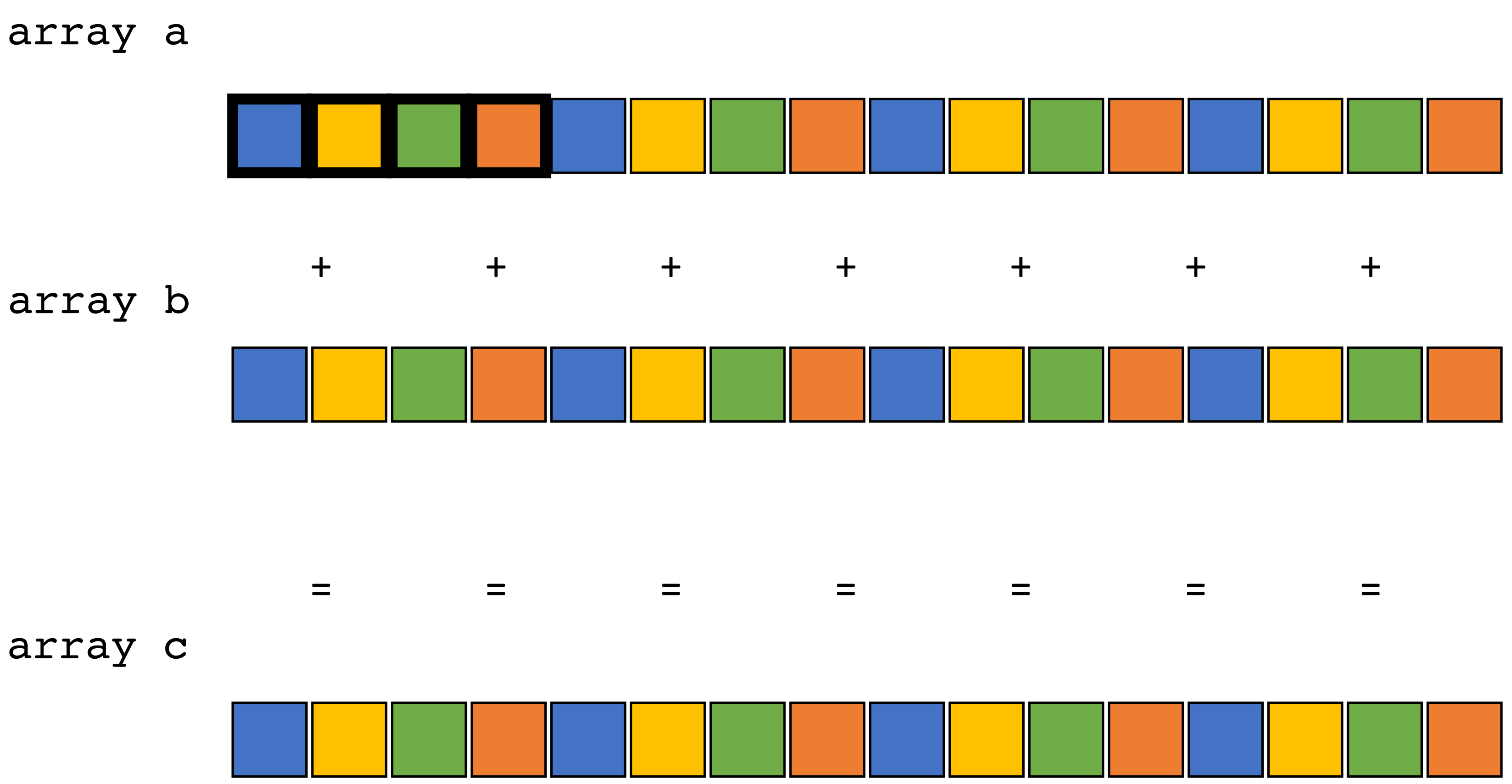
Computation can easily be divided into threads

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?

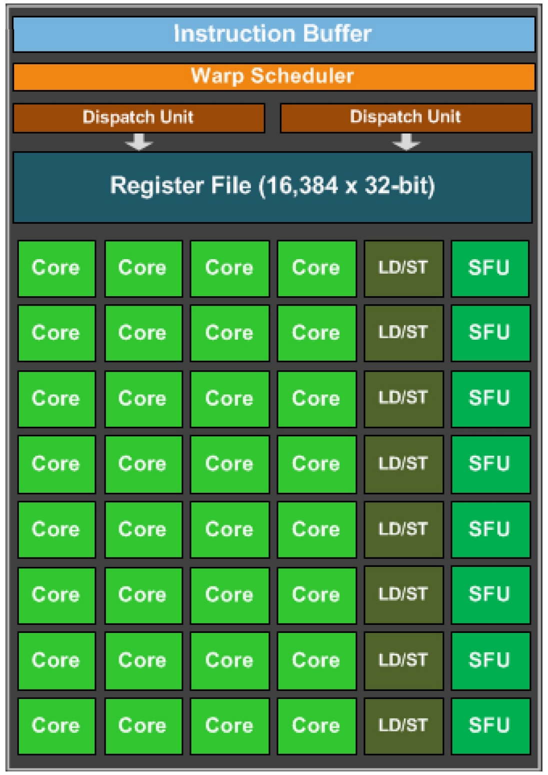
Coalesced memory accesses

Lets try it! What do we think? 😀

What else can we do?

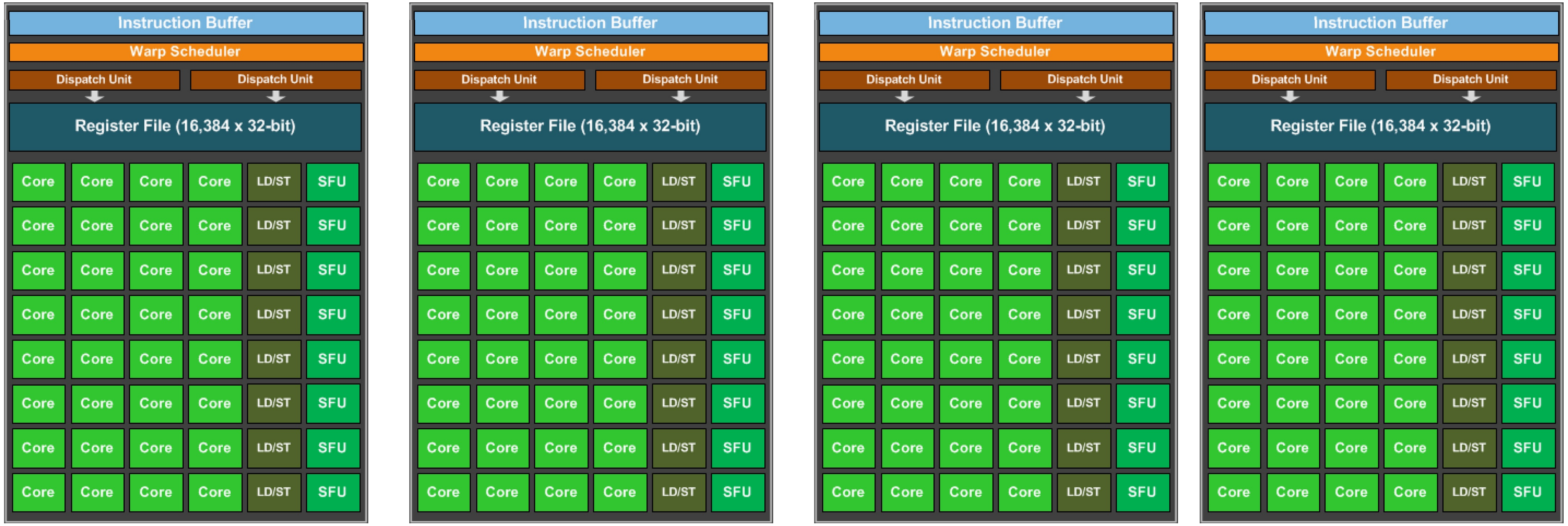
Multiple streaming multiprocessors

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



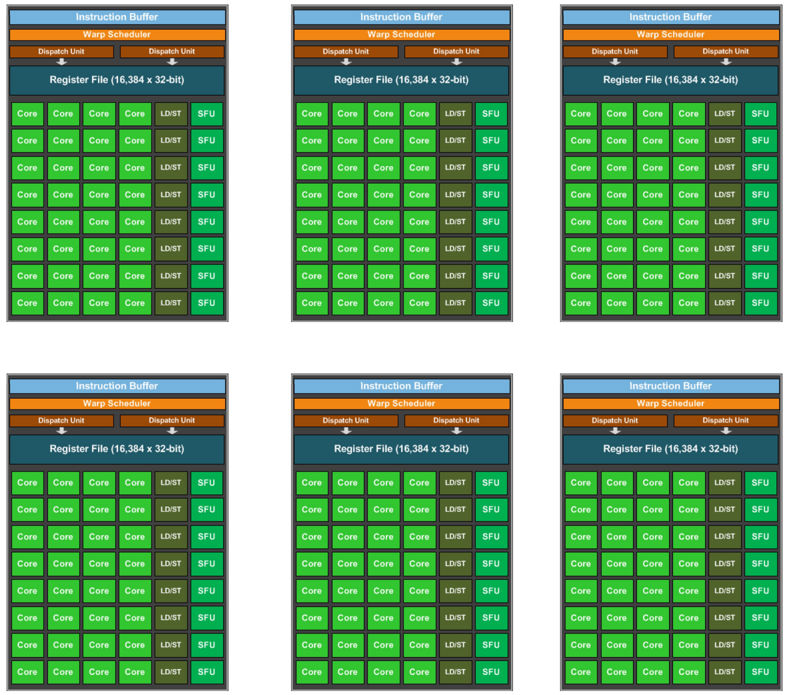
Multiple streaming multiprocessors

We've been talking only about 1 streaming multiprocessor, most GPUs have multiple SMs big ML GPUs have 32. My 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];
}
```

Need to recalculate some thread ids.

calling the function

Launch with many thread blocks

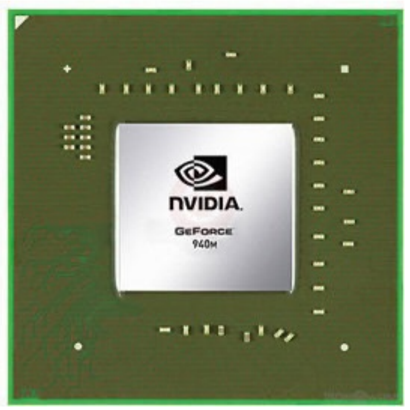
```
vector_add<<<1024,1024>>>(d_a, d_b, d_c, size);
```

Now we have 1 thread for each element

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

Final Round

The GPU in
my PhD laptop



Nvidia 940m
1.8 Billion transistors
75 TDP
Est. $130

Fight!

The CPU in
my professor
workstation

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

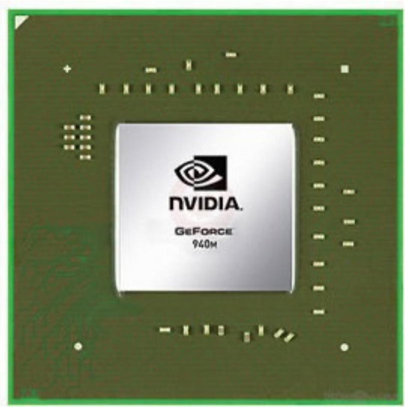
https://www.techpowerup.com/gpu-specs/geforce-940m.c2648
https://www.alibaba.com/product-detail/Intel-Core-i7-9700K-8-Cores_62512430487.html
https://www.prolast.com/prolast-elevated-boxing-rings-22-x-22/

Final Round

Nearly 4x faster!!

Fight!

The GPU in
my PhD laptop



Nvidia 940m
1.8 Billion transistors
75 TDP
Est. $130

The CPU in
my professor
workstation

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/

Next week

- GPU programming #2

- Get started on HW!