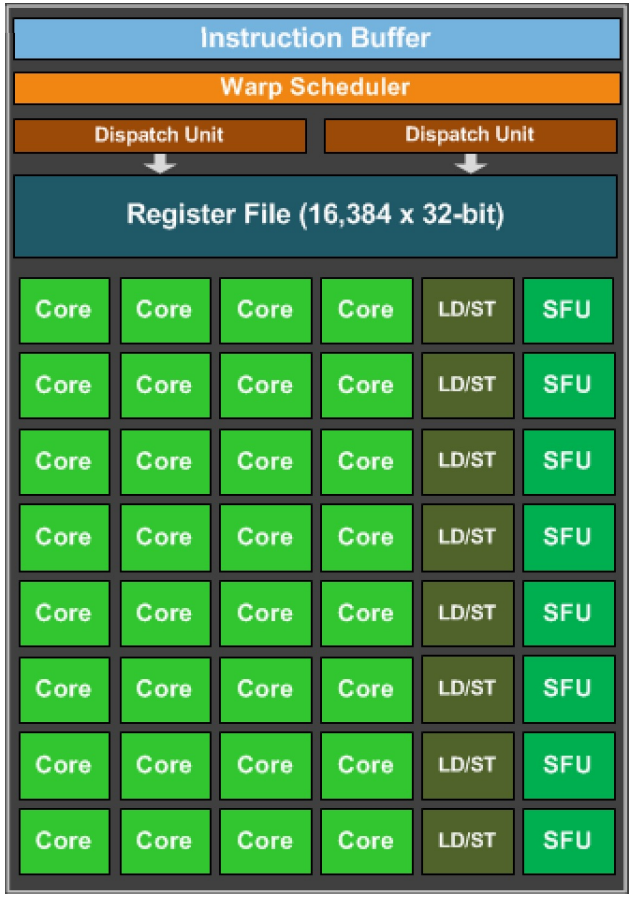
CSE113: Parallel Programming

March 7, 2022

- **Topics**:
  - Finish discussing HW 5
  - GPU programming



Announcements

- HW 4 was due on Friday

- HW 5 was released on Friday (technically Saturday AM... apologies!)
  - Please get started on it ASAP so that we can sort out technical issues sooner rather than later
  - Designed to be lighter than the previous homeworks.
  - Due by midnight the day before the final (March 16)

- HW 3 grades are released
  - Let us know ASAP if there are issues
  - If you are missing grades that should be there, definitely let us know!

Announcements

- Final is on March 17
  - I will release it by 8 AM, and you will have until midnight to turn it in
  - If you want to allocate time for it, our official final time is 4 PM to 7 PM
  - Same rules at the midterm:
    - Do not discuss with class mates
    - Do not google specific answers or ask questions on forums
    - You can use your notes, the slides, and the internet to google for general concepts.
  - worth 30% of your grade.

Announcements

- SETs are out!
  - Please fill them out; I know they are a pain and we're all busy
  - But it has an outsized effect on classes like this one
    - New class
    - New content
    - New professor
  - I would love to help

Quizes

- We will cancel quizzes for the rest of the quarter;
  - It’s a busy time for everyone and I want to make sure we can support you in HW 5 as much as possible.
  - If you think of good quiz questions let me know!

Review

Homework 5

Homework 5- requirements

- The browser
  - Google Chrome Canary
  - (if you have linux, Google Chrome Dev should work)

- Why do we need the Canary?
  - WebGPU is new and support is inconsistent on main (Although it is officially supported)

  - Perhaps more interesting is the shared array buffer.

  - Make sure you navigate to `http://localhost:8000`

Homework 5 - requirements

Node.js and local webservers

- Permission issues
  - you can try running with sudo (generally considered bad)
  - a stack overflow thread with installation options

Javascript

- logging
- variables
- objects
- shared array buffers

What does the solution look like?

- Demo

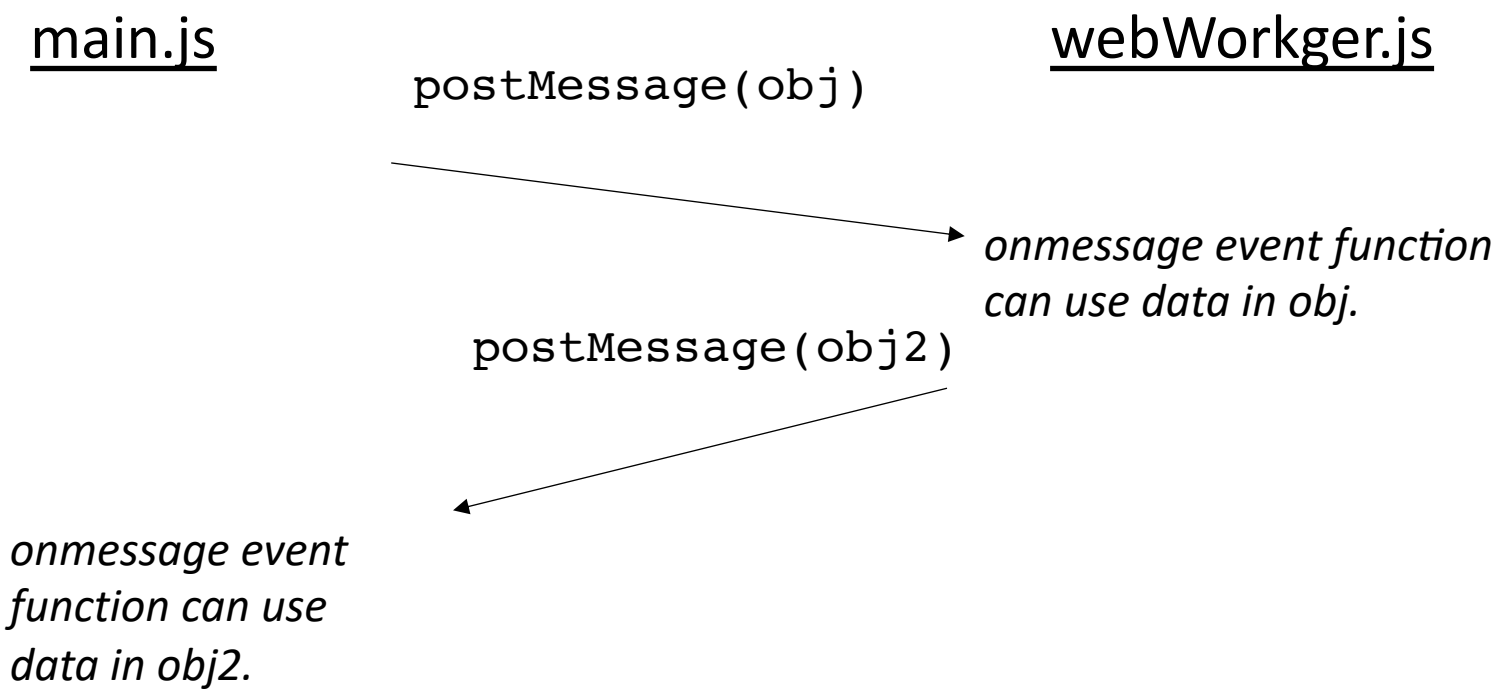
New material

Web Workers demo

Overview: Web Workers

- Create a new worker with a new .js file (this is done for you)
  - Nothing happens on creation
- File contains an `onmessage` event function
- Main file calls `postMessage` to start the thread along with an object argument.
- Worker sends a message back to the main file (`postMessage`), it can catch the data with an `onmessage` event.

Overview: Web Workers



Your Homework (part 1 and part 2)

- part 1 you only modify the Web Worker.
  - You are given code to do all of the Web Worker interface (sending message, posting messages, etc).
  - You just need to update the particles every timestep

- part 2 you need to modify the Web Worker for it to be multithreaded
  - Most of your web worker from part 1 will apply.
  - You will need extra arguments
  - You will need to modify main.js to launch multiple web workers and figure out how to make sure they are all finished before drawing and calling the next iteration of updates.

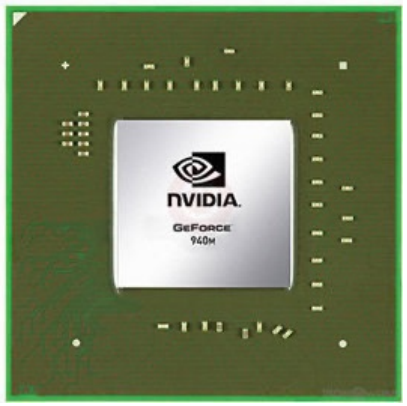
Your Homework (part 3)

- part 3
  - You will only modify the GPU kernel code.
  - WebGPU simply has too much boiler plate.
  - We will discuss CUDA mappings to WebGPU on Wednesday

On to the GPU part of the lecture!

Programming a GPU

The GPU in
my PhD laptop



Nvidia 940m
1.8 Billion transistors
33 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/

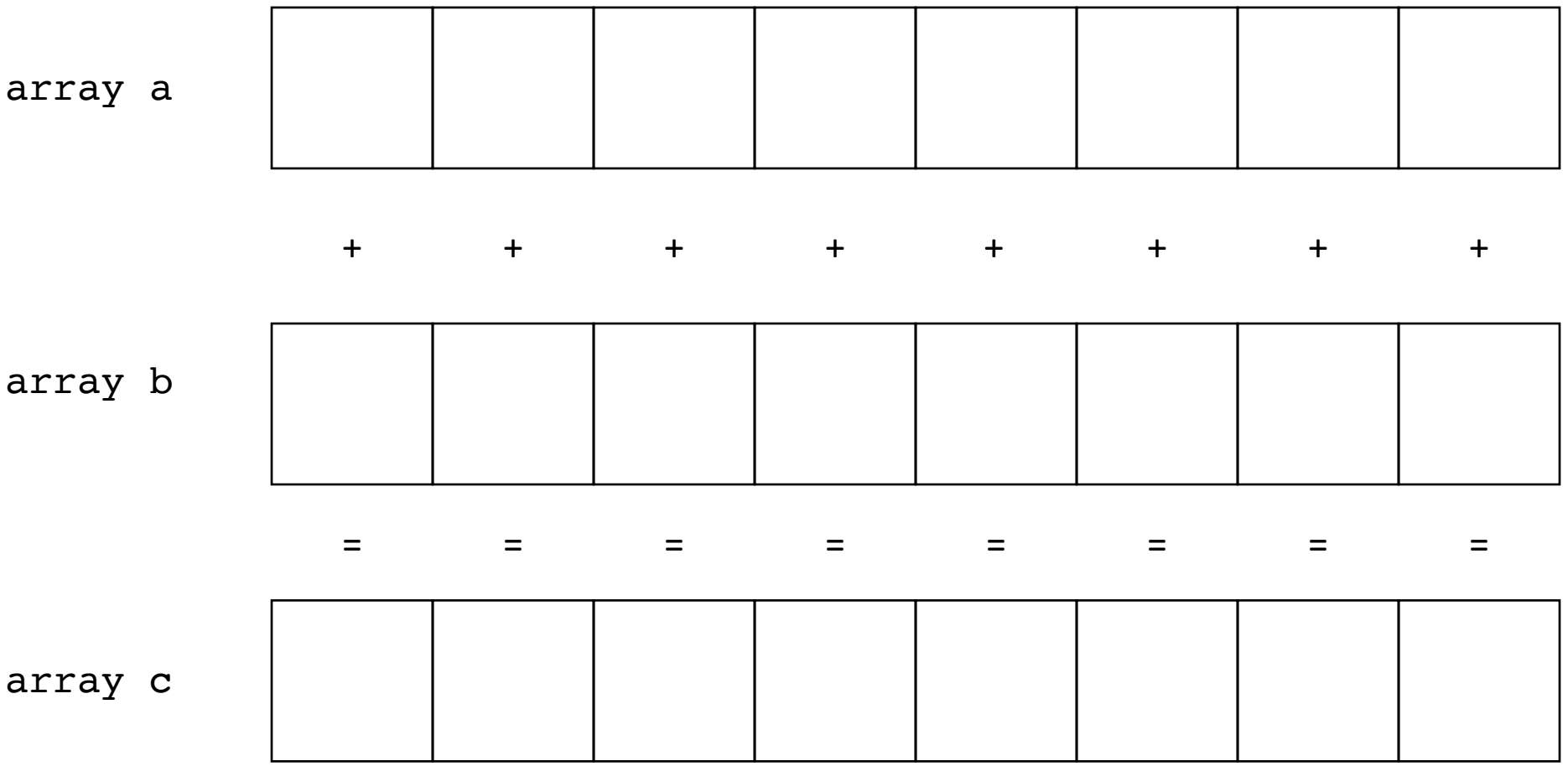
Programming a GPU

- 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

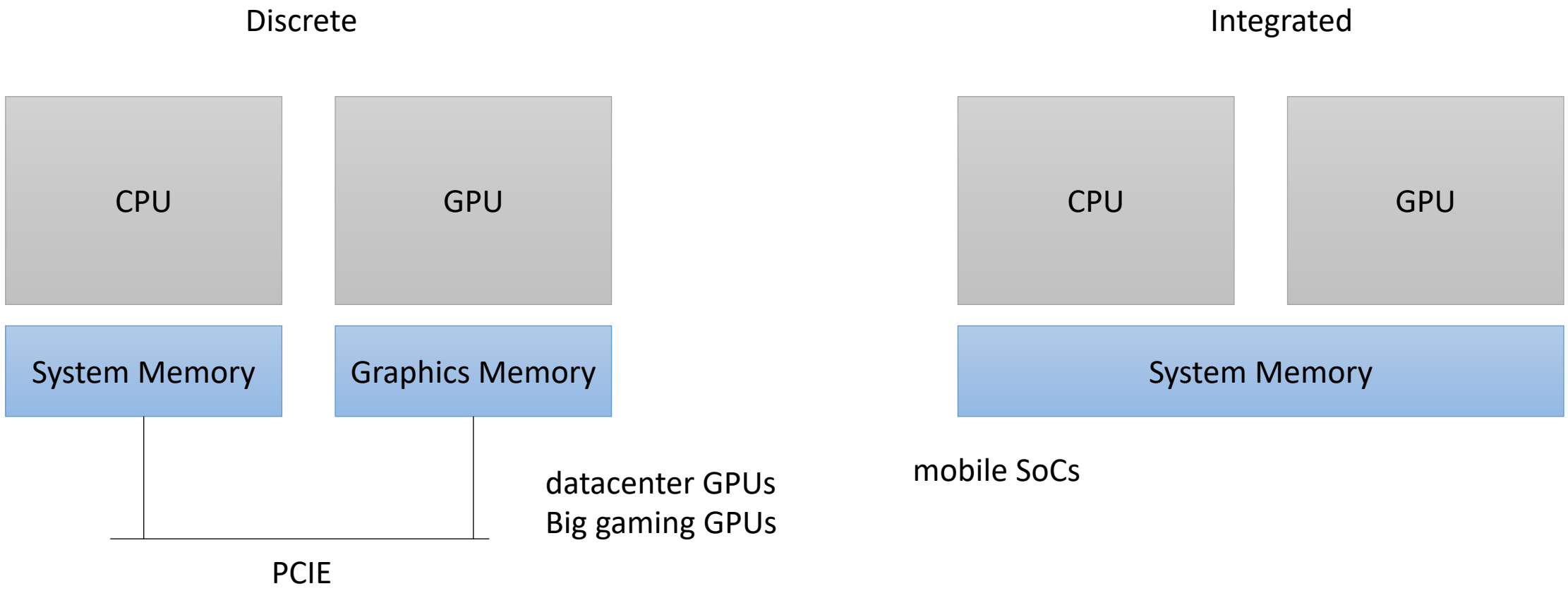
- Its going to take a bit of work....

GPU set up

- We need to allocate and initialize memory

GPU set up

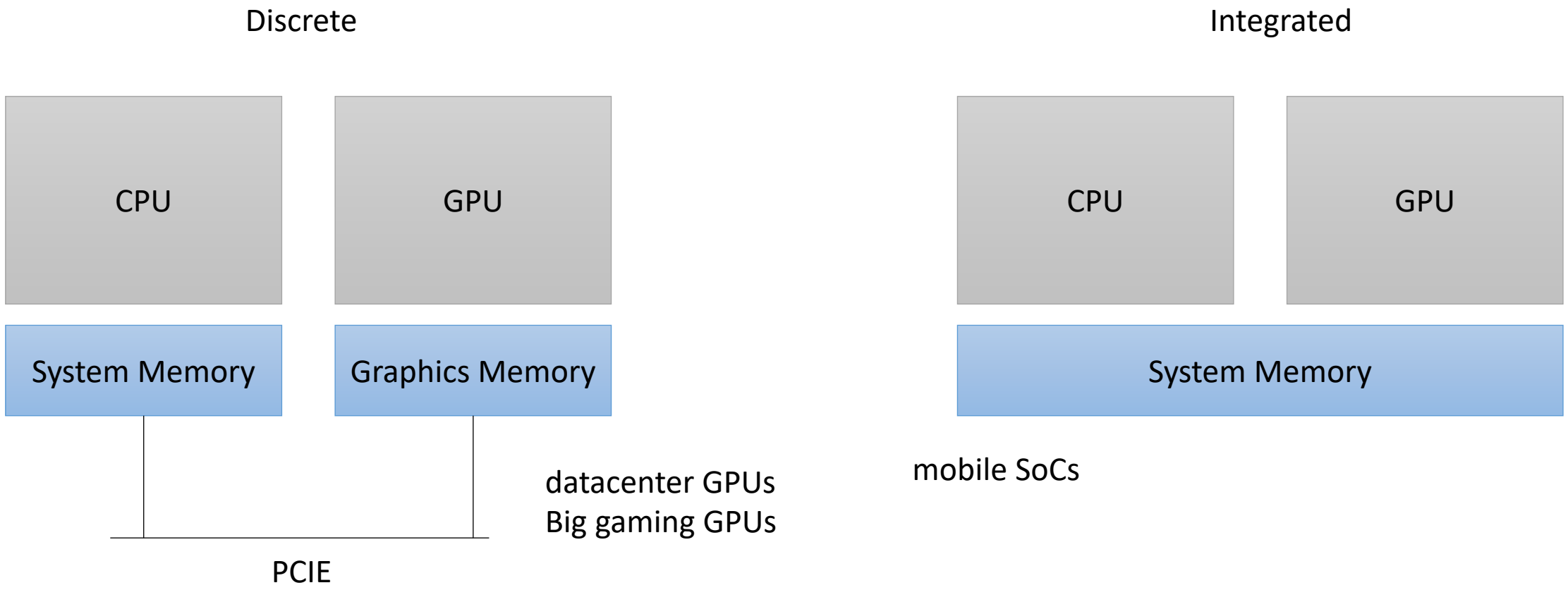
- GPUs come in two flavors



GPU set up

Pros and cons of each?

- GPUs come in two flavors

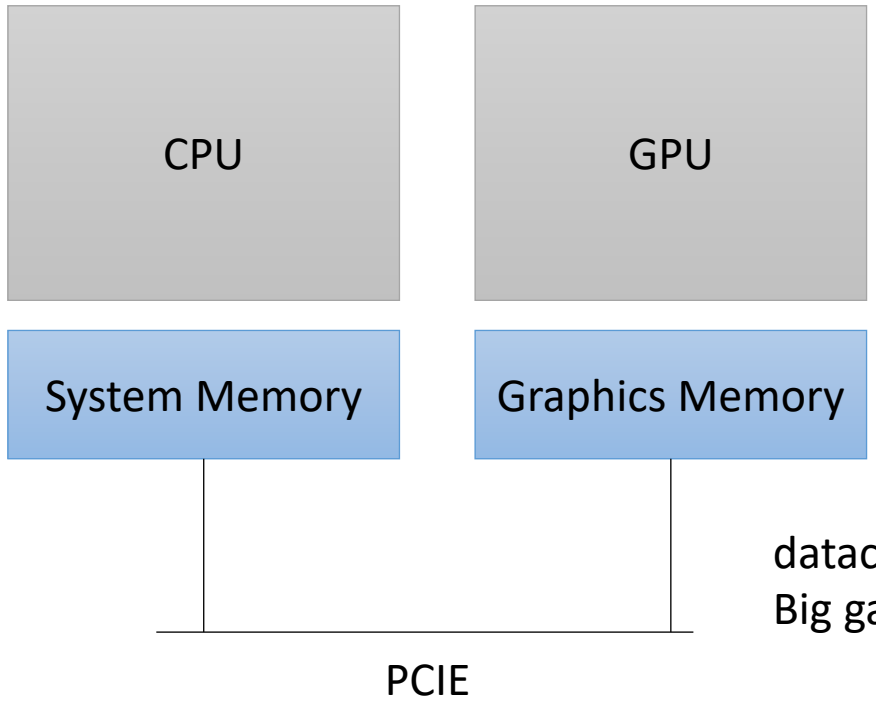


GPU set up

- GPUs come in two flavors

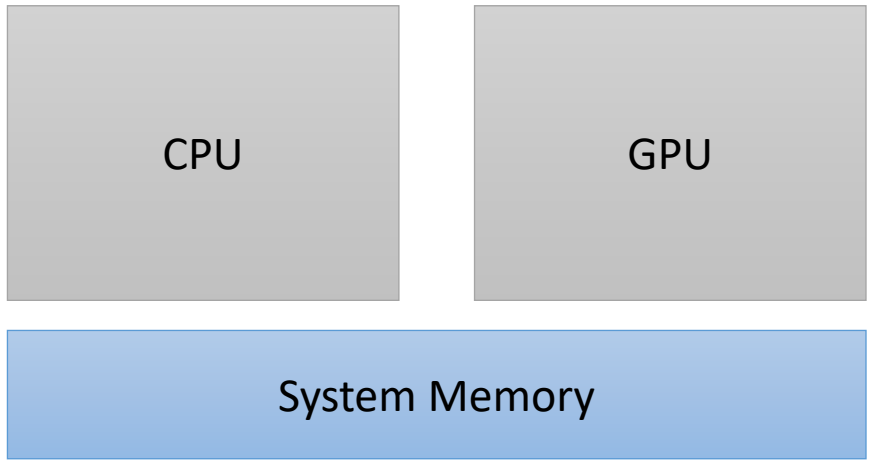
Pros and cons of each?
* Different types of memory for discrete
* Swappable for discrete
* More energy efficient for integrated
* Better memory utilization for integrated
* More efficient communication between CPU and GPU

Discrete



datacenter GPUs
Big gaming GPUs

Integrated



mobile SoCs

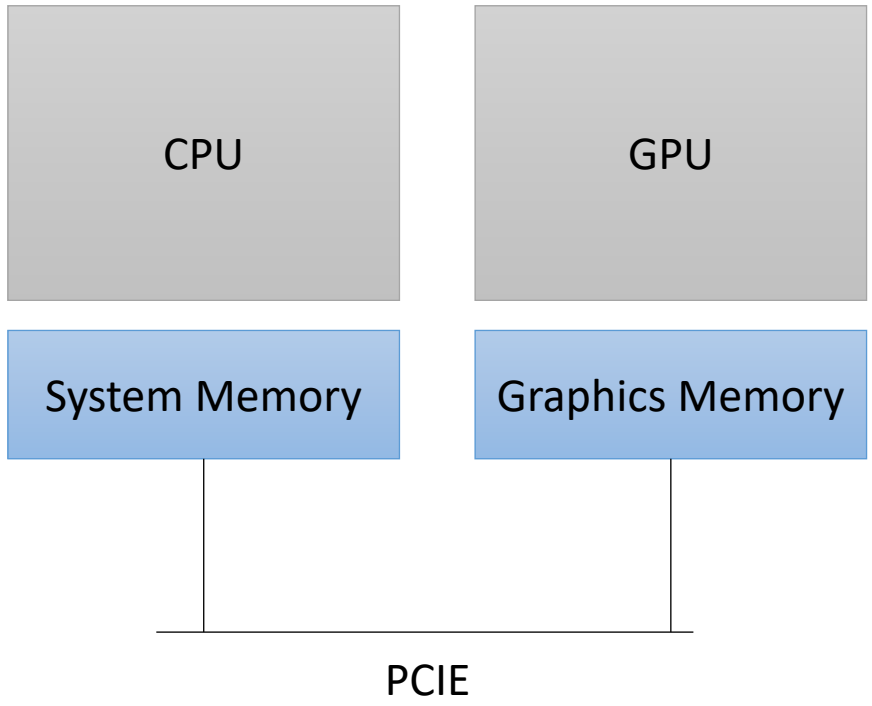
GPU set up

- GPUs come in two flavors

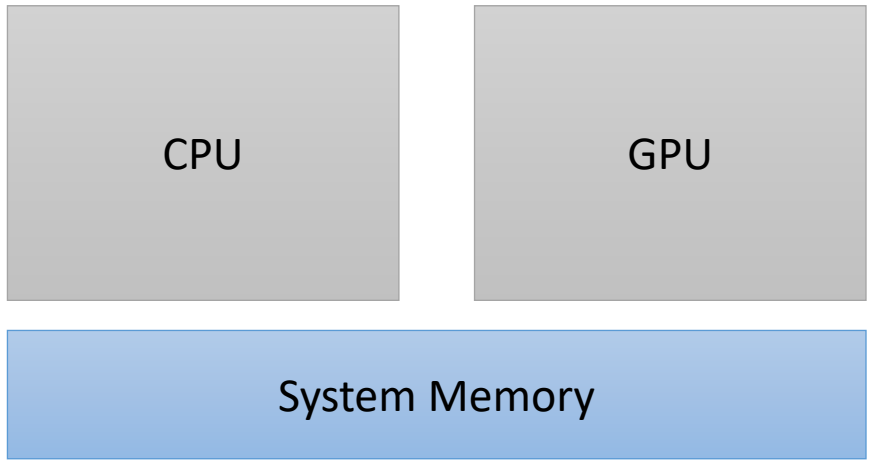
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



Integrated



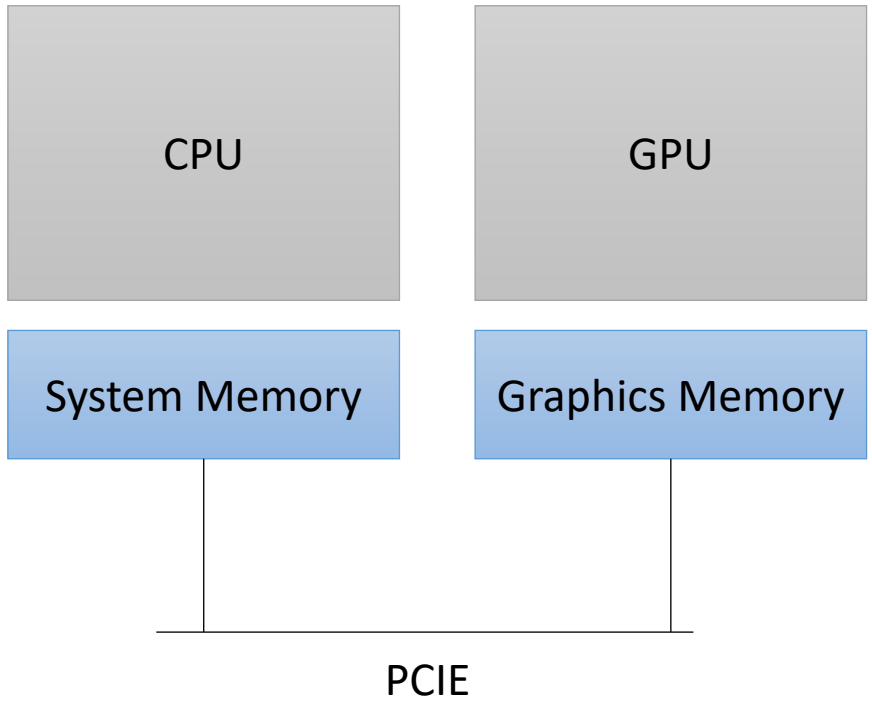
GPU set up

- GPUs come in two flavors

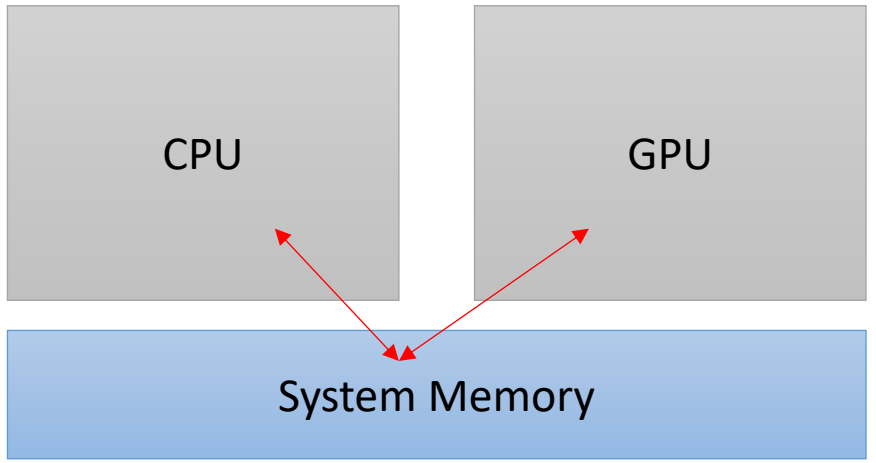
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



Integrated



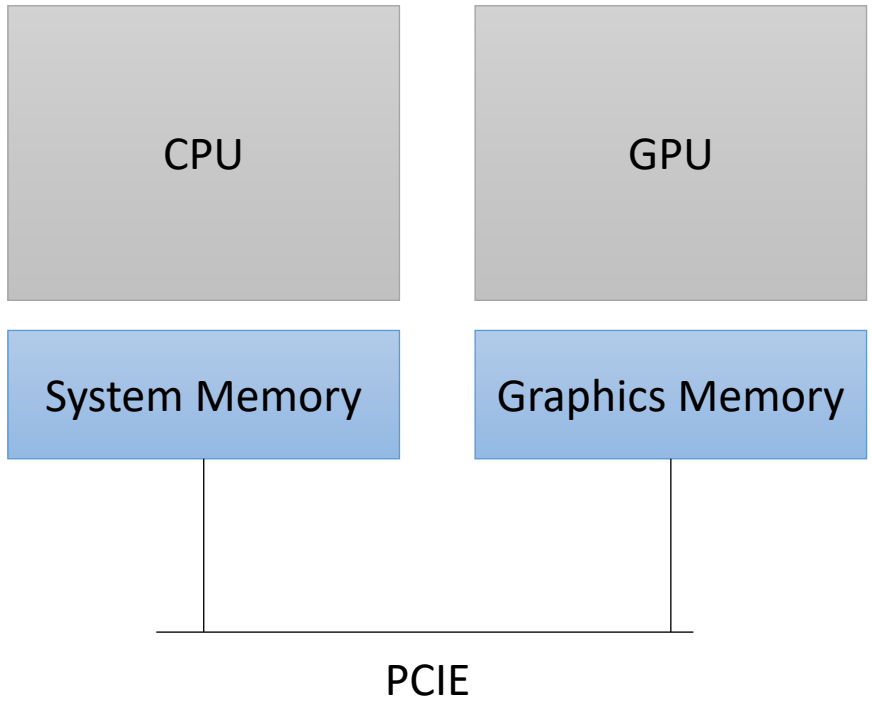
GPU set up

- GPUs come in two flavors

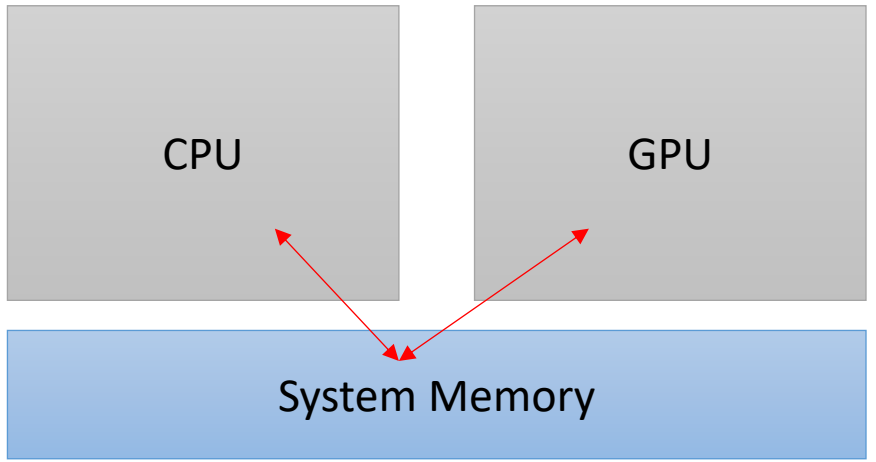
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



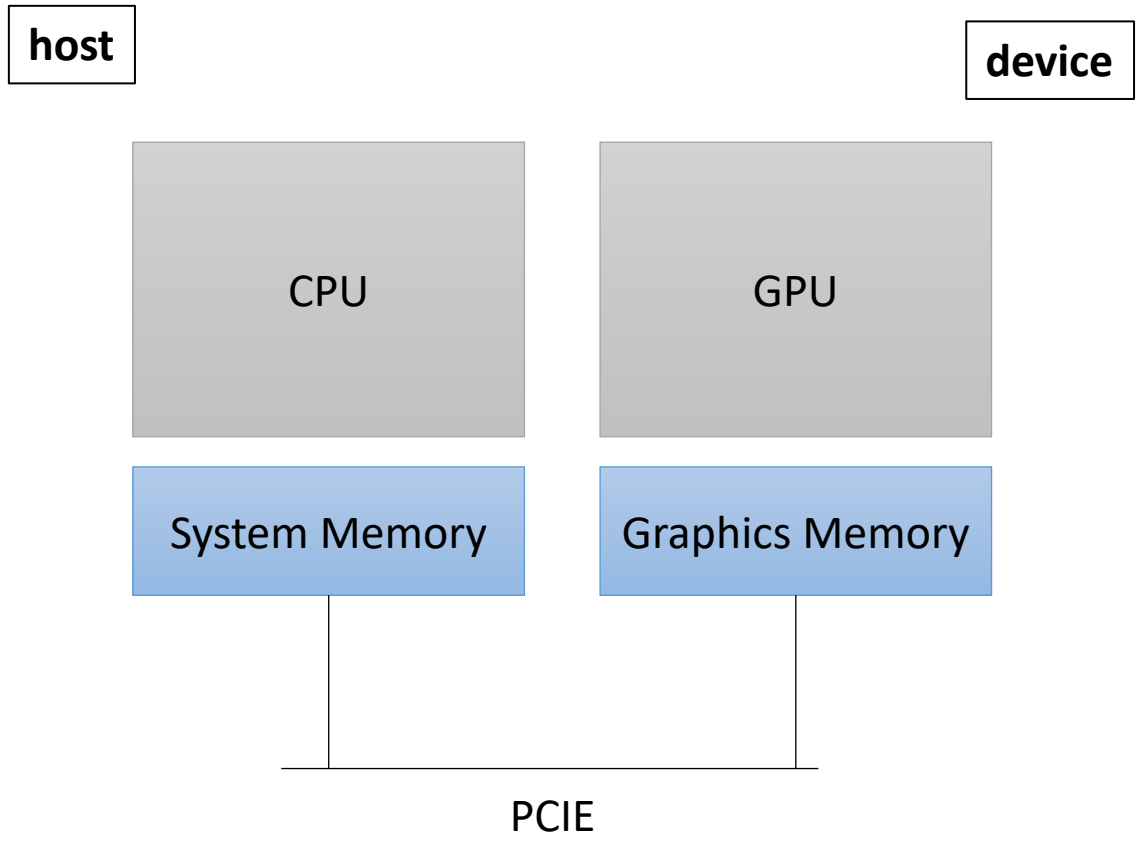
Integrated



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

GPU set up

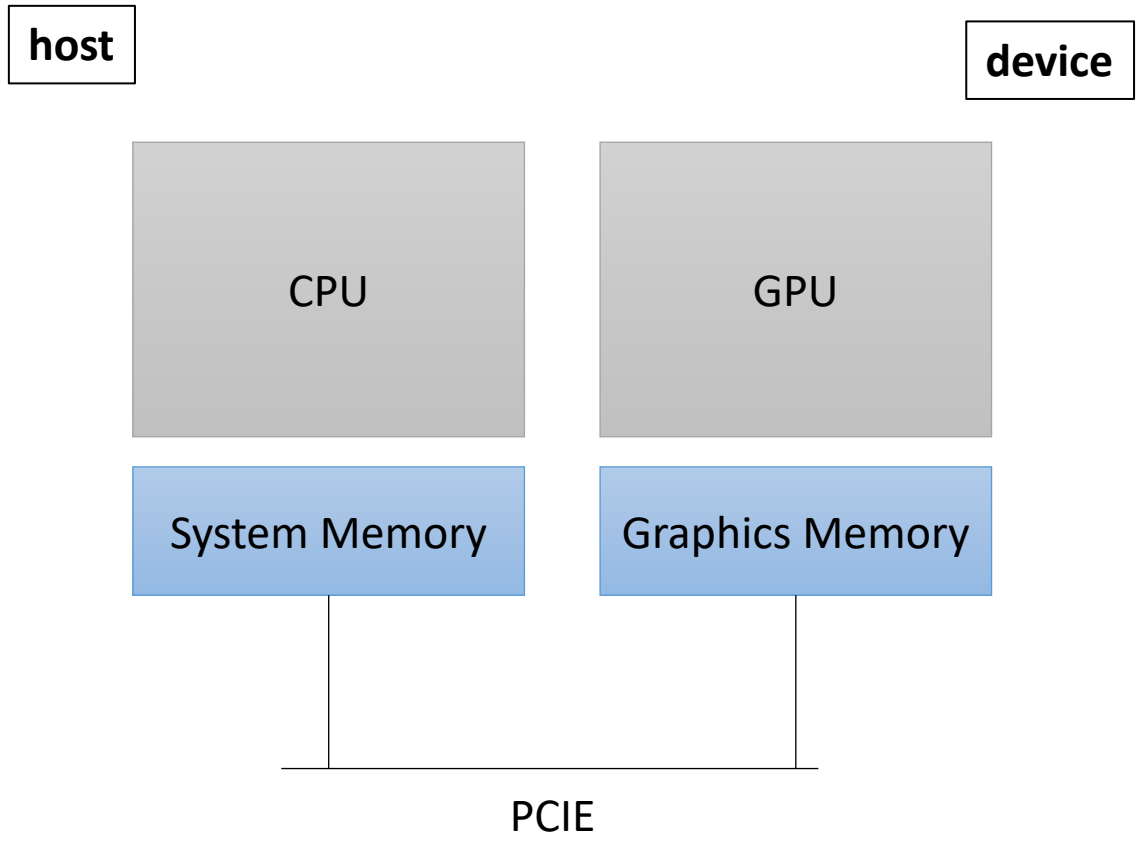
- Our heterogeneous, parallel, programming model



GPU set up

- Our heterogeneous, parallel, programming model

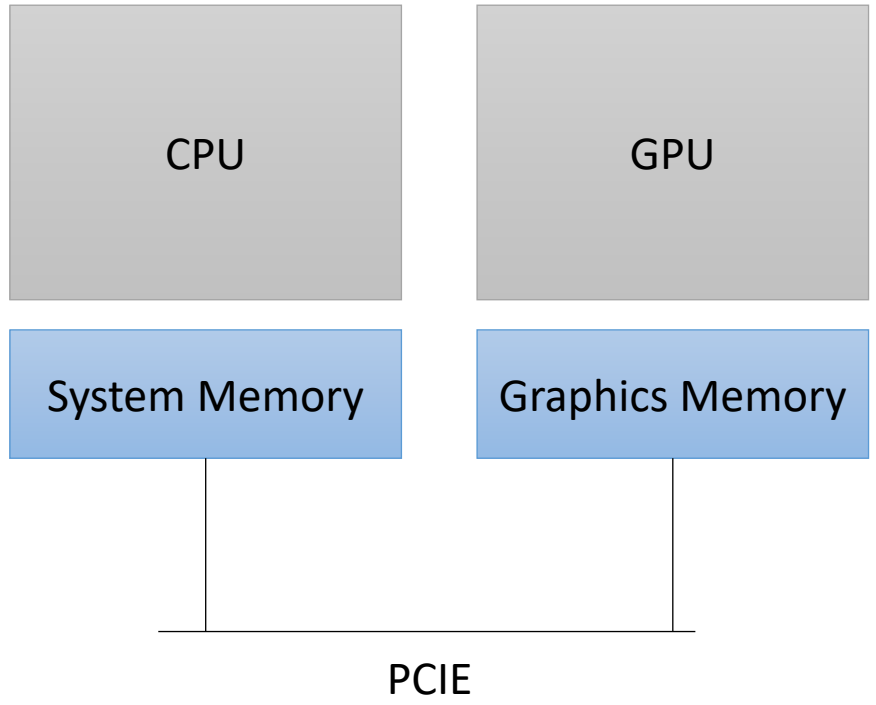
The host (CPU) will write a C++-like program that allocates and sets up memory on the GPU. The host will then call a GPU program called a kernel.



GPU set up

How do we allocate memory on a CPU?

- Our heterogeneous, parallel, programming model

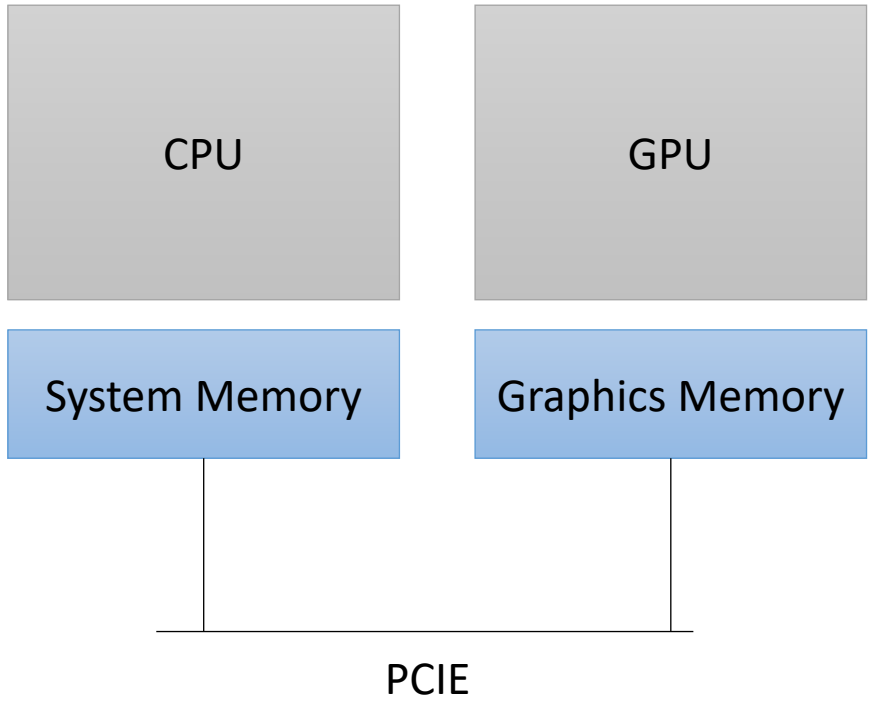


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

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

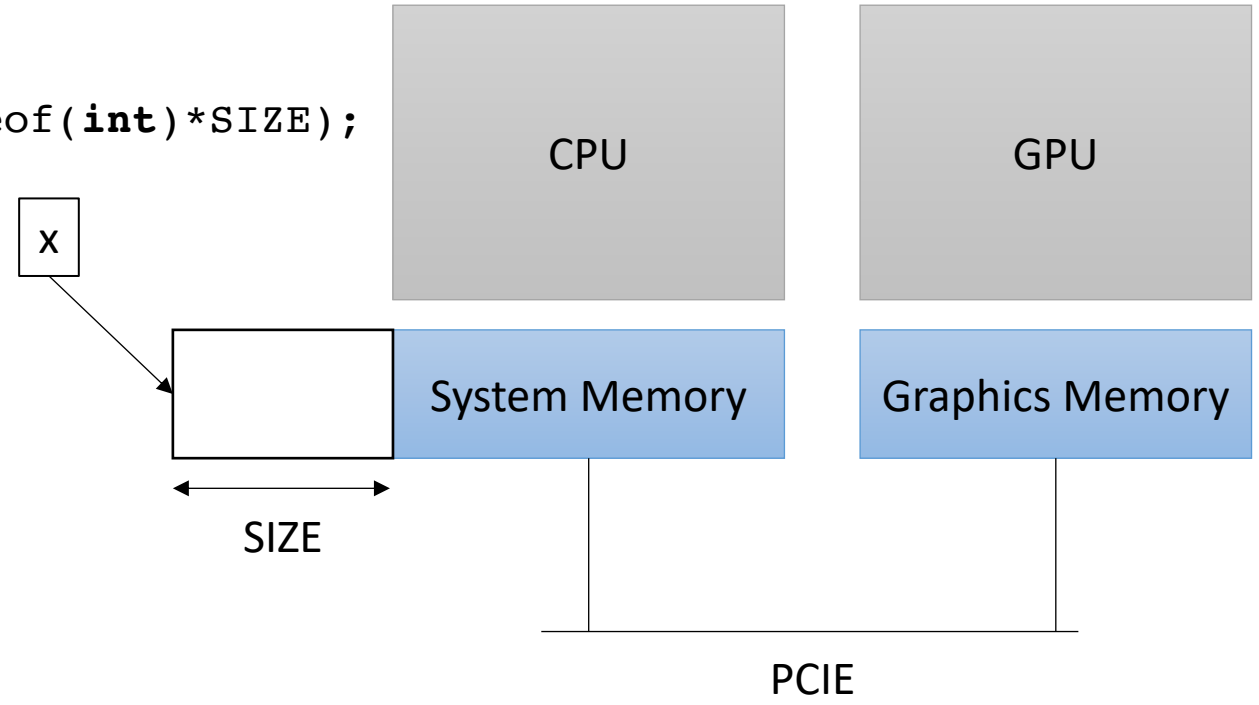


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

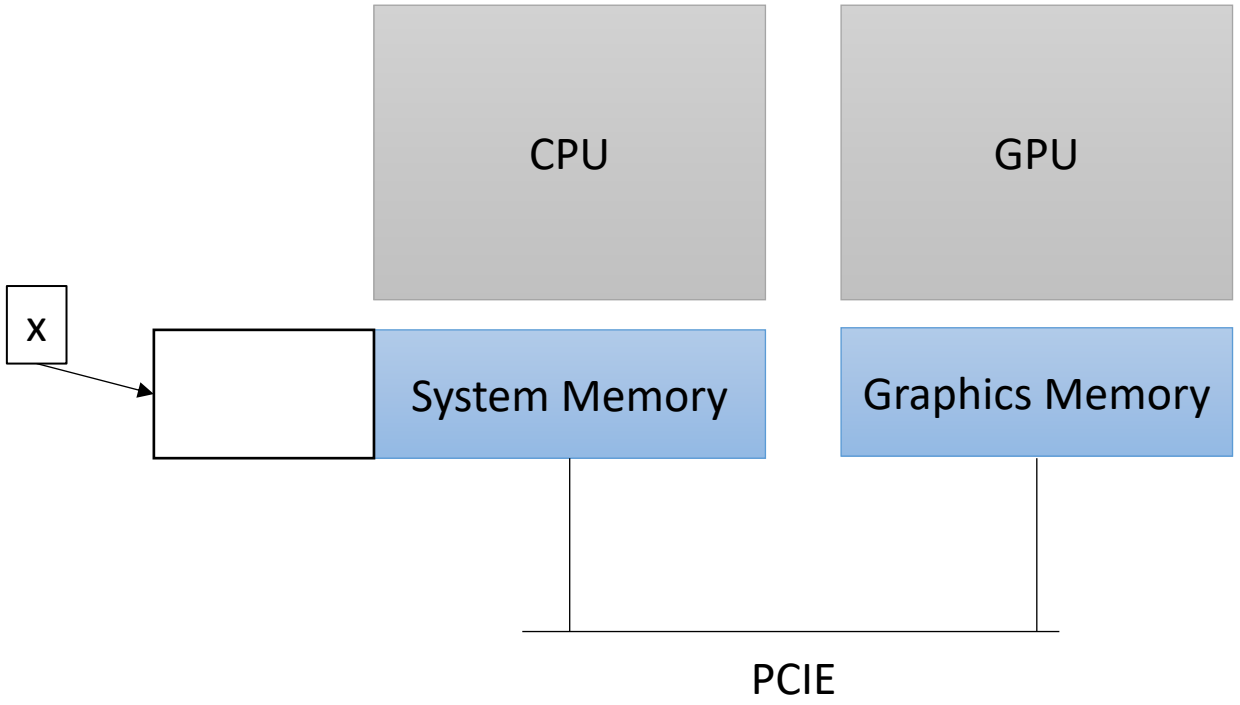
```
int *x = (int*) malloc(sizeof(int)*SIZE);
```



GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

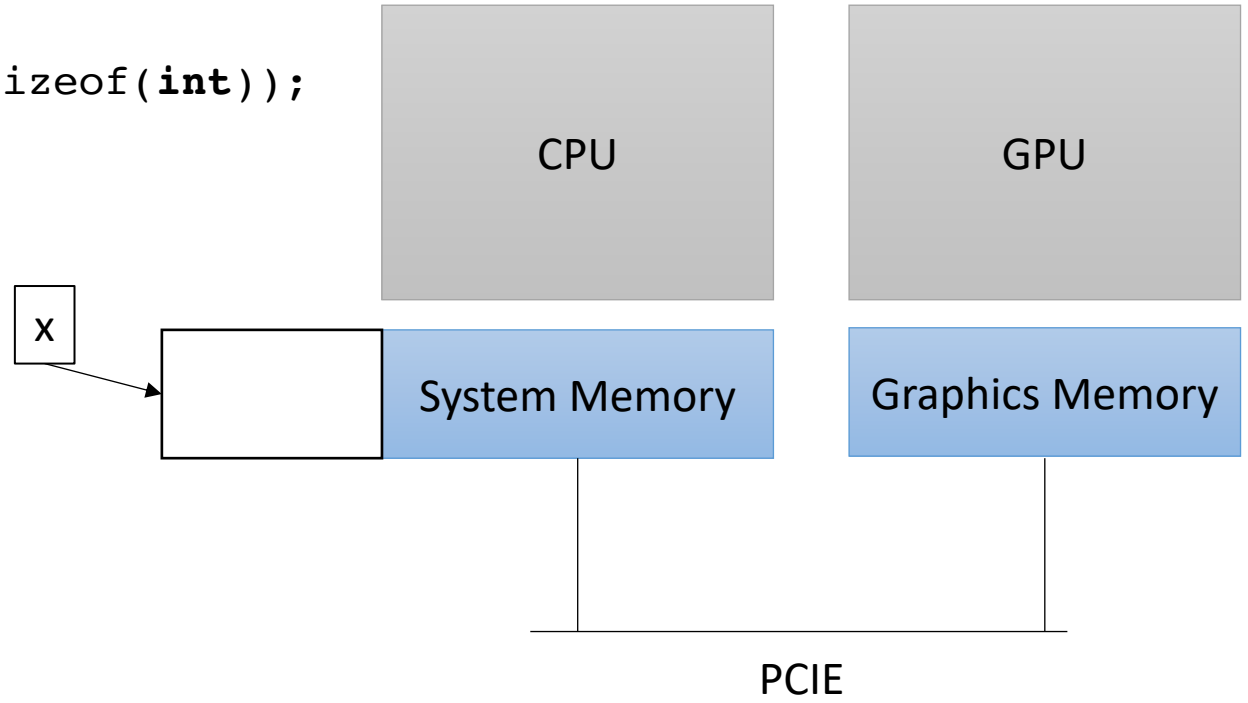


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

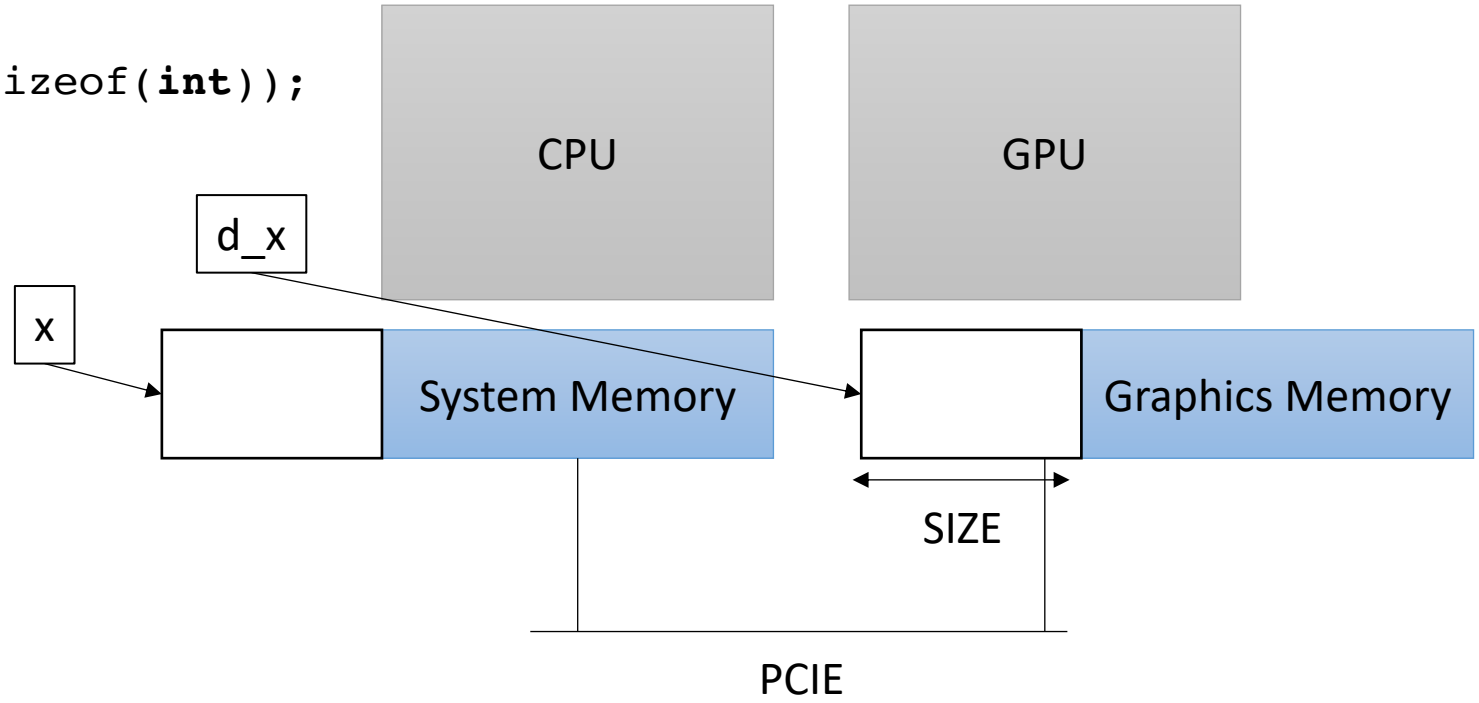


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

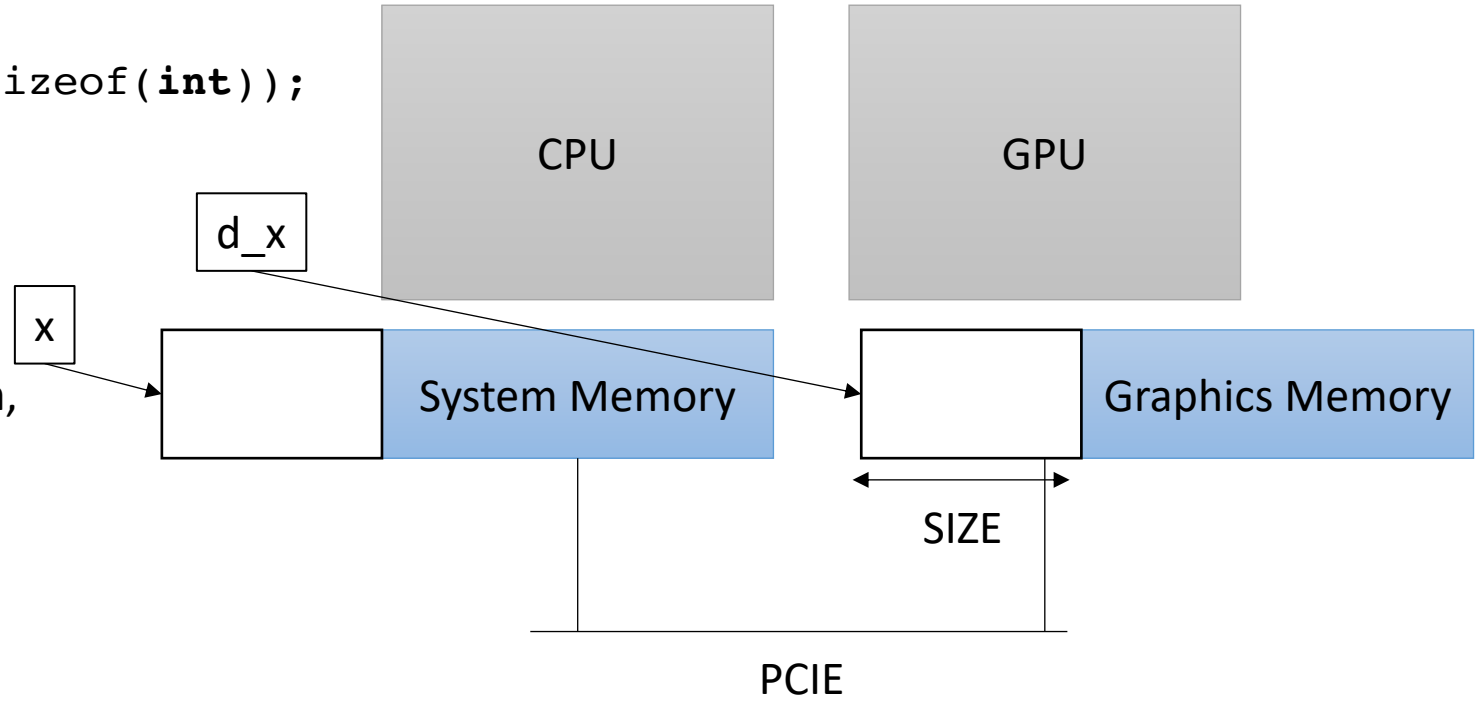


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

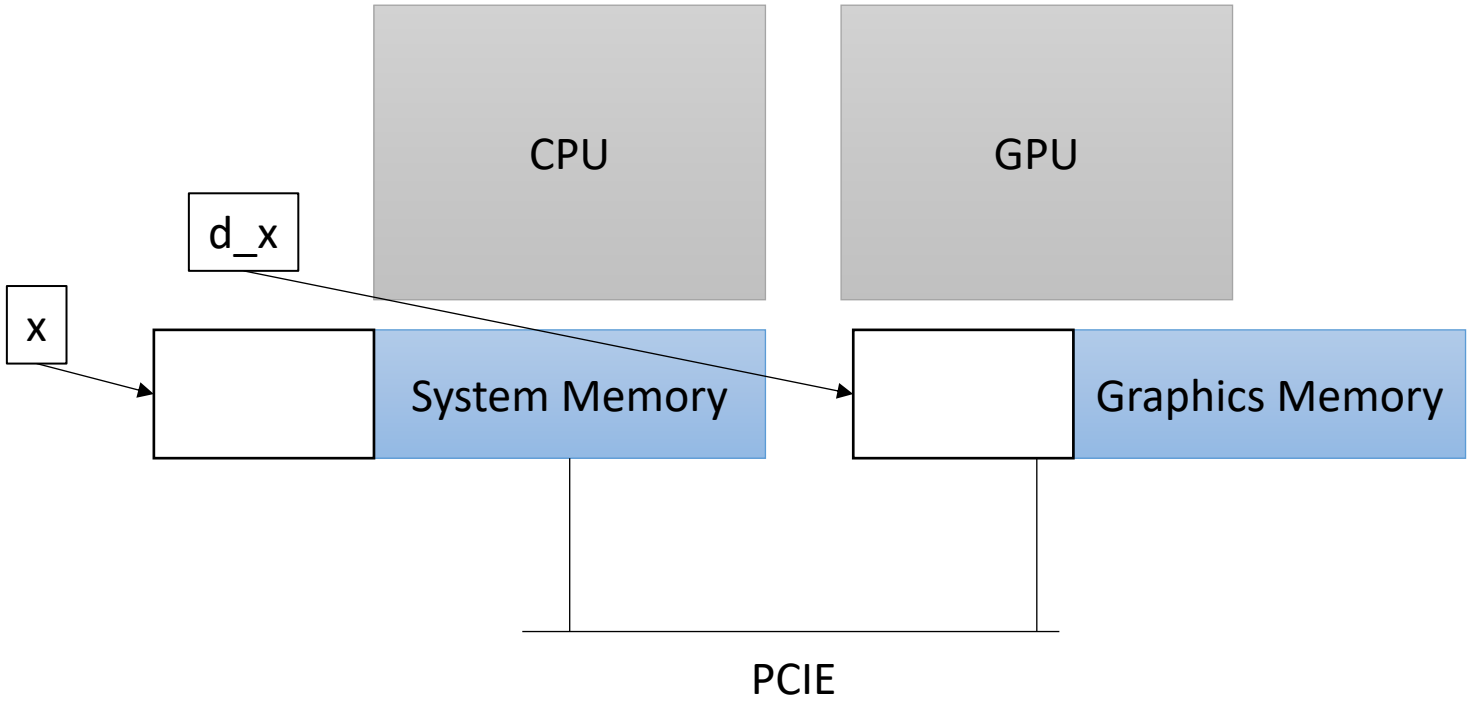


`d_x` is a pointer, in the CPU program, that points to memory on the GPU.

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

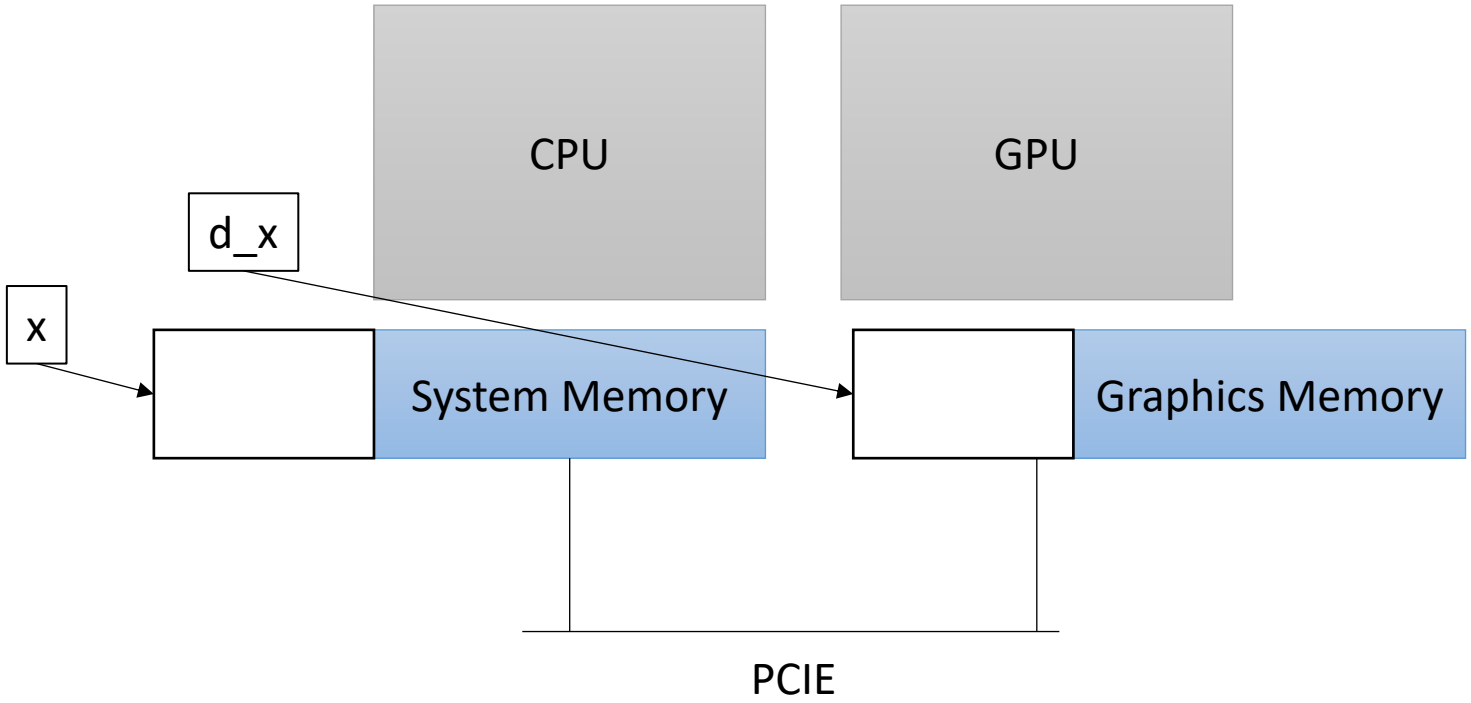


GPU set up

- 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



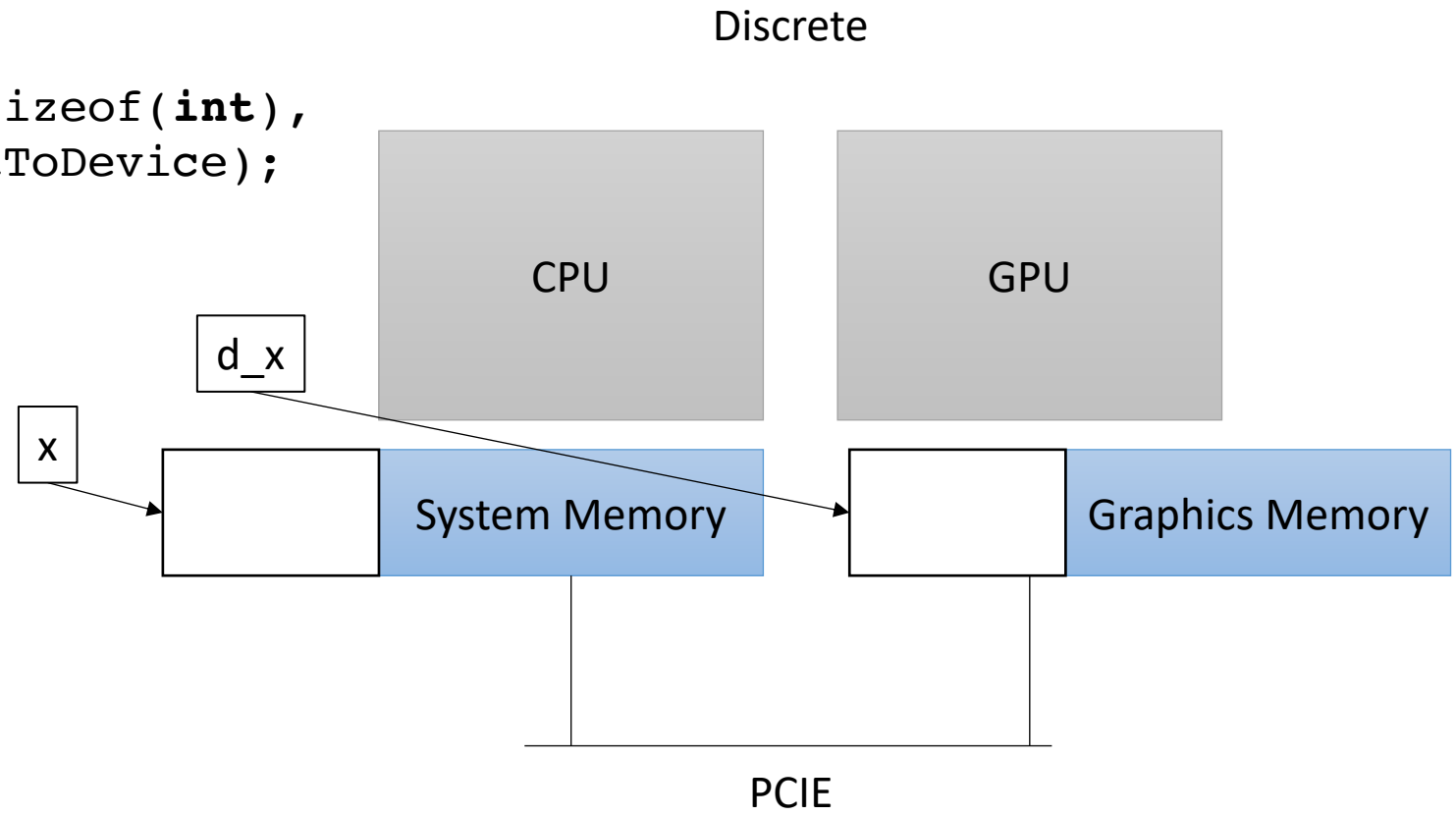
GPU set up

- 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);
```



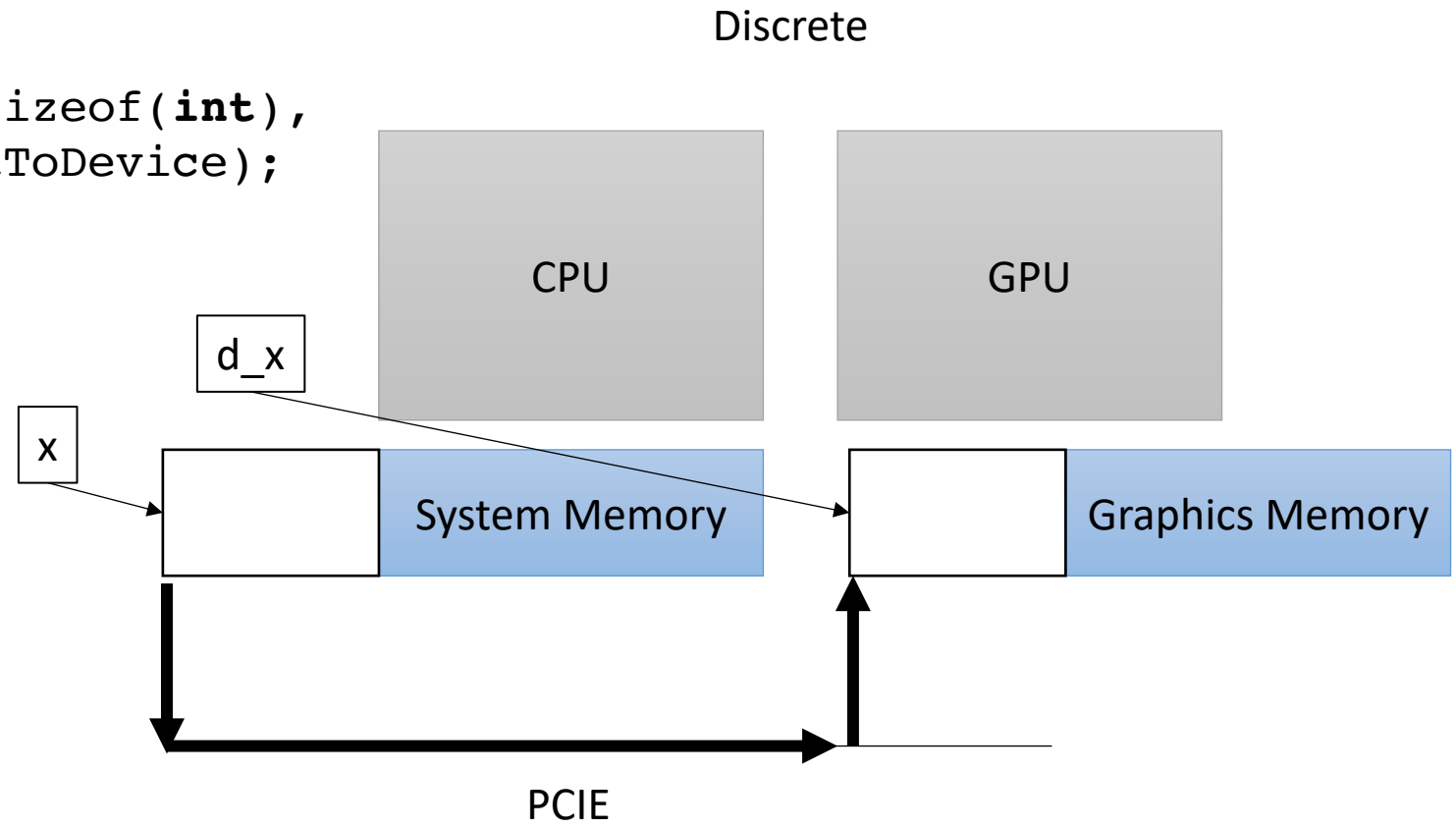
GPU set up

- 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);
```



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

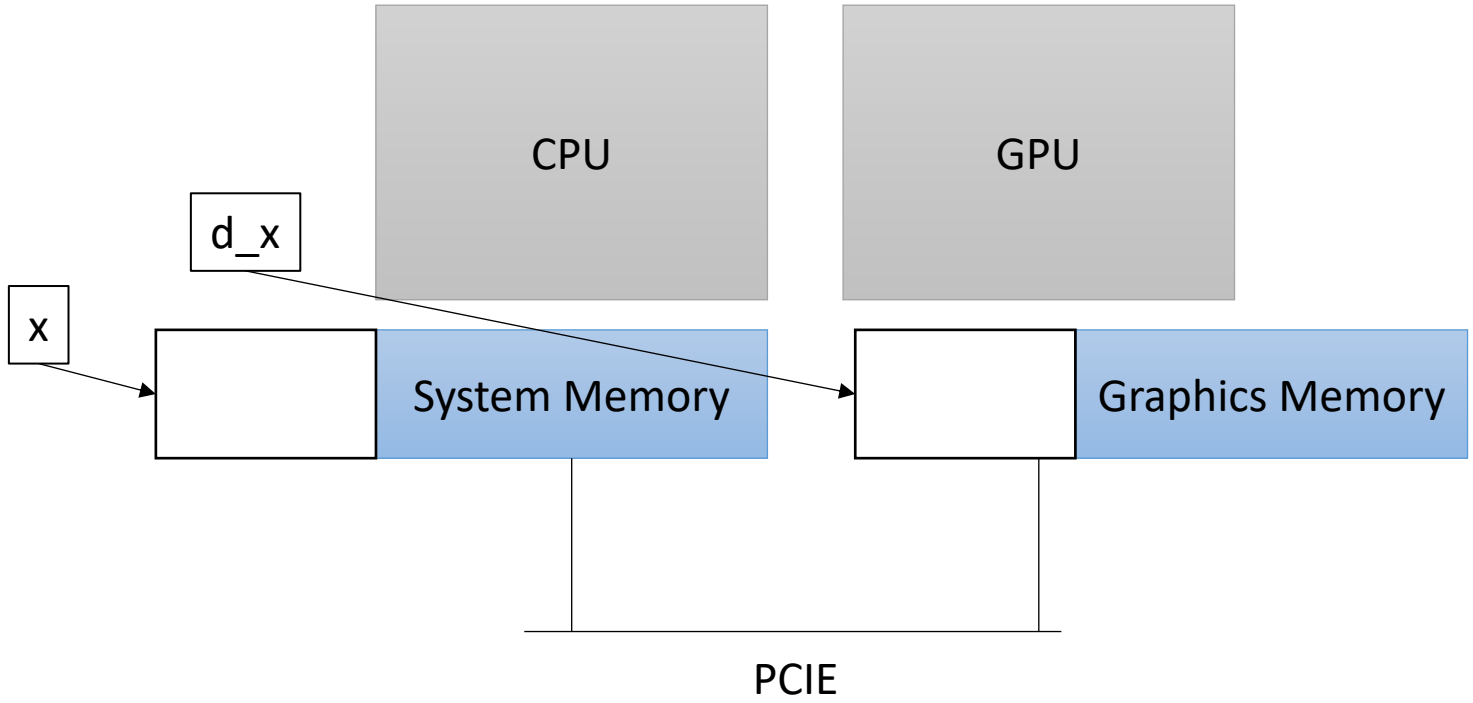
calling the function

```
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);
```

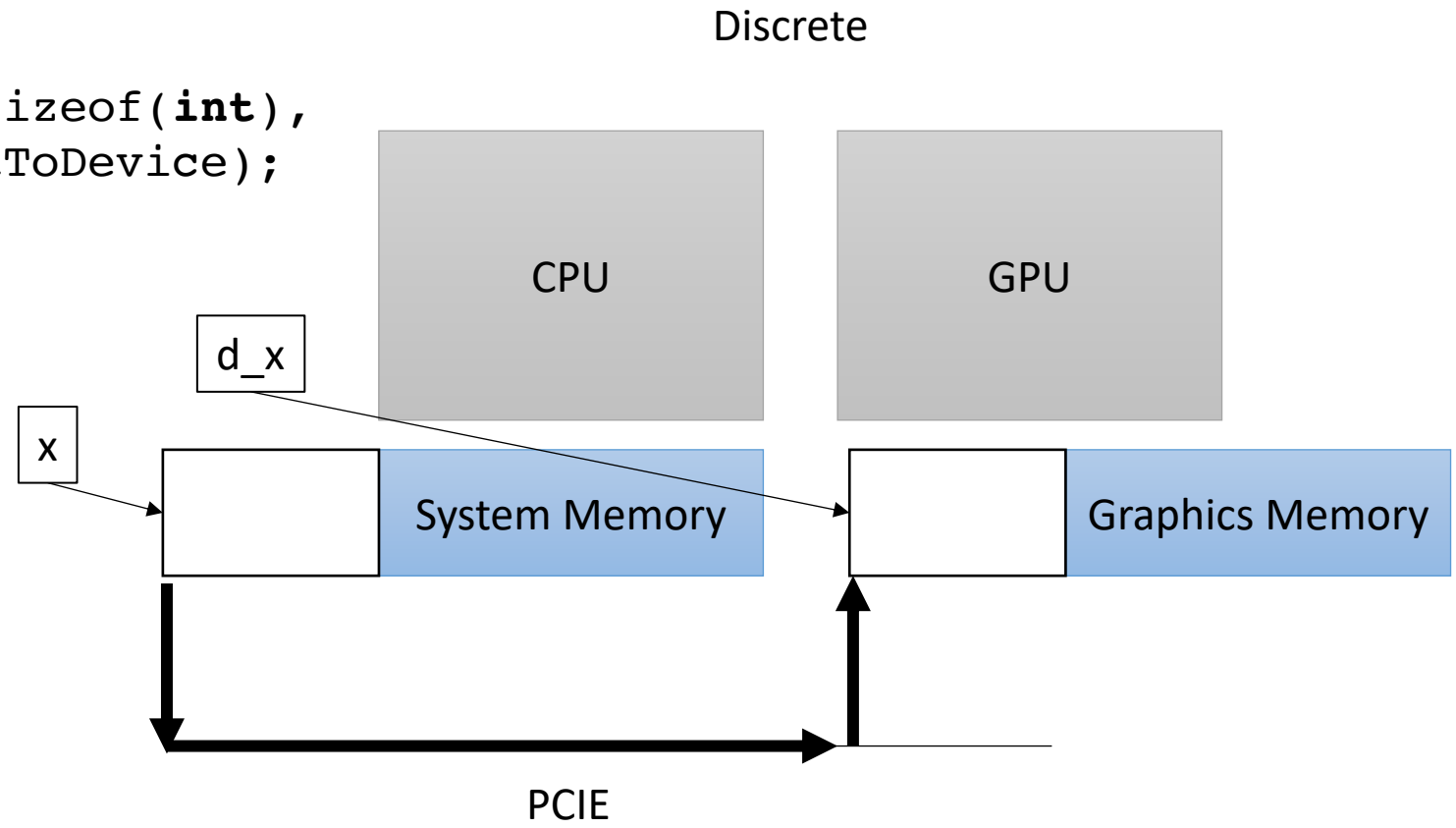
The GPU Program

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

GPU set up

- Our heterogeneous, parallel, programming model

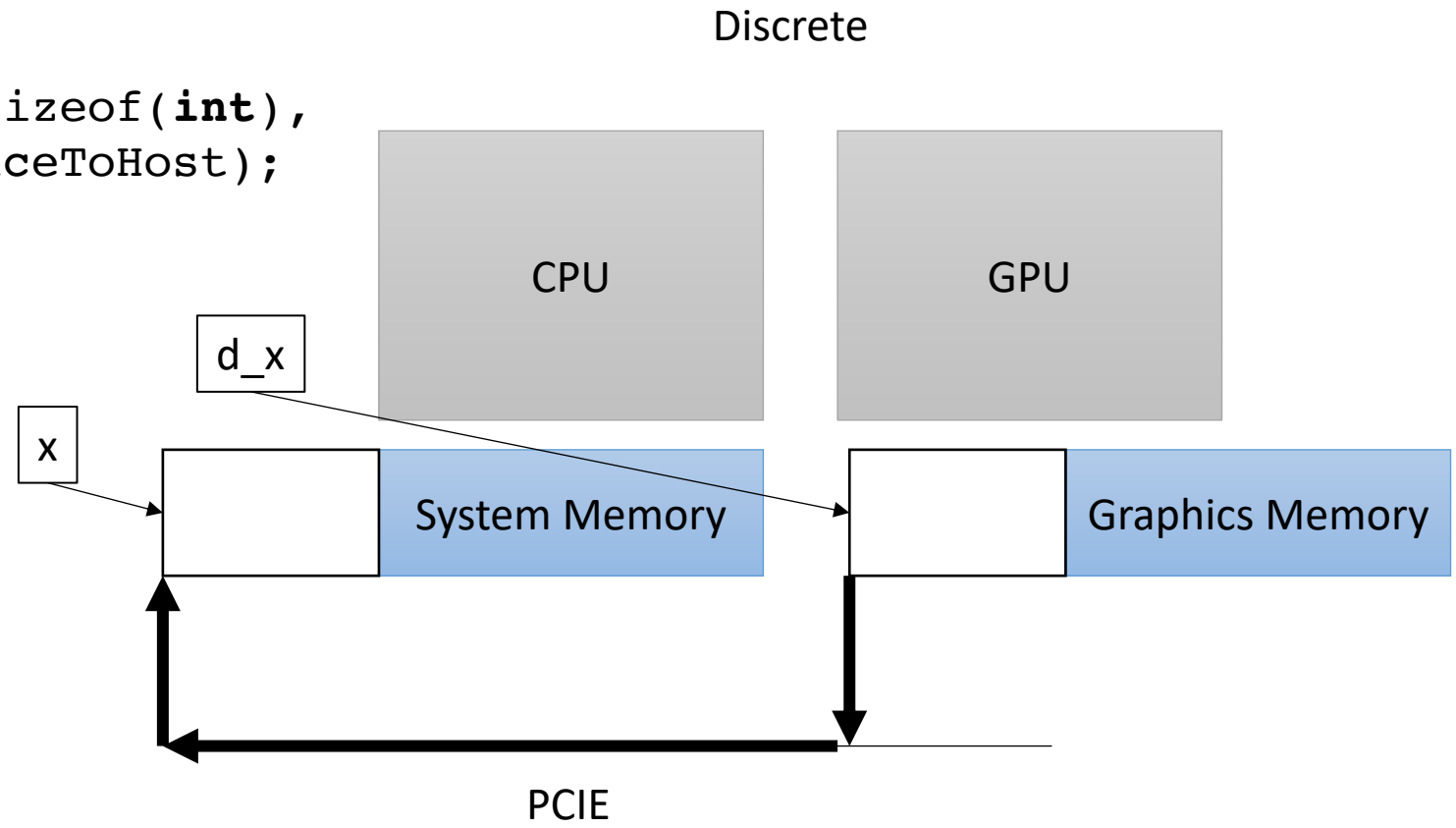
```
//initialize x on host
cudaMemcpy(d_x, x, SIZE*sizeof(int),
           cudaMemcpyHostToDevice);
```



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

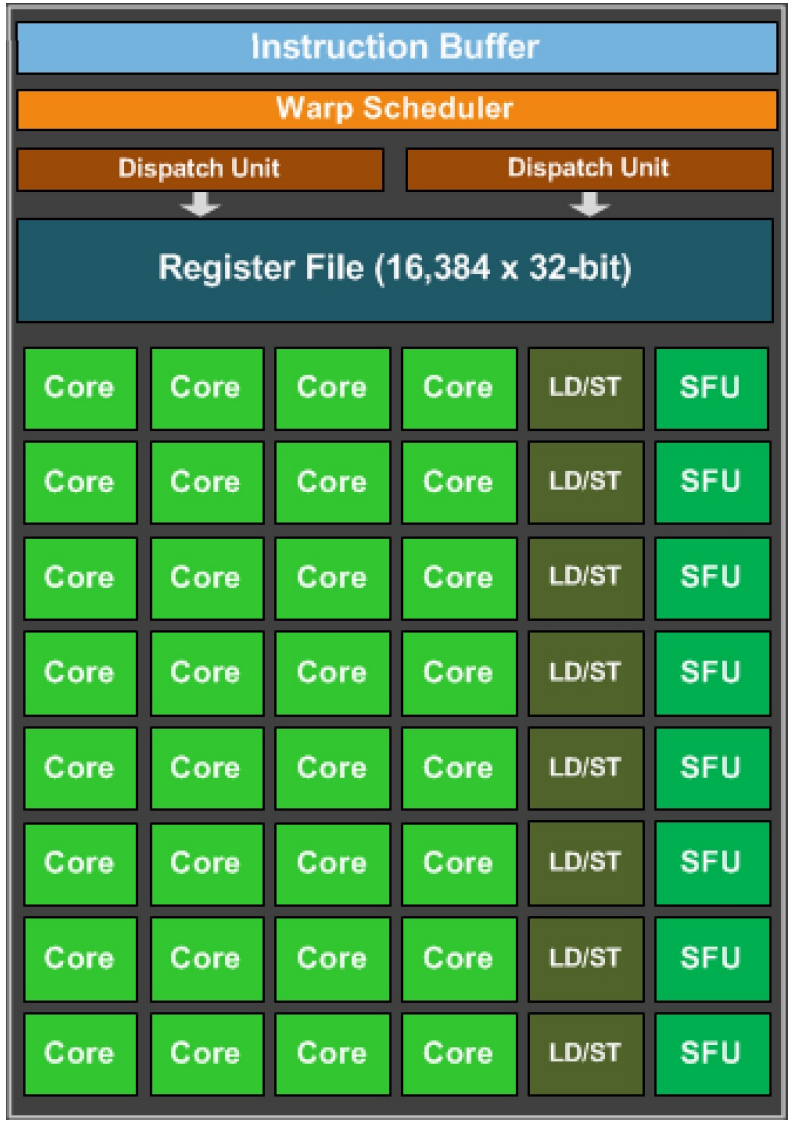
The GPU Program

It didn’t do so well...

First parallelization attempt

- Lets look at some GPU documentation.

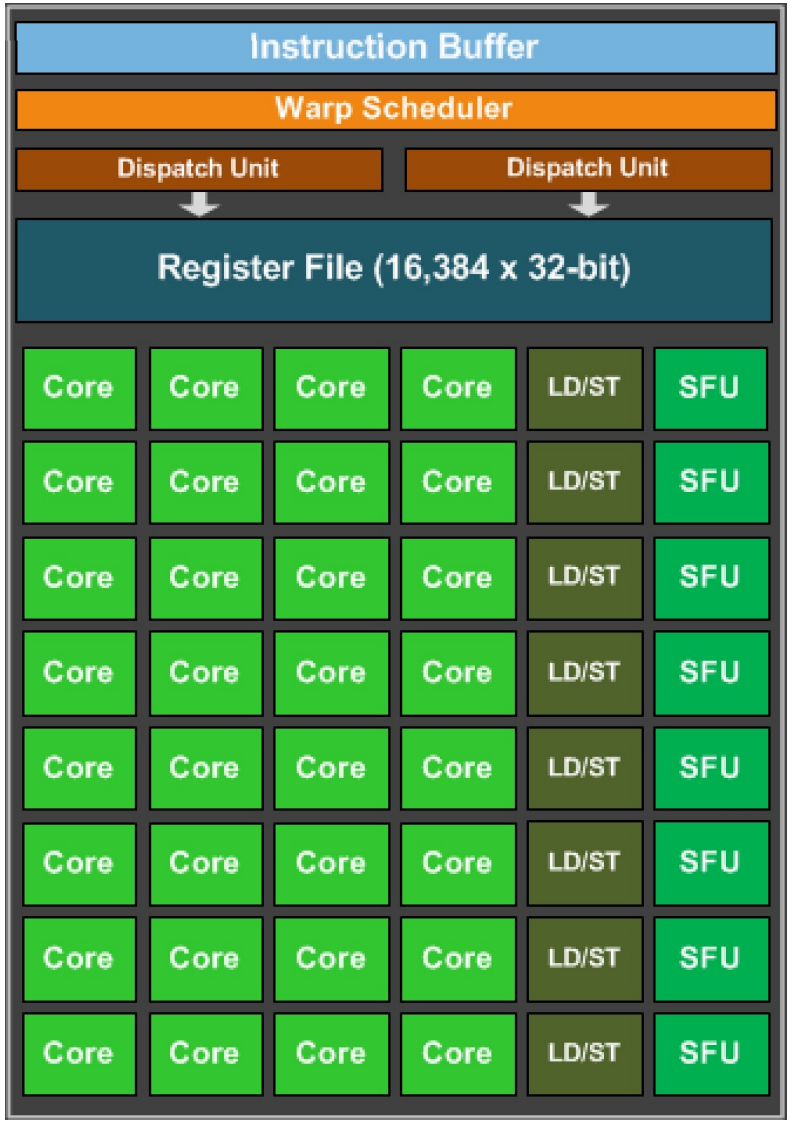
- The Maxwell whitepaper shows a diagram of one of the GPU cores



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

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

number of threads

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

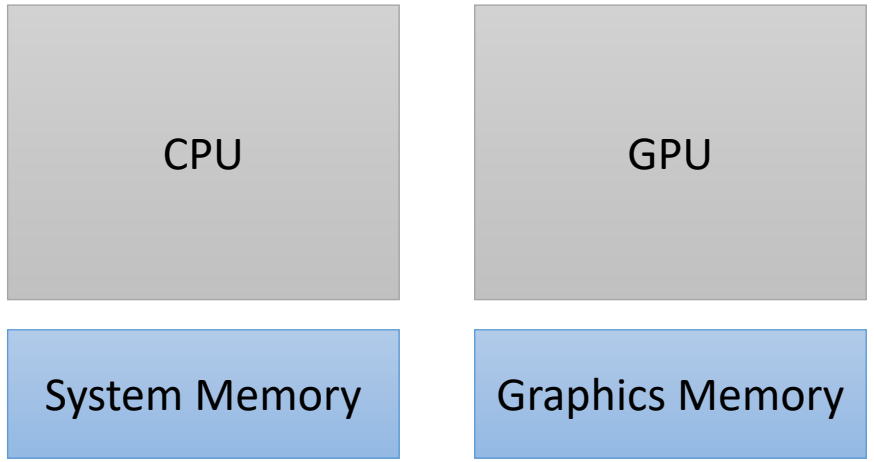
First parallelization attempt

Lets try it! What do we think?

First parallelization attempt

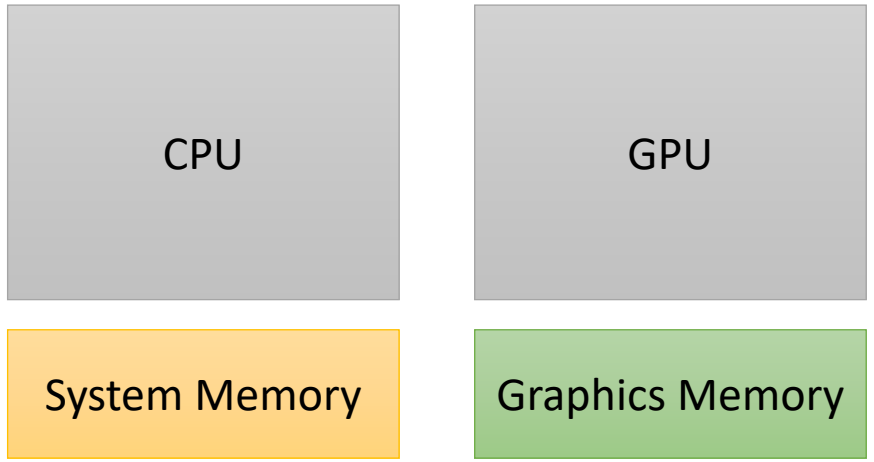
Getting better but we have a long ways to go!

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

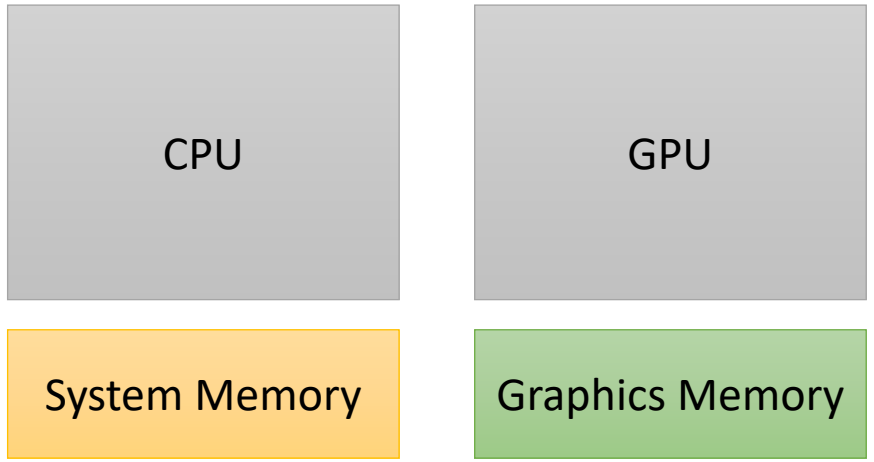
16-lane highway on a windy road driven by semi trucks

GPU Memory

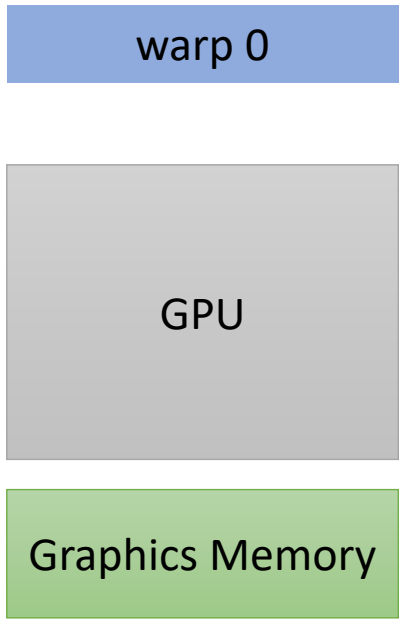
bandwidth:
~**700 GB/s** for GPU
~**50 GB/s** for CPUs

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

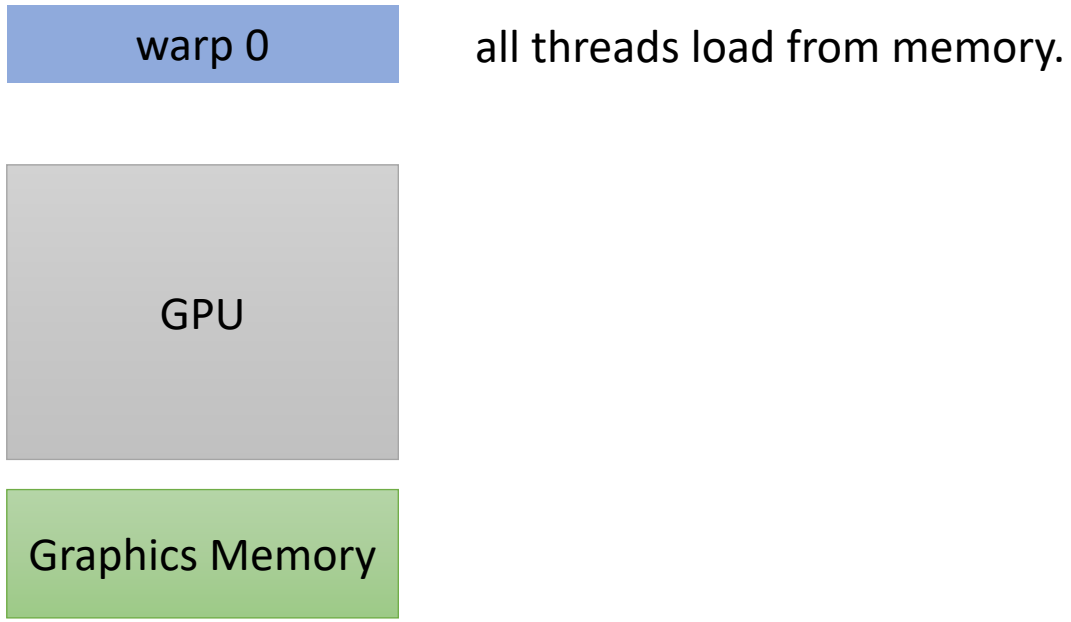
Cache Latency:
~**28** cycles for L1 hit for GPU
~**4** cycles for L1 hit on CPUs



Preemption and concurrency?

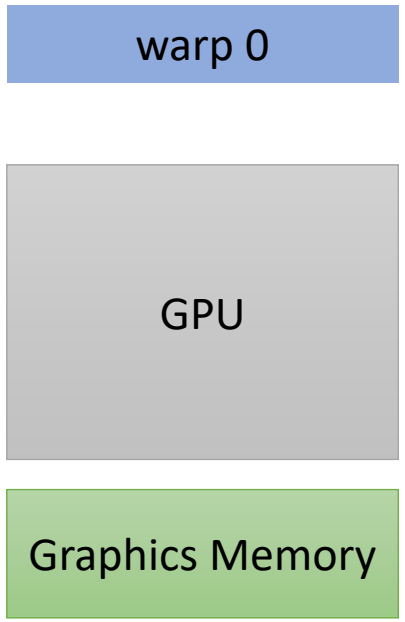


Preemption and concurrency?



all threads load from memory.

Preemption and concurrency?



all threads load from memory.

600 cycles!

Preemption and concurrency?

warp 0

warp 1

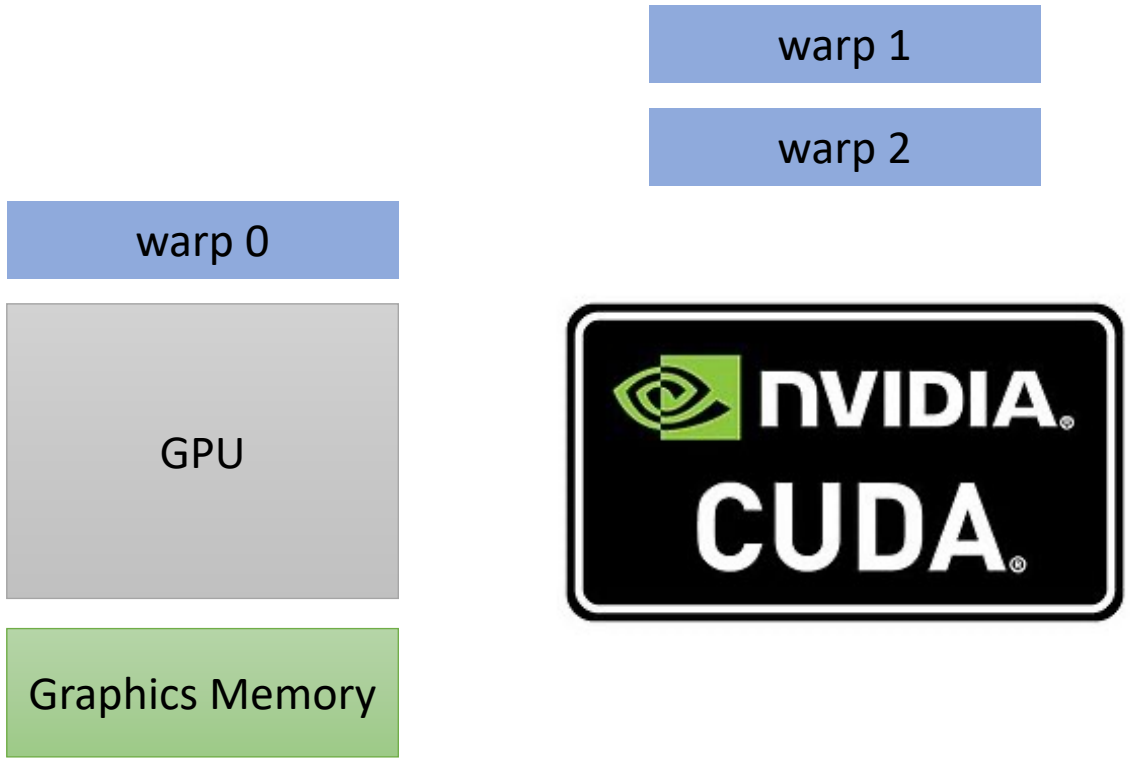
warp 2

GPU

Graphics Memory

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

warp 2

warp 0

GPU

Graphics Memory

We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

warp 1

warp 2

warp 0

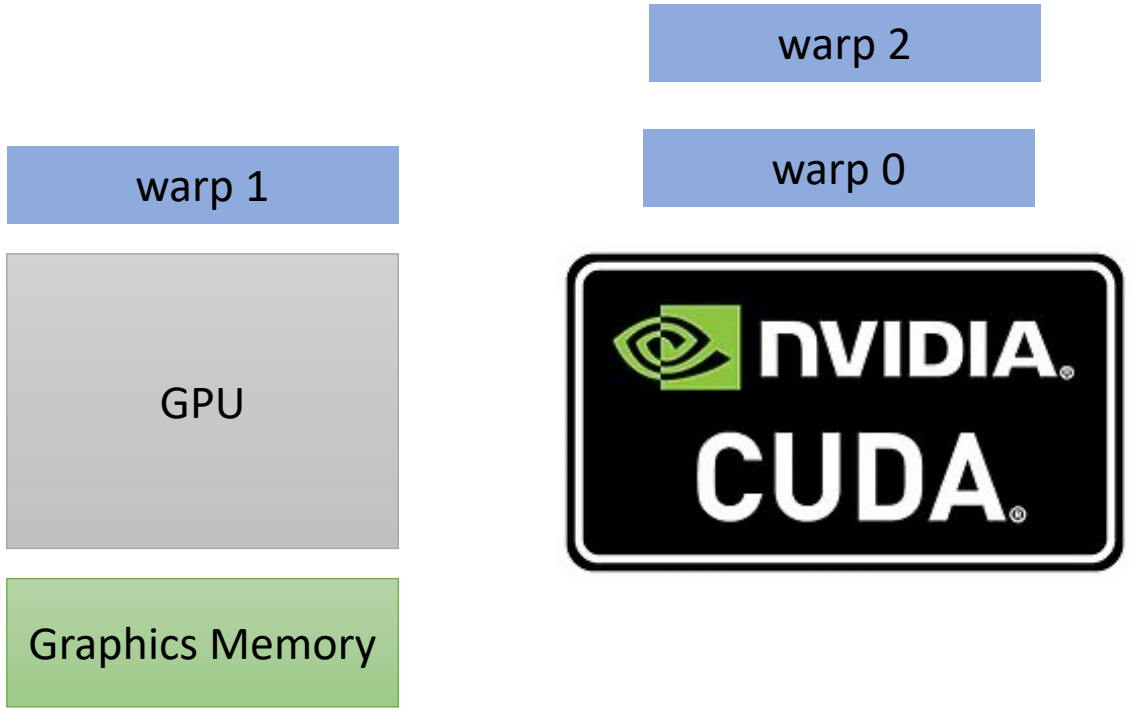
GPU

Graphics Memory

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

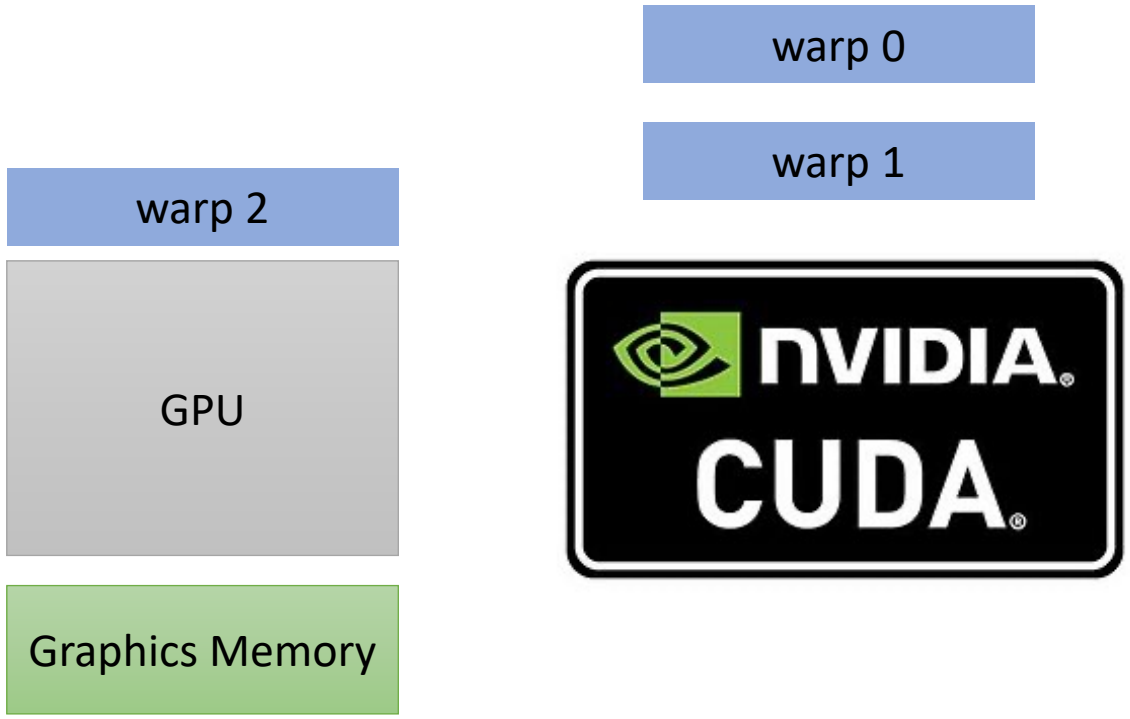
warp 2

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

warp 2

GPU

Graphics Memory

warp 0

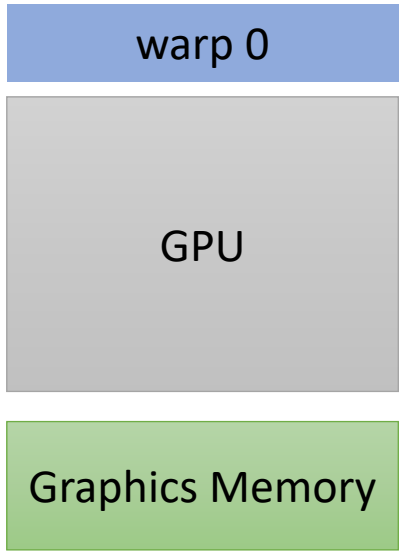
warp 1

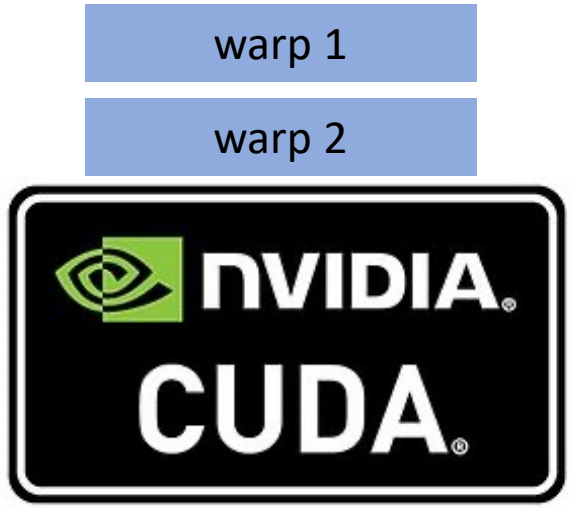
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!





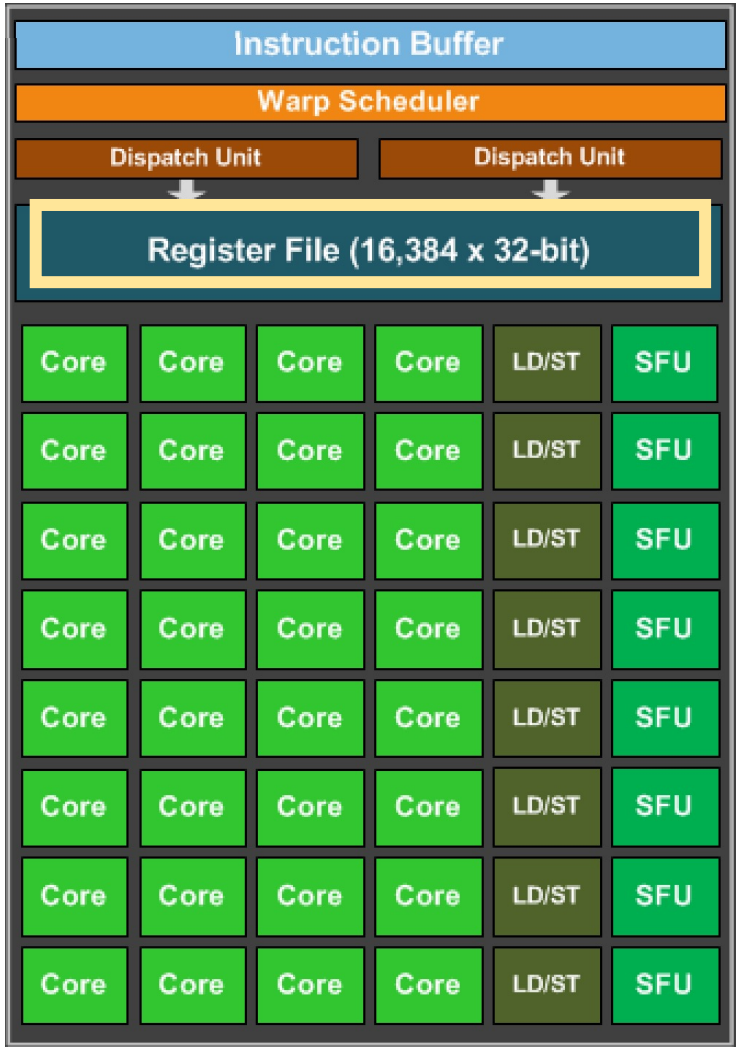
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

But wait, I thought preemption was expensive?

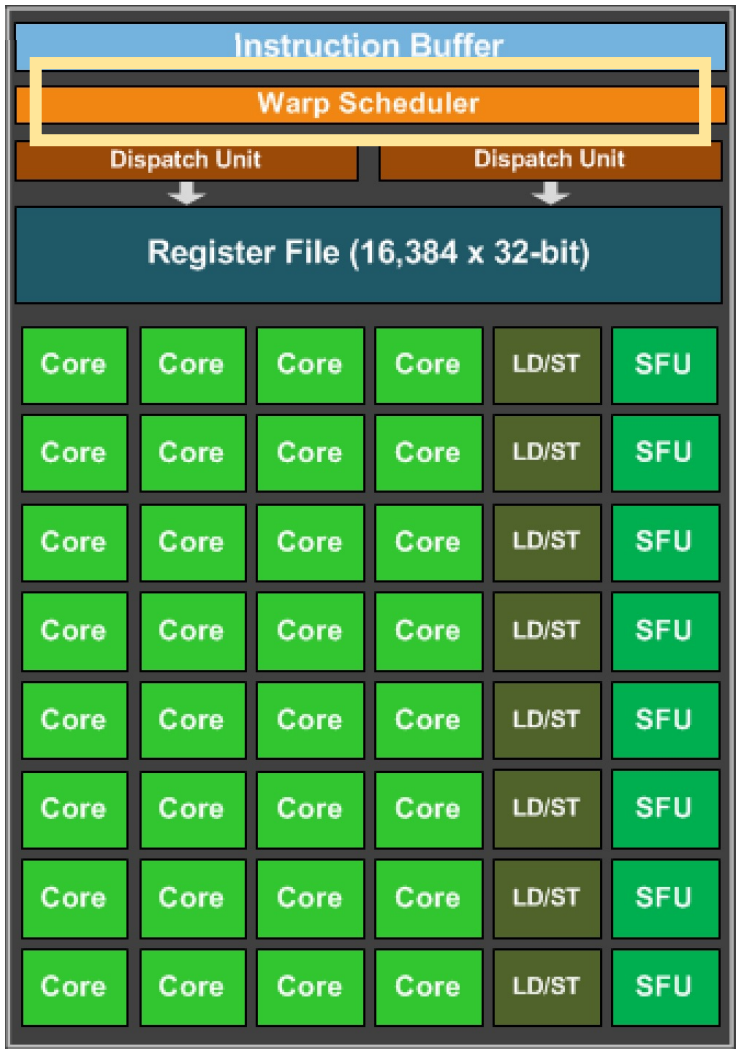
Preemption and concurrency?



But wait, I thought preemption was expensive?

Registers all stay on chip

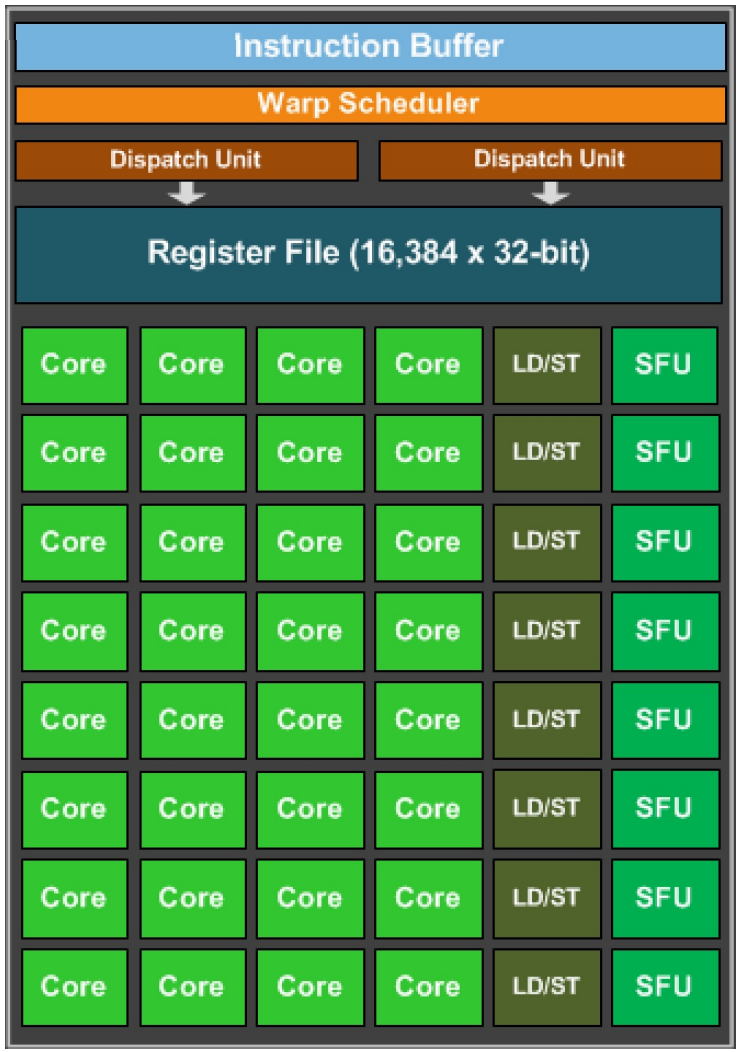
Preemption and concurrency?



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!

See you on Wednesday!

- We will continue optimizing the GPU program!

- Get started on HW 5!