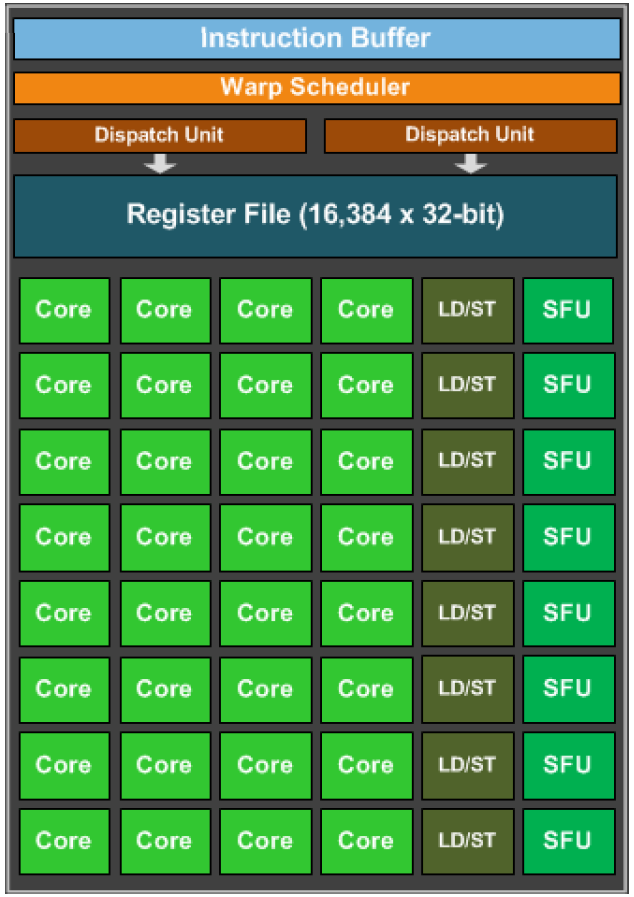
CSE113: Parallel Programming

- **Topics**:
  - Intro to GPUs



Announcements

- Midterm graded.
  Tips for improving understanding going into the final:
  - Attend class, study the slides
  - Do readings
  - Experiment with homework
  - Focus on understanding and problem solving

Announcements

- HW2 grades announced by the end of this week.
- HW3 due today (+ 3 days if you need it)

Announcements

- Rest of the quarter schedule, so that we can make it through
- Module 4: GPU for three sessions
- Module 5: advanced topics – Concurrent sets, barriers and memory models

Previous quiz + Review

Previous quiz + Review

A DOALL Loop must have:

- ◯ A loop variable that starts at 0 and is incremented by 1
- ◯ loop iterations that are independent
- ◯ be unrolled and interleaved
- ◯ not access any memory locations

Previous quiz + Review

A DOALL Loop must have:

- A loop variable that starts at 0 and is incremented by 1
-> loop iterations that are independent
- be unrolled and interleaved
- not access any memory locations

Previous quiz + Review

Which one of the following is NOT a drawback of a global workstealing parallel schedule

- ◯ Requires a concurrent data structure
- ◯ Contention on shared cache lines
- ◯ Contention on a single location with RMWs

Previous quiz + Review

Which one of the following is NOT a drawback of a global workstealing parallel schedule

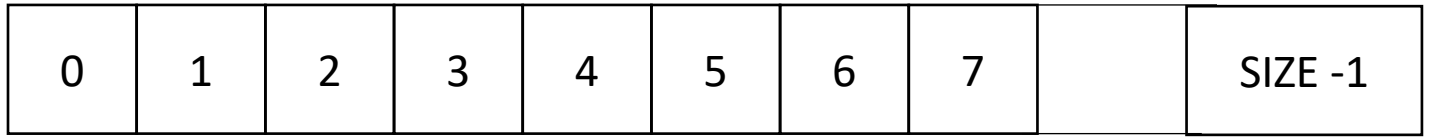
-> ◯ Requires a concurrent data structure

◯ Contention on shared cache lines

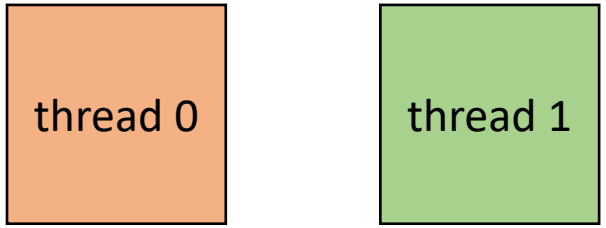
◯ Contention on a single location with RMWs

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

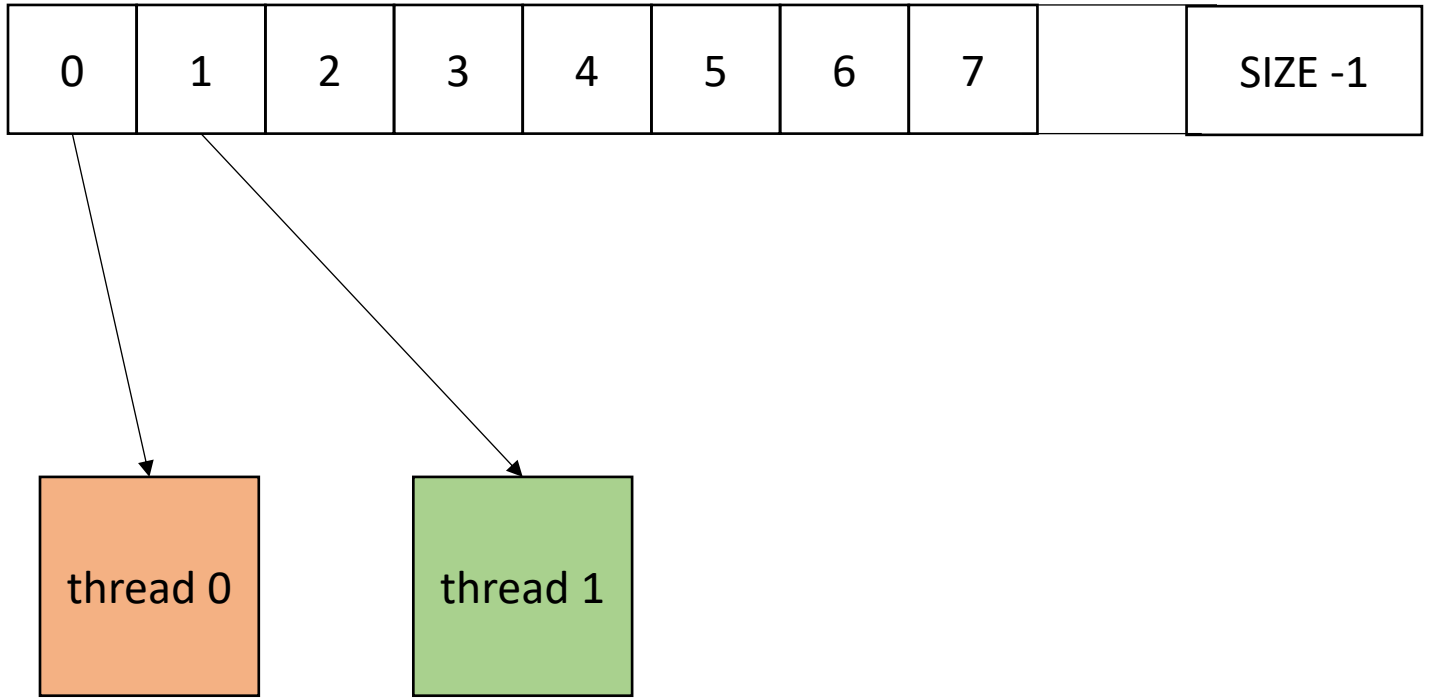


cannot color initially!



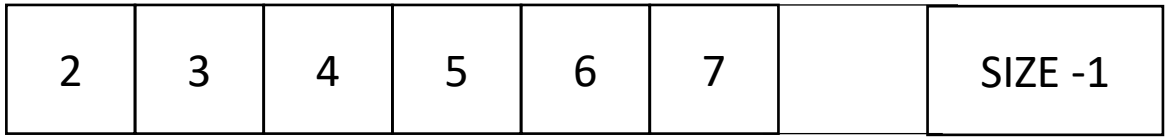
Work stealing - global implicit worklist

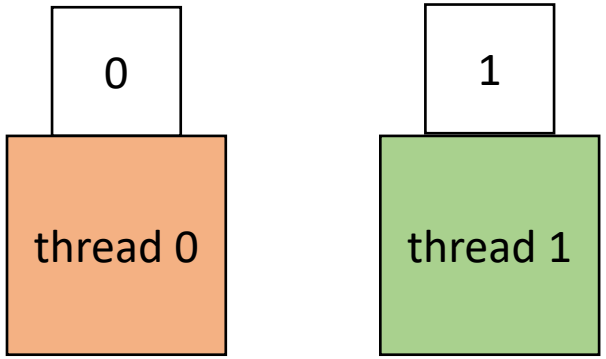
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

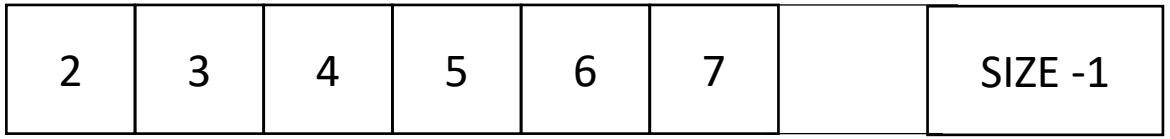
- Global worklist: threads take tasks (iterations) dynamically

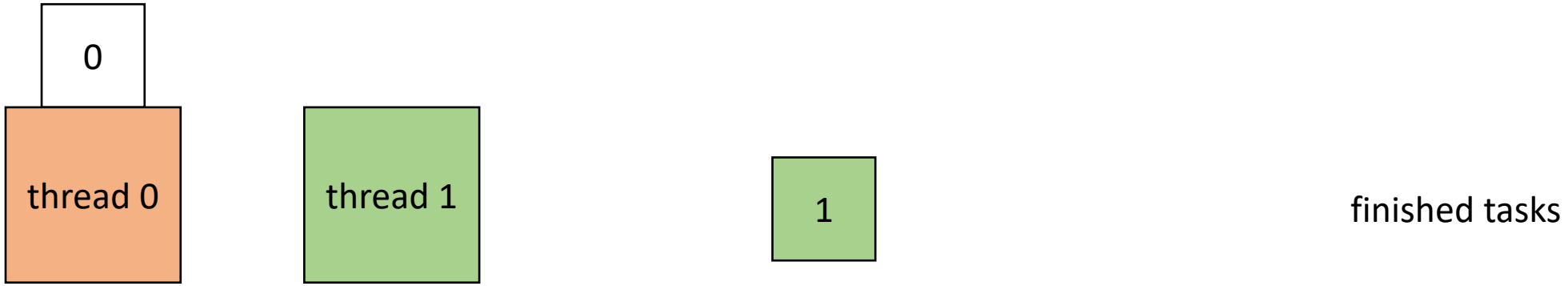




Work stealing - global implicit worklist

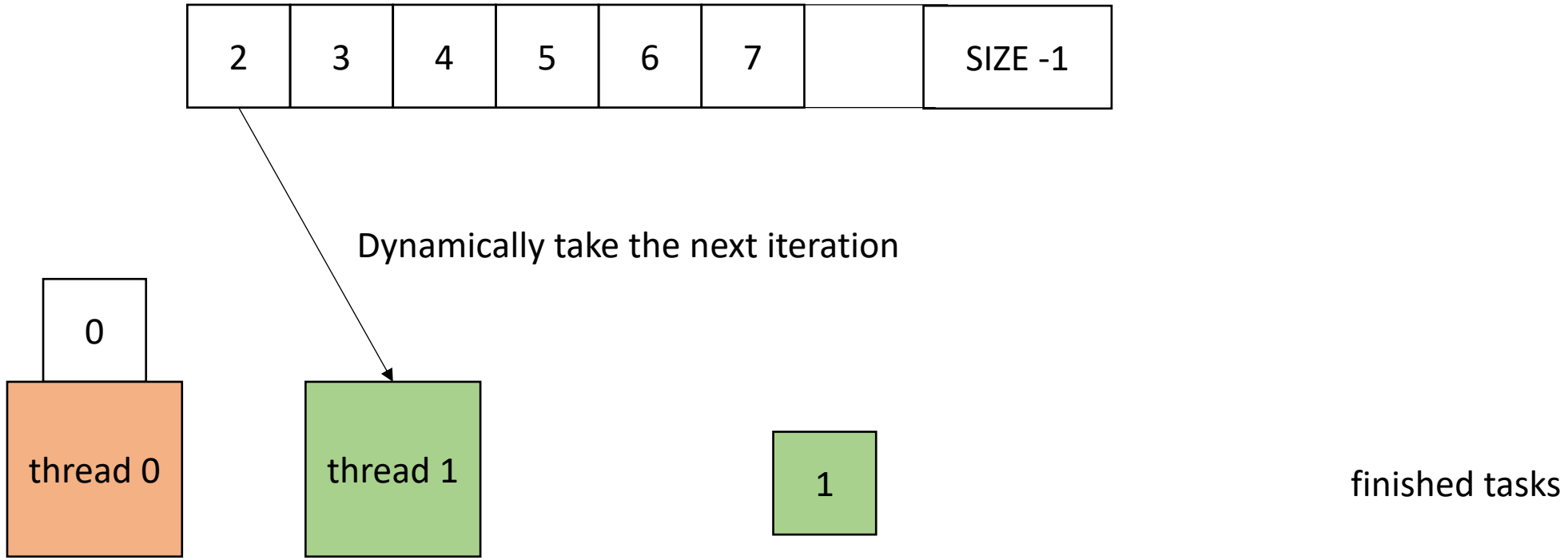
- Global worklist: threads take tasks (iterations) dynamically





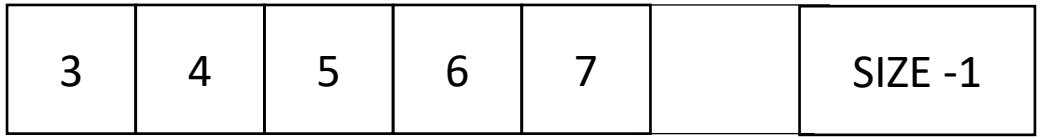
Work stealing - global implicit worklist

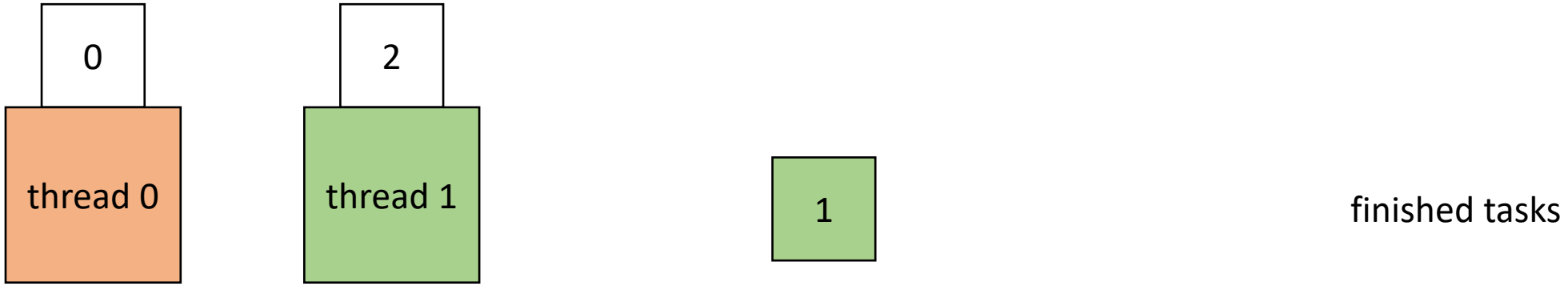
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

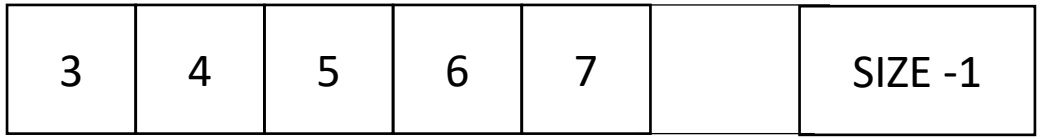
- Global worklist: threads take tasks (iterations) dynamically

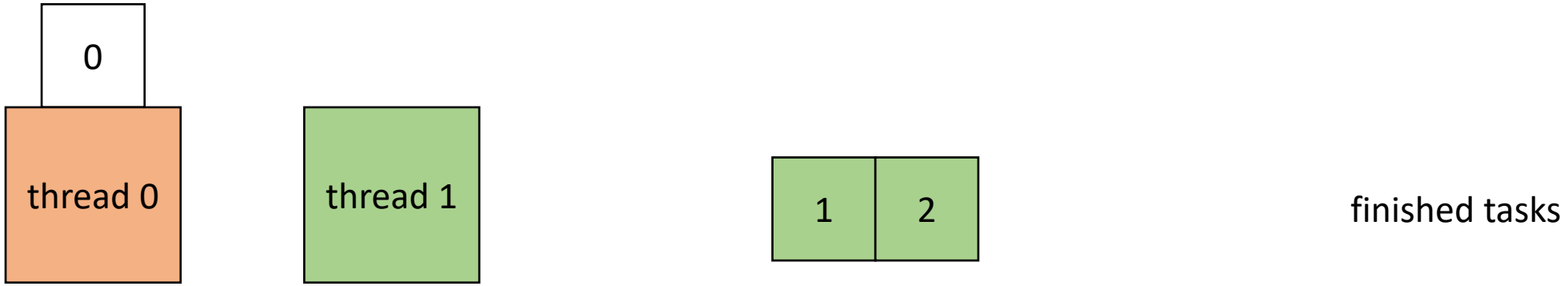




Work stealing - global implicit worklist

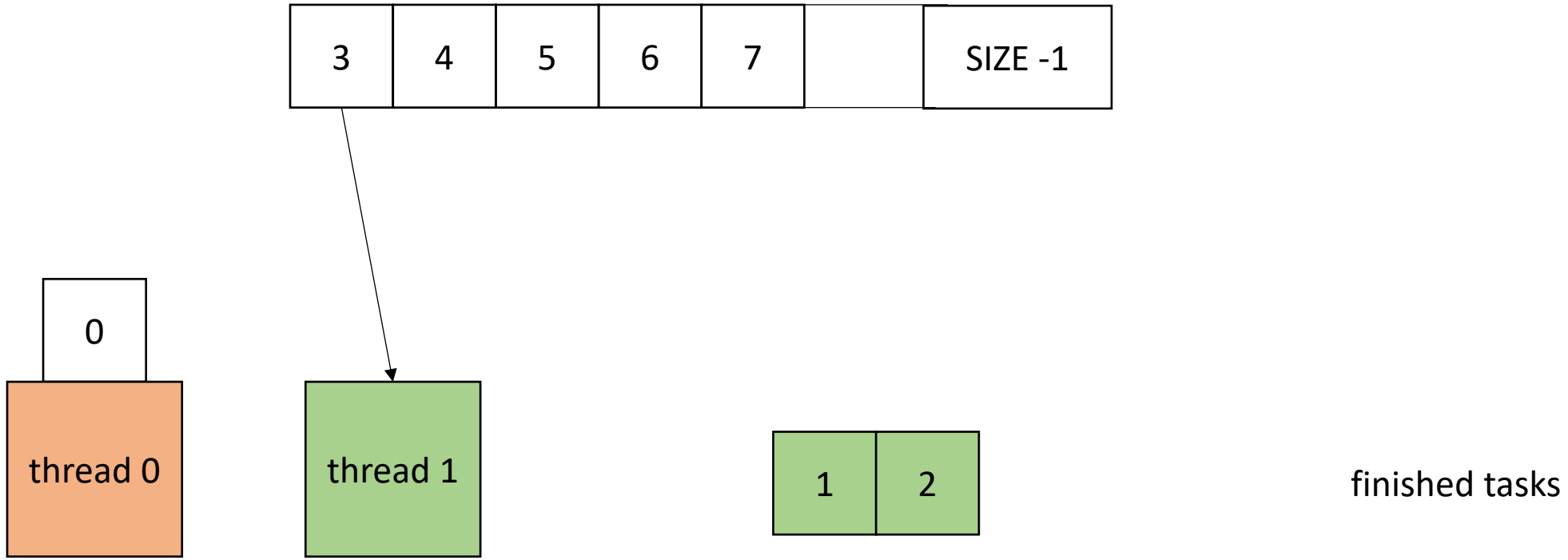
- Global worklist: threads take tasks (iterations) dynamically





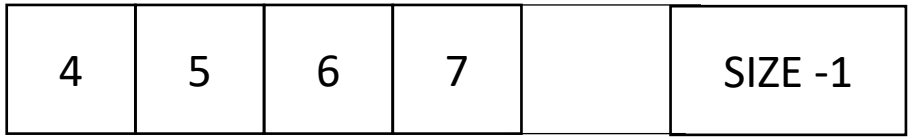
Work stealing - global implicit worklist

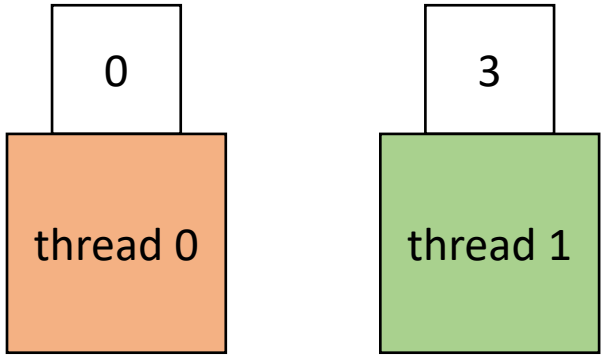
- Global worklist: threads take tasks (iterations) dynamically

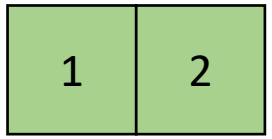


Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



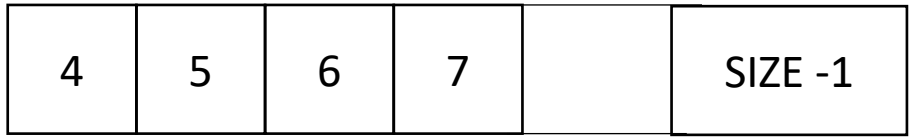


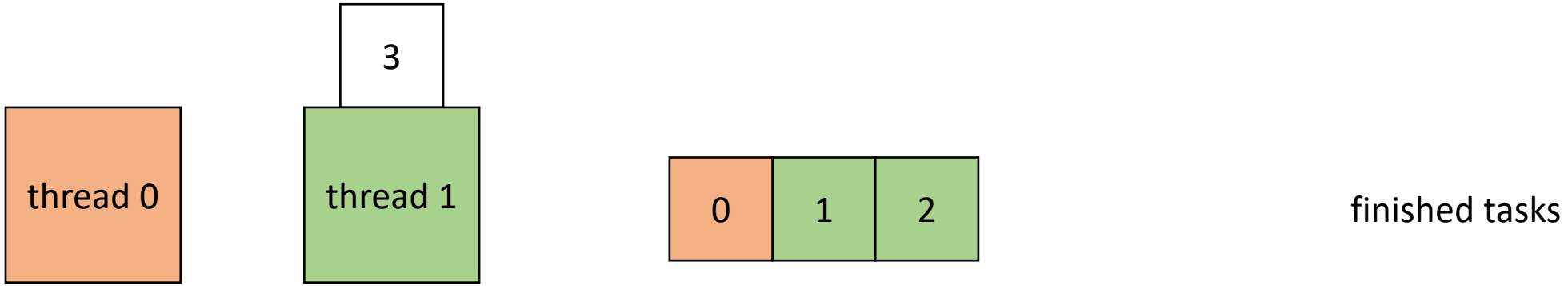


finished tasks

Work stealing - global implicit worklist

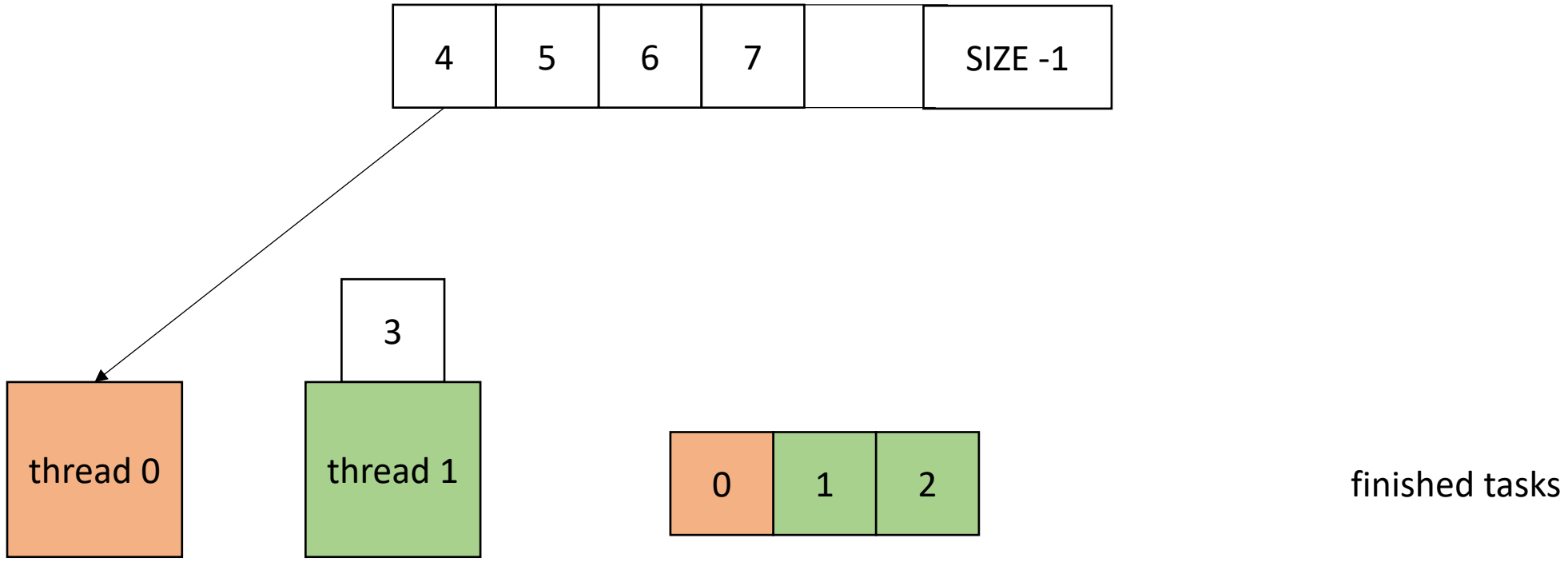
- Global worklist: threads take tasks (iterations) dynamically





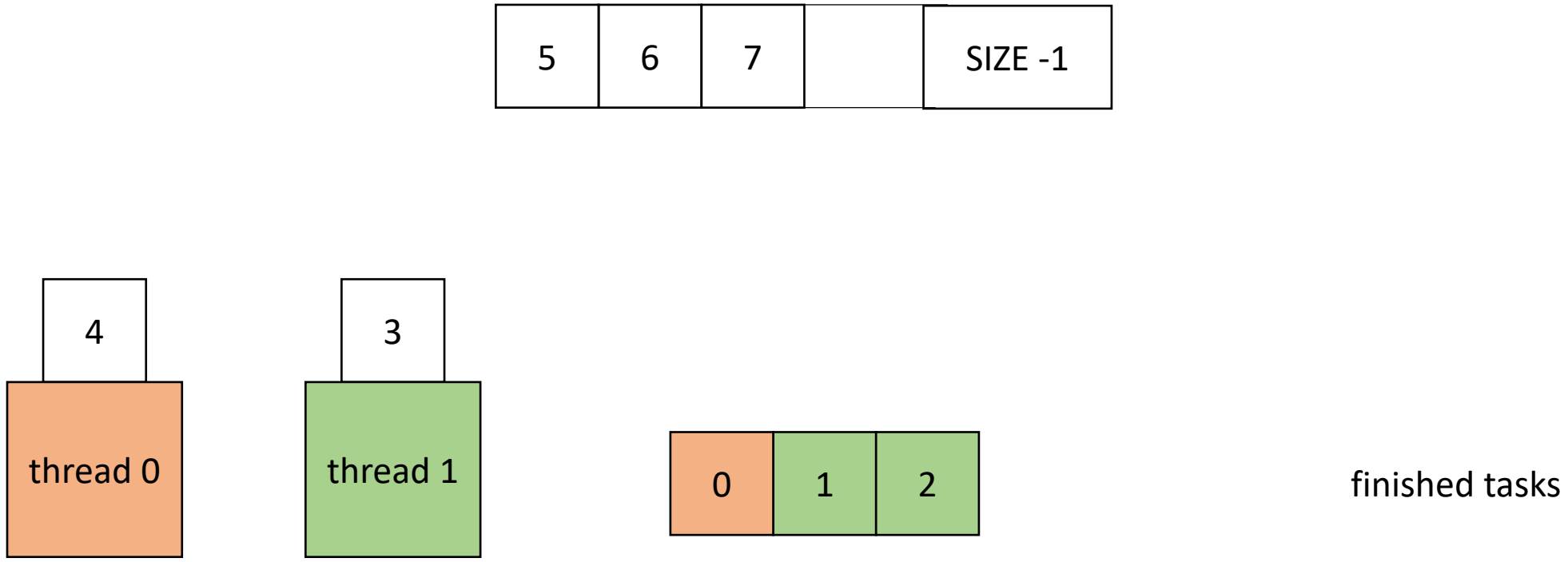
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Previous quiz + Review

Which of the following is NOT an overhead of the local worklist workstealing parallel schedule (that we studied in class)

- ◯ Initialization of the queues
- ◯ Checking a global variable to ensure all work is completed
- ◯ Managing concurrent enqueues to the worklists

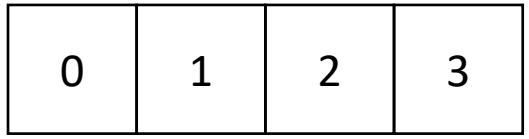
Previous quiz + Review

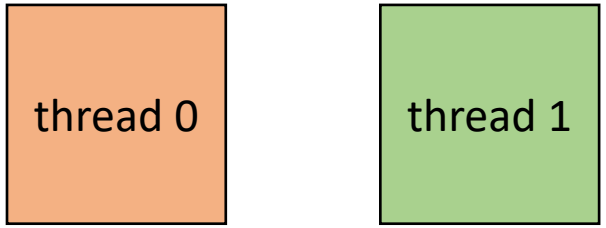
Which of the following is NOT an overhead of the local worklist workstealing parallel schedule (that we studied in class)

- ◯ Initialization of the queues
- ◯ Checking a global variable to ensure all work is completed
- -> ◯ Managing concurrent enqueues to the worklists

Work stealing - local worklists

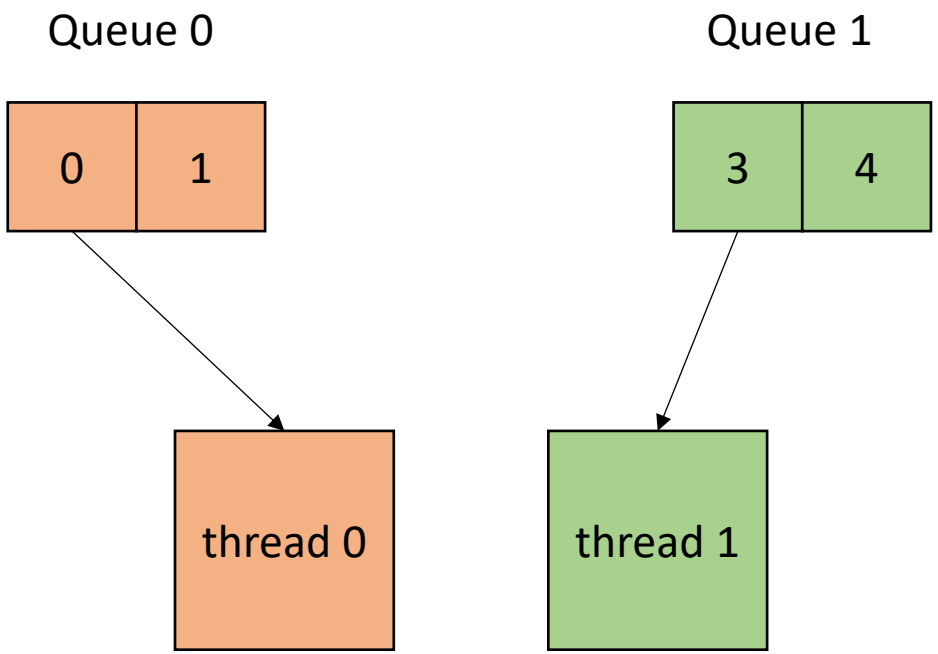
- local worklists: divide tasks into different worklists for each thread





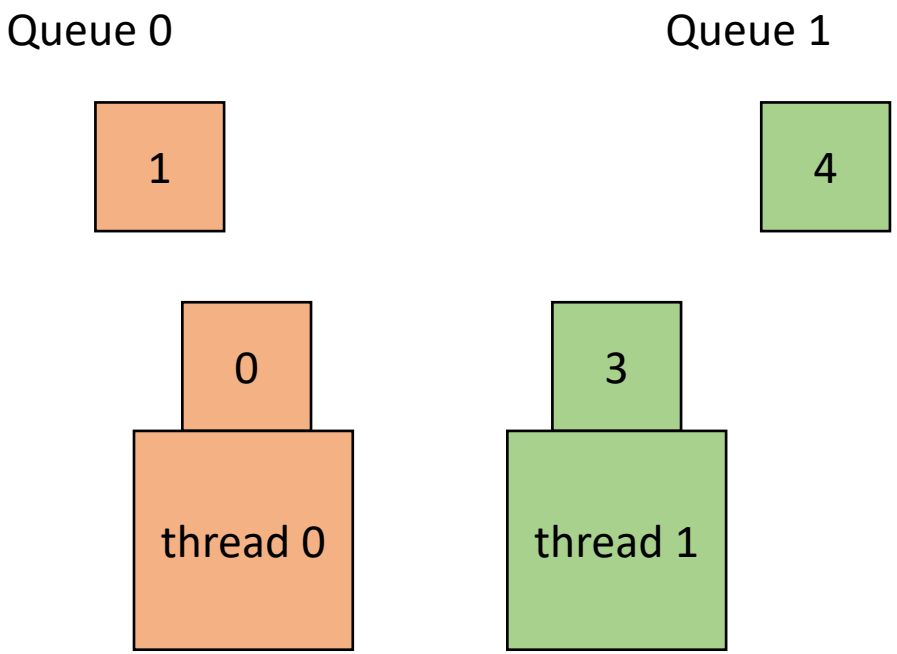
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



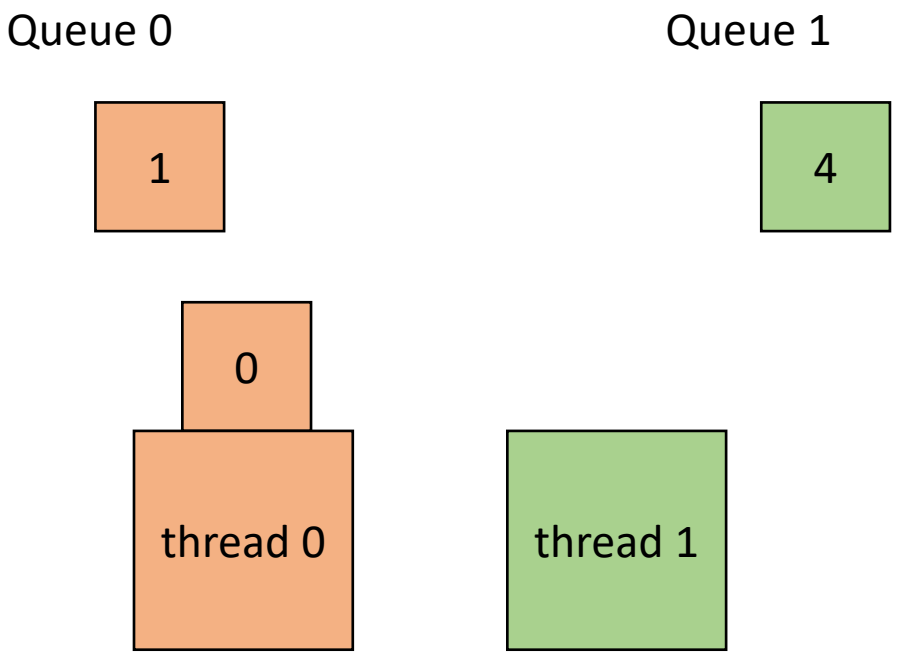
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



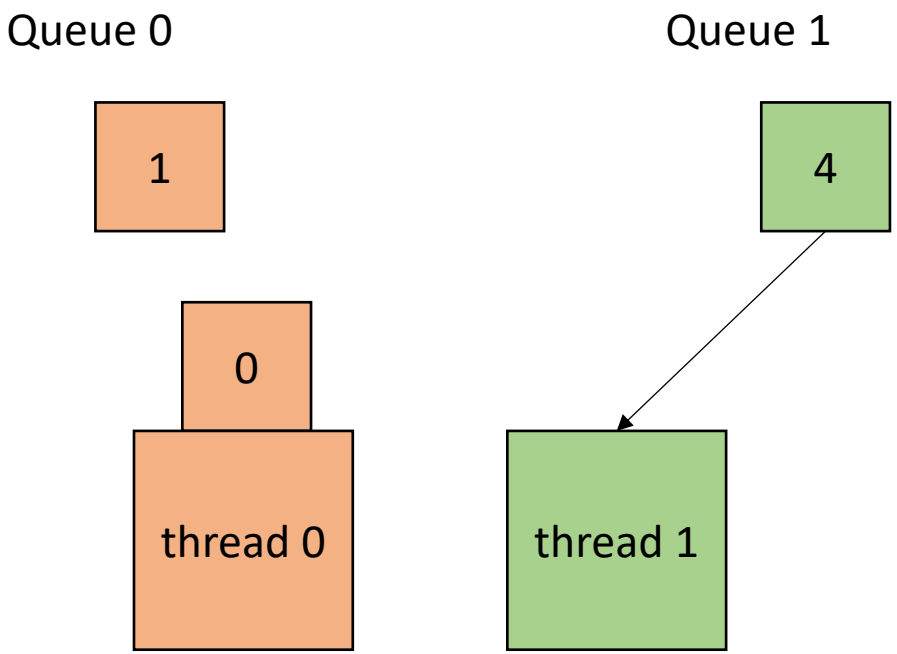
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



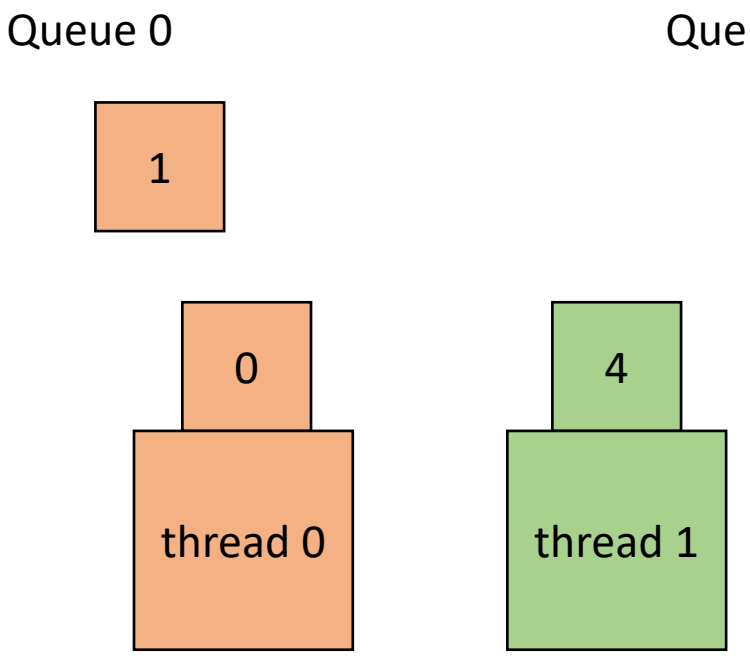
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



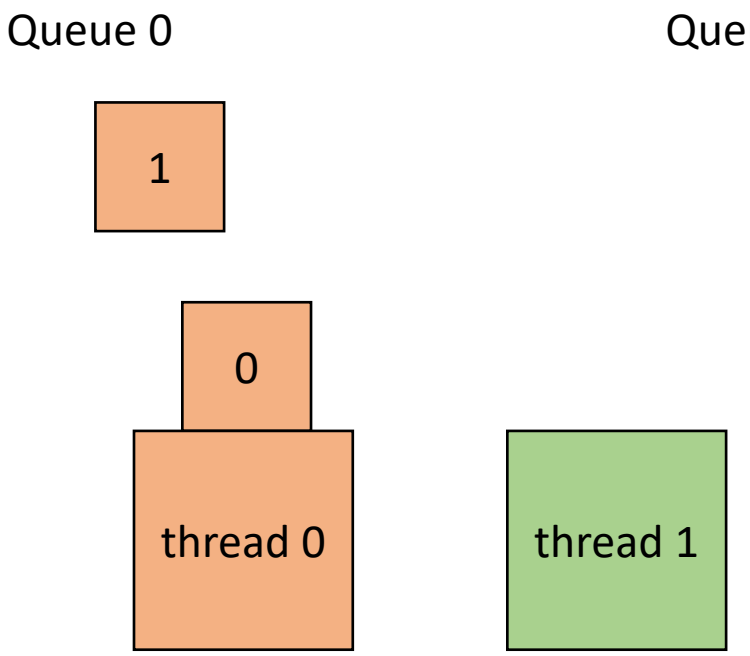
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



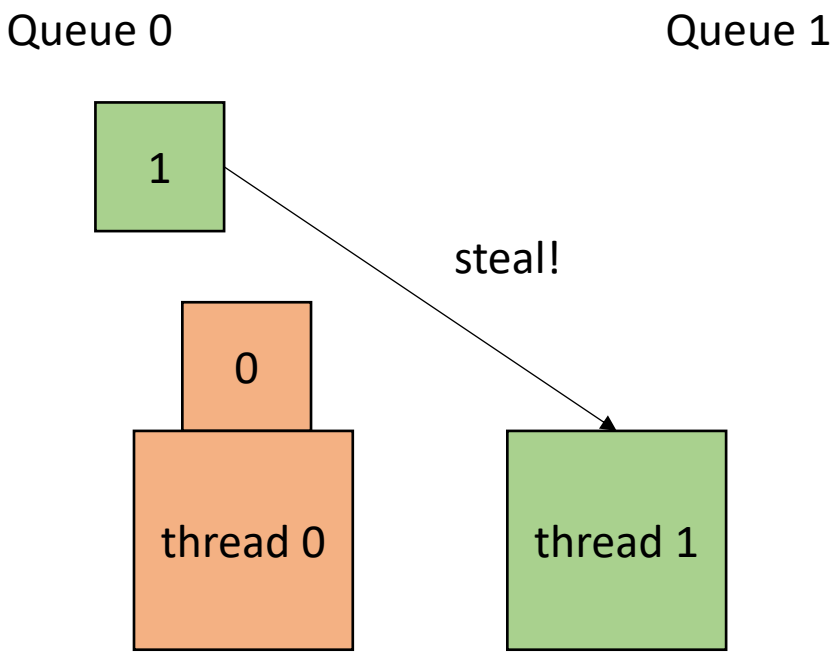
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

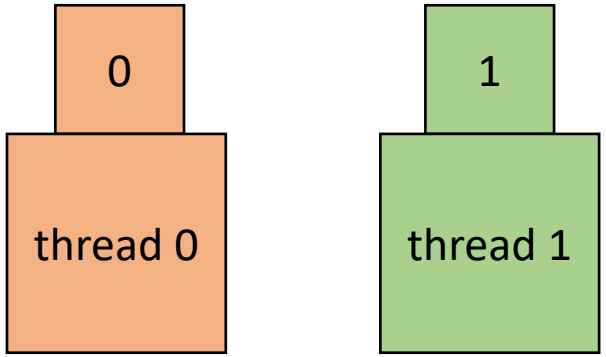


Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

Queue 0

Queue 1



Previous quiz + Review

Which of the following solutions can guarantee that a static schedule will not be out of bounds?

- ◯ The last thread always checks to get the minimum between the end of the array or the value allocated
- ◯ The last thread always get the end of the array
- ◯ The last thread never receives more than N tasks

Previous quiz + Review

Which of the following solutions can guarantee that a static schedule will not be out of bounds?

-> ◯ The last thread always checks to get the minimum between the end of the array or the value allocated

◯ The last thread always get the end of the array

◯ The last thread never receives more than N tasks

Previous quiz + Review

Write a few sentences about the pros and cons of using local workstealing queues over the global implicit worklist

Previous quiz + Review

Write a few sentences about the pros and cons of using local workstealing queues over the global implicit worklist

Simple vs. More efficient

Previous quiz + Review

Given what we've learned: what role do you believe the compiler should play in parallelizing DOALL loops?

For example, should it: (1) identify them? (2) parallelize them? (3) pick a parallel schedule?

There is no right or wrong answer here, but it is interesting to think about!

On to new stuff!

- GPUs

GPUs: a brief history

First chapter of CUDA by Example
https://www.techspot.com/article/650-history-of-the-gpu/

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

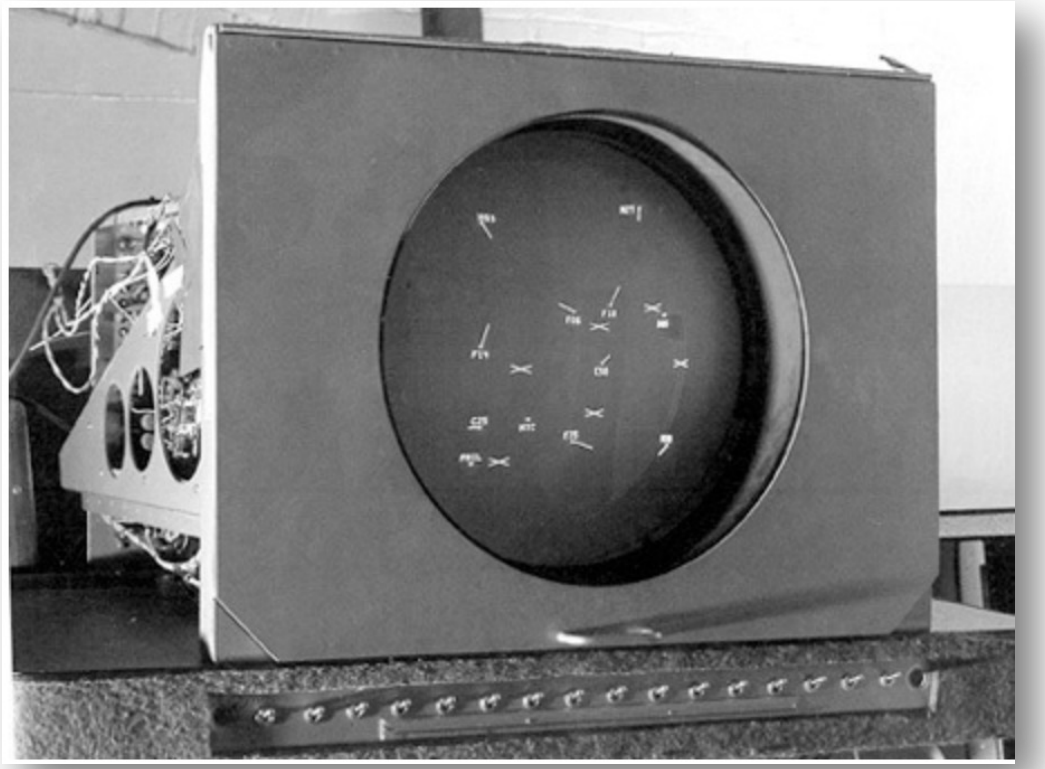


Image from: https://ohiostate.pressbooks.pub/graphicshistory/chapter/2-1-whirlwind-and-sage/

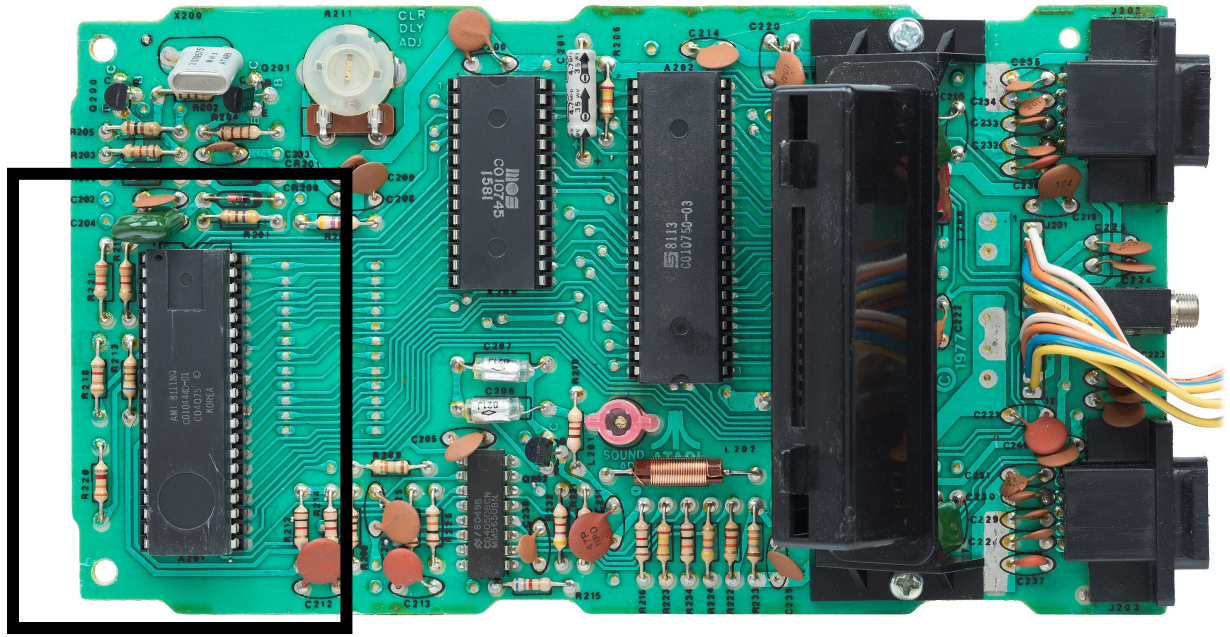
Specialization

- 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 for CPUs 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

River Raid Game

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

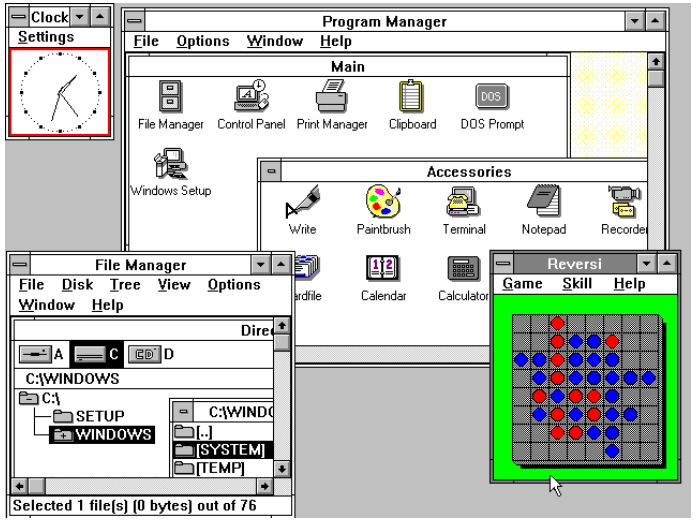
https://en.wikipedia.org/wiki/Microsoft_Windows

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

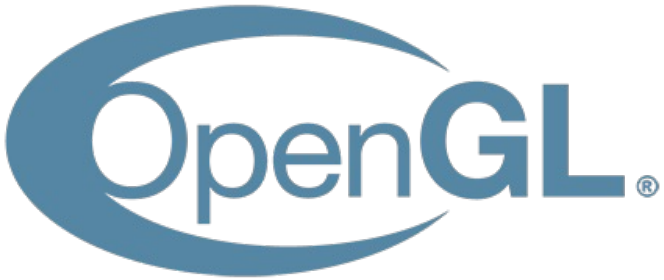
OS integration

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

- New APIs started appearing, to enable GUI programs



Windows 3 (1990)



1992

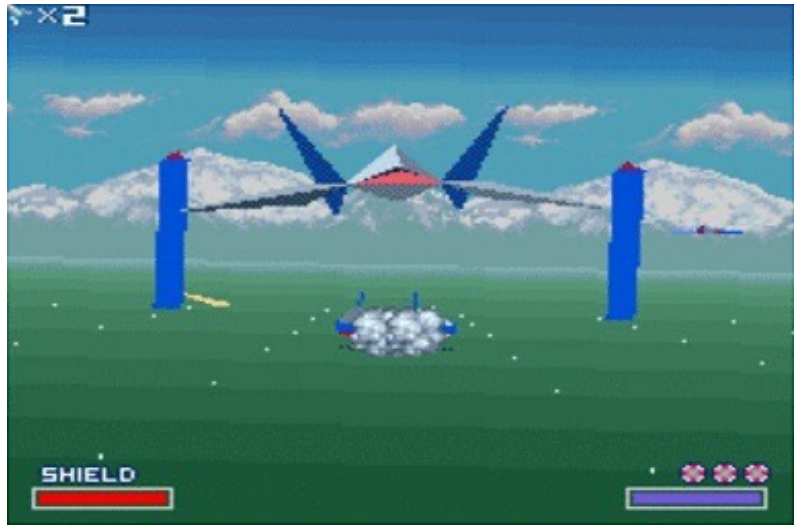
1995

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 (Super Mario developer) really wanted a 3D flight simulator though
- Worked with a British software company to develop...



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

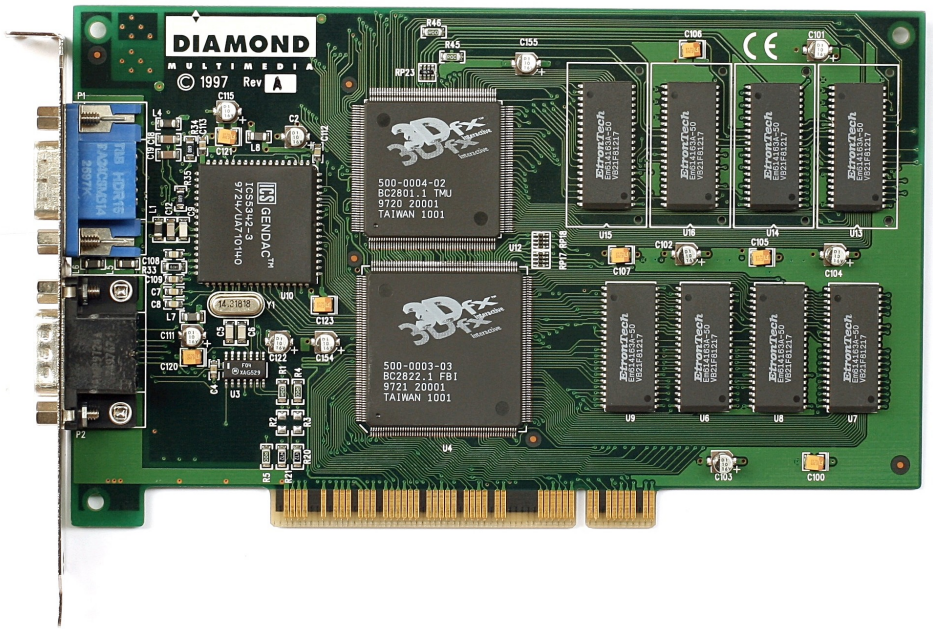
3D graphics in consoles (1993)

- 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 (acquired by AMD)
  - 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
  - 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 to counter Microsoft's DirectX:
  - 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 big for compute
  - Vulkan unifies compute + graphics (mostly graphics)

- Apple deprecated Khronos group standards to support Metal in 2018
  - iPhone used to have GPUs from a company called Imagination, a British company. Apple took their engineers.
  - Our laptops may have GPUs from Intel or ADM. Now apple laptops has apple GPUs.

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

Where are we now?

- Nvidia CUDA is widely used, driving many HPC and ML applications
  - Nvidia heavily invested in its software stack and made it accessible, and this made it widely used.
  - Nvidia GPUs are not designed power efficient, and are not used much in mobile devices.
- OpenCL is used to program other GPUs (although it is not as widely used)
- Metal is used for Apple devices
- Vulkan has momentum, but mostly in graphics

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

GPU Shortages?

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

GPUs in 2024

- Nvidia is now the 4th most valuable company in the world (1.9T)
  - More valuable than Google currently (1.7T)
- Graphics are still a valuable domain, but AI seems to be more valuable
  - Valuable to learn general purpose GPU programming (GPGPU)
- Many companies make their own GPUs:
  - AMD, Apple, Qualcomm, ARM, Intel, Imagination, Broadcom
- Many startups in the Valley making new AI chips similar to GPUs

Teaching GPU programming

- This is difficult!
- Nvidia GPUs have the most straightforward programming model (CUDA).
- It is extremely difficult to get a class of 120 students access to Nvidia GPUs these days.
  - AWS? Expensive and often oversubscribed w.r.t. GPUs
  - Department? ML folks get priority and super computing clusters are painful

Plan

- The GPU programming lectures will use CUDA
  - It is widely used
  - The programming model is straightforward

- Homework will use WebGPU, because it is widely supported
  - *There are more non-Nvidia GPUs in this room than Nvidia GPUs*

Plan

- The homework uses Javascript as its ”CPU” language, and WebGPU as its ”GPU” language.

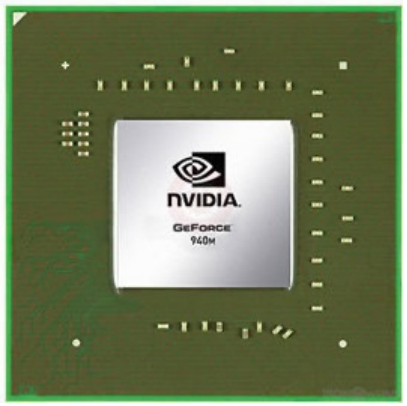
- We have provided generous skeletons for the homework. We can go over some javascript, but it is a high-level language and should not be hard to figure out what you need to do.

- The WebGPU portion is straight forward and I will provide a mapping directly from what we talked about to what you need.

Start on GPU lectures

Programming a GPU

Tiny GPU in an embedded system



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

Fight!

The CPU

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/

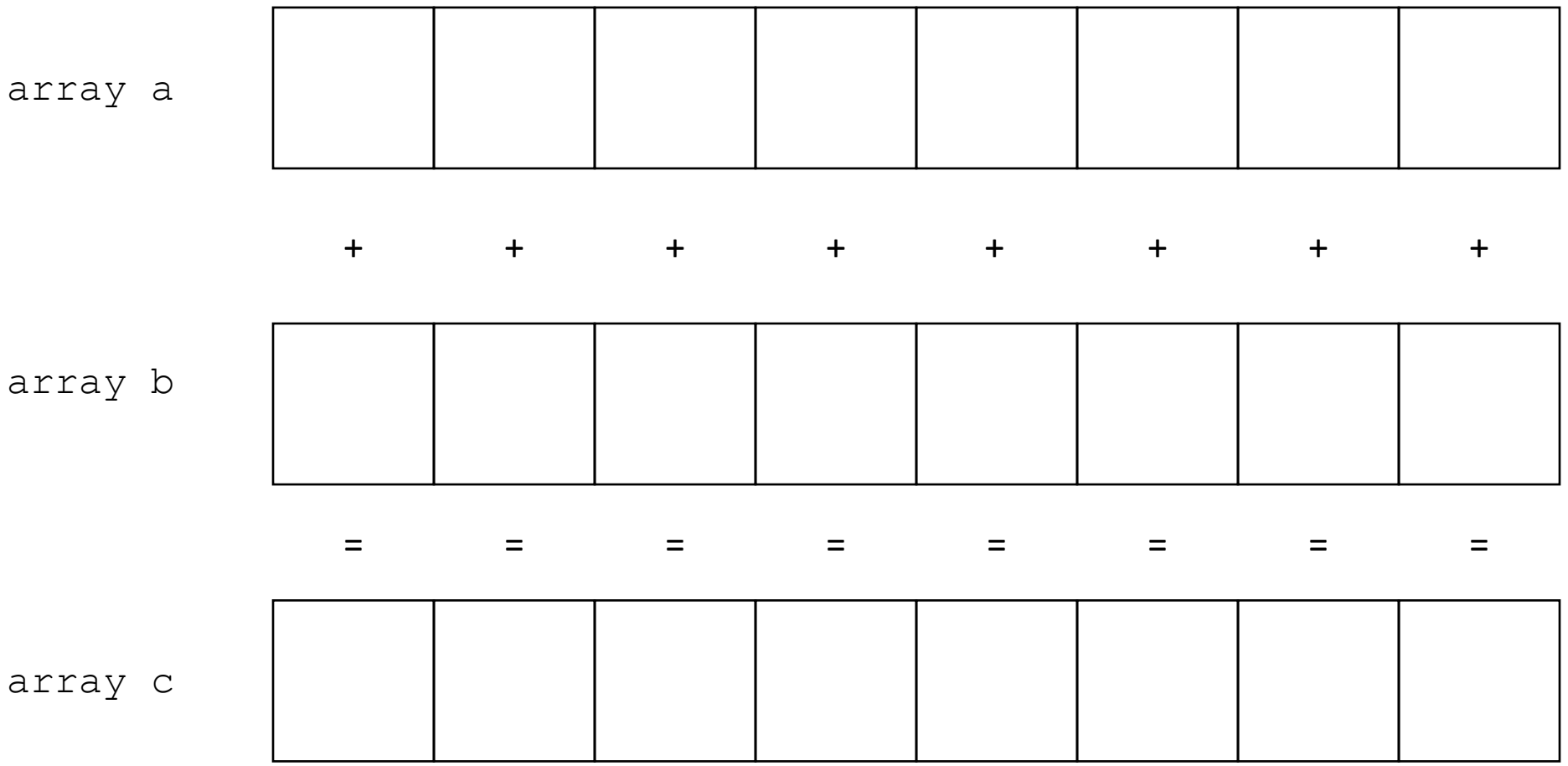
Programming a GPU

- The problem: Vector addition

Embarrassingly parallel

Computation can easily be divided into threads

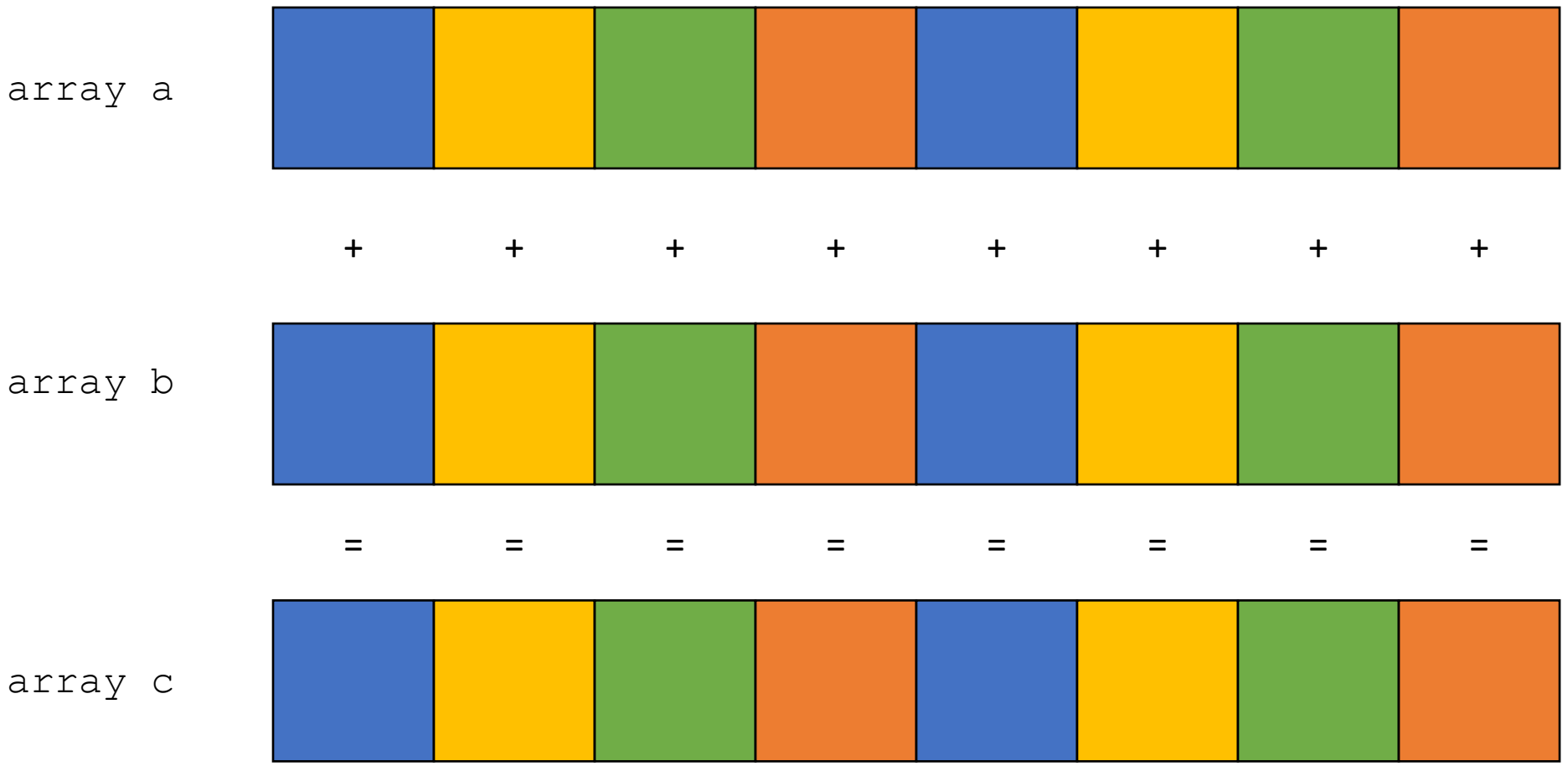
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

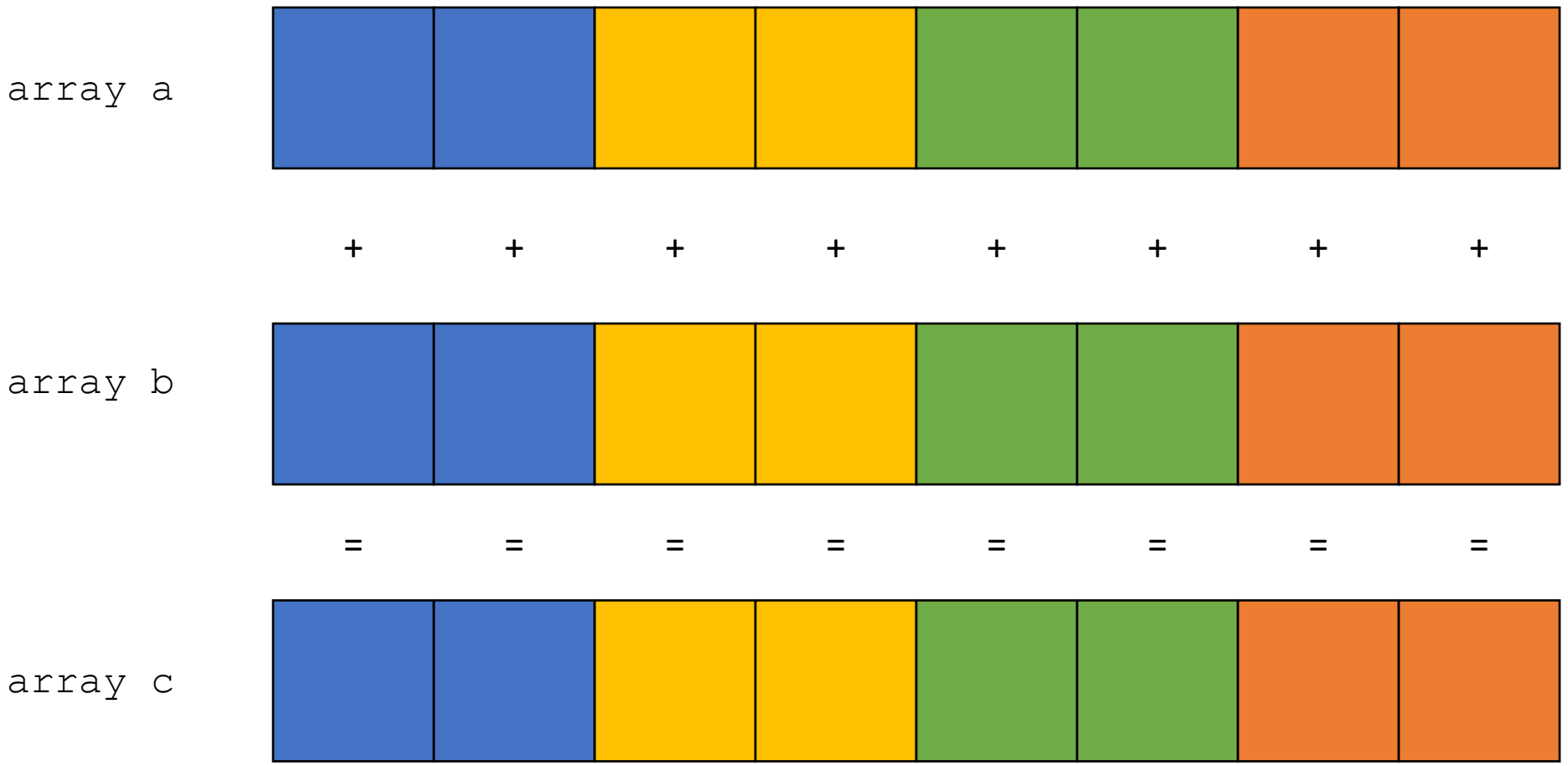
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

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

```
void vec_add(double *a, double *b, double *c, int size, int
tid, int num_threads) {
  int chunk = size / num_threads;
  int start = chunk * tid;
  int end = start + chunk;
  for (int j = 0; j < OUTER; j++) {
    for (int i = start; i < end; i++) {
      a[i] = b[i] + c[i];
    }
  }
}
```

```
for (int i = 0; i < num_threads; i++)
  thread_array[i] = thread(vec_add, a, b, c, SIZE, i,
num_threads);

for (int i = 0; i < num_threads; i++)
  thread_array[i].join();
```

Now for the GPU

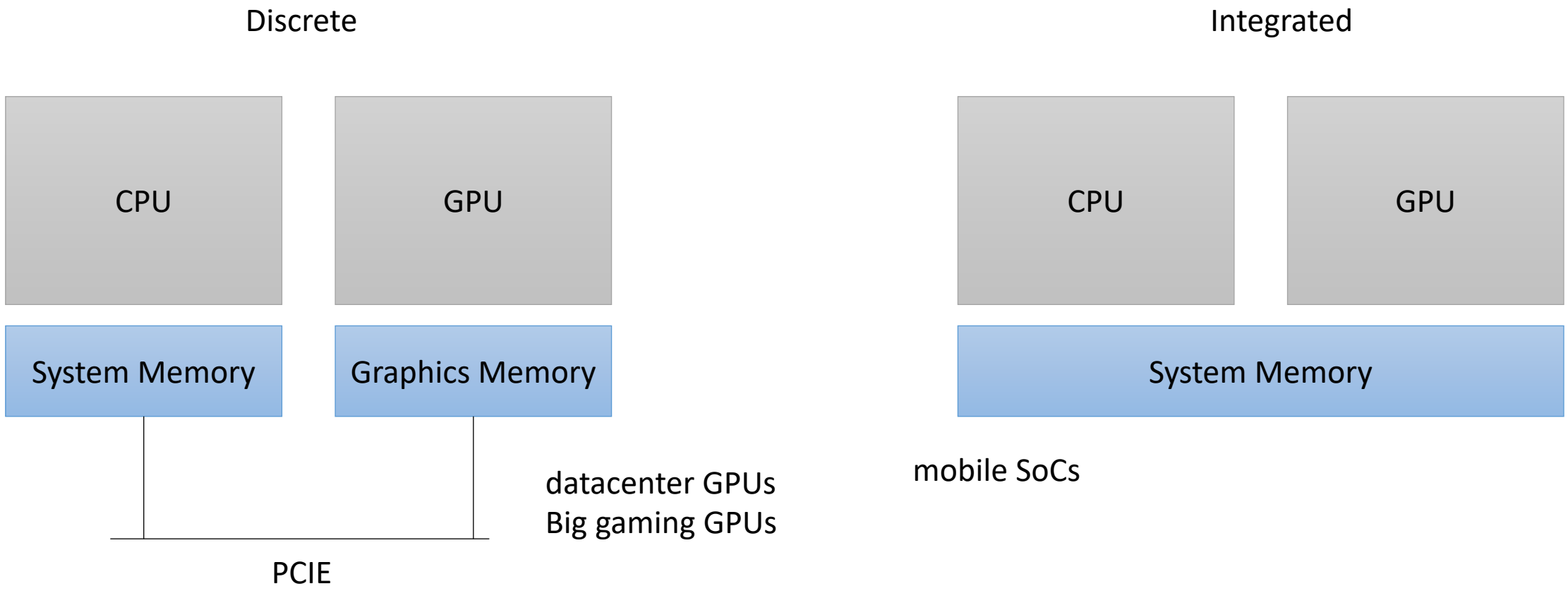
- Its going to take a bit of work....

GPU set up

- We need to allocate and initialize memory

GPU set up

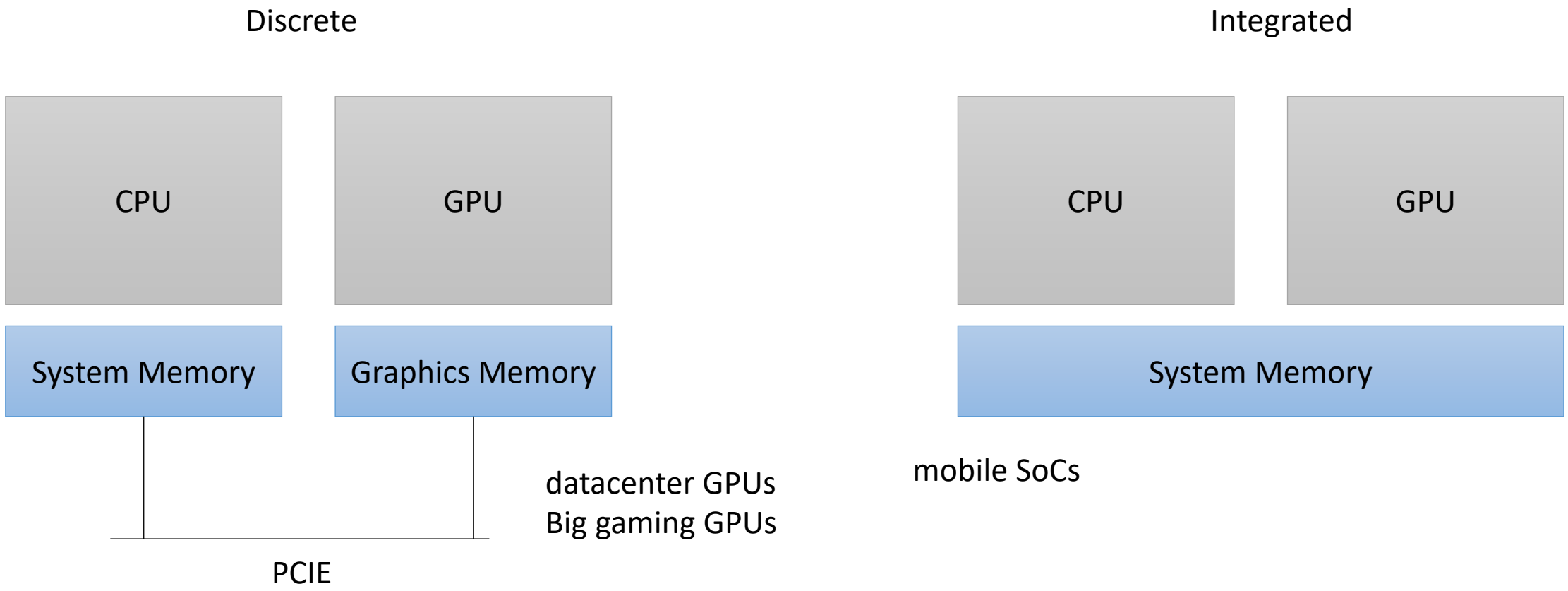
- GPUs come in two flavors



GPU set up

Pros and cons of each?

- GPUs come in two flavors

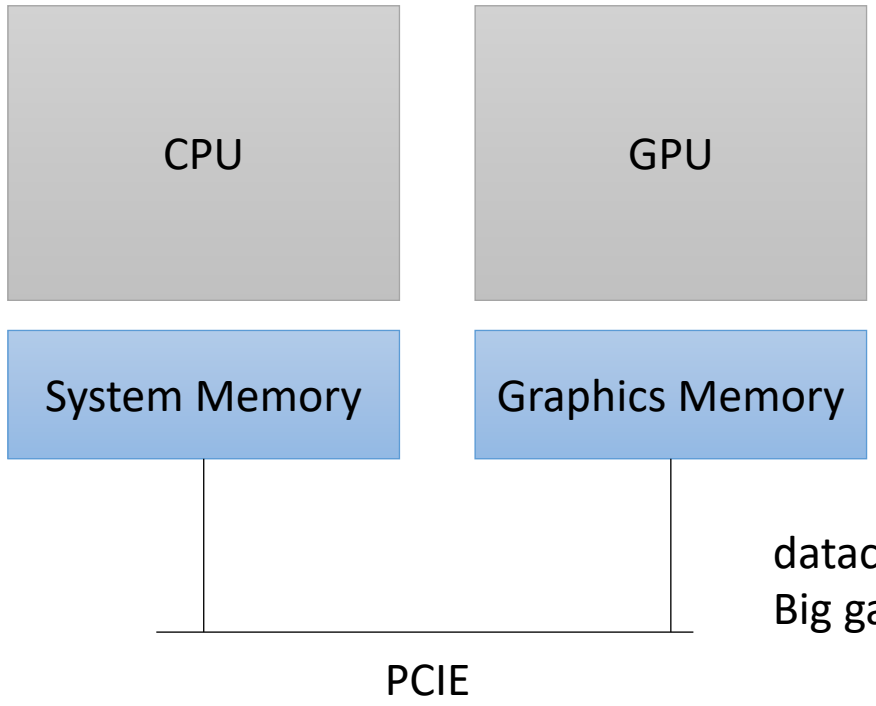


GPU set up

- GPUs come in two flavors

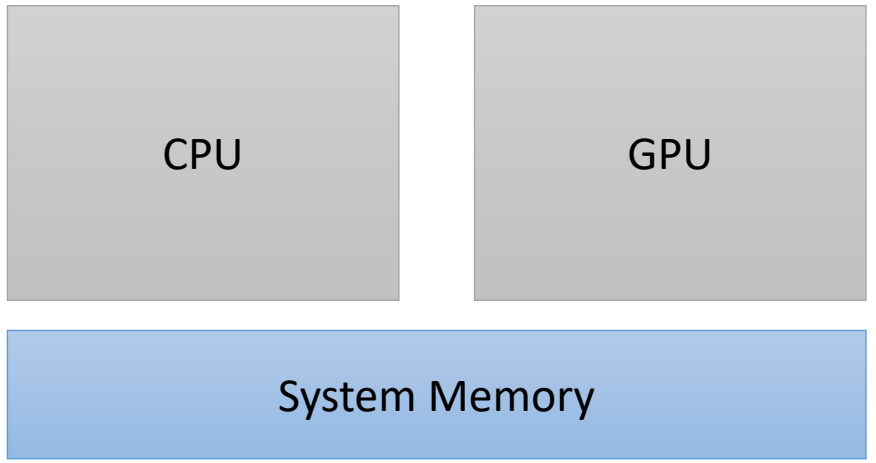
Pros and cons of each?
*Different types of memory for discrete
*Swappable for discrete
*More energy efficient for integrated
*Better memory utilization for integrated

Discrete



datacenter GPUs
Big gaming GPUs

Integrated



mobile SoCs

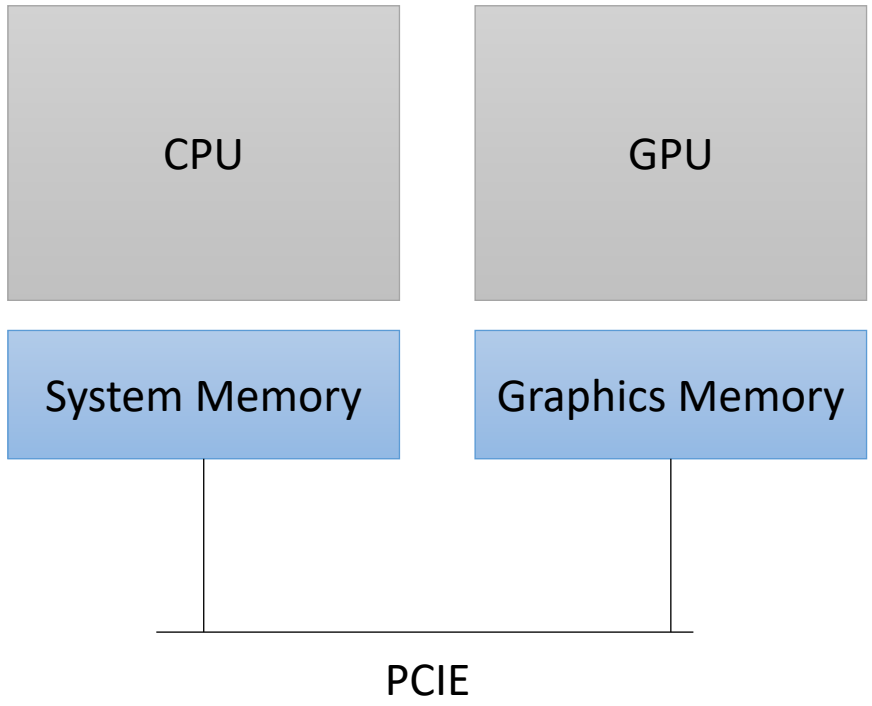
GPU set up

- GPUs come in two flavors

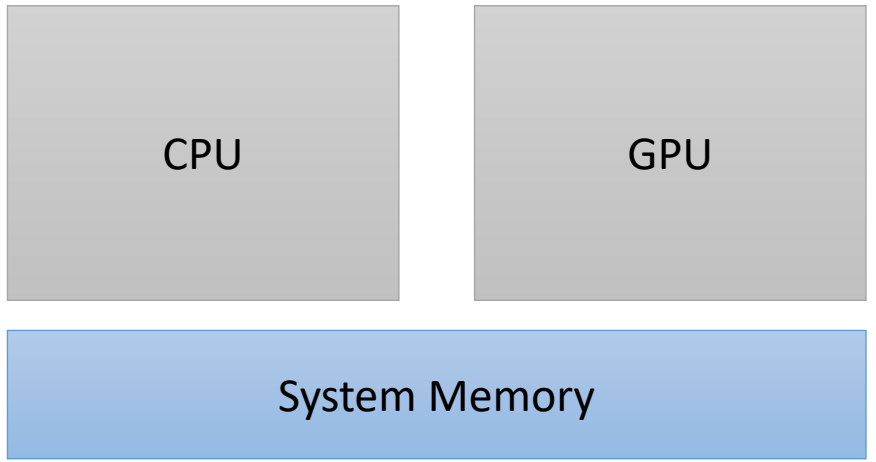
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



Integrated



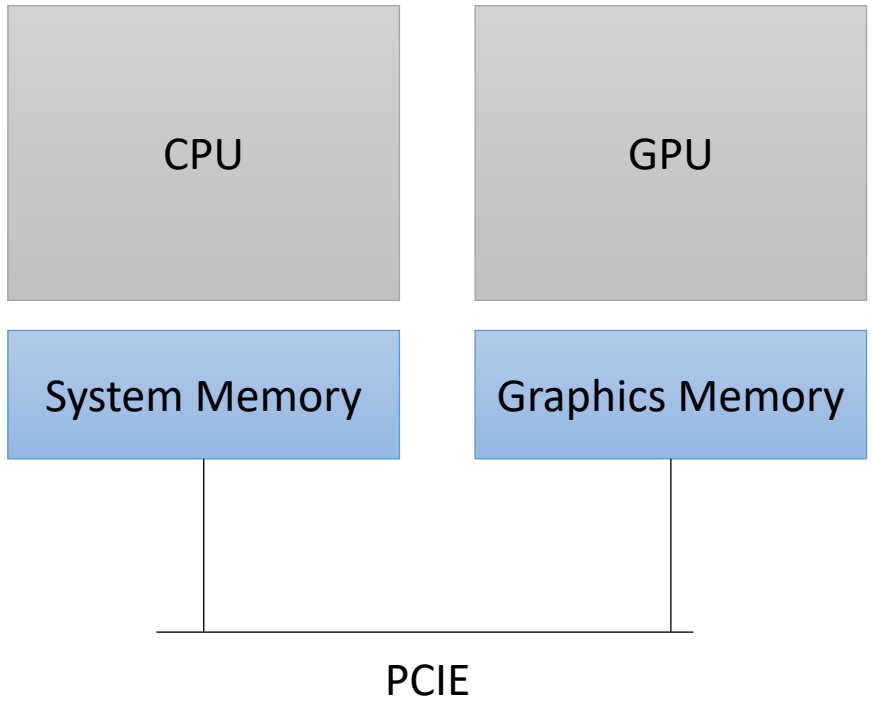
GPU set up

- GPUs come in two flavors

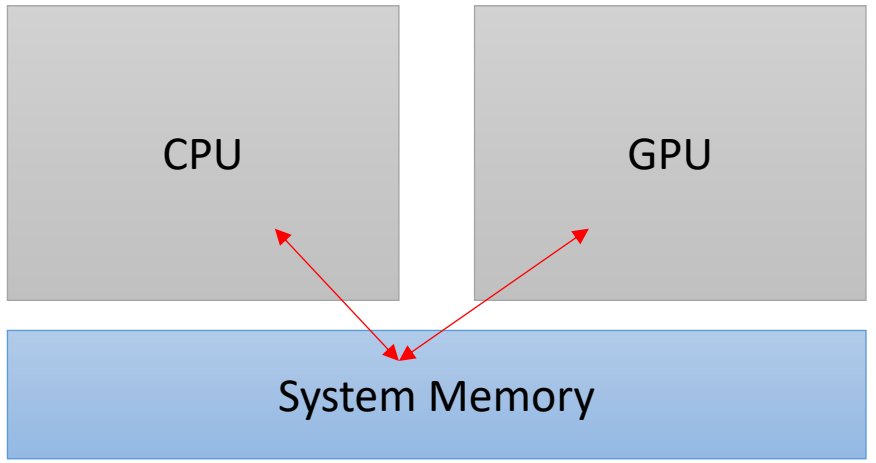
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



Integrated



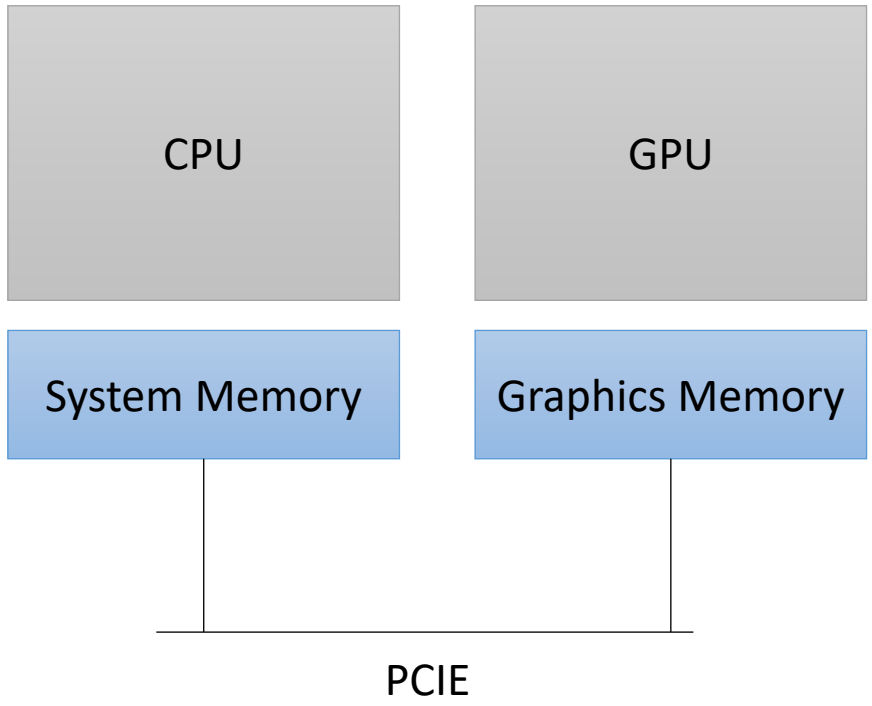
GPU set up

- GPUs come in two flavors

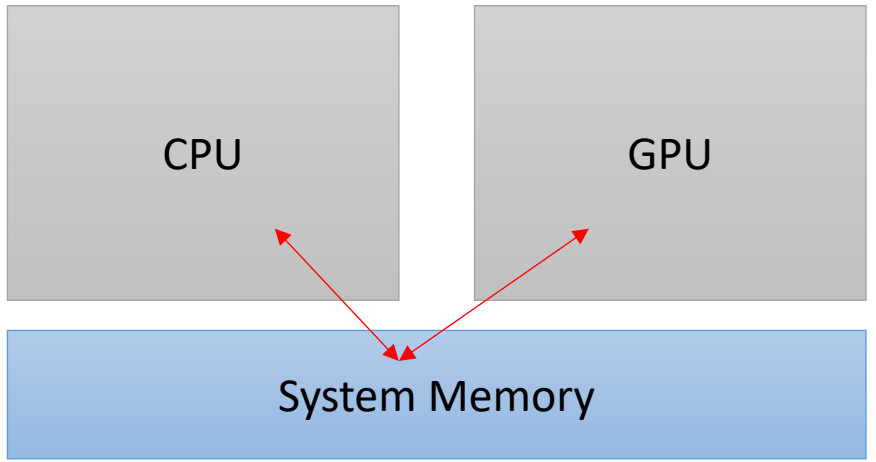
Although mobile GPUs share the system memory, Most still require you to program as if they didn't have shared memory.

Why?

Discrete



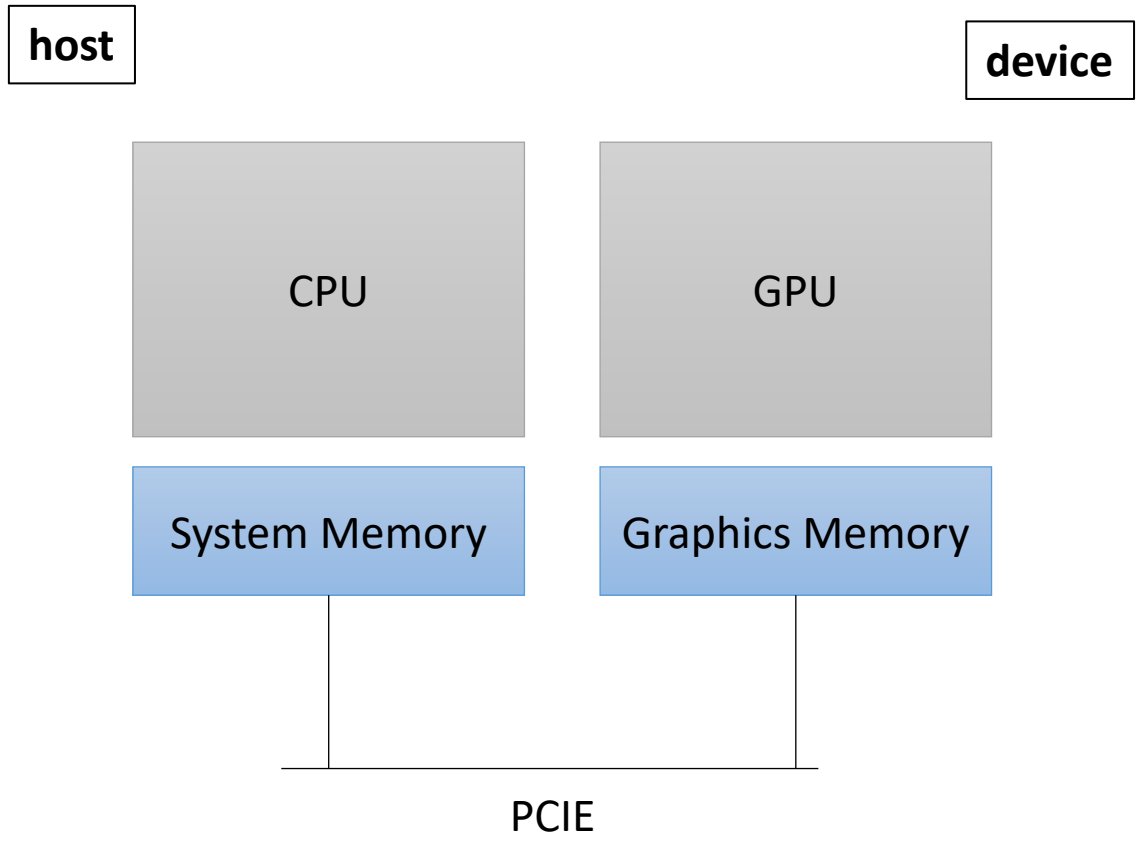
Integrated



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

GPU set up

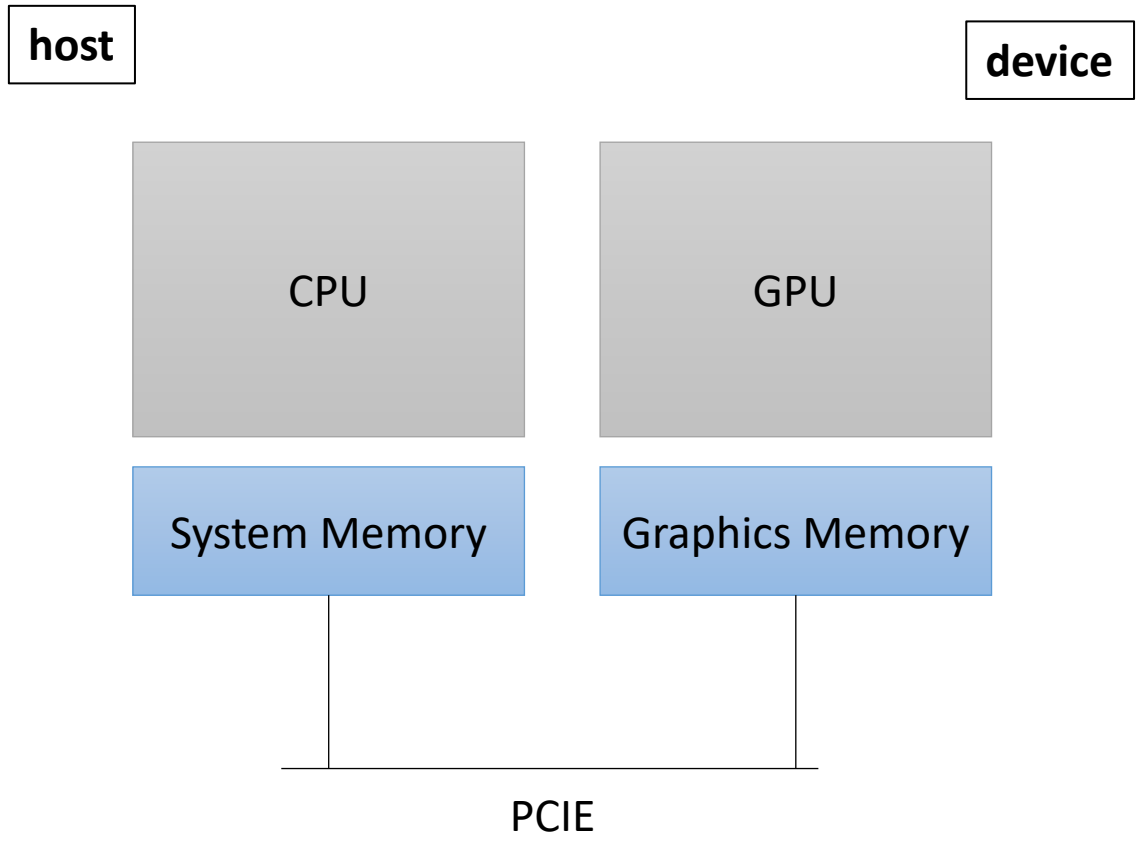
- Our heterogeneous, parallel, programming model



GPU set up

- Our heterogeneous, parallel, programming model

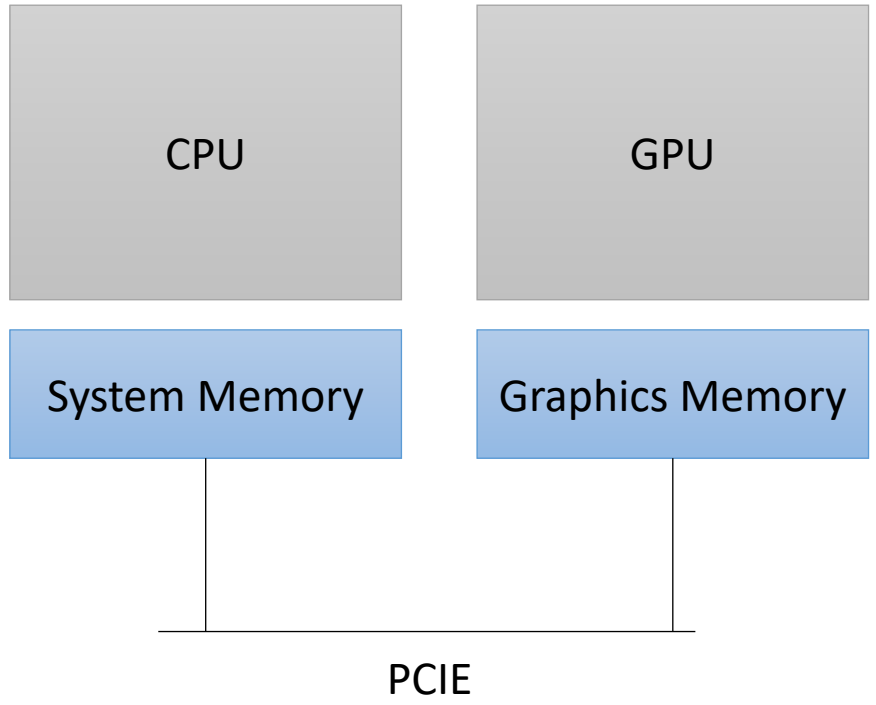
The host (CPU) will write a C++-like program that allocates and sets up memory on the GPU. The host will then call a GPU program called a kernel.



GPU set up

How do we allocate memory on a CPU?

- Our heterogeneous, parallel, programming model

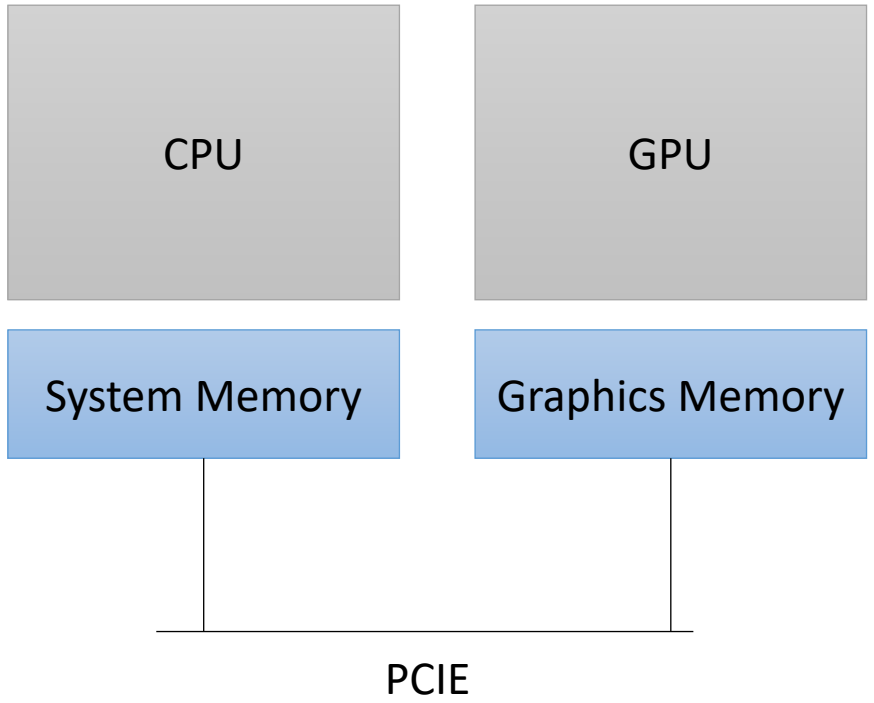


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

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

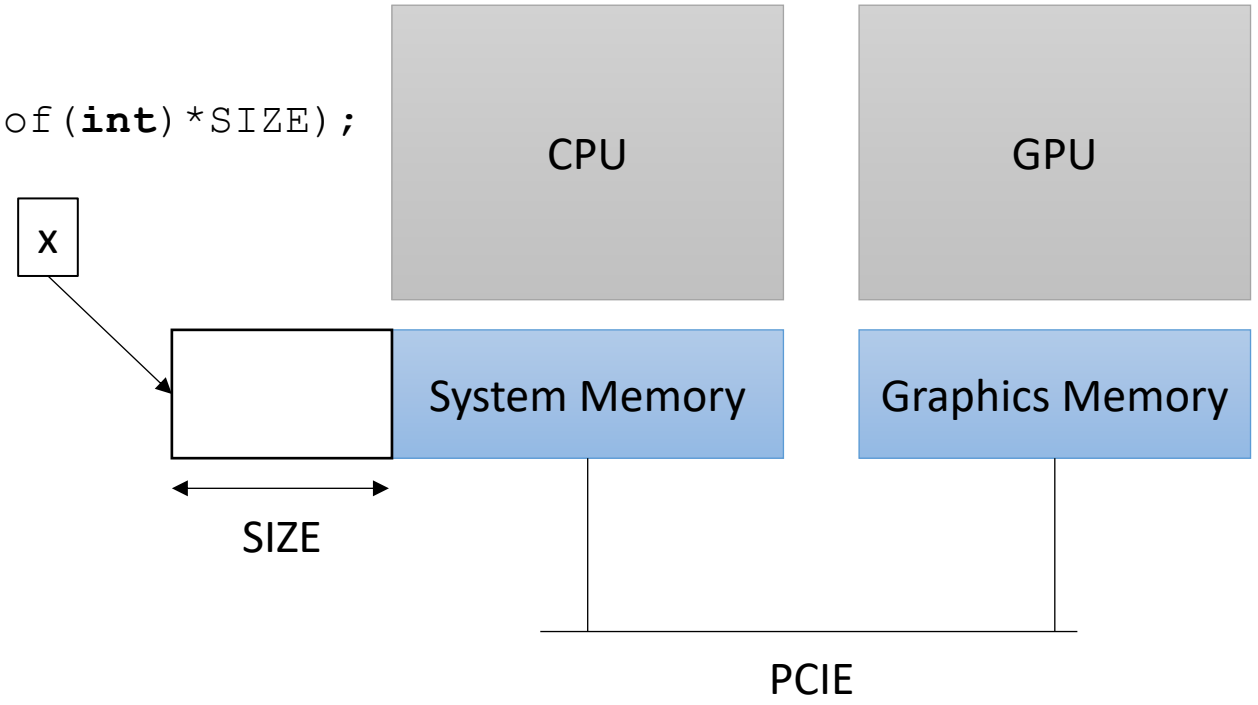


GPU set up

How do we allocate CPU memory on the host?

- Our heterogeneous, parallel, programming model

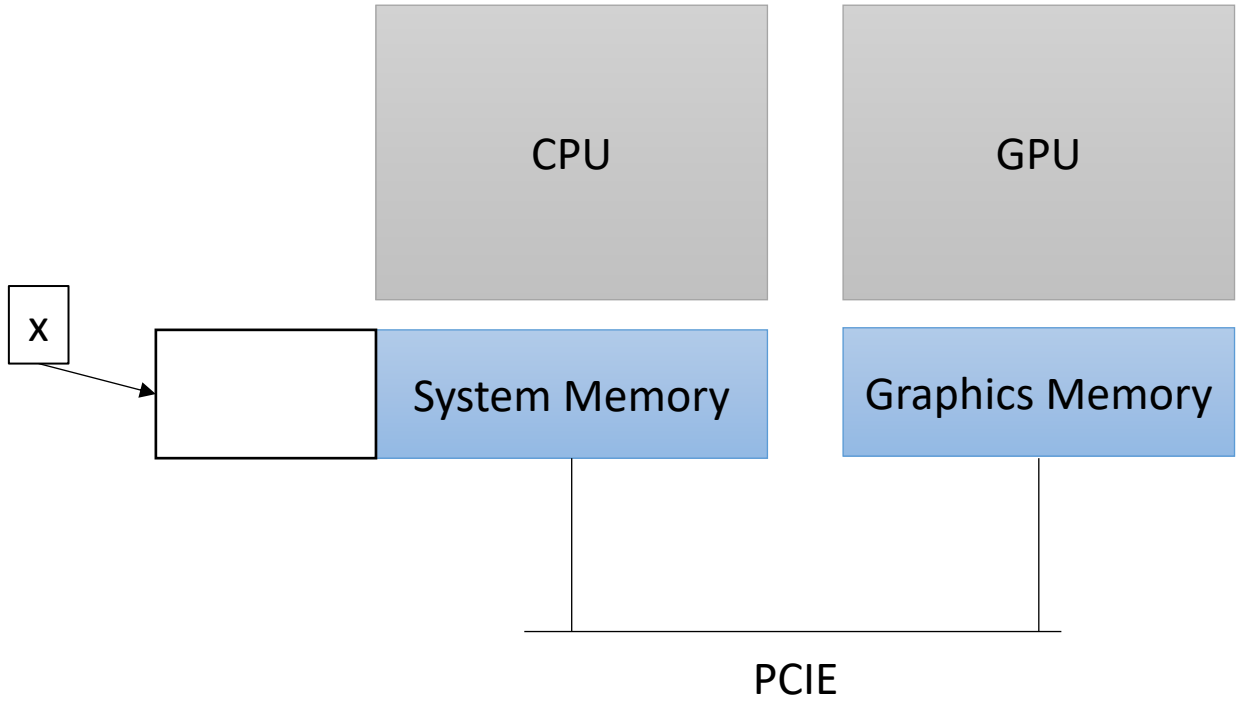
```
int *x = (int*) malloc(sizeof(int)*SIZE);
```



GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

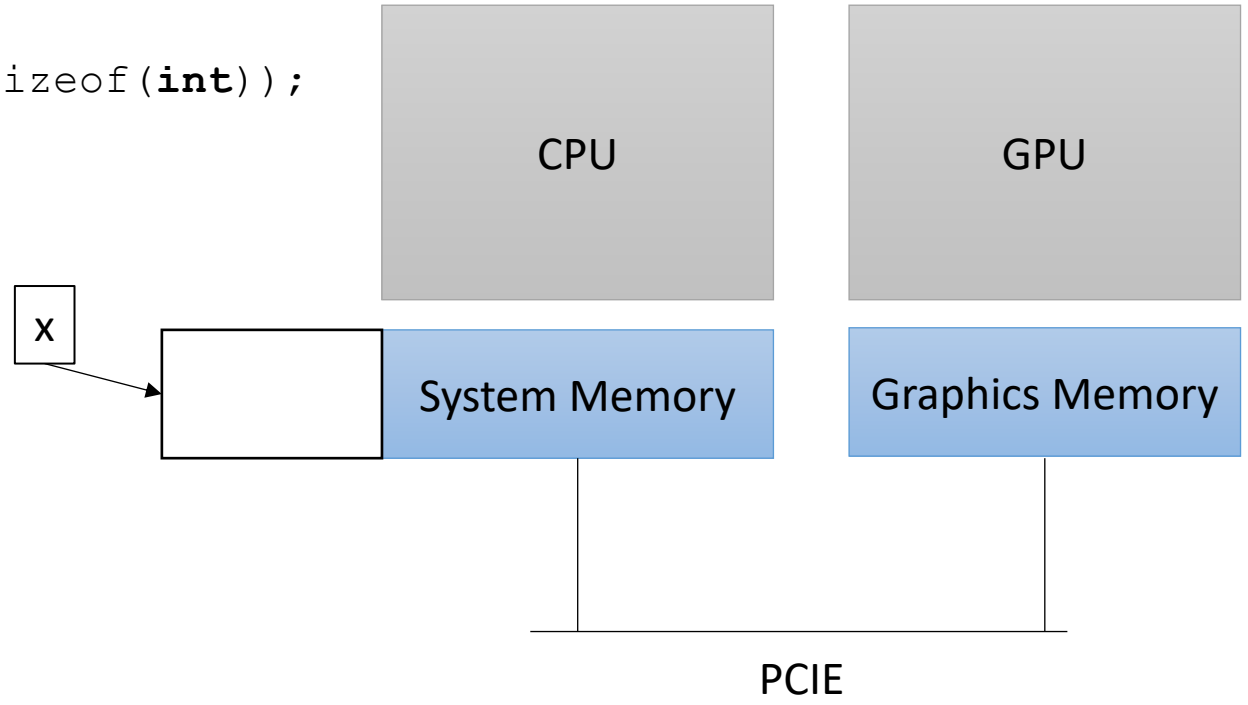


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

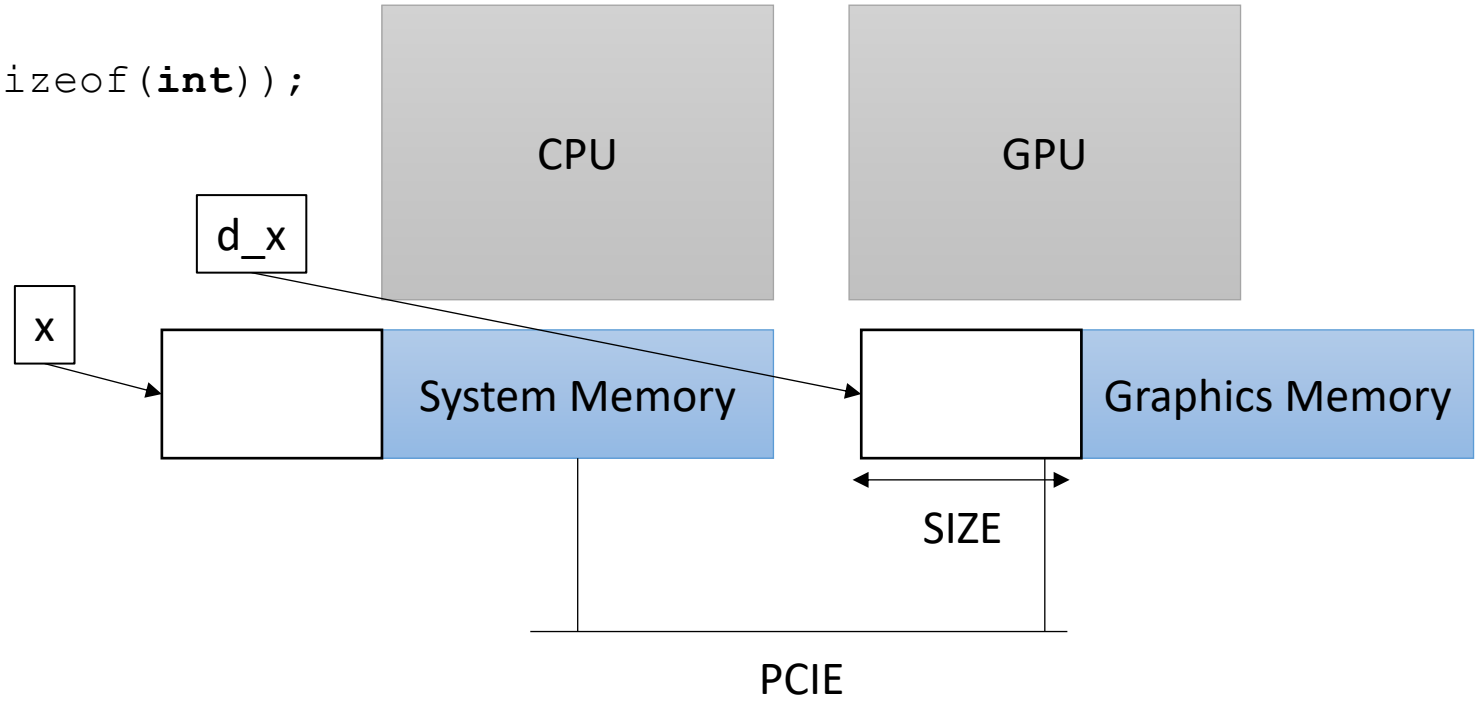


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

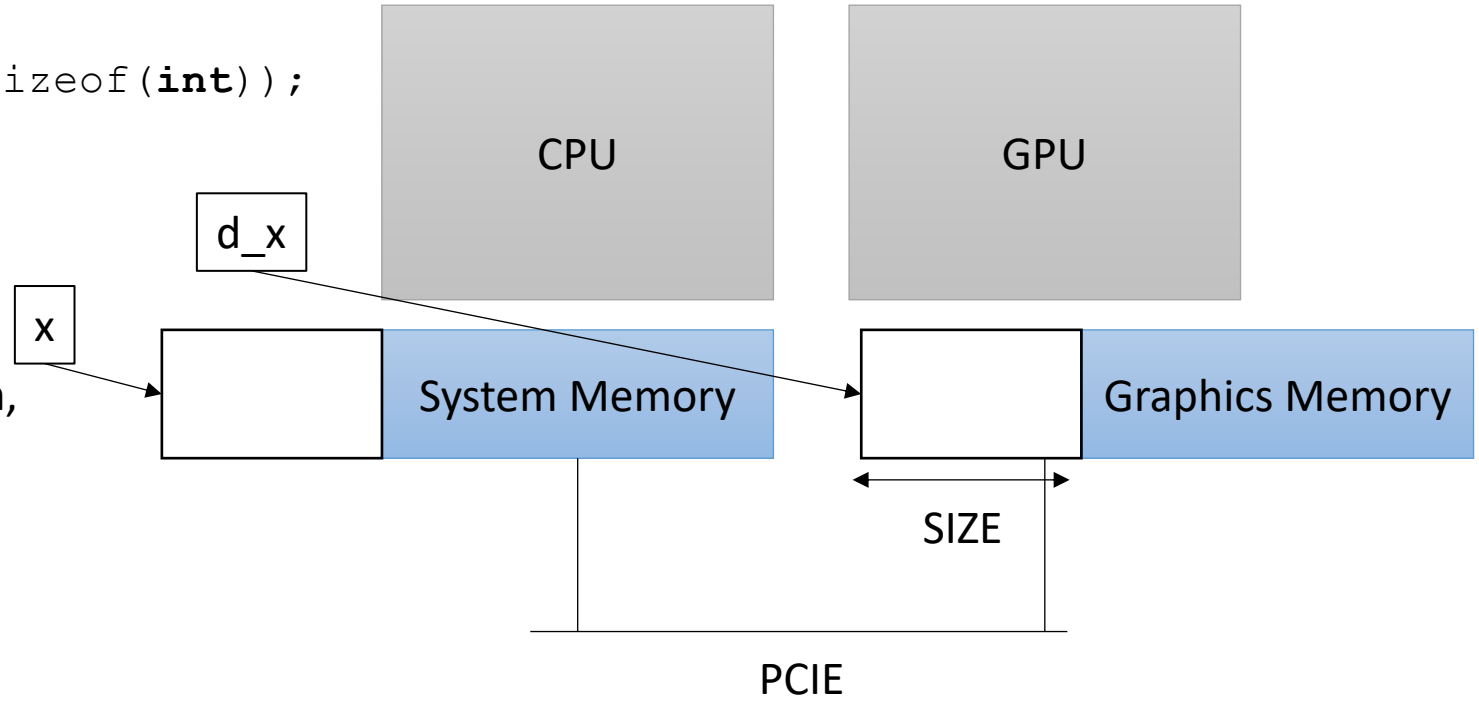


GPU set up

We need to allocate GPU memory on the host

- Our heterogeneous, parallel, programming model

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

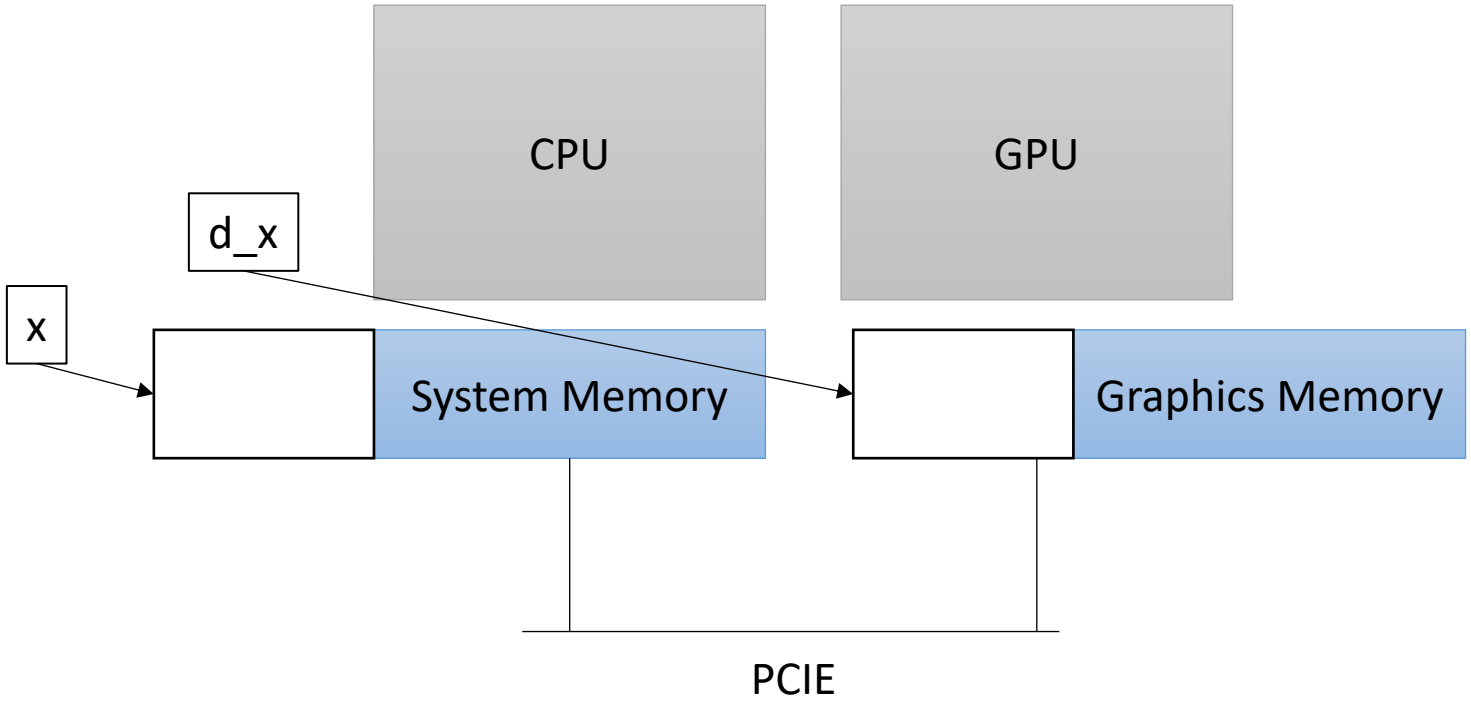


`d_x` is a pointer, in the CPU program, that points to memory on the GPU.

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

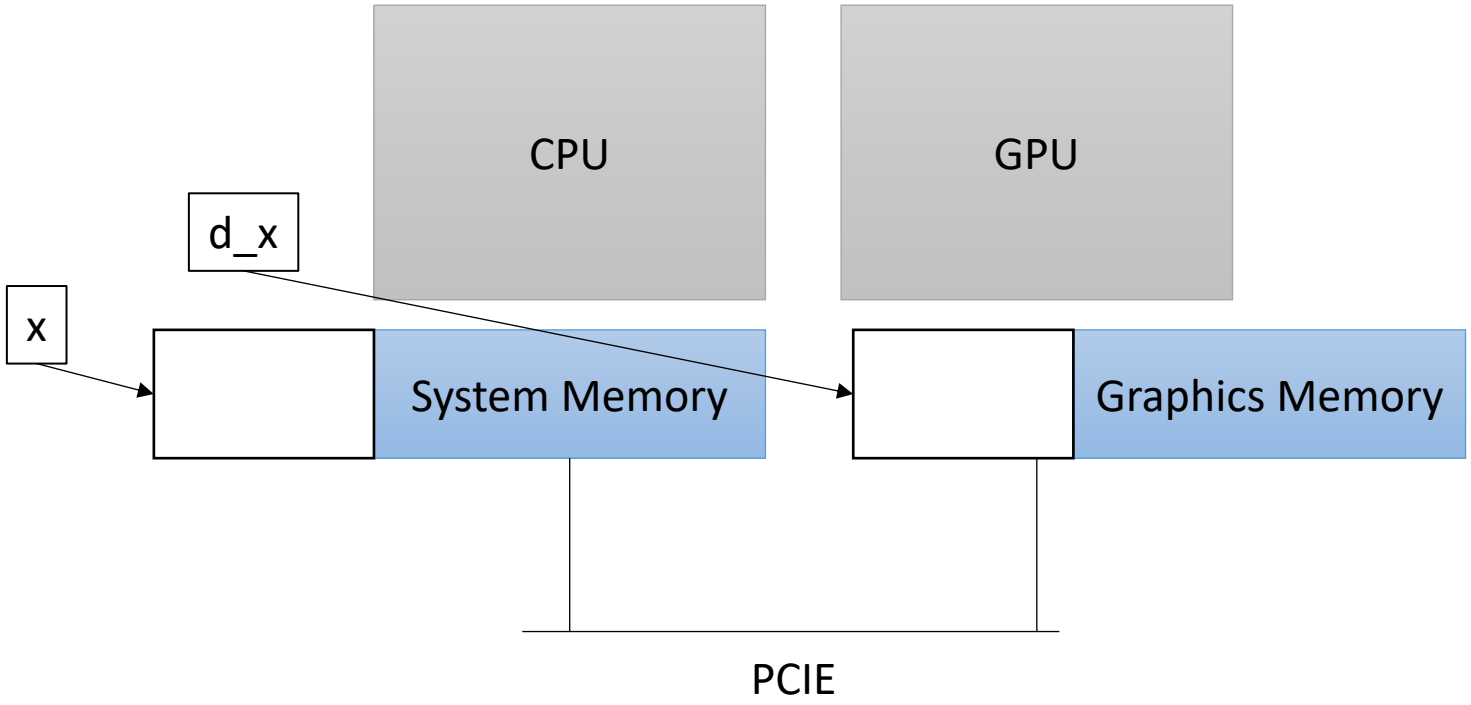


GPU set up

- 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



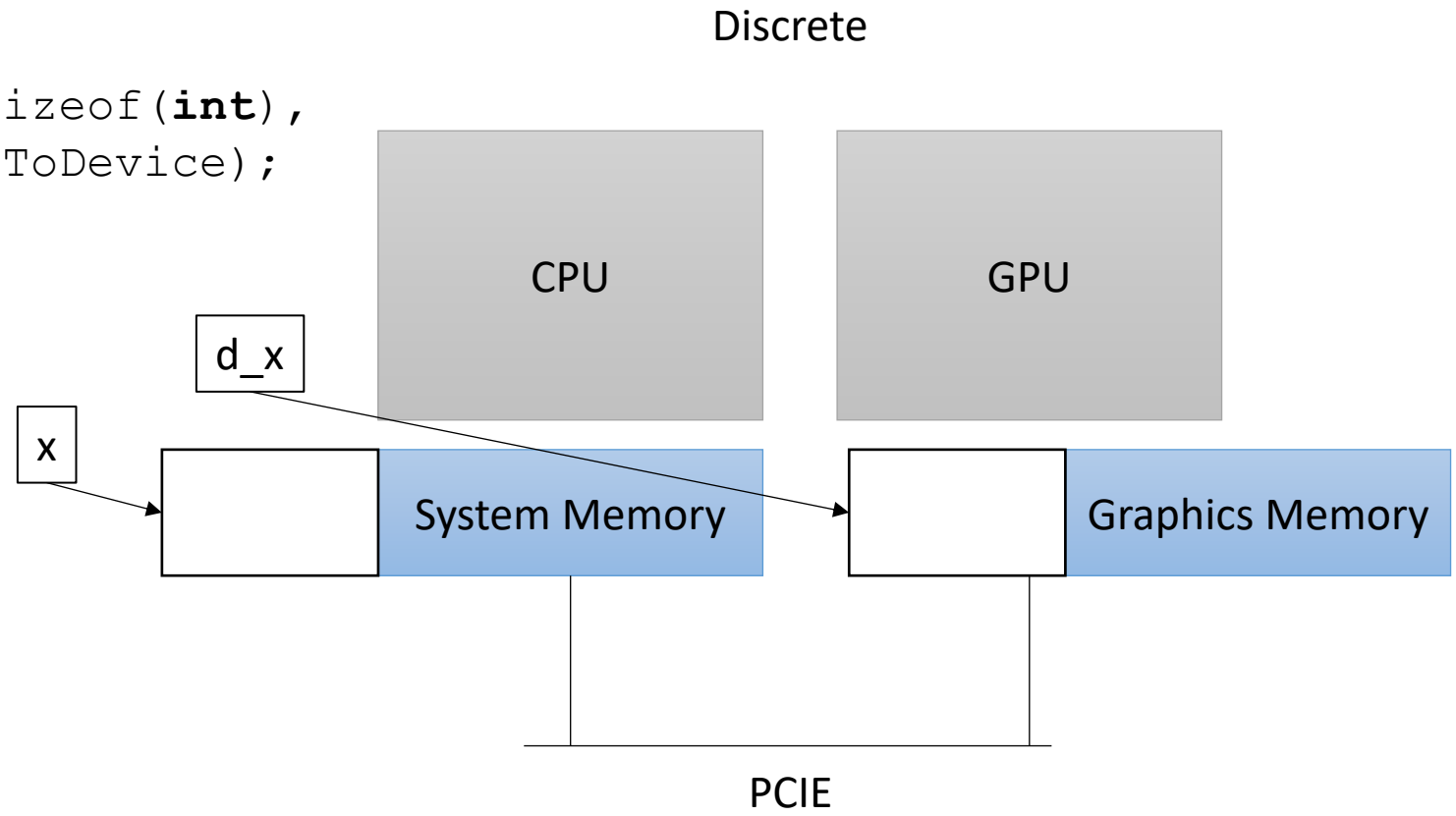
GPU set up

- 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);
```



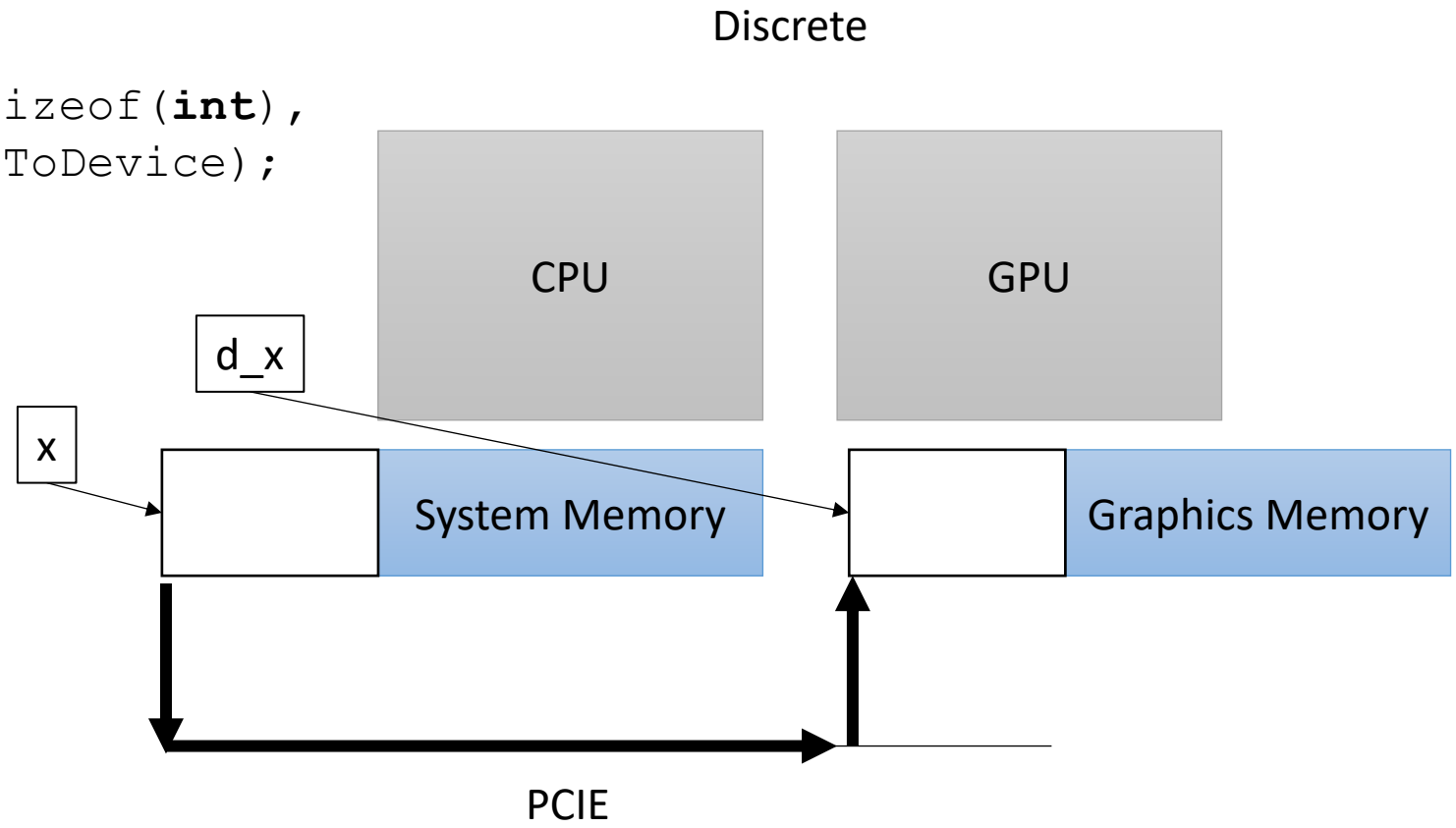
GPU set up

- 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?

```
float *a = new float[SIZE];
float *b = new float[SIZE];
float *c = new float[SIZE];

for (int i = 0; i < SIZE; i++) {
  a[i] = 0.0f;
  b[i] = i;
  c[i] = 1.0f;
}

float *d_a, *d_b, *d_c;
int e = 0;
e = cudaMalloc(&d_a, SIZE*sizeof(float));
e |= cudaMalloc(&d_b, SIZE*sizeof(float));
e |= cudaMalloc(&d_c, SIZE*sizeof(float));

e |= cudaMemcpy(d_a, a, SIZE*sizeof(float), cudaMemcpyHostToDevice);
e |= cudaMemcpy(d_b, b, SIZE*sizeof(float), cudaMemcpyHostToDevice);
e |= cudaMemcpy(d_c, c, SIZE*sizeof(float), 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

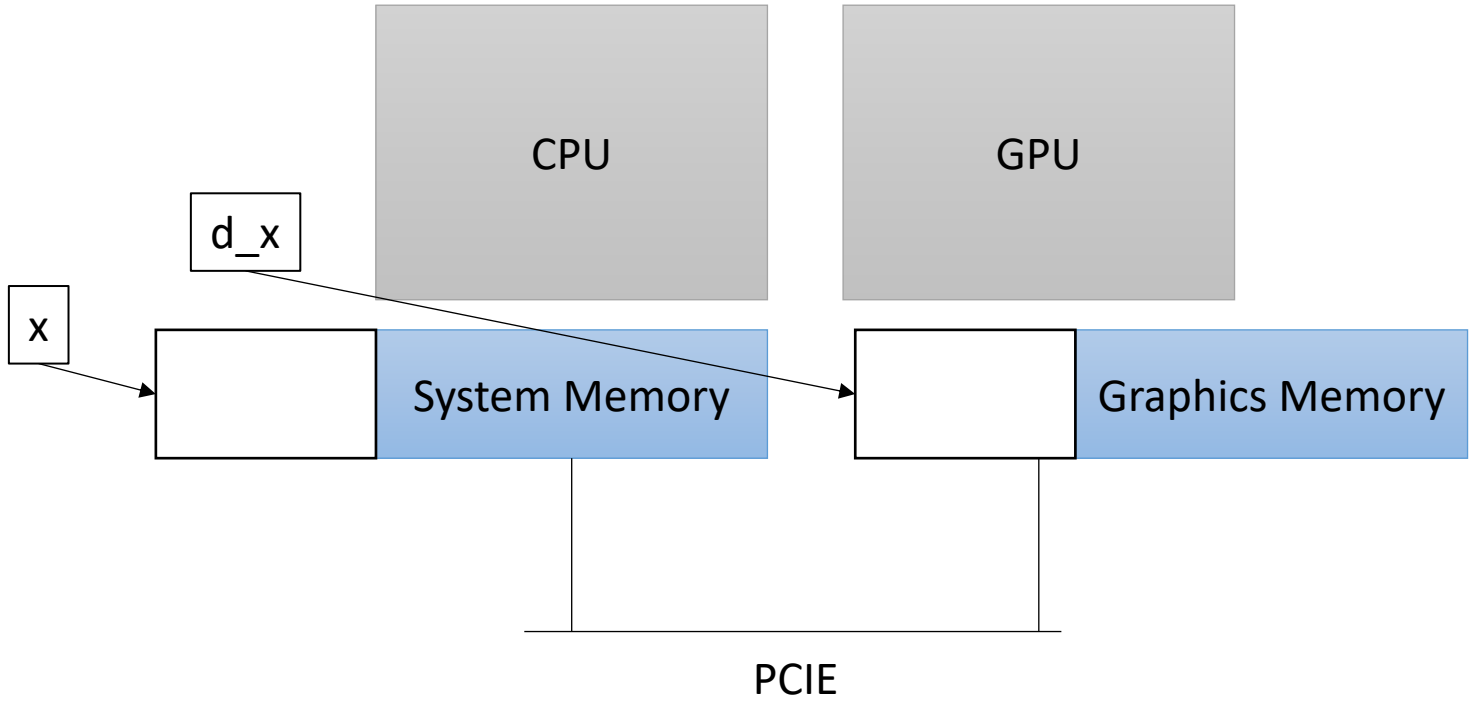
calling the function

```
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

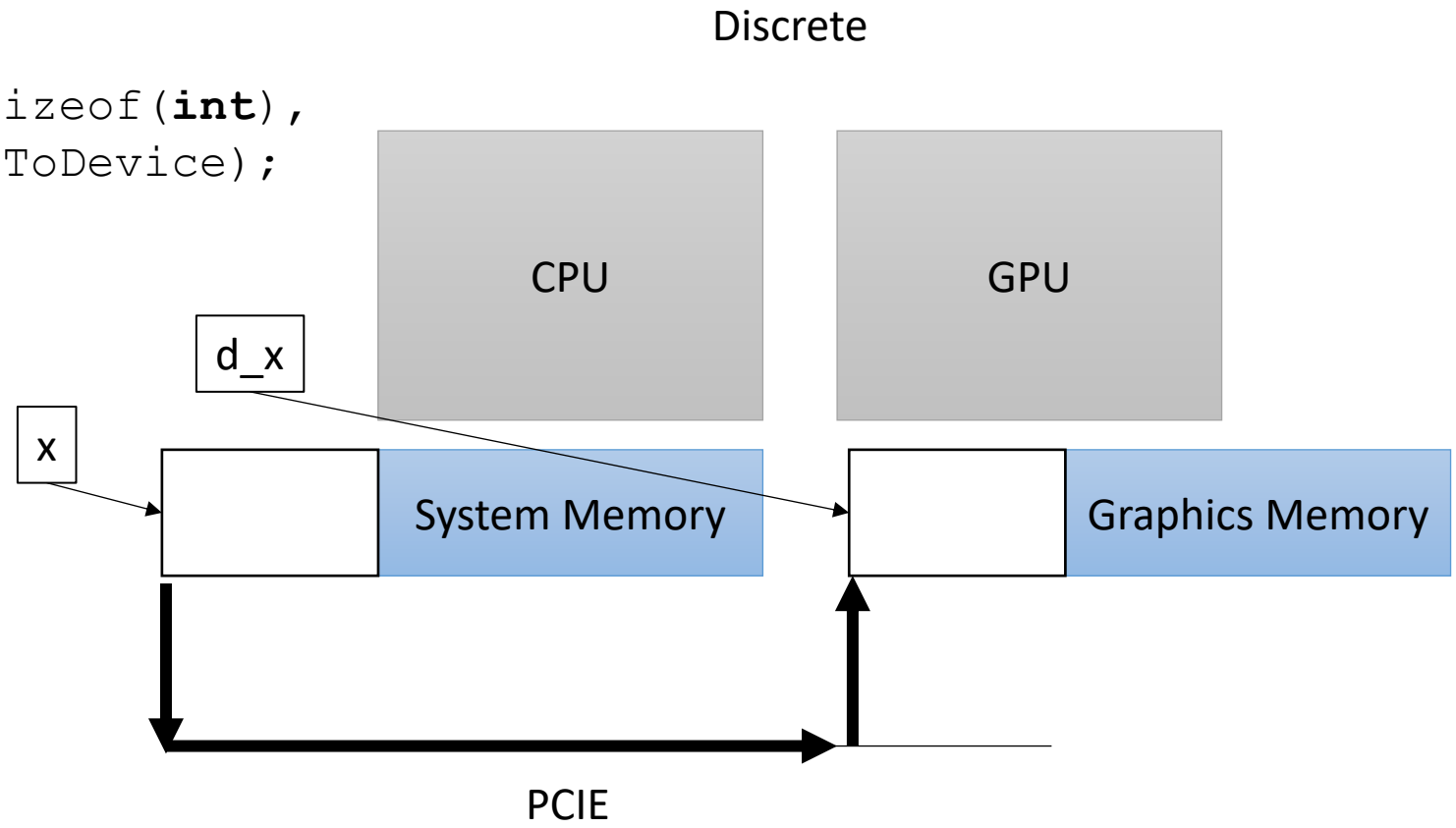
```
vector_add<<<1,1>>>(d_a, d_b, d_c, size);
e |= cudaDeviceSynchronize();
```

```
$ nvcc main.cu -o main
```

GPU set up

- Our heterogeneous, parallel, programming model

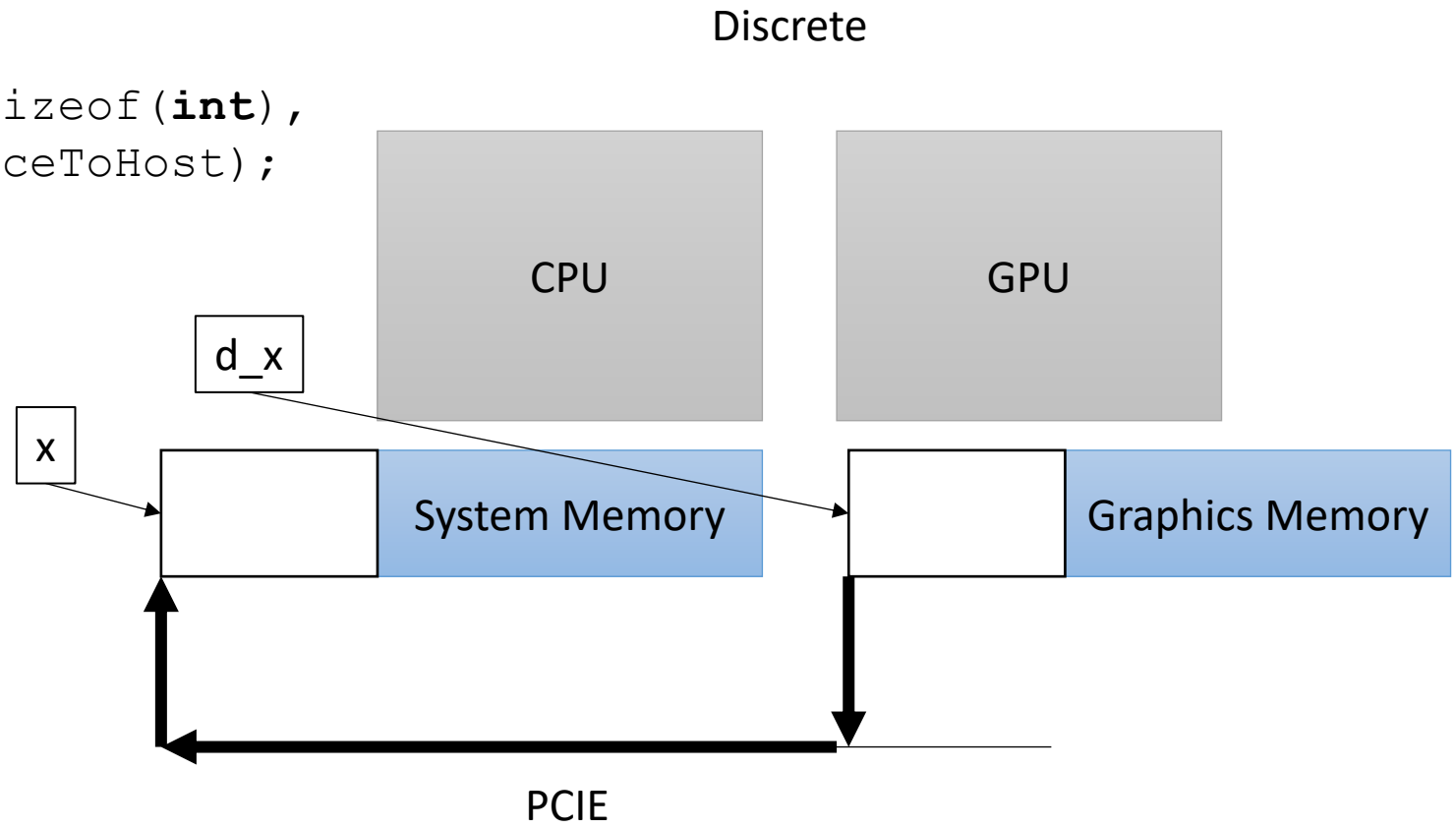
```
//initialize x on host
cudaMemcpy(d_x, x, SIZE*sizeof(int),
           cudaMemcpyHostToDevice);
```



GPU set up

- Our heterogeneous, parallel, programming model

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



The GPU Program

- The system expects the GPU kernel to be fast to render graphics.
- It kills the kernel if it takes more than ~5 seconds.
- If GPU is used for both display and CUDA.

The GPU Program

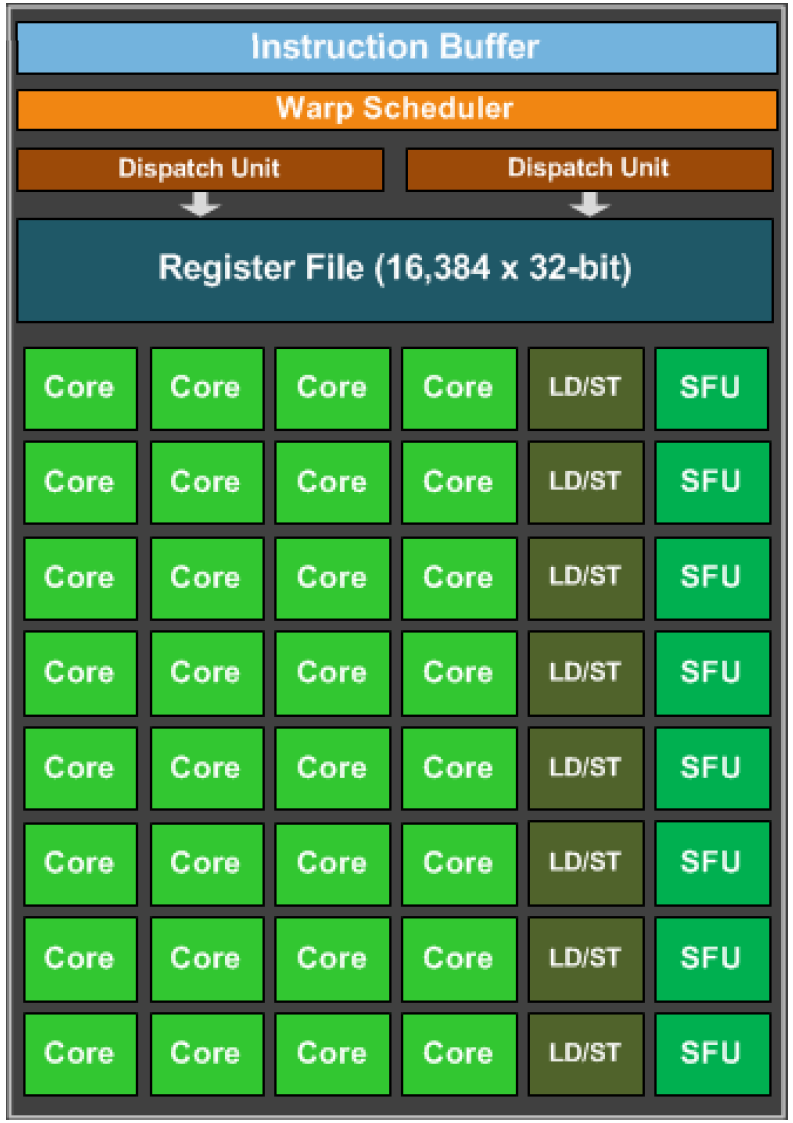
1000x slower on GPU?
It didn’t do so well...

One GPU core is much slower than a single CPU core

First parallelization attempt

- Lets look at some GPU documentation.

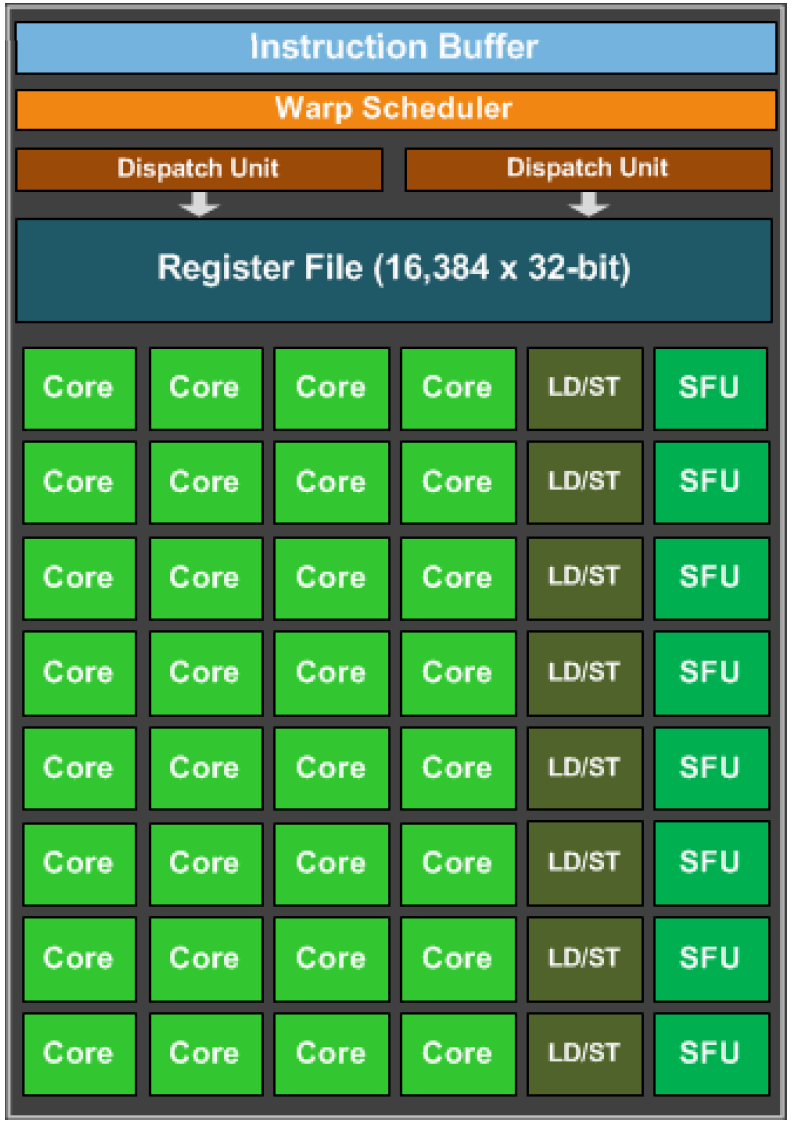
- The Maxwell whitepaper shows a diagram of one of the GPU cores



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

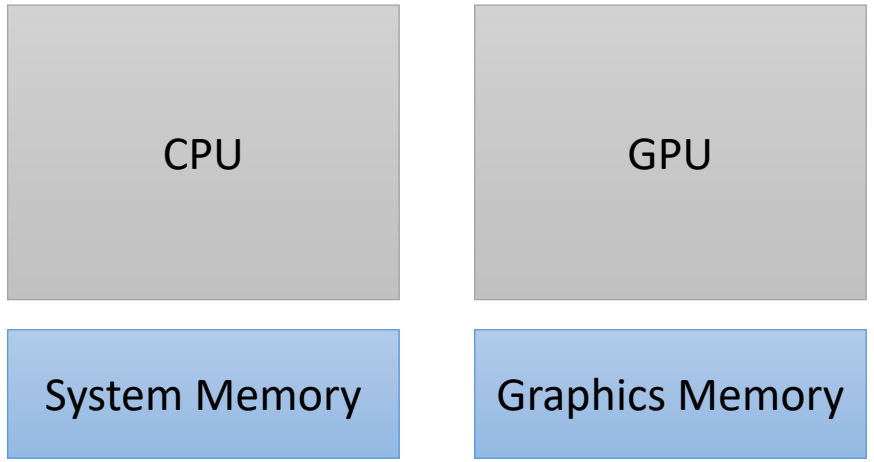
First parallelization attempt

That gives us about 14x speedup!

First parallelization attempt

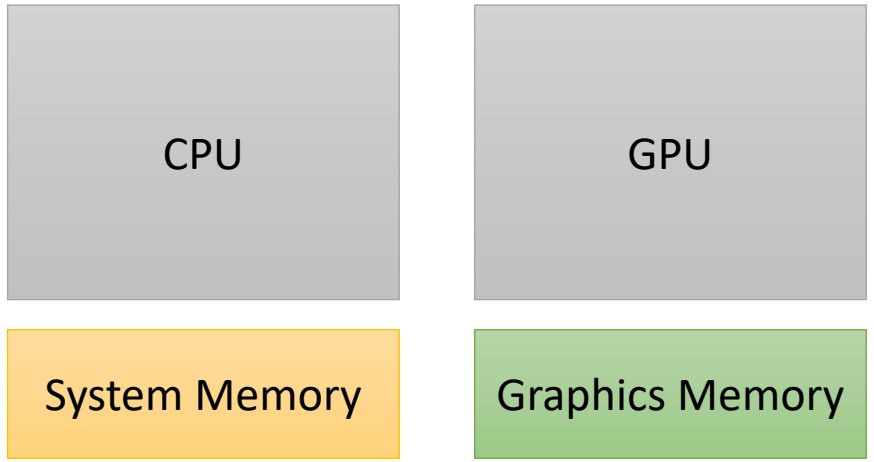
Getting better but we have a long ways to go!

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

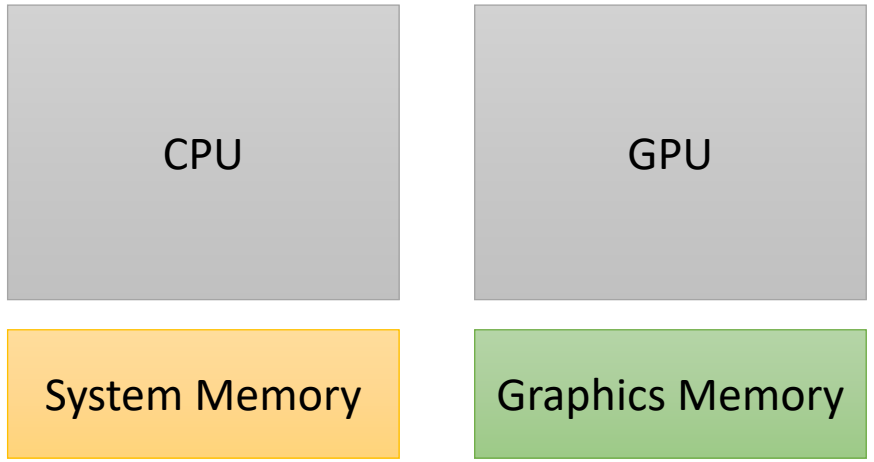
16-lane highway on a windy road driven by semi trucks

GPU Memory

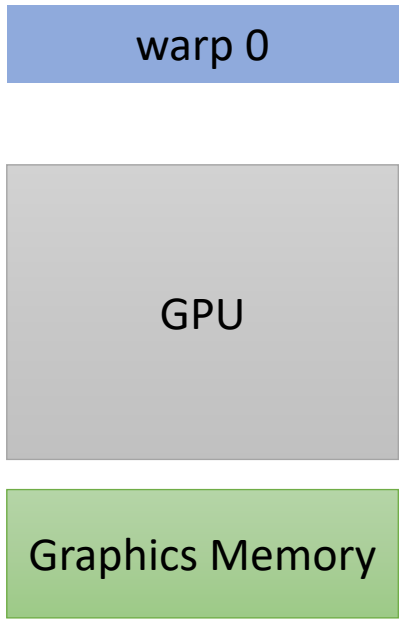
bandwidth:
~**700 GB/s** for GPU
~**50 GB/s** for CPUs

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

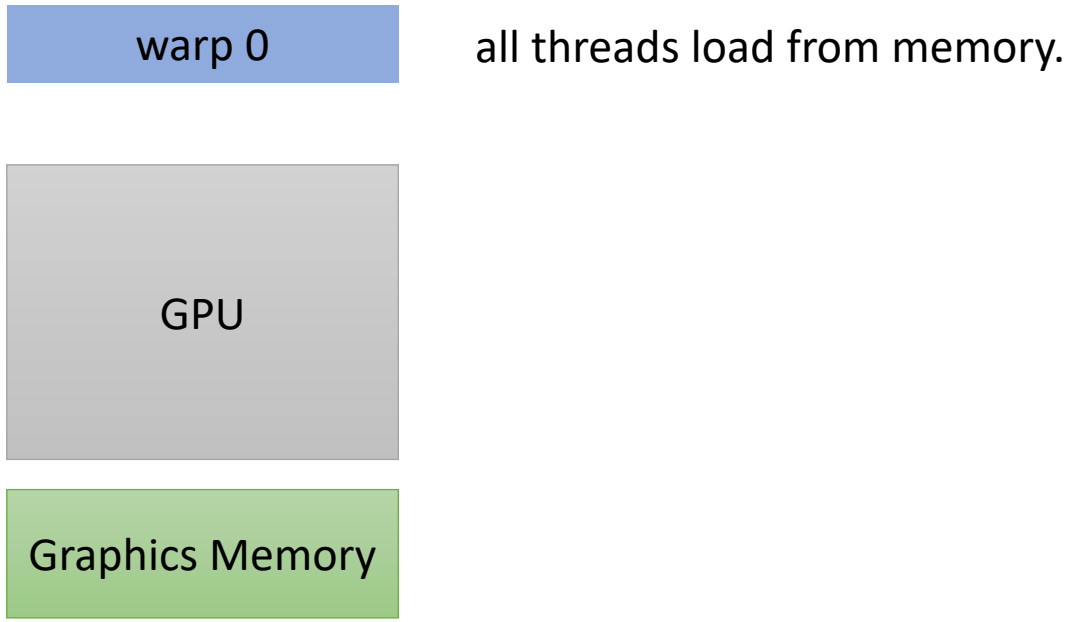
Cache Latency:
~**28** cycles for L1 hit for GPU
~**4** cycles for L1 hit on CPUs



Preemption and concurrency?

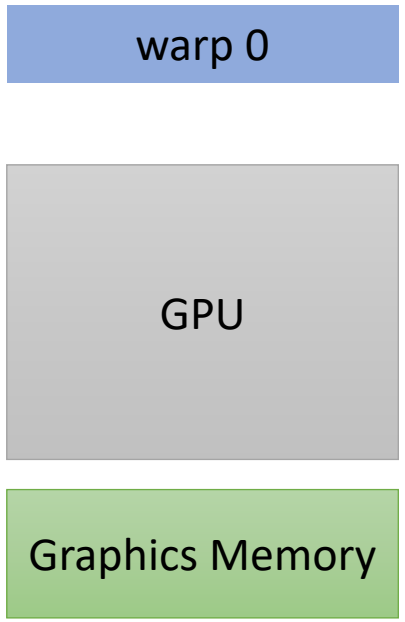


Preemption and concurrency?



all threads load from memory.

Preemption and concurrency?



all threads load from memory.

600 cycles!

Preemption and concurrency?

warp 0

warp 1

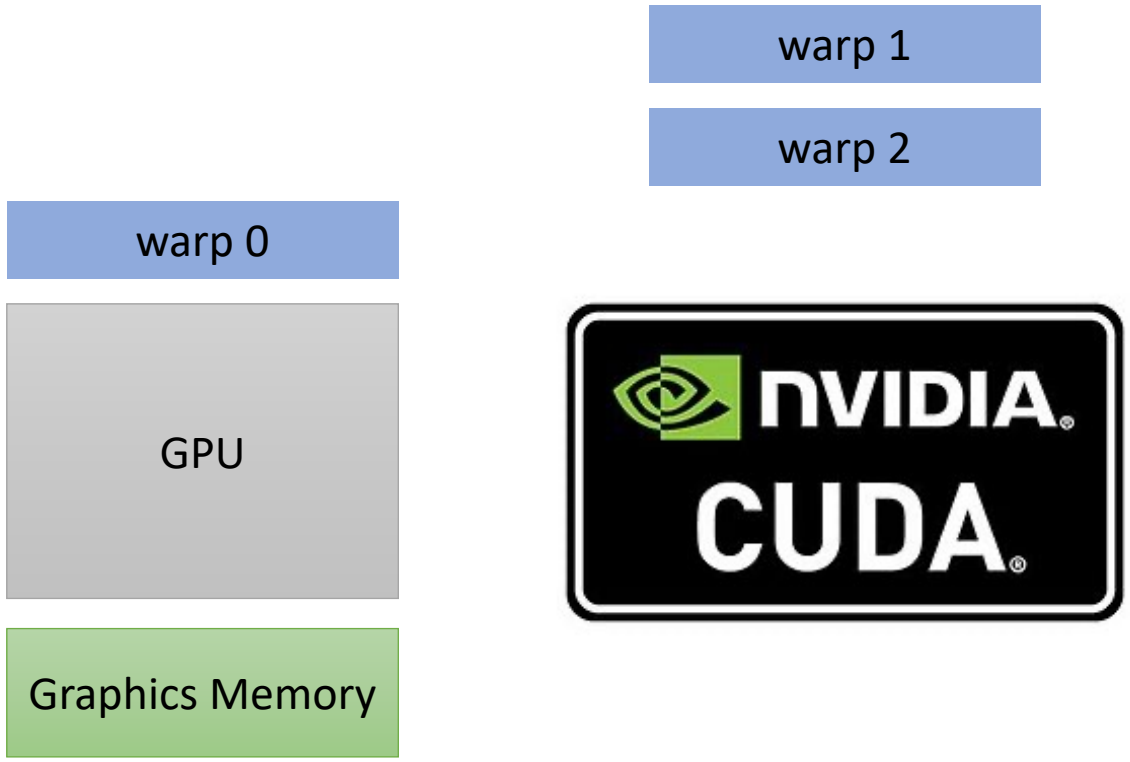
warp 2

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

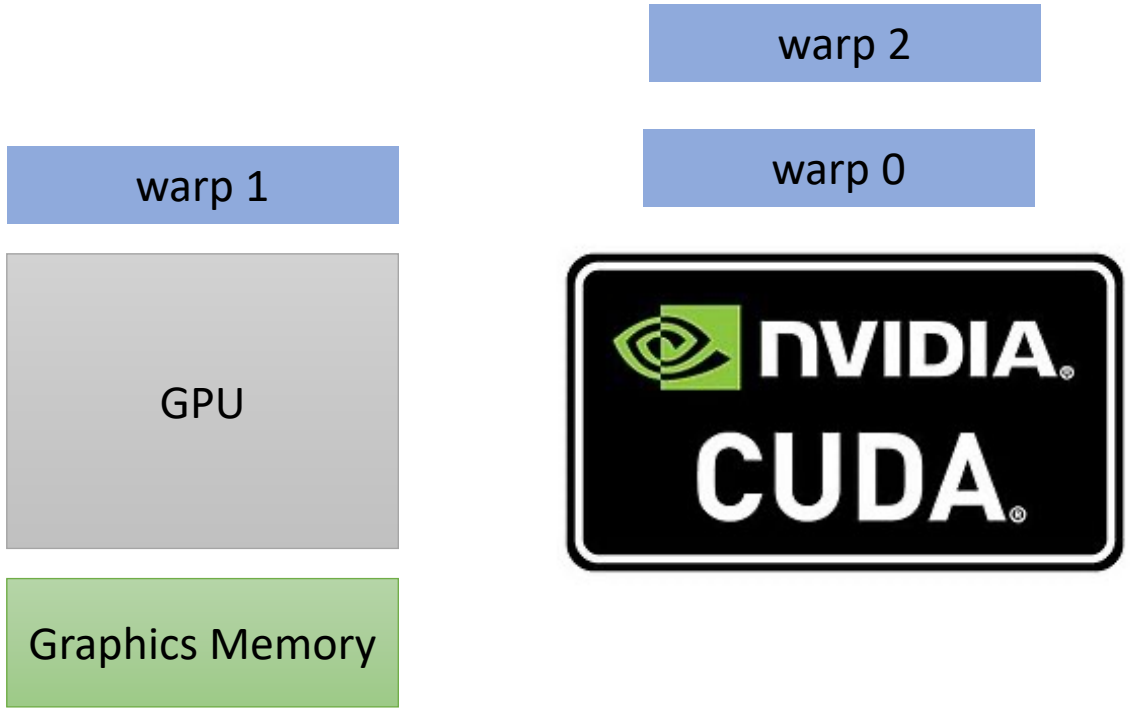
GPU

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

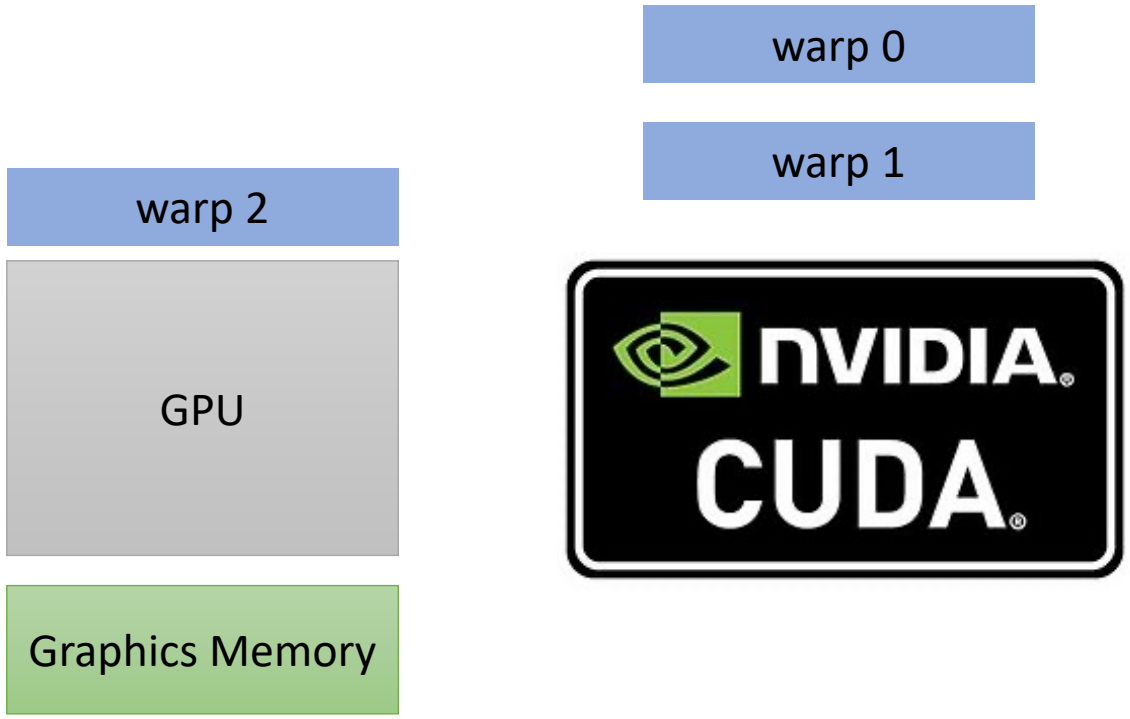
warp 2

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

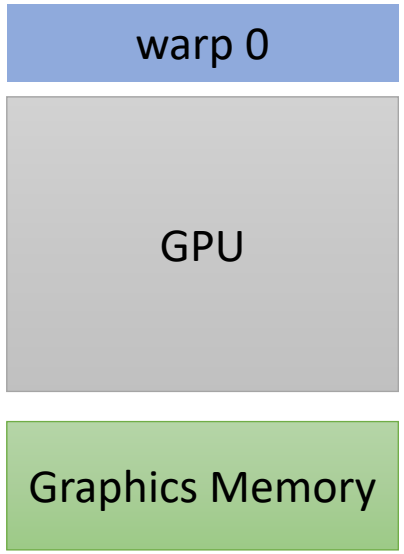
warp 1

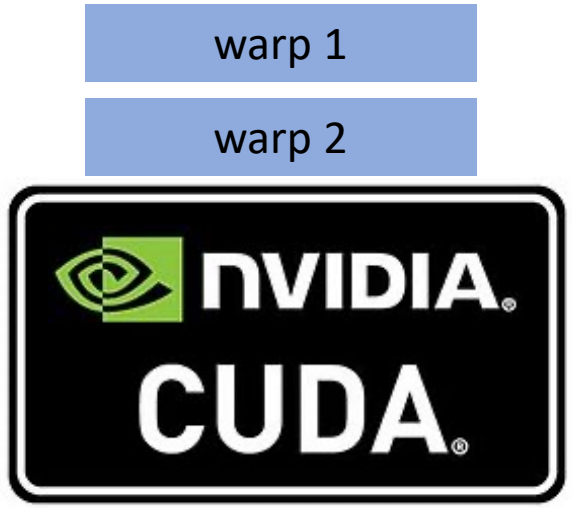
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!





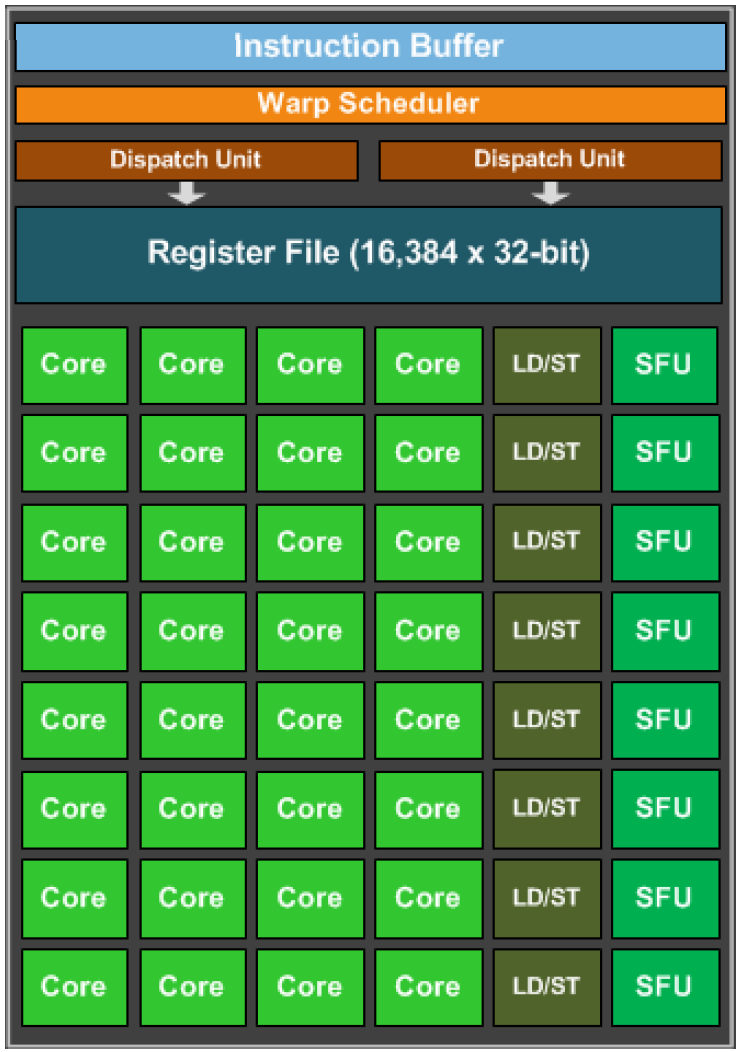
preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?

But wait, I thought preemption was expensive?

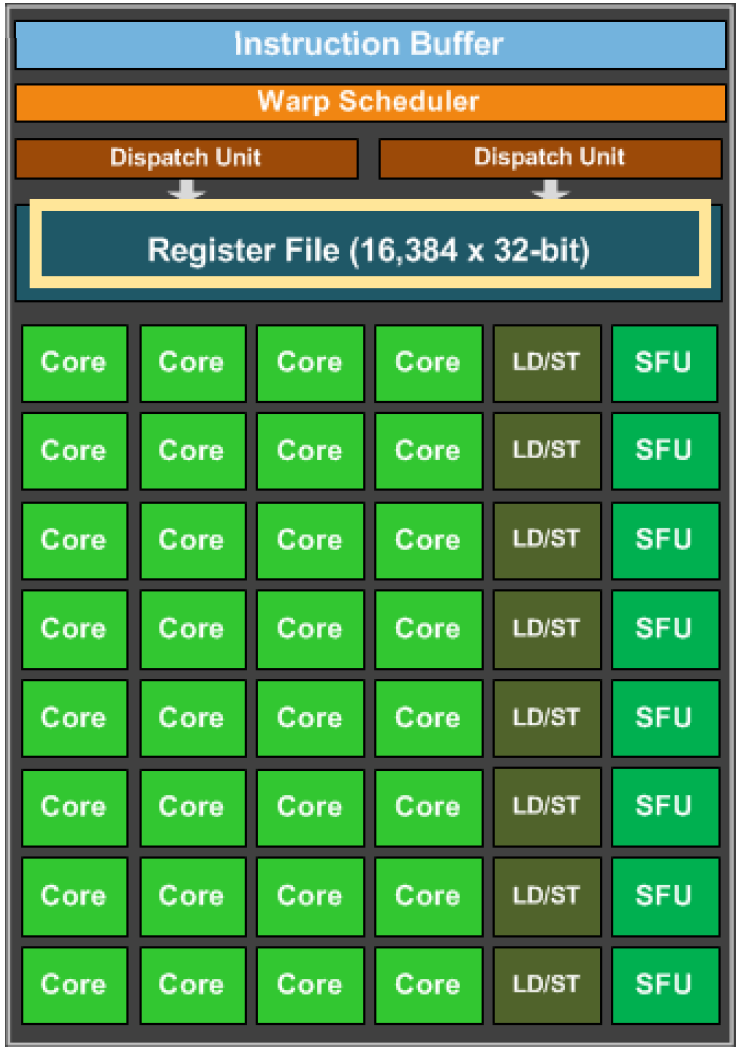
Preemption and concurrency?



But wait, I thought preemption was expensive?

bound on number of warps: 32

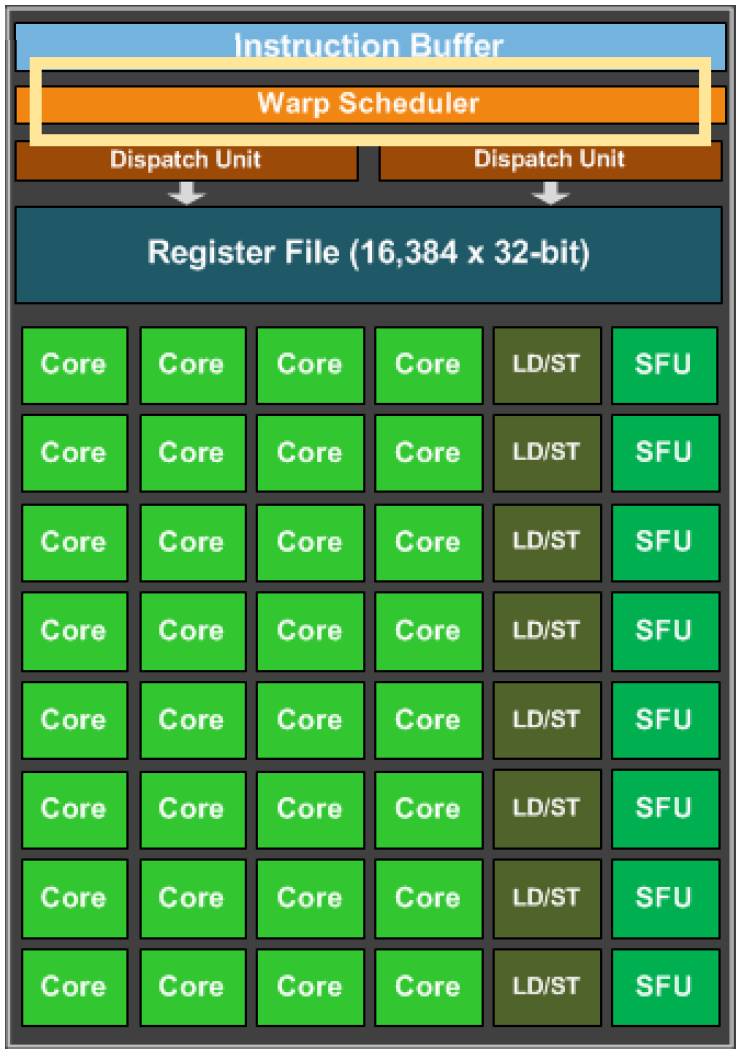
Preemption and concurrency?



But wait, I thought preemption was expensive?

Registers all stay on chip. Large register file.

Preemption and concurrency?



But wait, I thought preemption was expensive?

dedicated scheduler logic

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

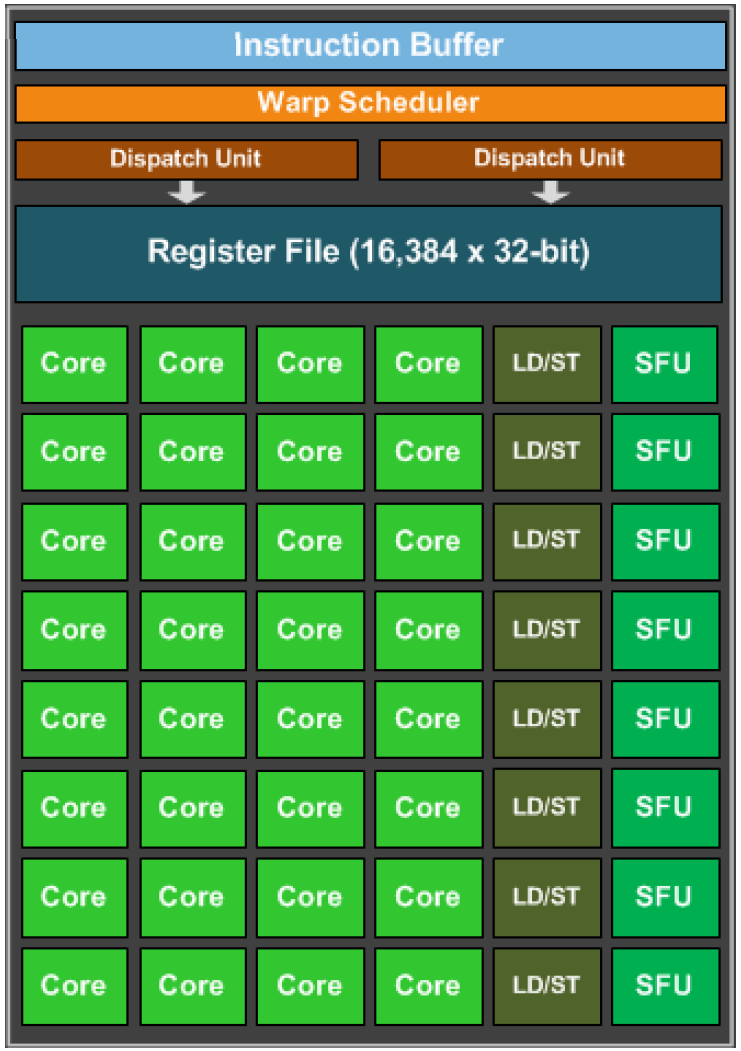
This is about 2x faster.

Concurrent warps

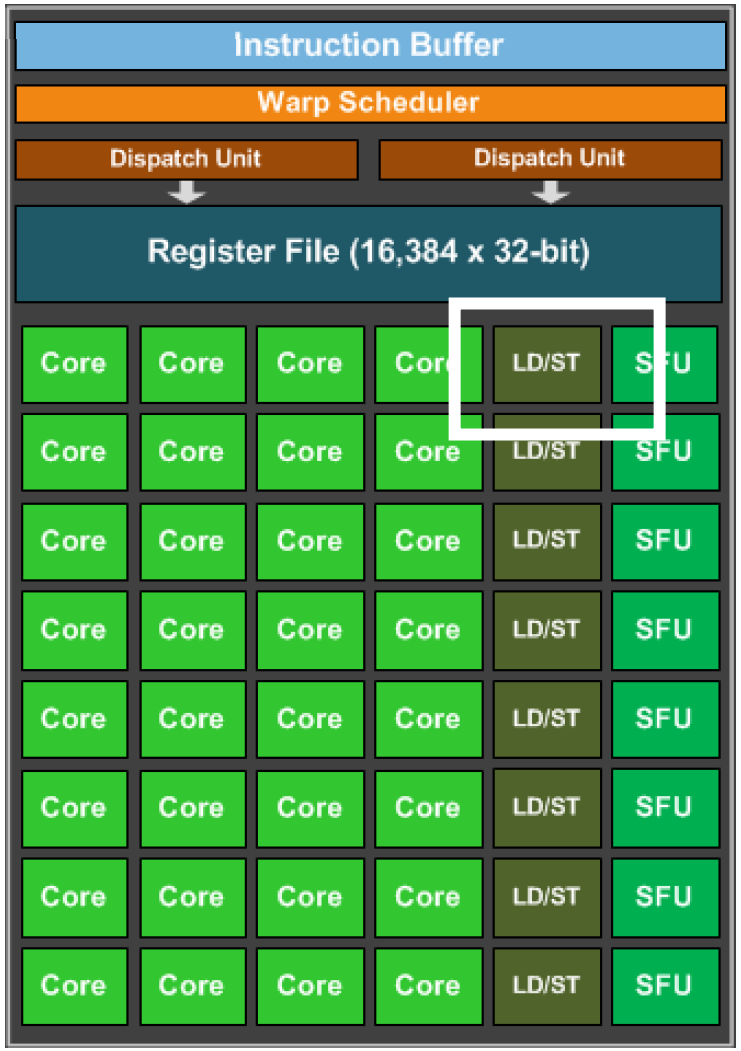
This is about 2x faster.

Getting better!

Optimizing memory accesses

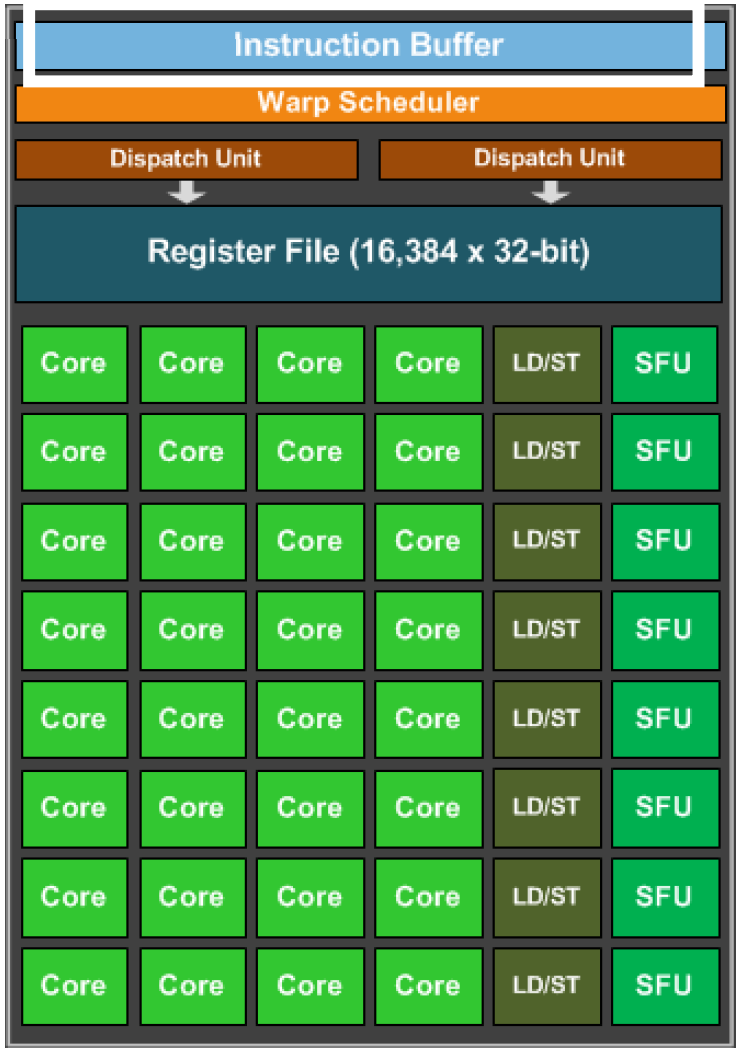


Optimizing memory accesses



This is the load/store unit. The hardware component responsible for issuing loads and stores.
Shared between 4 cores.

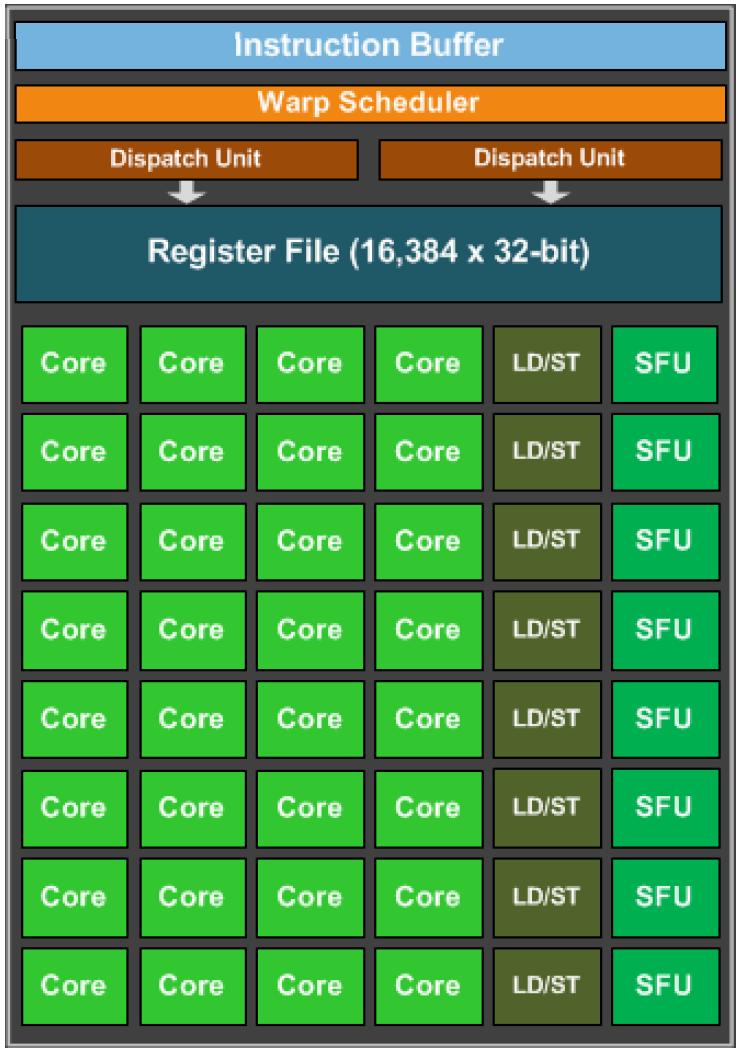
Optimizing memory accesses



This is the instruction cache
Shared between all cores of the warp.

This is the load/store unit. The hardware component responsible for issuing loads and stores.
Shared between 4 cores.

Warp execution



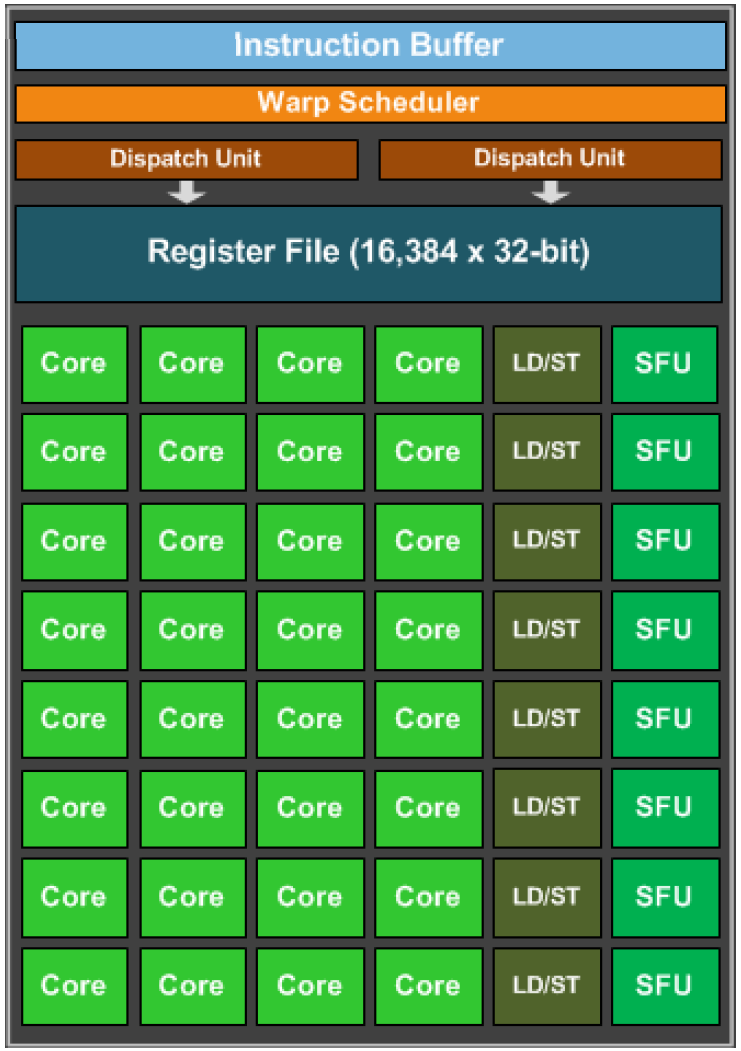
Groups of 32 threads are called a “warp”

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”

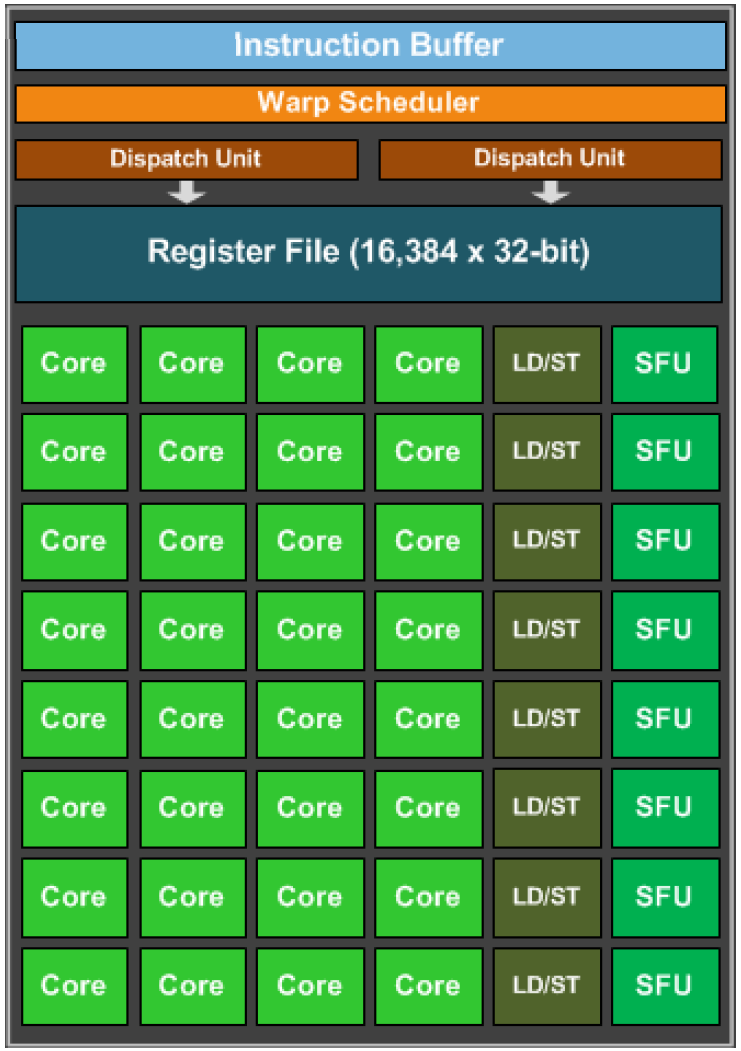
They are executed in lock-step, i.e. they all execute the same instruction at the same time



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

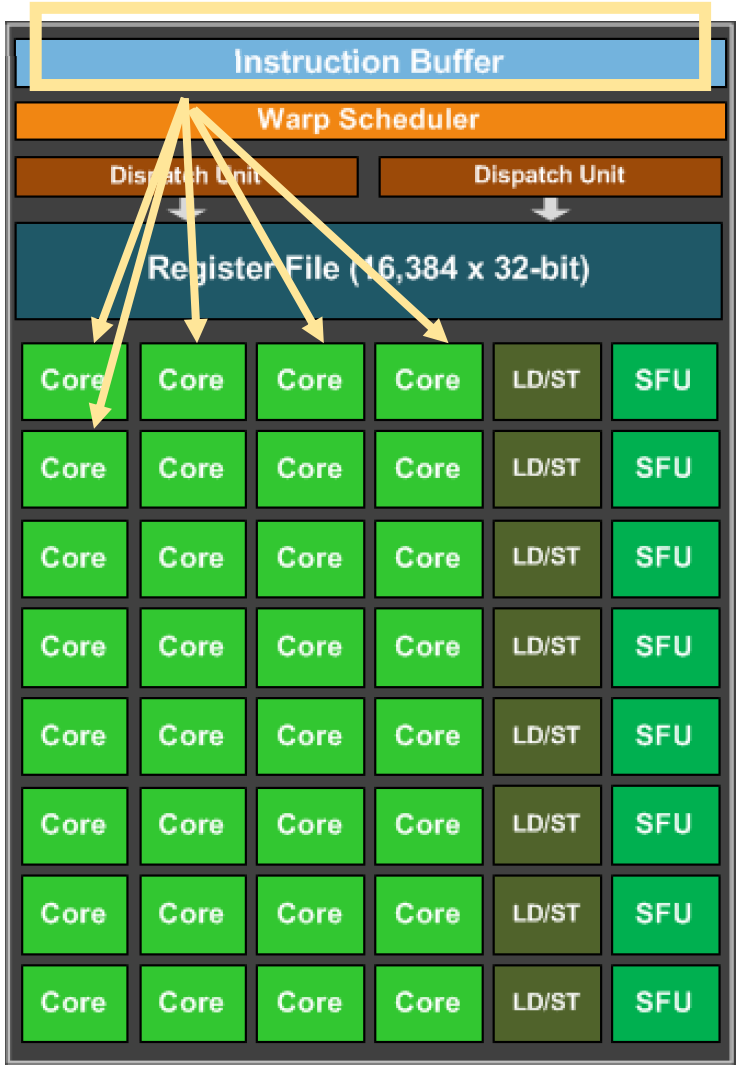
Program:

```
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.

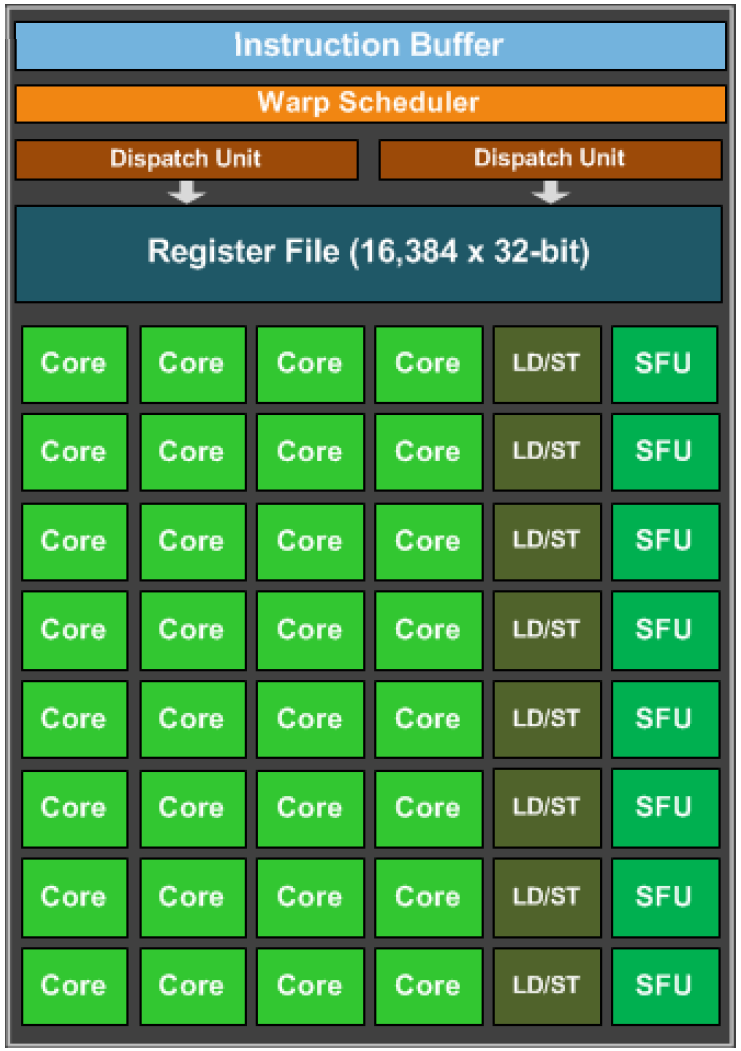
Program:

```
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



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

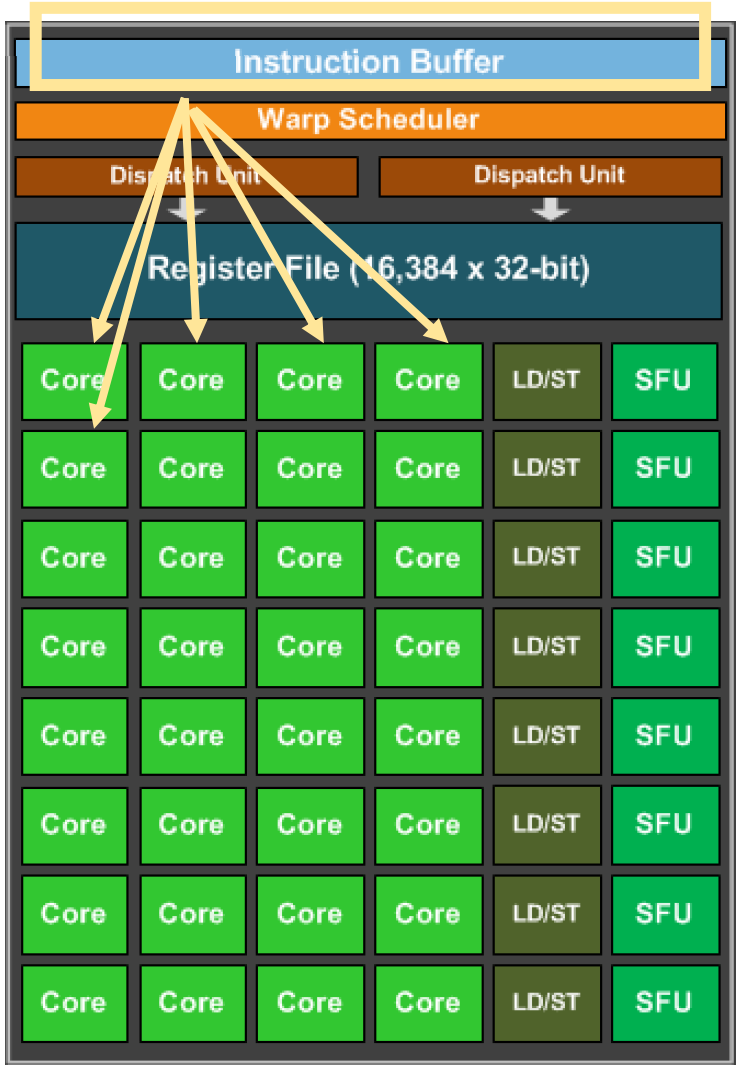
Program:

```
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:

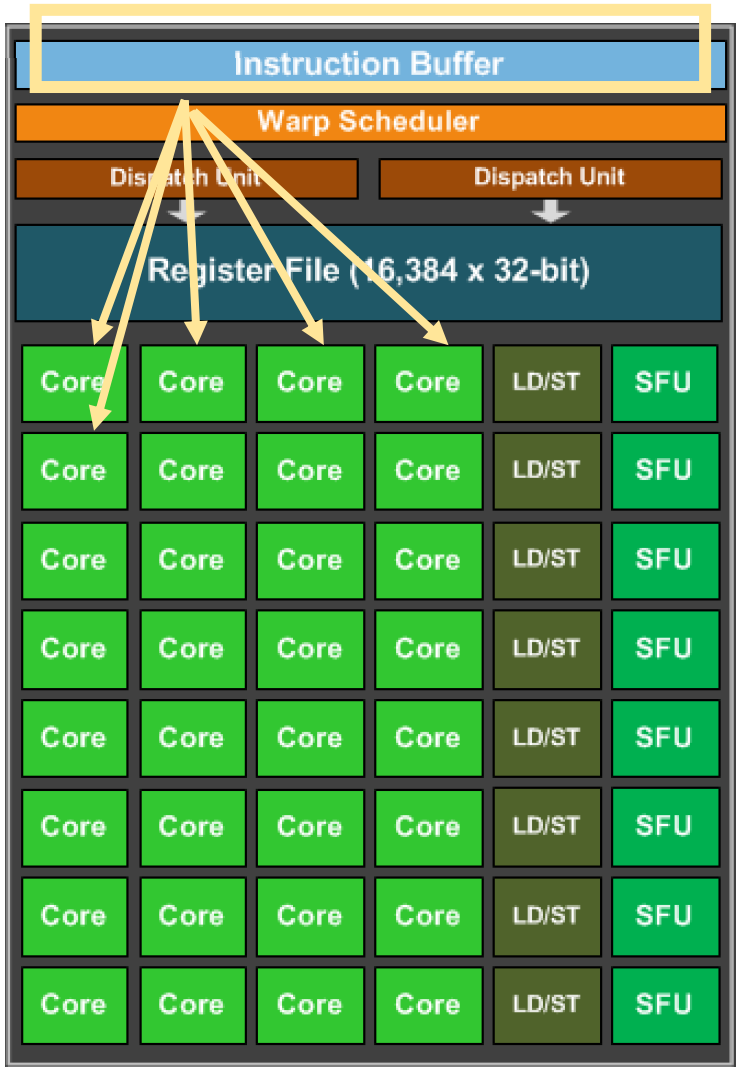
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

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:

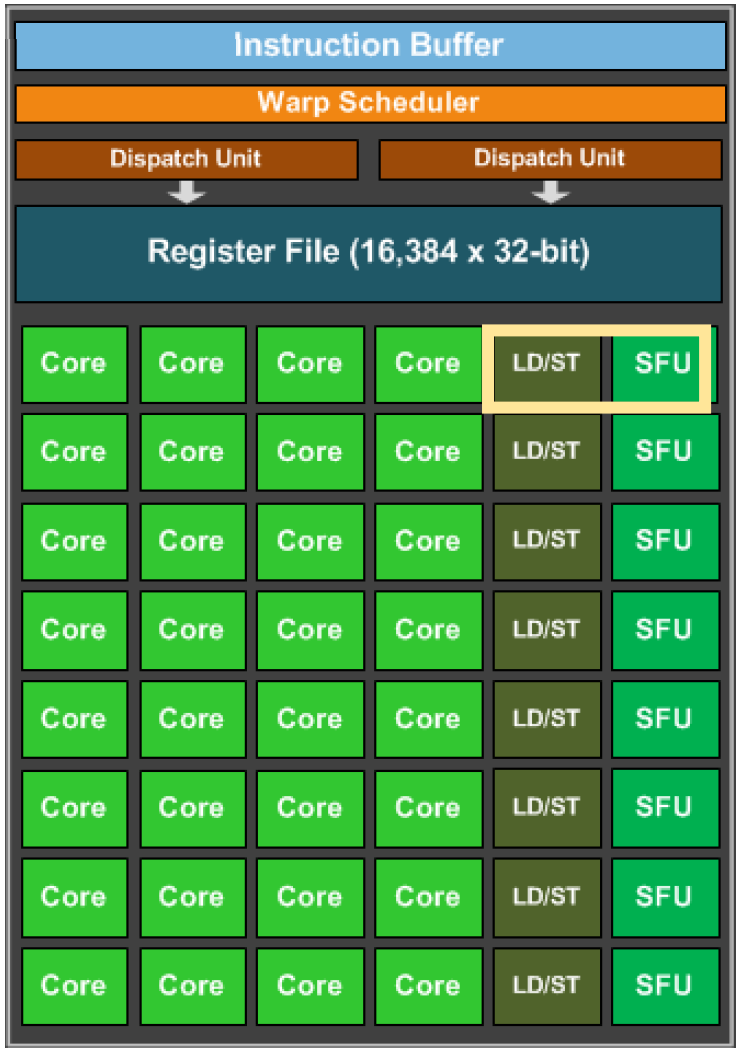
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

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

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



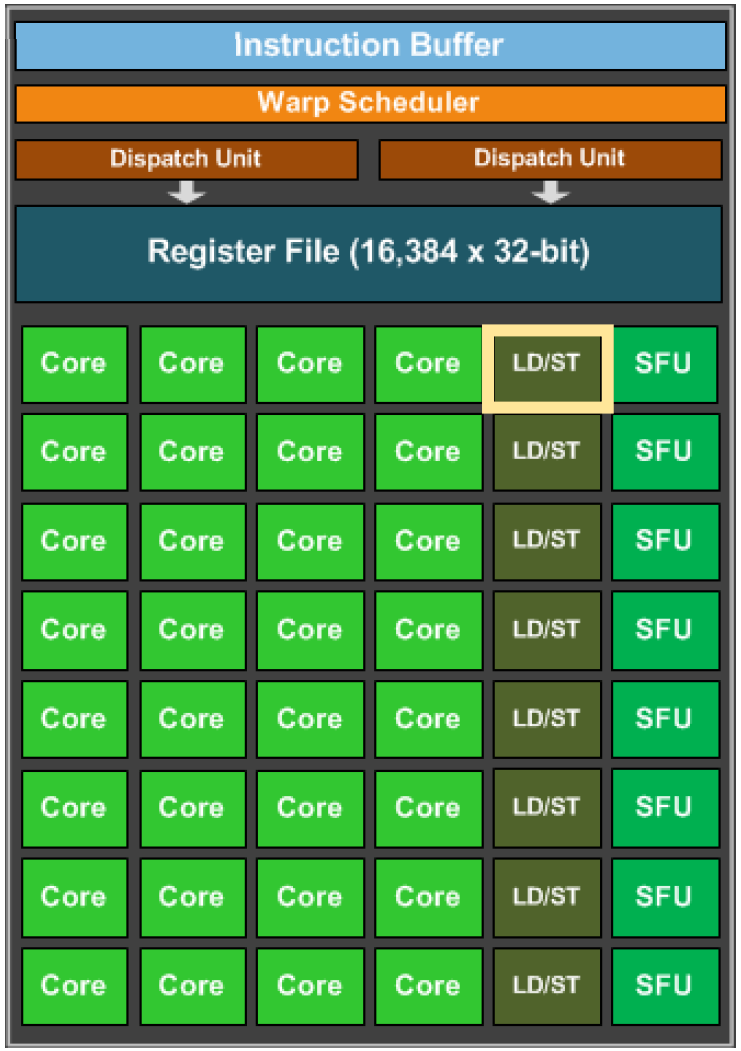
Cores share expensive HW units (load/store and special functions)

Program:

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

Warp execution

Lets look closer at memory

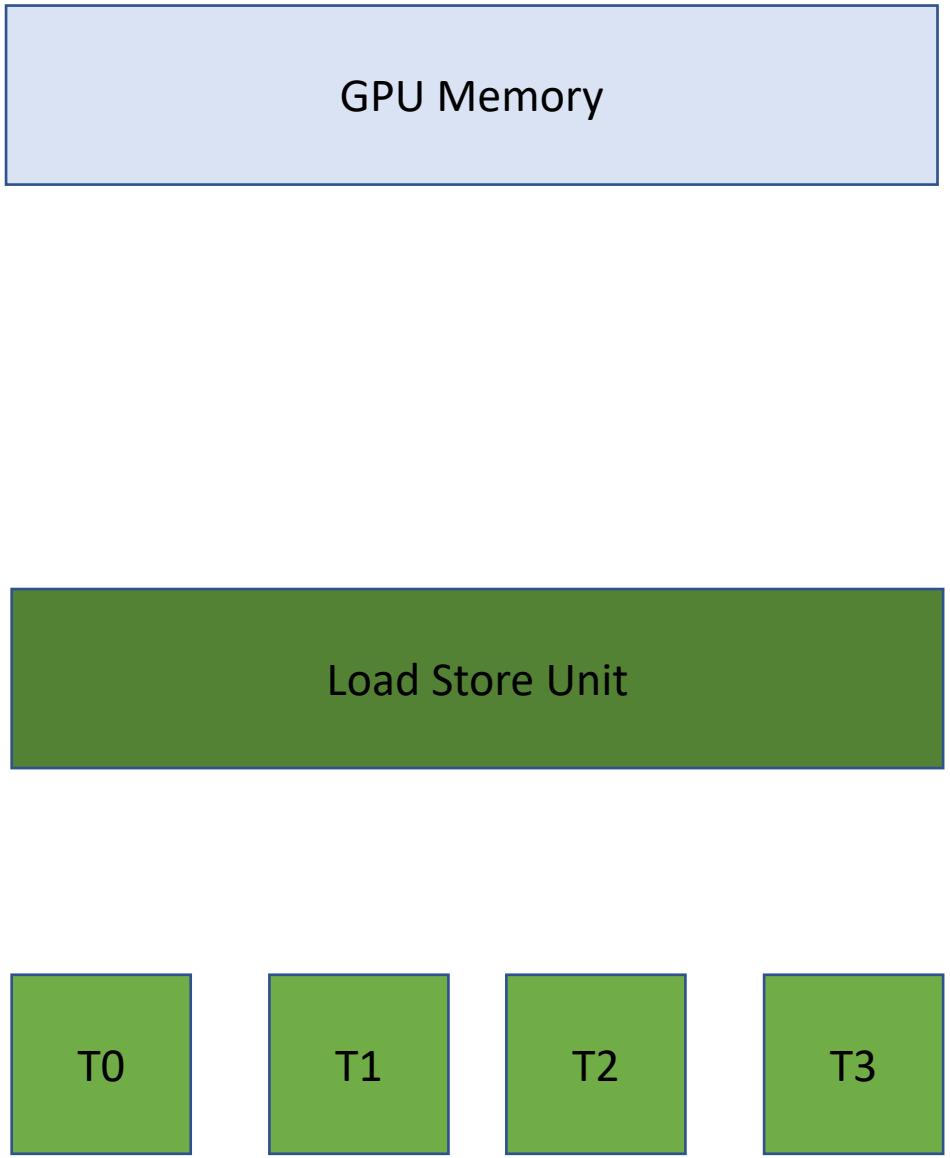


Program:

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

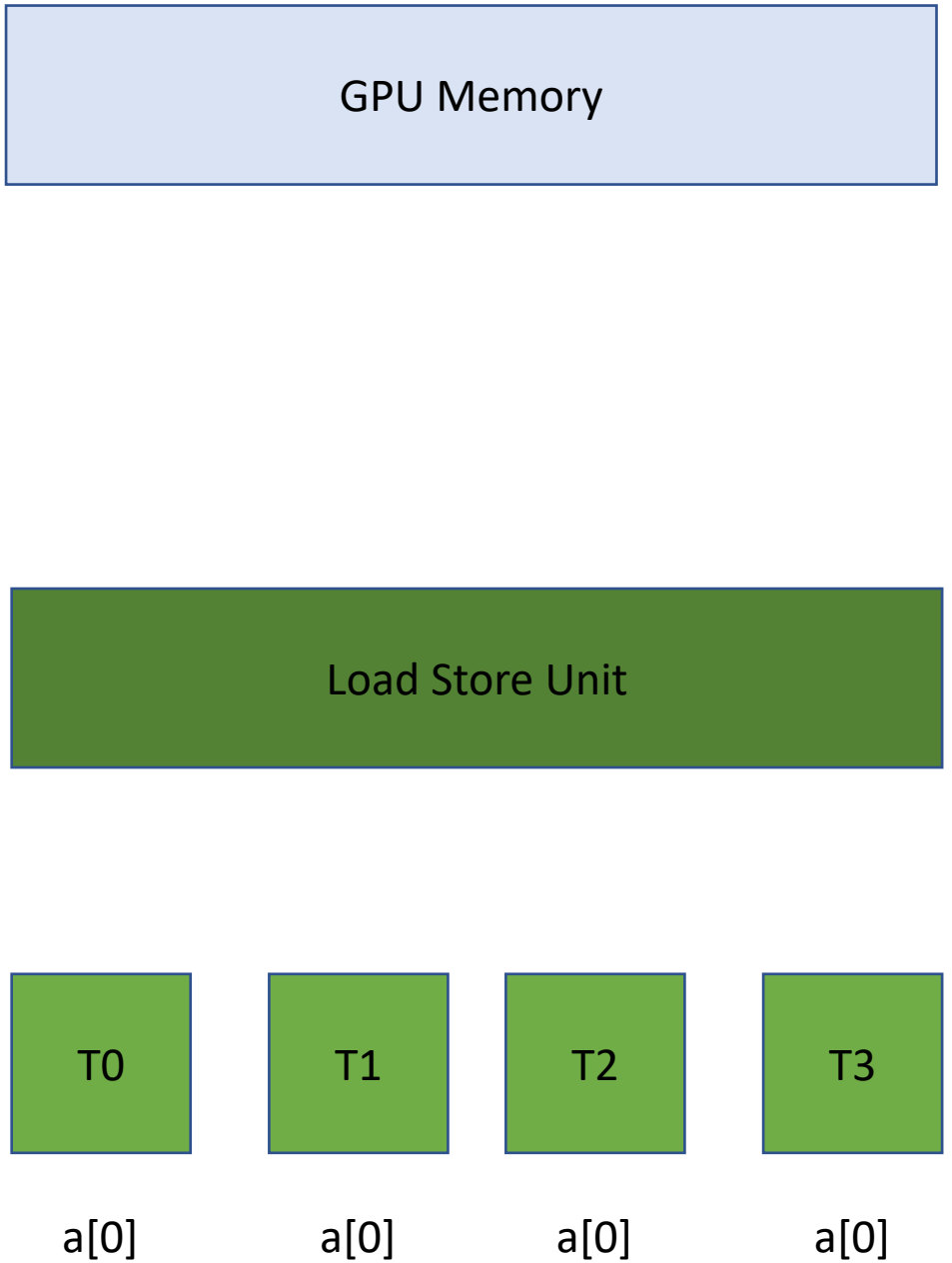
4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen



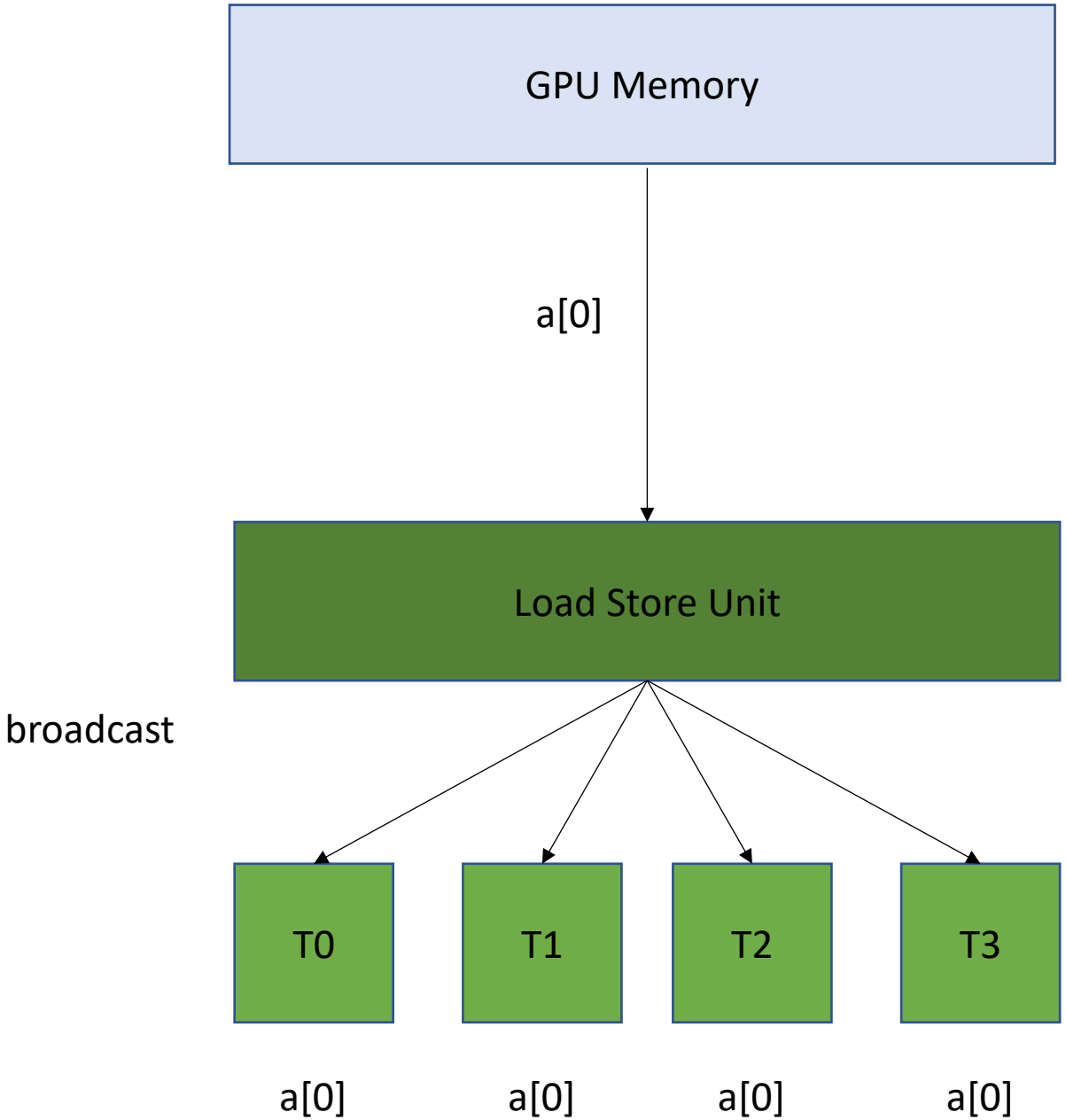
4 cores are accessing memory. What can happen

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.

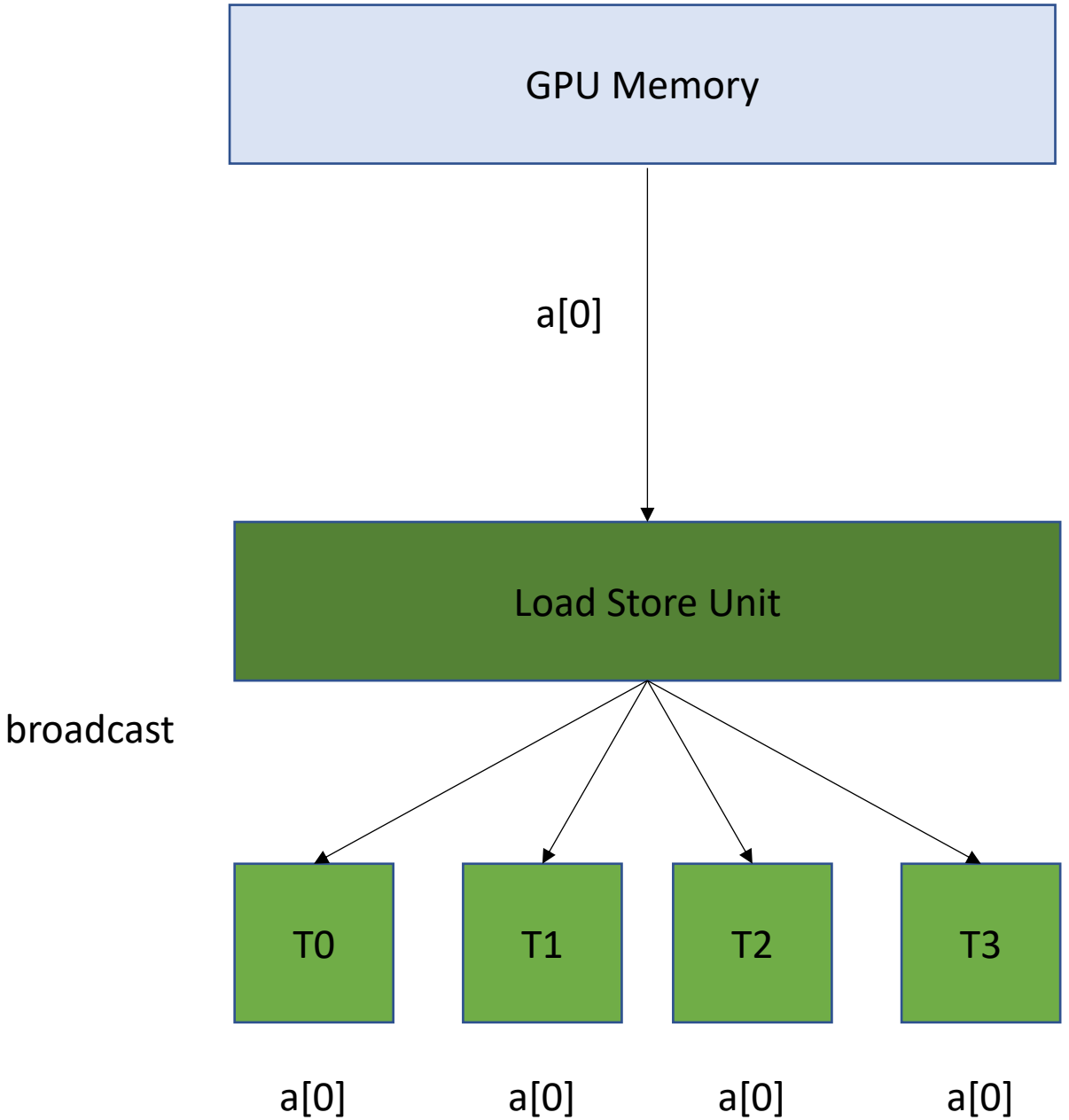


4 cores are accessing memory. What can happen

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

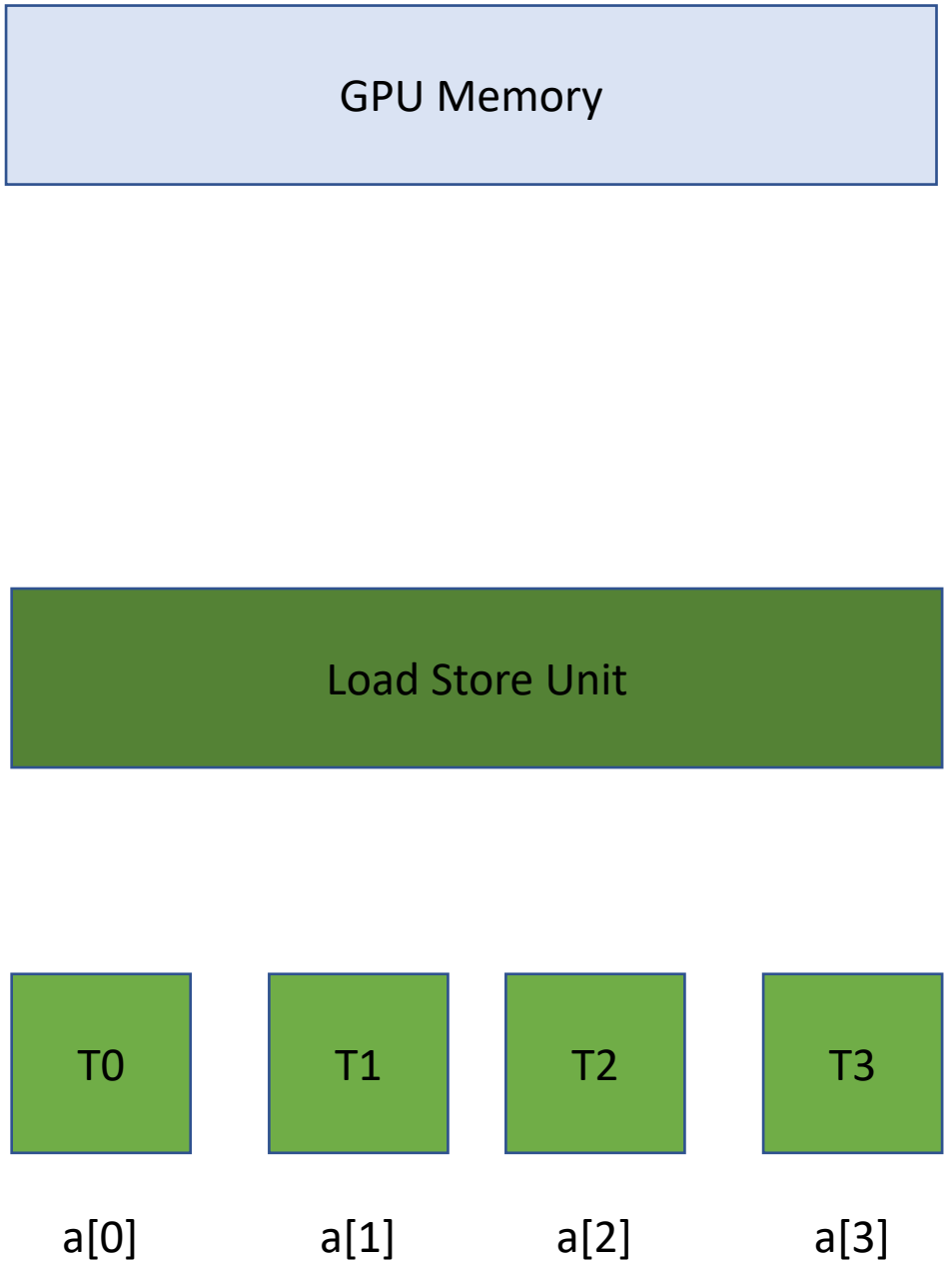
1 request to GPU memory

Efficient, but probably not too common.



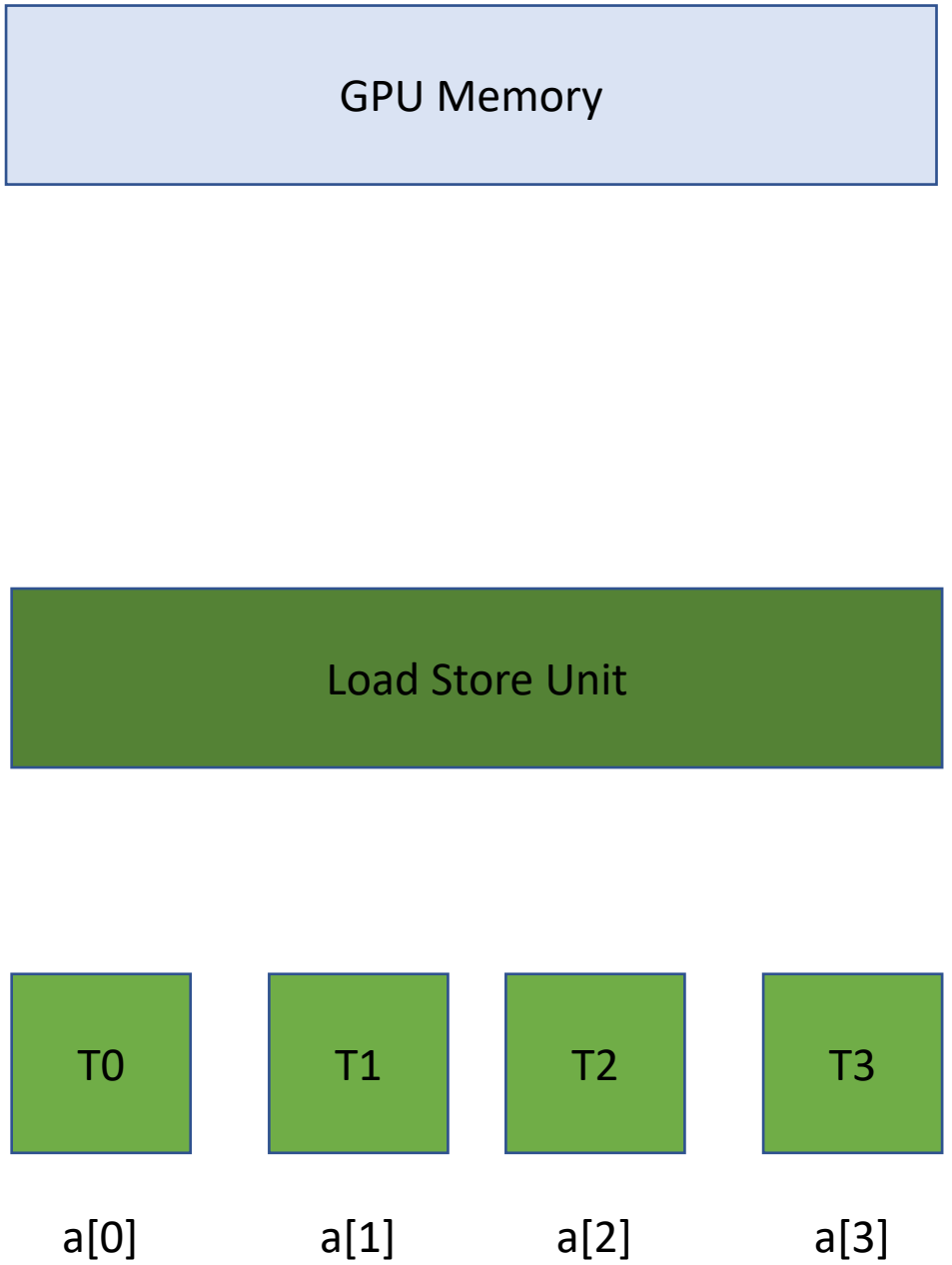
4 cores are accessing memory. What can happen

Read contiguous values



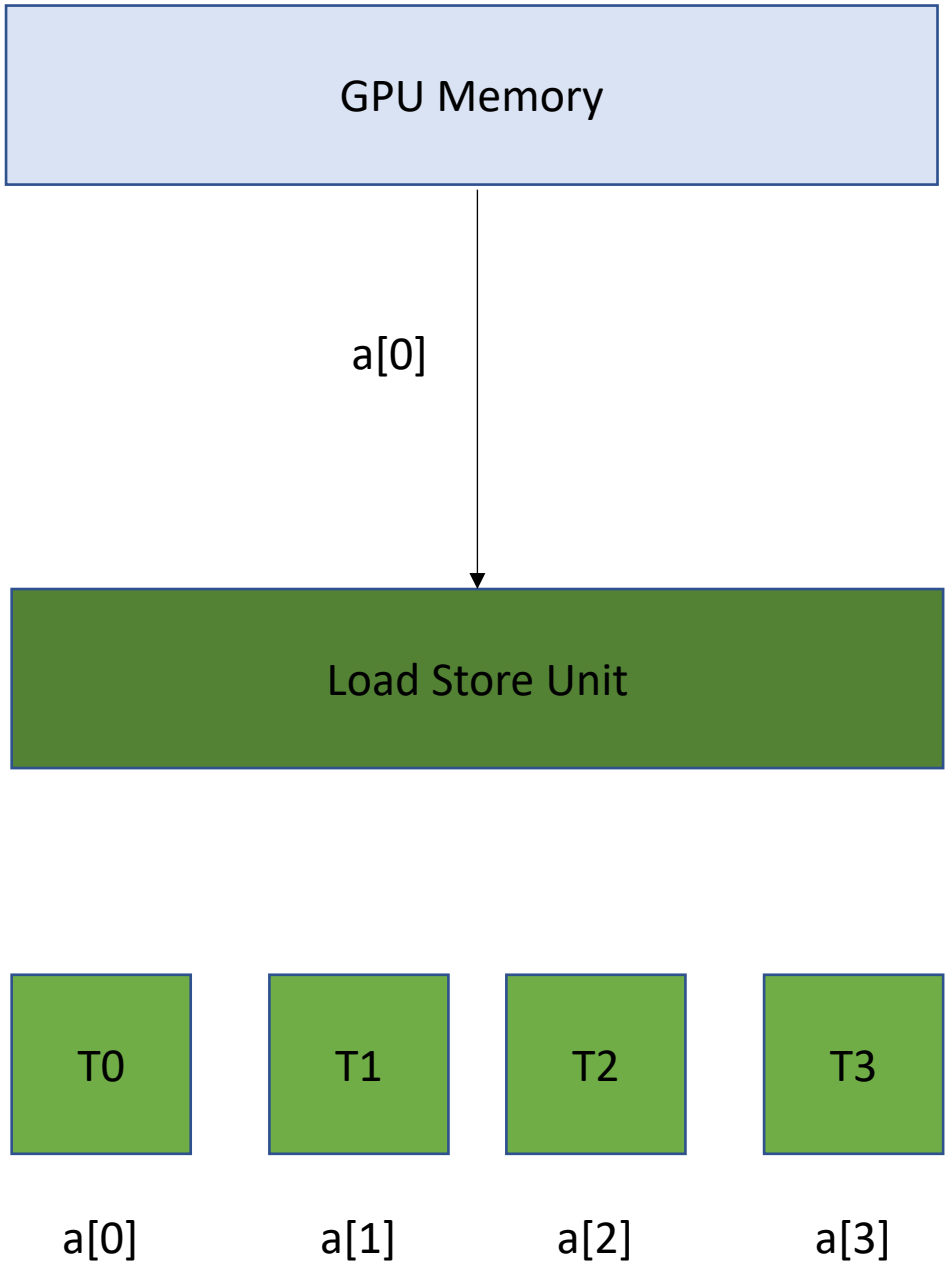
4 cores are accessing memory. What can happen

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

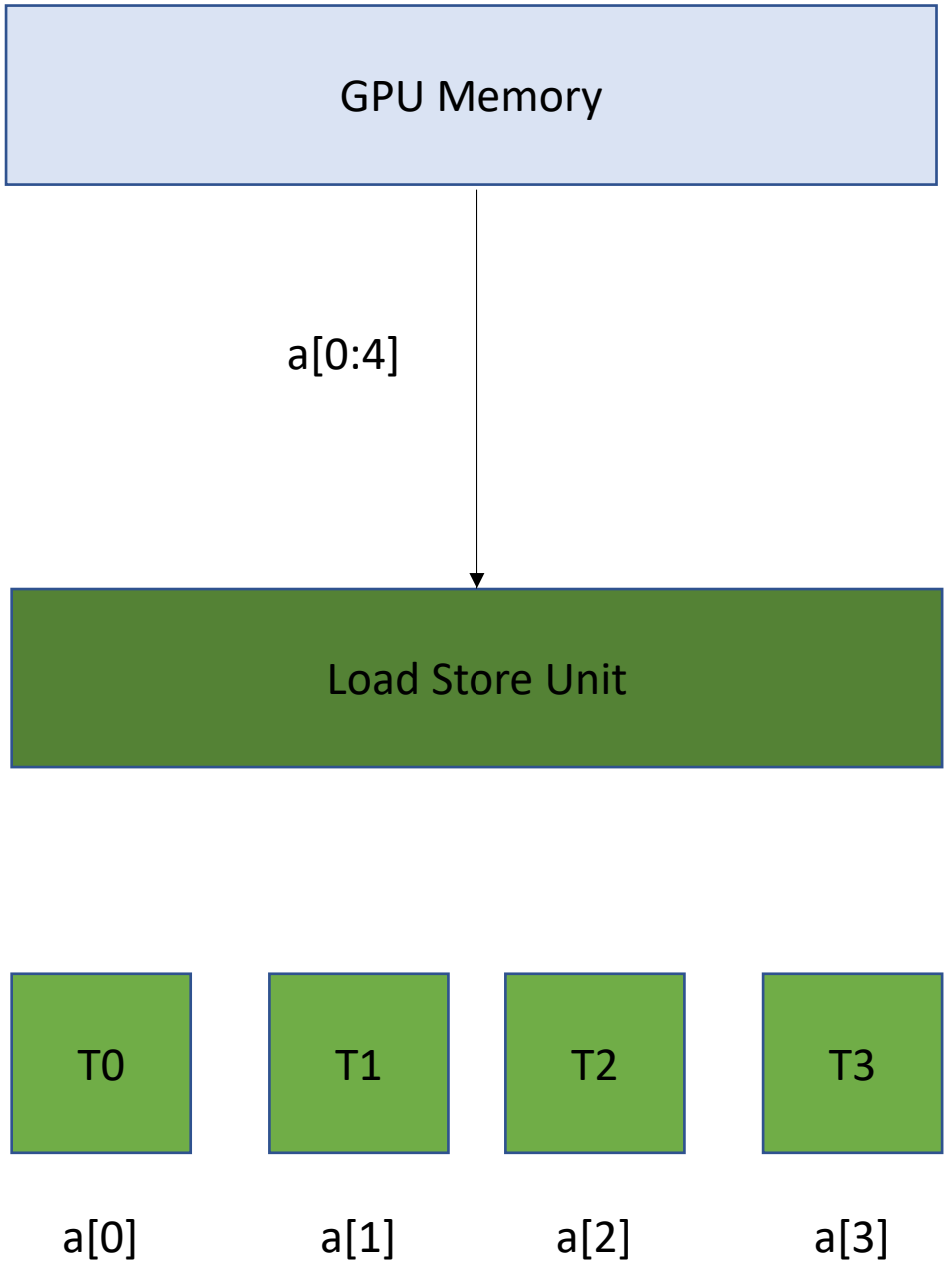
Read contiguous values
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes



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

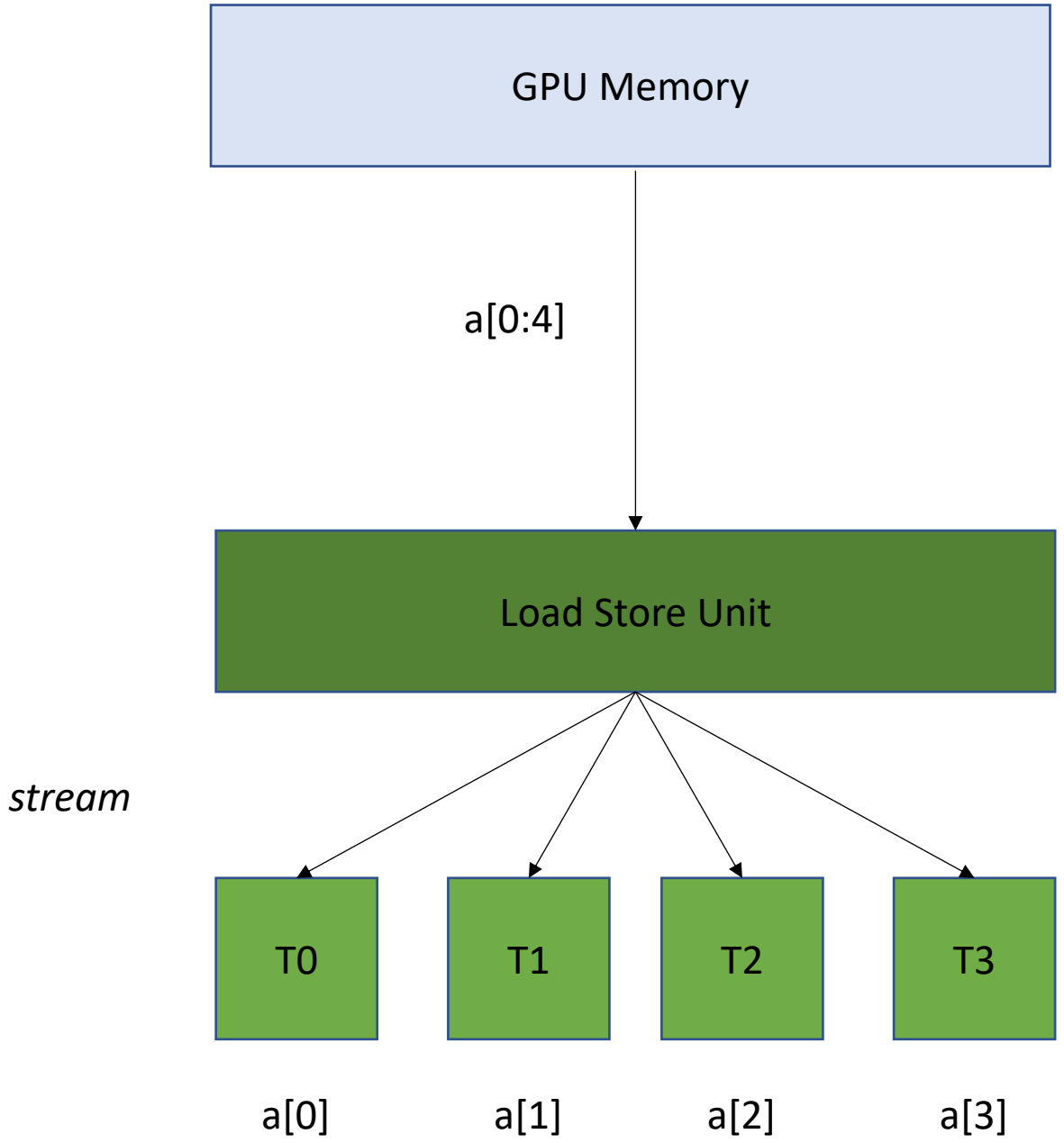


4 cores are accessing memory. What can happen

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

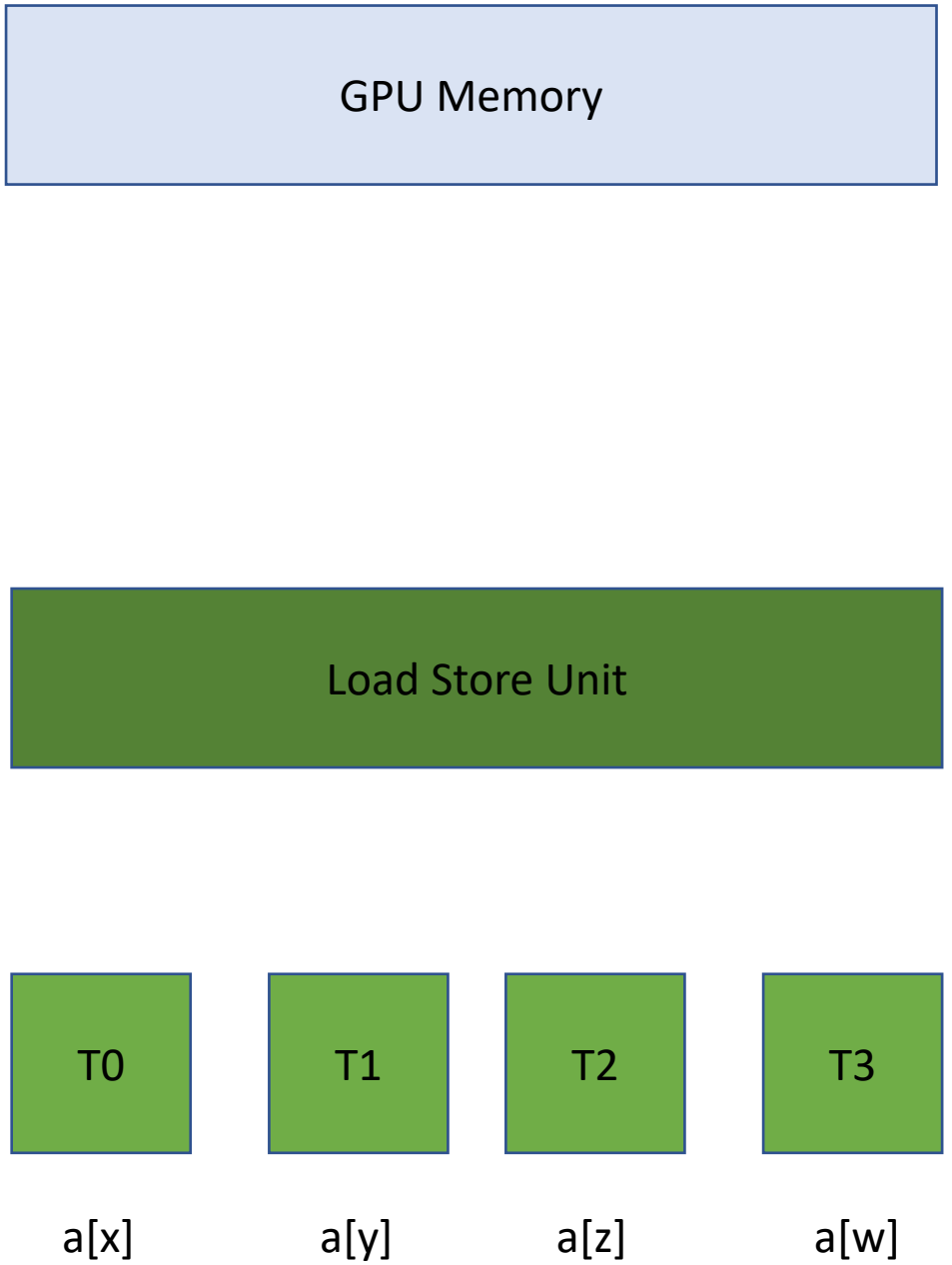


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

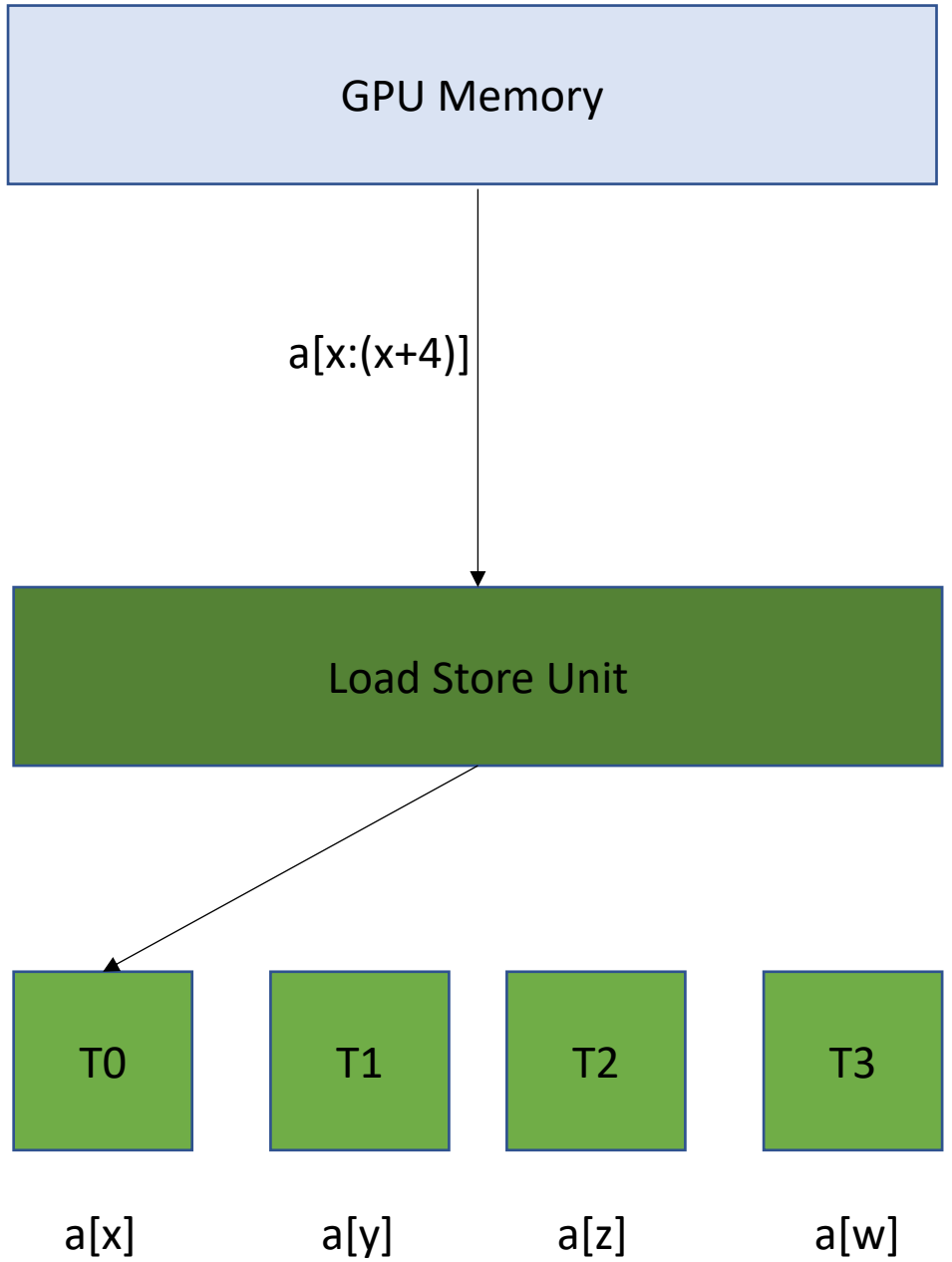


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

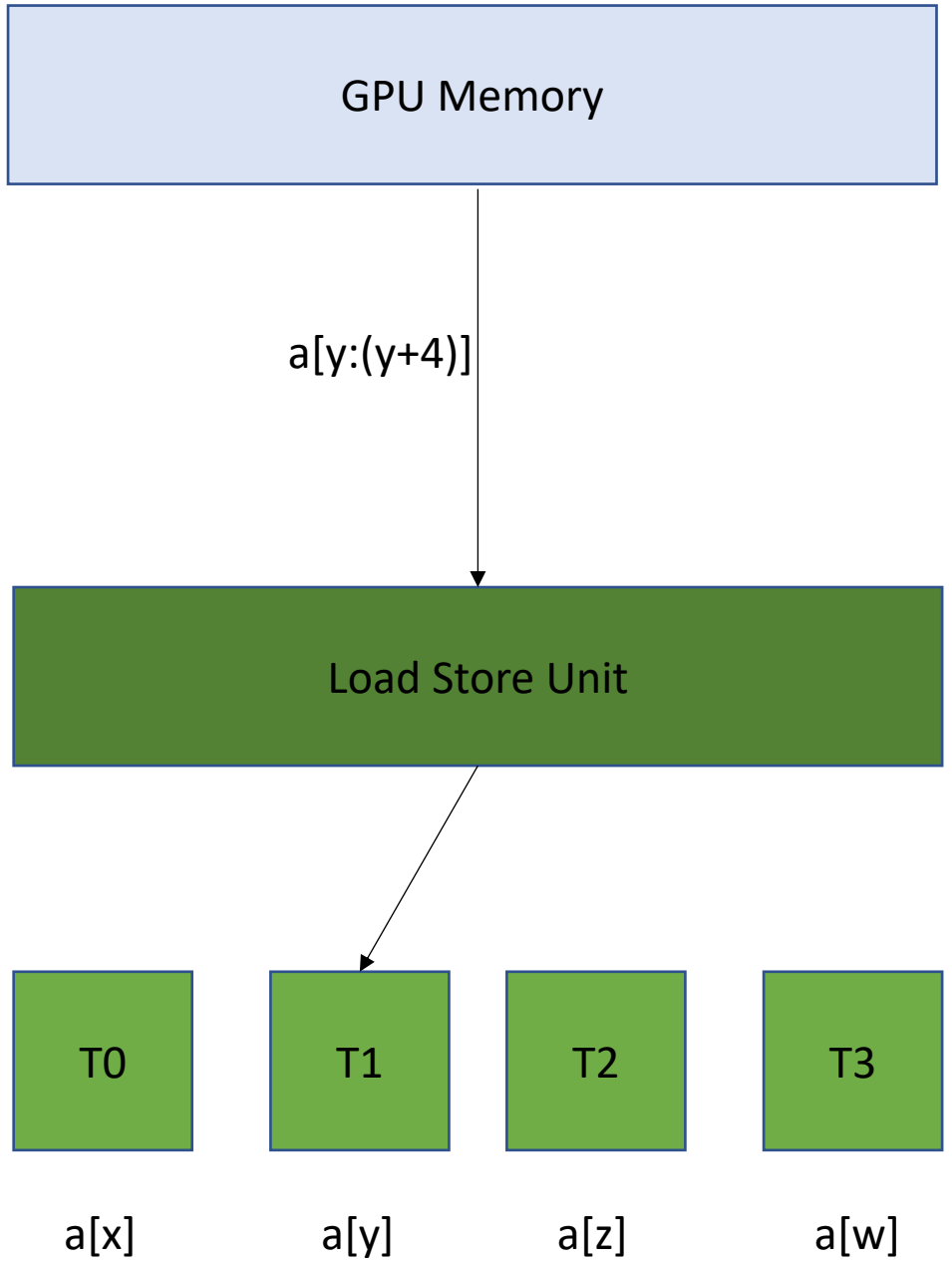


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

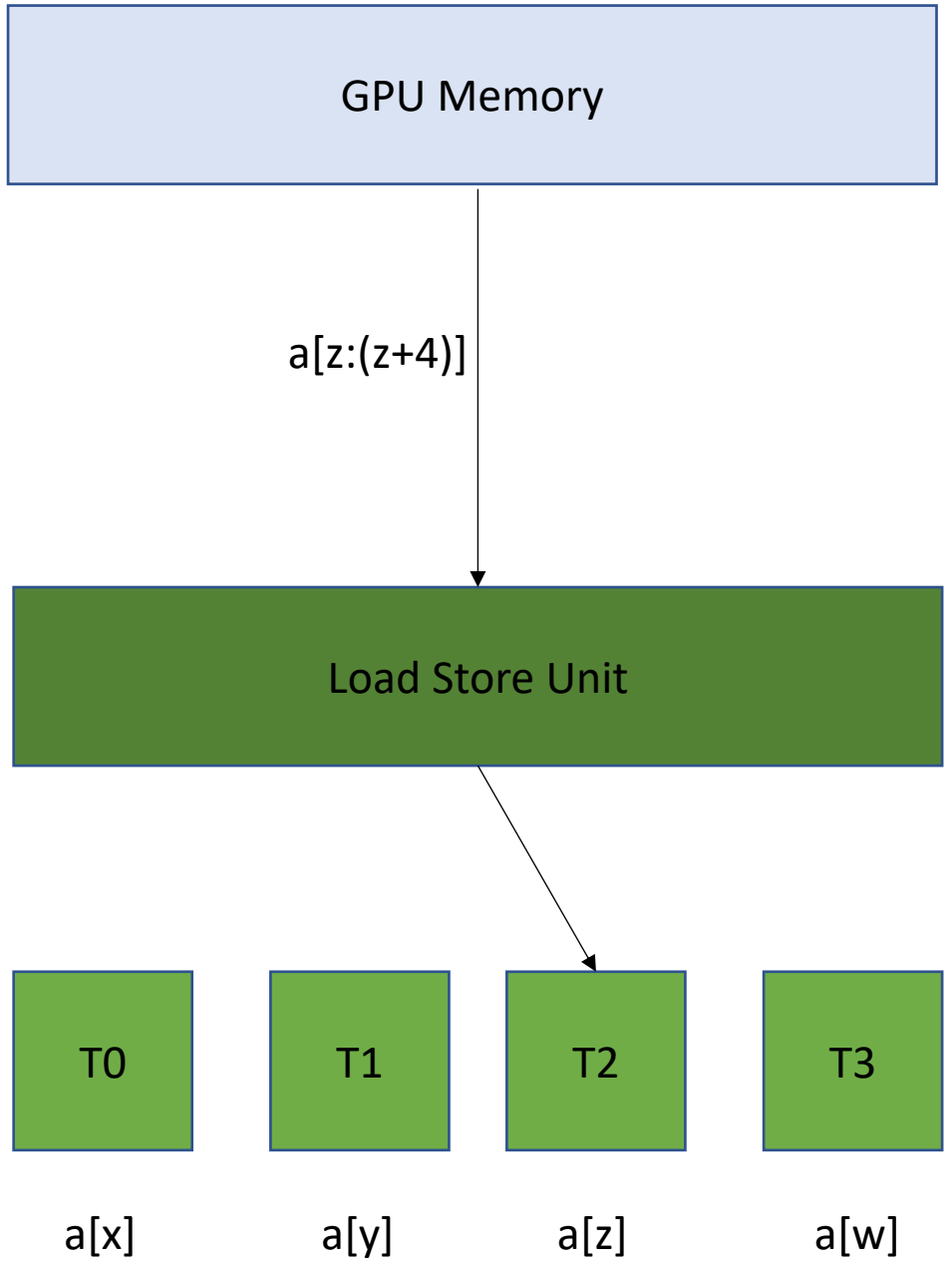


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

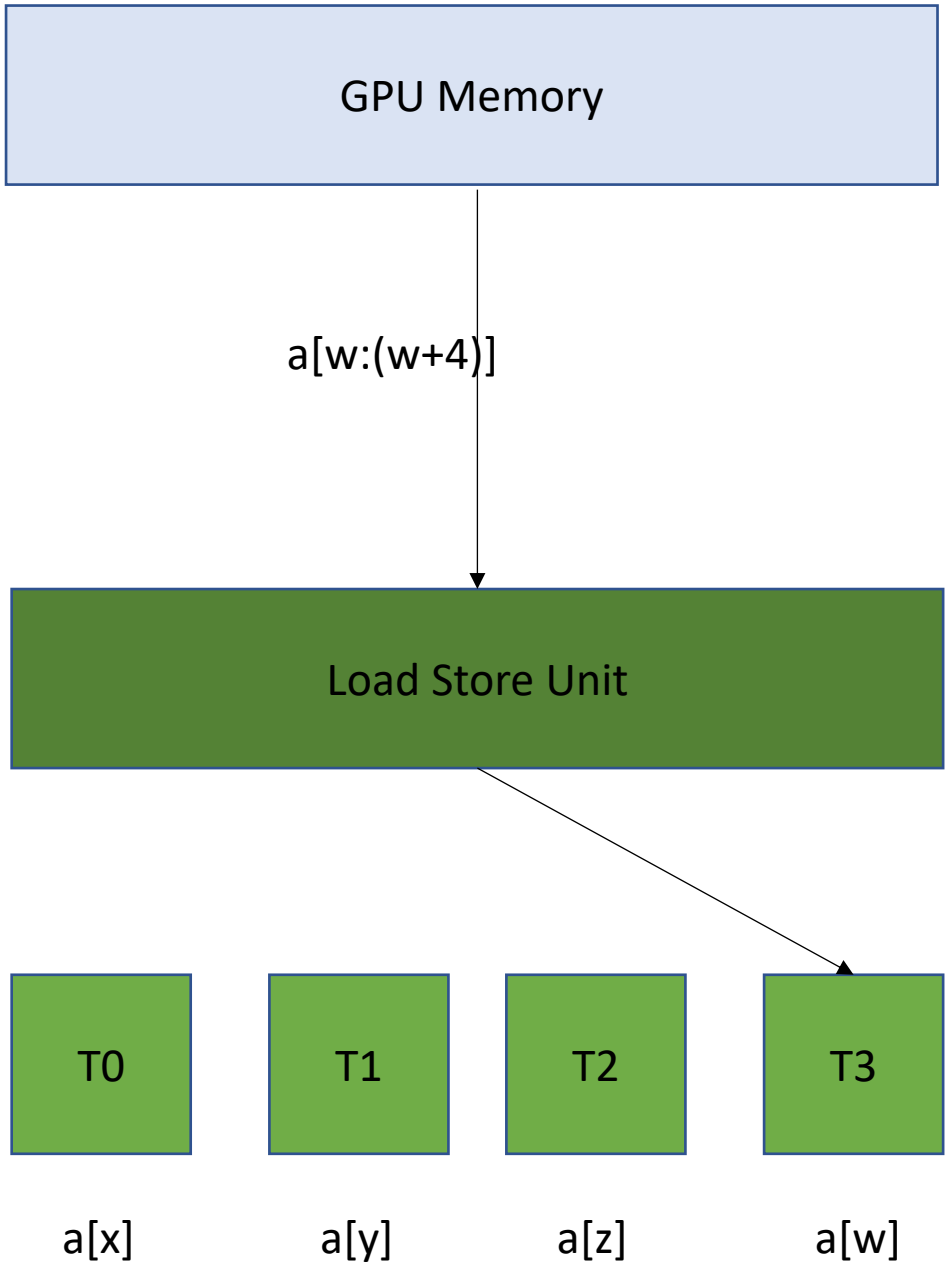


4 cores are accessing memory. What can happen

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];
   }
}
```

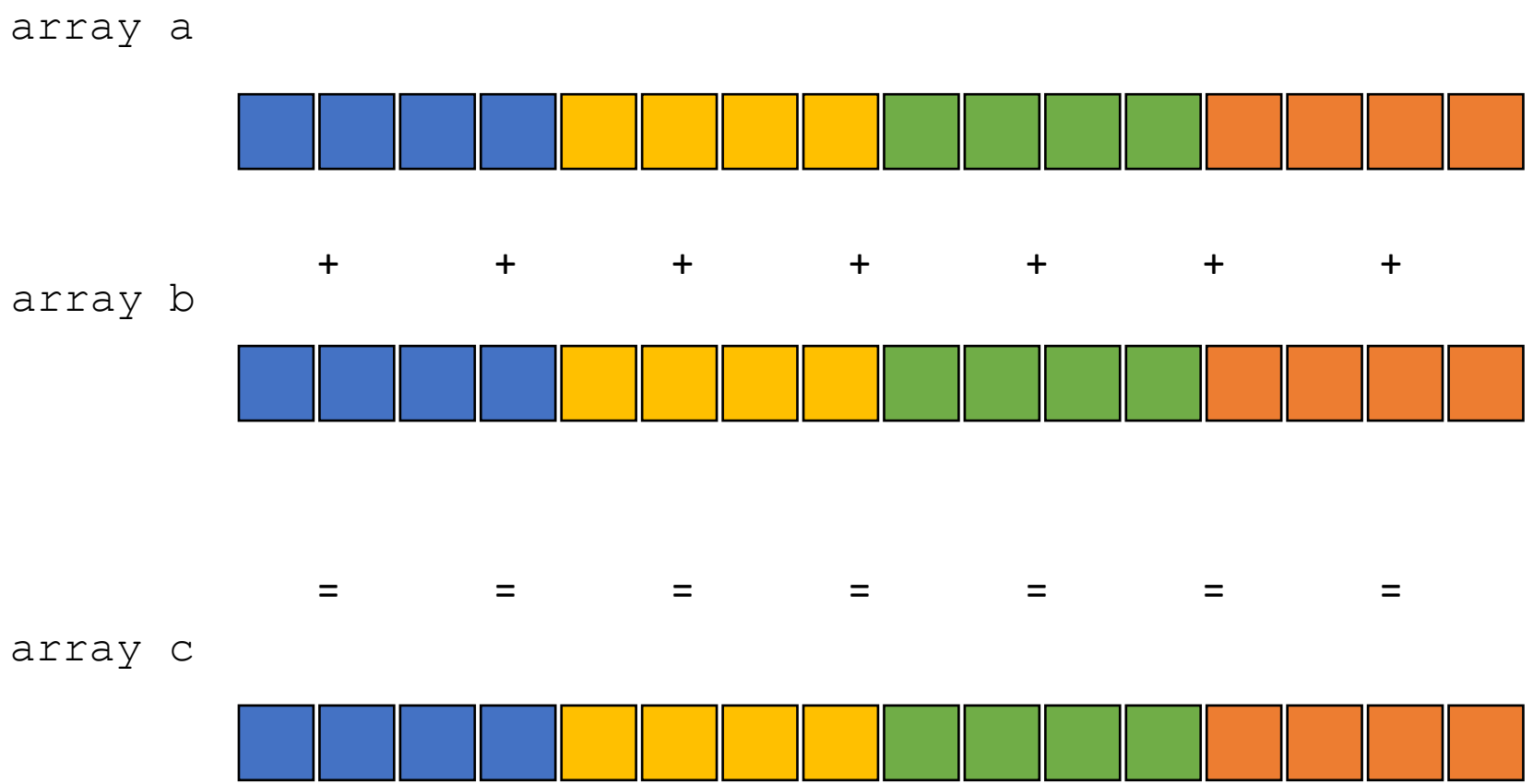
calling the function

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

Chunked Pattern

Computation can easily be divided into threads

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

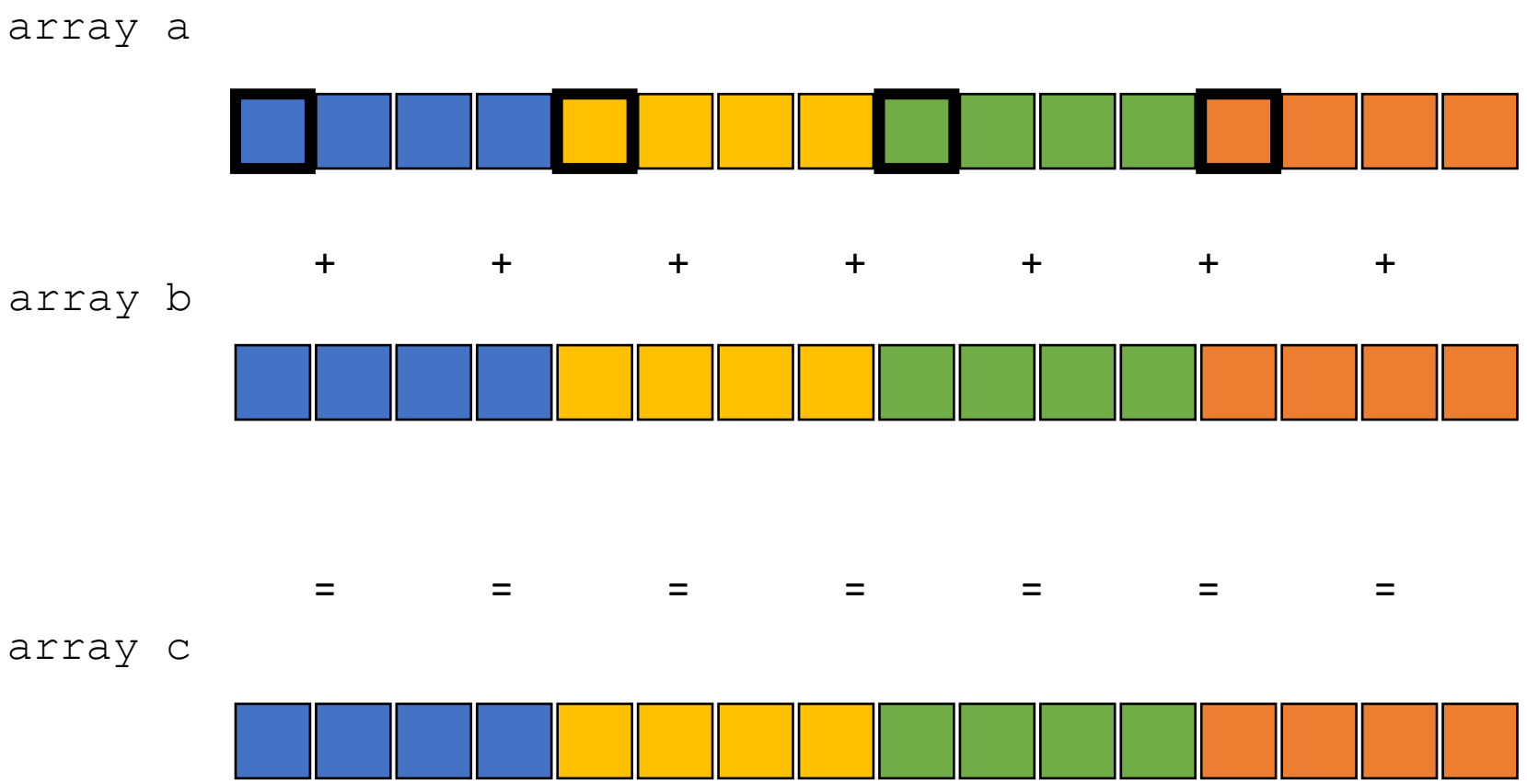


Chunked Pattern

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

Computation can easily be divided into threads

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

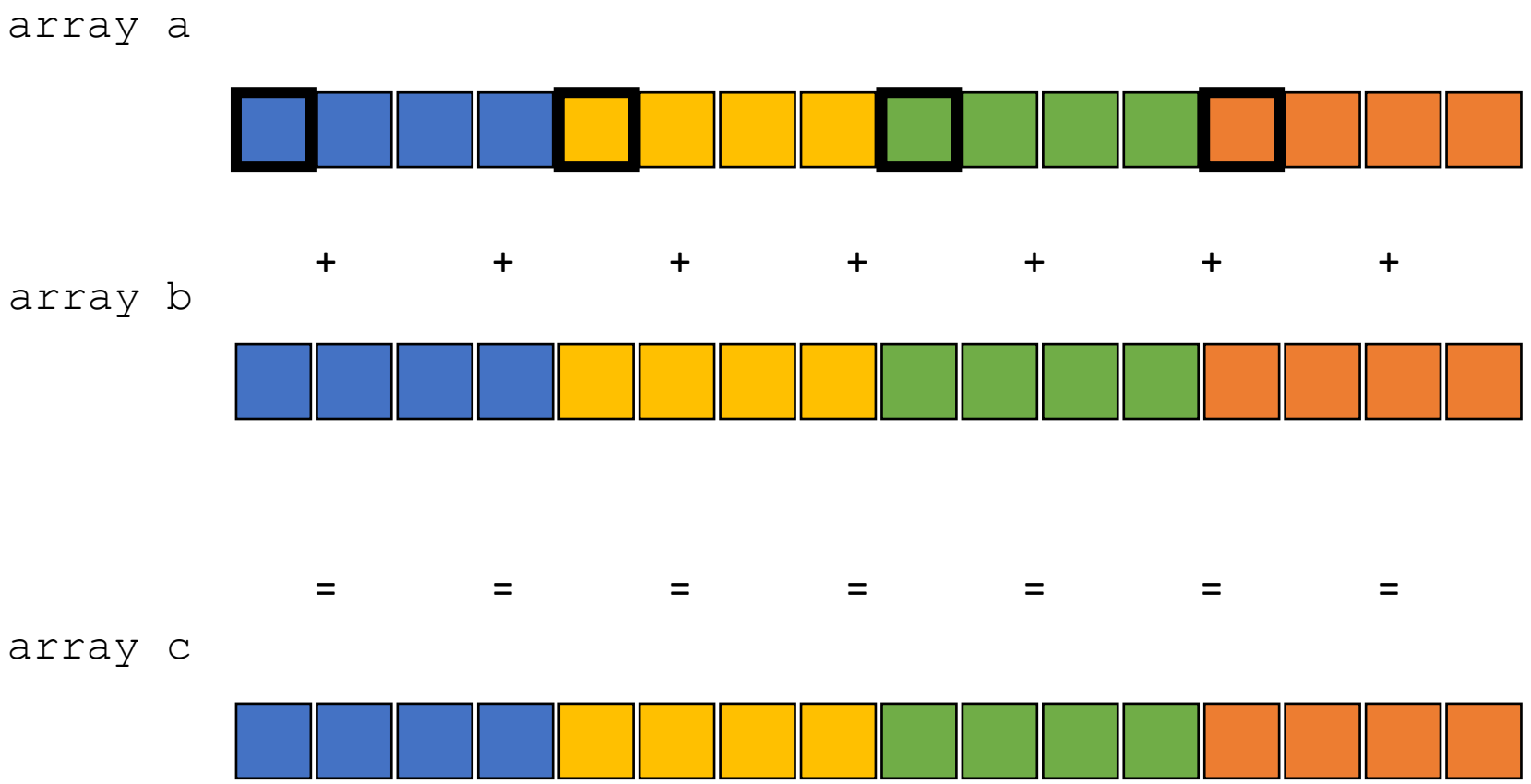


Chunked Pattern

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

Computation can easily be divided into threads

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

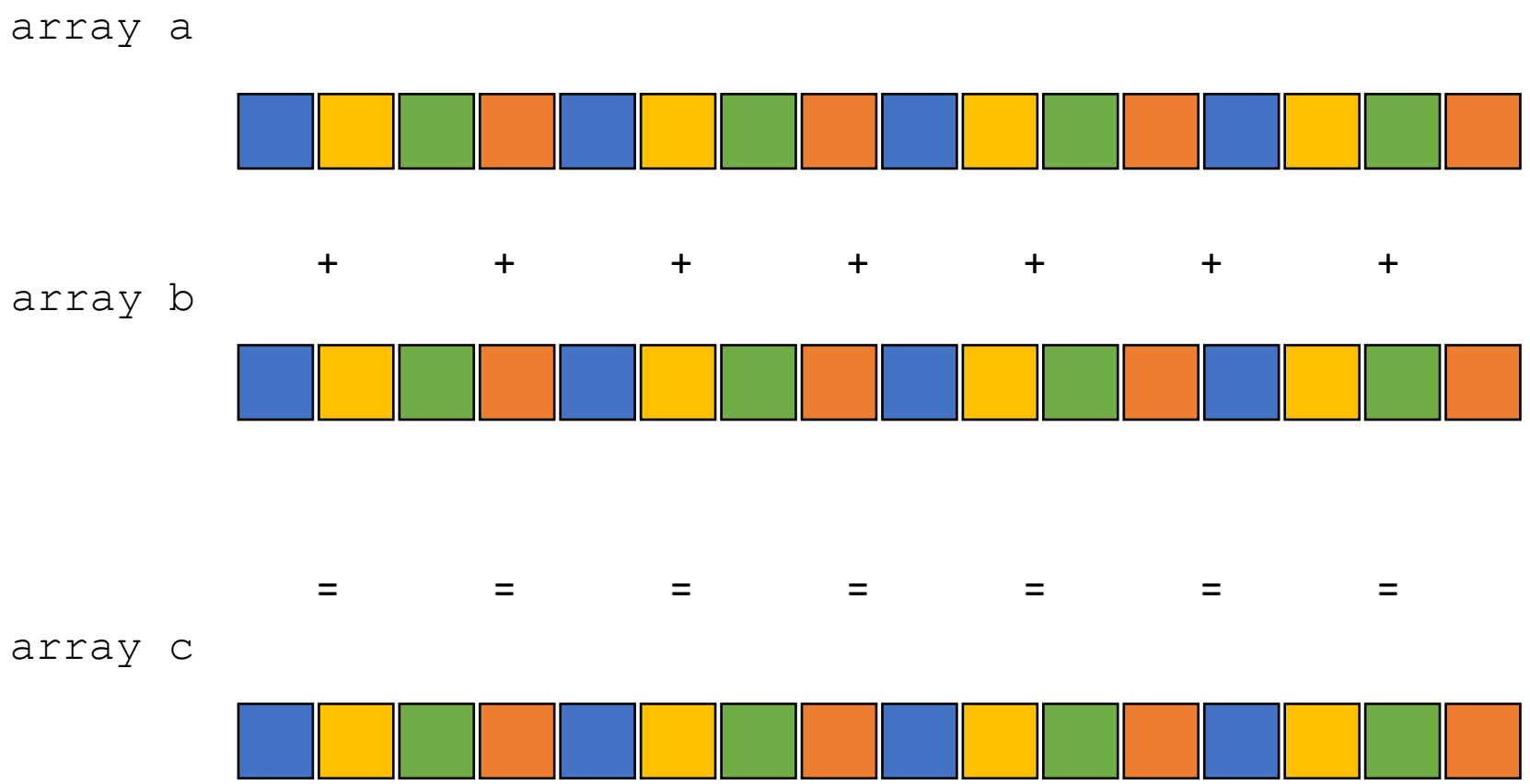


How can we fix this

Stride Pattern

Computation can easily be divided into threads

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

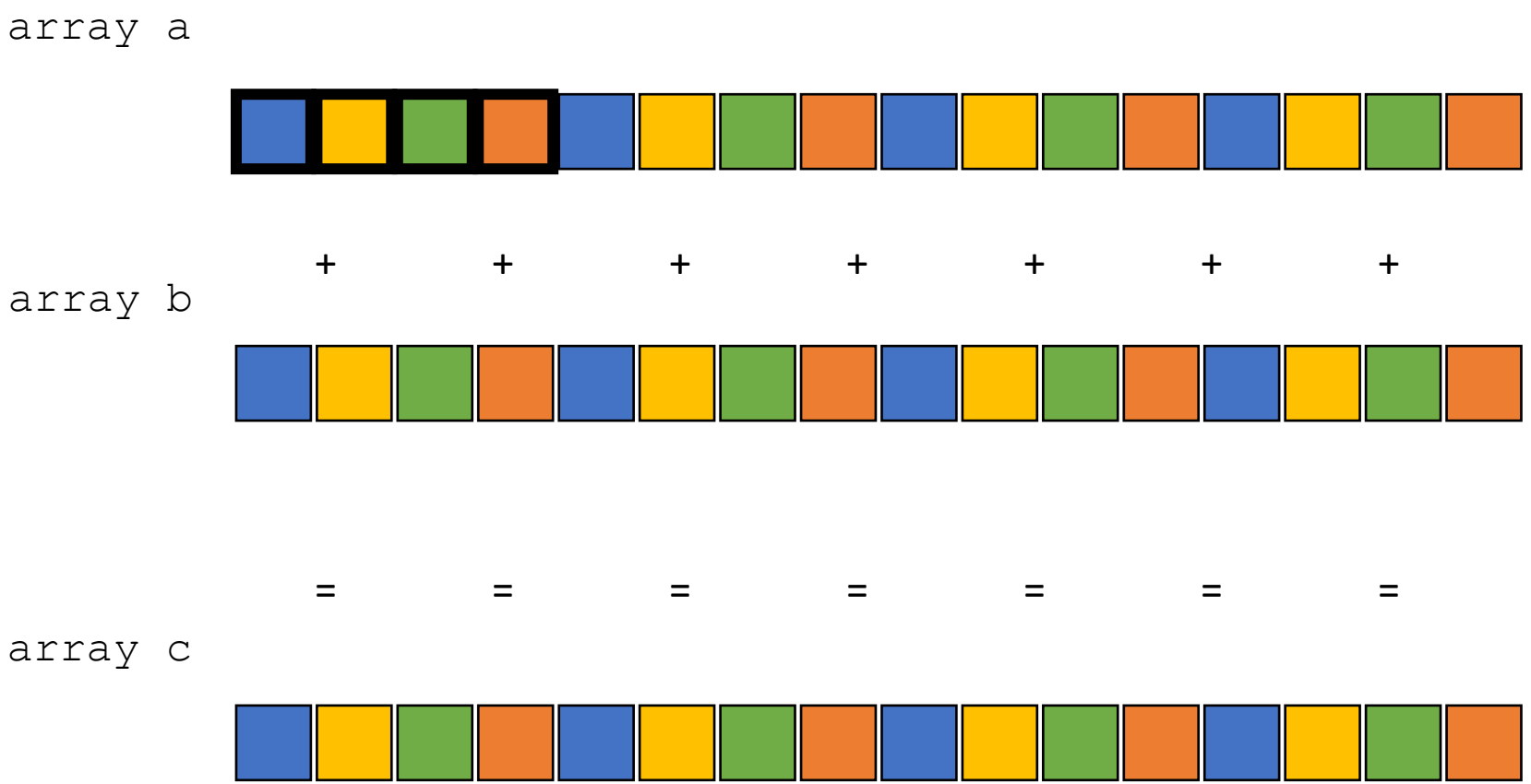


Stride Pattern

What sort of pattern is this?

Computation can easily be divided into threads

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



Go back to our program

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

calling the function

Lets change this to a stride pattern

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

Go back to our program

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

calling the function

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

Coalesced memory accesses

This is about 4x faster.

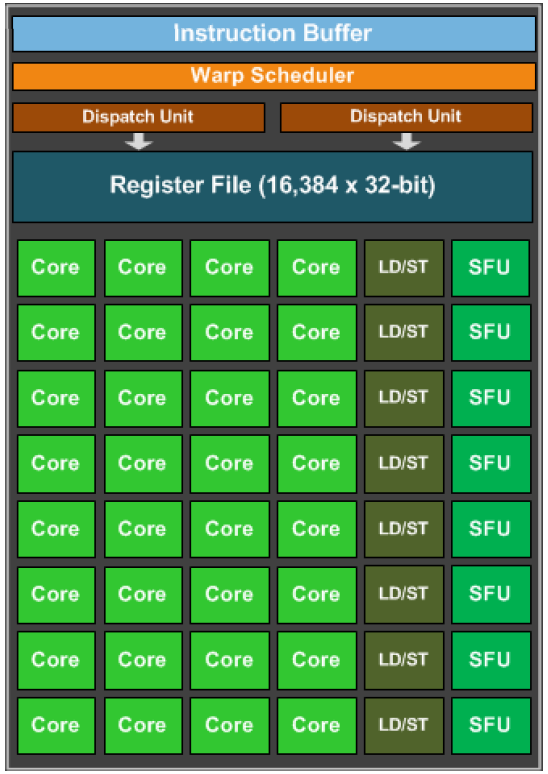
Coalesced memory accesses

This is about 4x faster.

What else can we do?

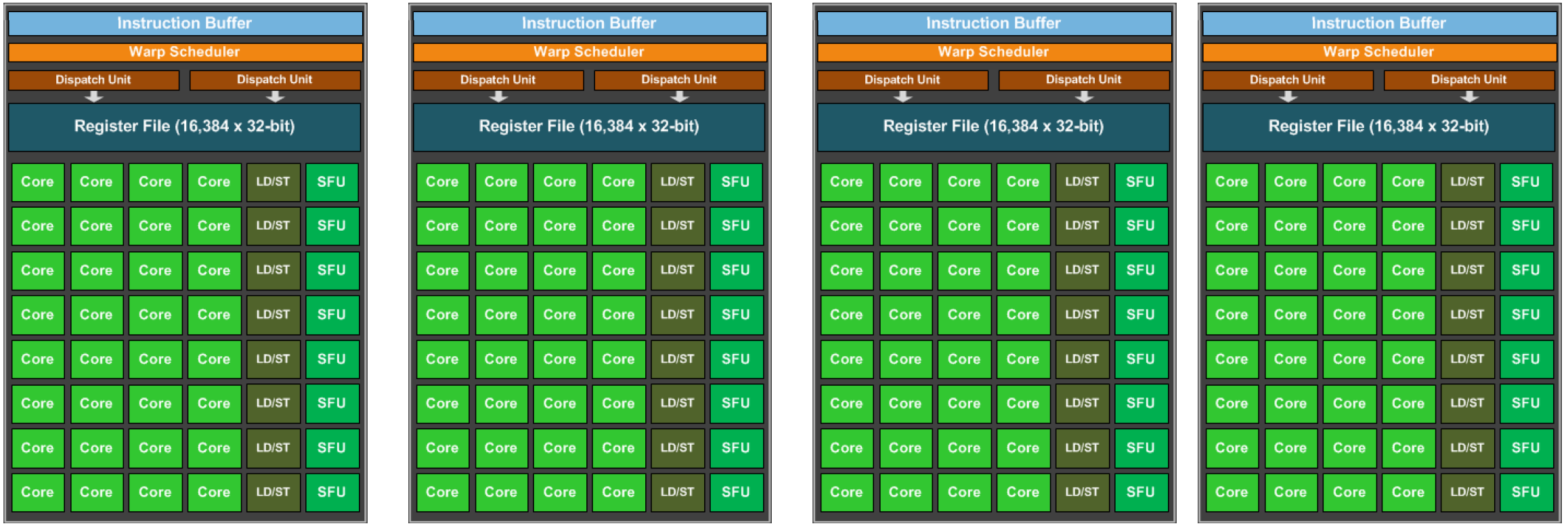
Multiple streaming multiprocessors

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



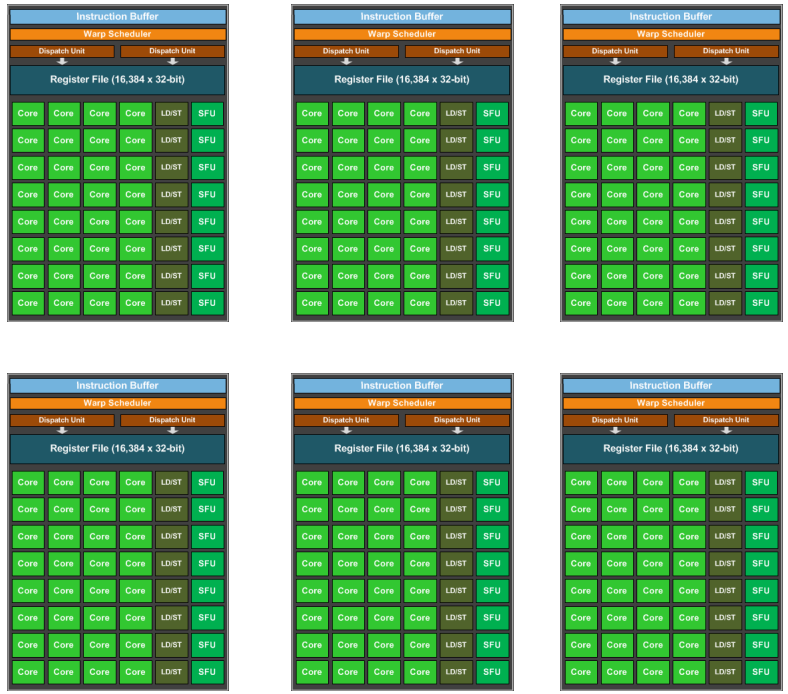
Multiple streaming multiprocessors

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



Multiple streaming multiprocessors

CUDA provides virtual streaming multiprocessors called **blocks**
They hide memory latency
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

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

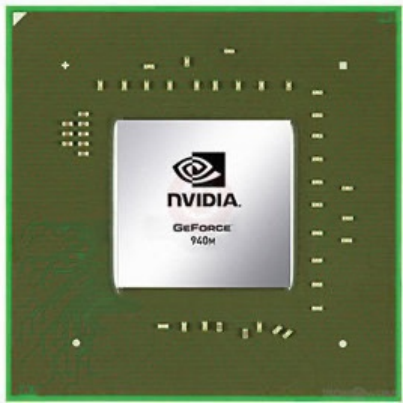
Launch with many thread blocks

Now we have 1 thread for each element

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

Final Round

Tiny GPU in an embedded system



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

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

With 10x energy efficiency, GPU is fast as CPU.
With the same energy efficiency, GPU is about 33x as CPU.