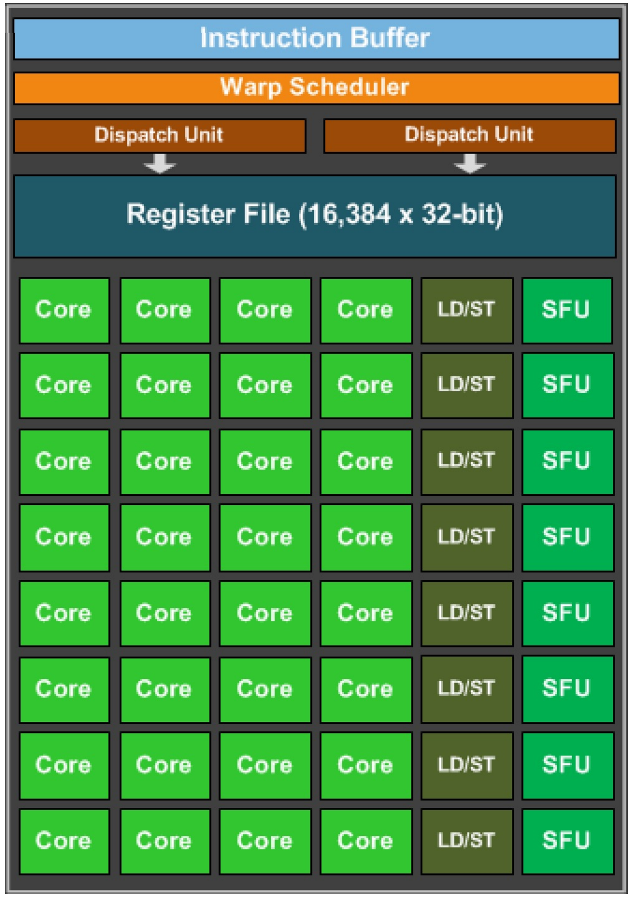
CSE113: Parallel Programming

March 11, 2022

- **Topics**:
  - Conclude GPU Programming
  - Homework 5 WebGPU
  - Conclude class



Announcements

- HW 5 is out
  - Hopefully you have started!
  - Due the day before the final
  - LATE HW 5 WILL NOT BE ACCEPTED
    - This is not my policy, this is the university policy!
  - I will hold office hours Tuesday from 3 - 5 PM.
  - TAs and Tutors will NOT have office hours
- HW 3 grades are released
  - Let us know ASAP if there are issues
- We are grading HW 4 right now

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.
    - Late final will not be accepted
        - This is university policy, not mine

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 teach this in the future, SET feedback will help me do that

Quizzes

- No more quizzes!

Review

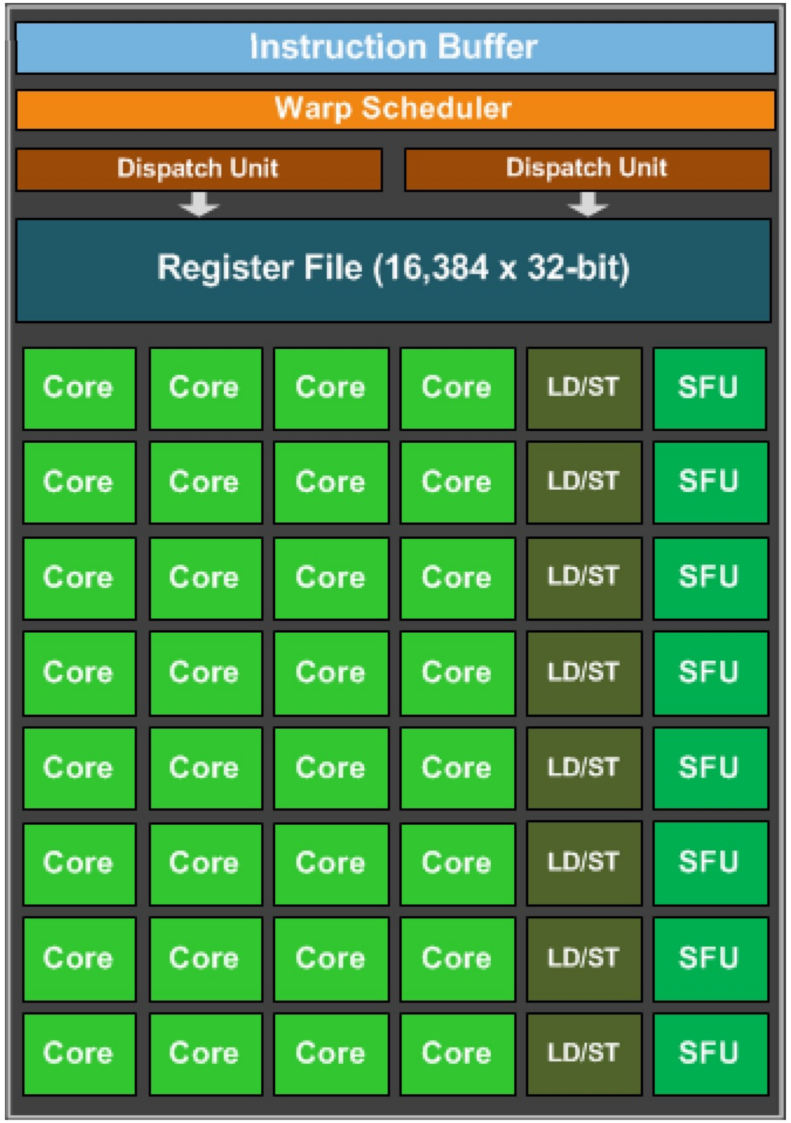
Optimizing GPU code

Parallelism

Called a streaming multiprocessor

woah, 32 cores!

We should parallelize our application!


Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU
Core Core Core Core LD/ST SFU

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) {
  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];
 }
}
```

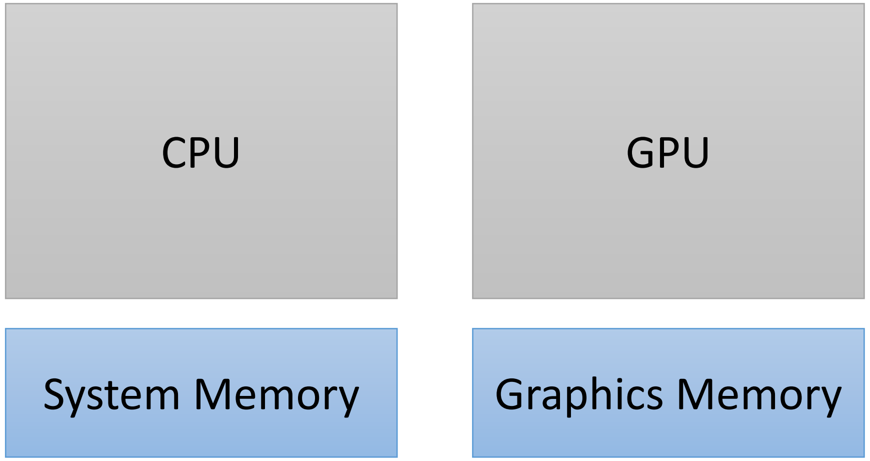
calling the function

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

number of threads
thread id

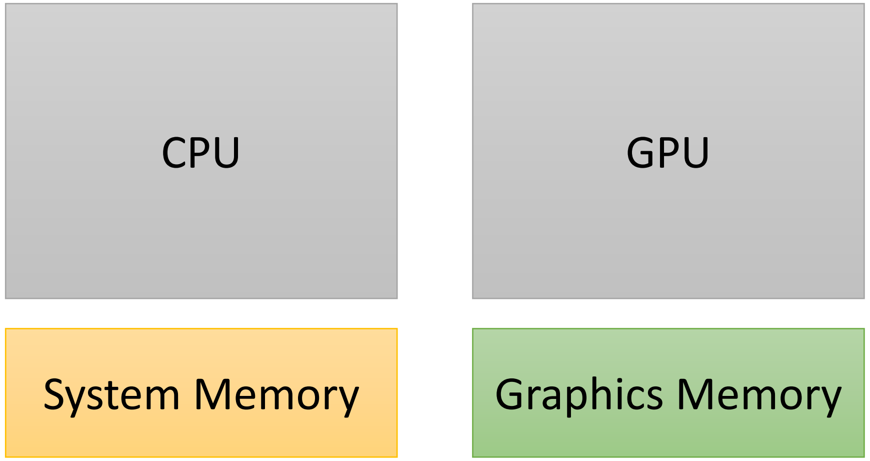
Concurrency

GPU Memory



GPU Memory

CPU Memory:
Fast: Low Latency
Easily saturated: Low Bandwidth
Scales well: up to 1 TB
DDR



CPU

GPU

System Memory

Graphics Memory

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

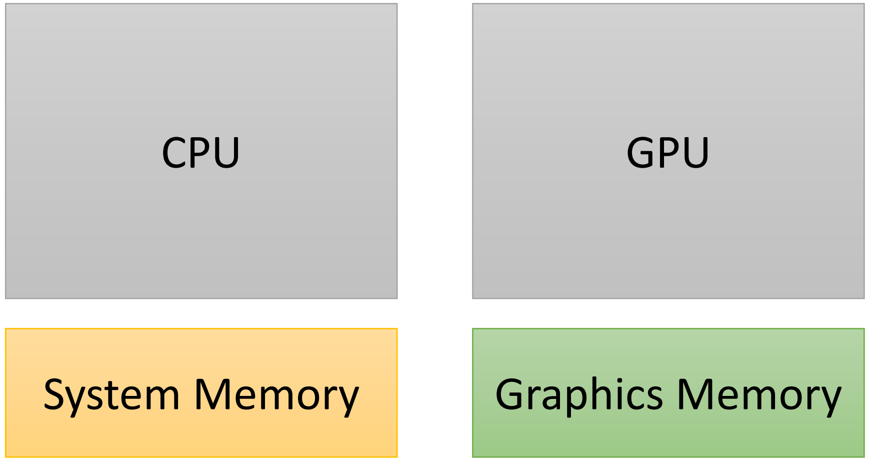
16-lane highway on a windy road driven by semi trucks

GPU Memory

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

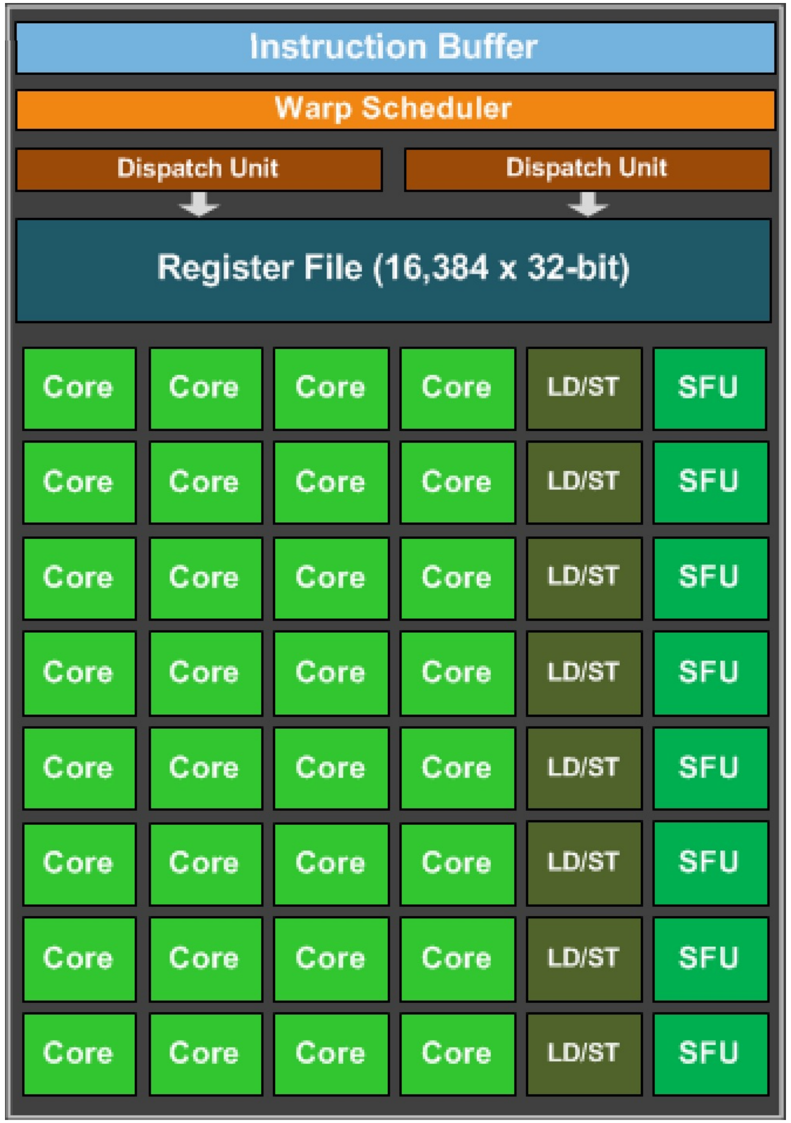
memory Latency:
~**600** cycles for GPU memory
~**200** cycles for CPU memory

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



Warps

A warp is a group of 32 threads that execute in parallel on a streaming multiprocessor



Instruction Buffer

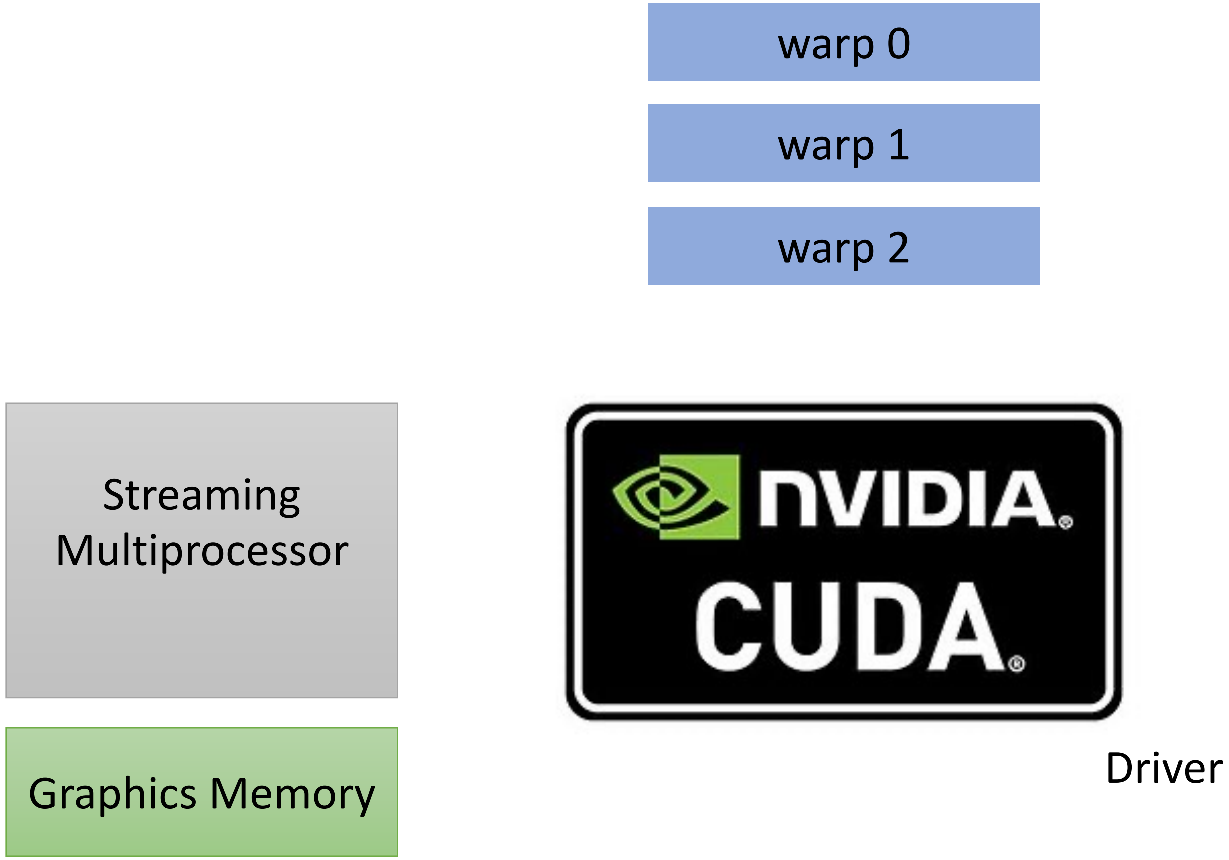
Warp Scheduler

Dispatch Unit | Dispatch Unit

Register File (16,384 x 32-bit)

Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU

Preemption and concurrency?



warp 0

warp 1

warp 2

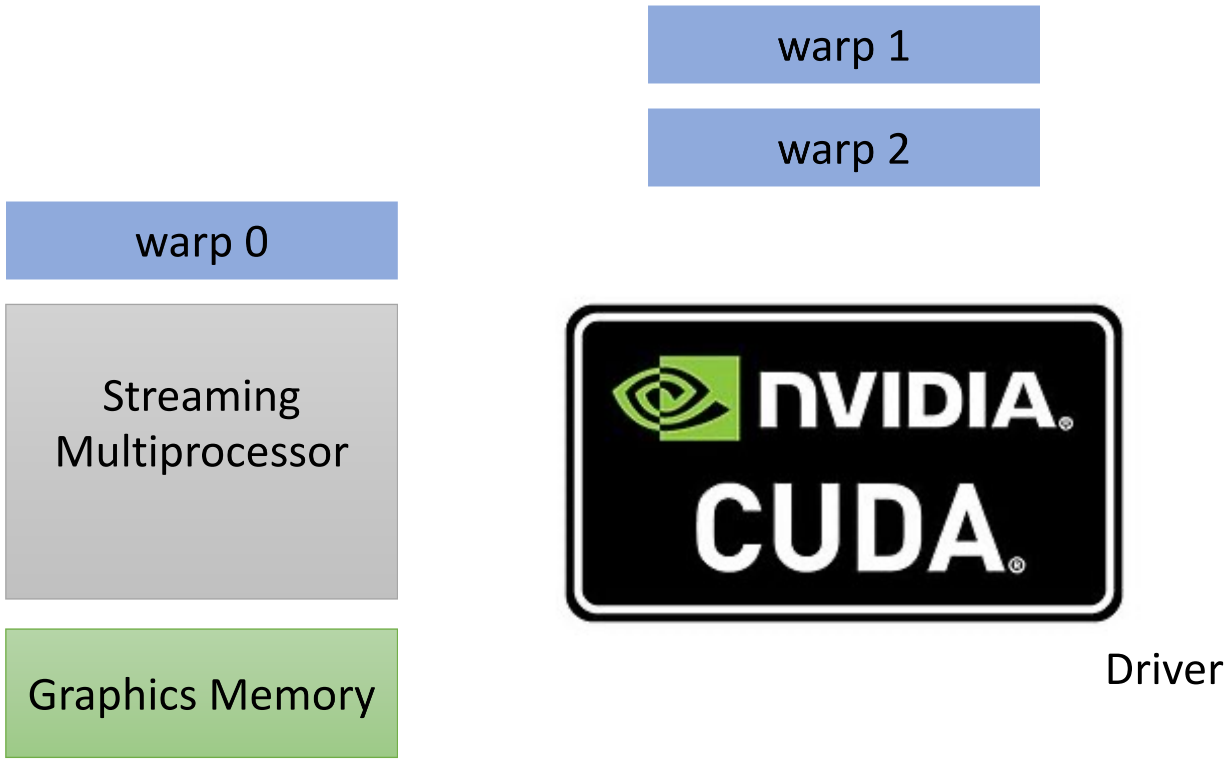
Streaming Multiprocessor

Graphics Memory

Driver

We can hide latency through preemption and concurrency!

Preemption and concurrency?



warp 1

warp 2

warp 0

Streaming Multiprocessor

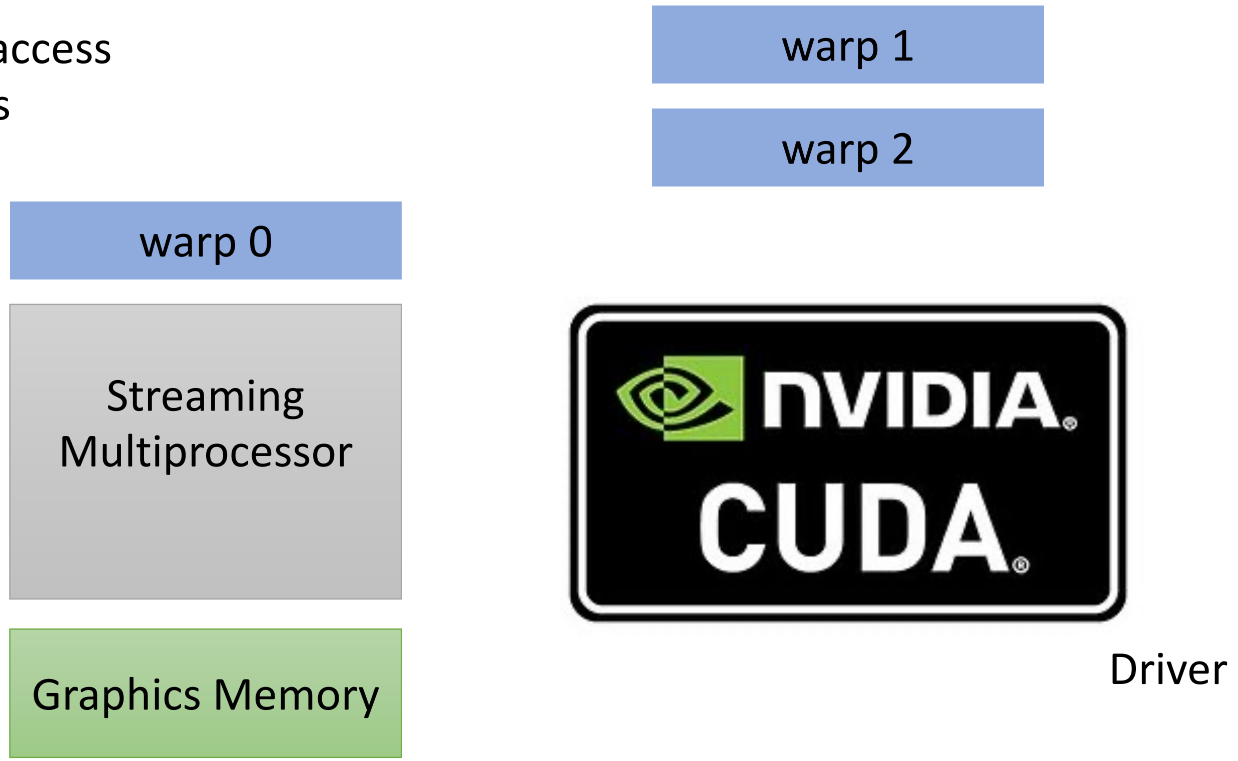
Graphics Memory

Driver

We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles



warp 1

warp 2

warp 0

Streaming Multiprocessor

Graphics Memory

Driver

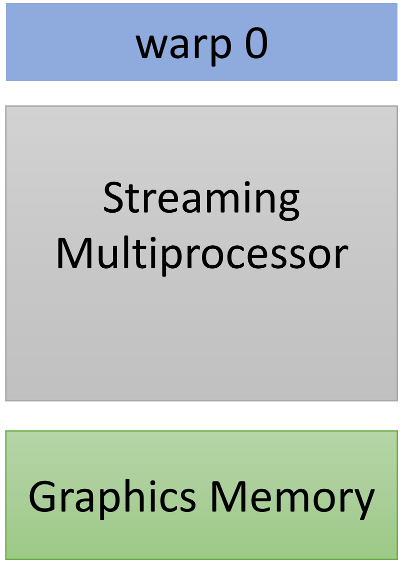
We can hide latency through preemption and concurrency!

Preemption and concurrency?

warp 1

warp 2

memory access
600 cycles

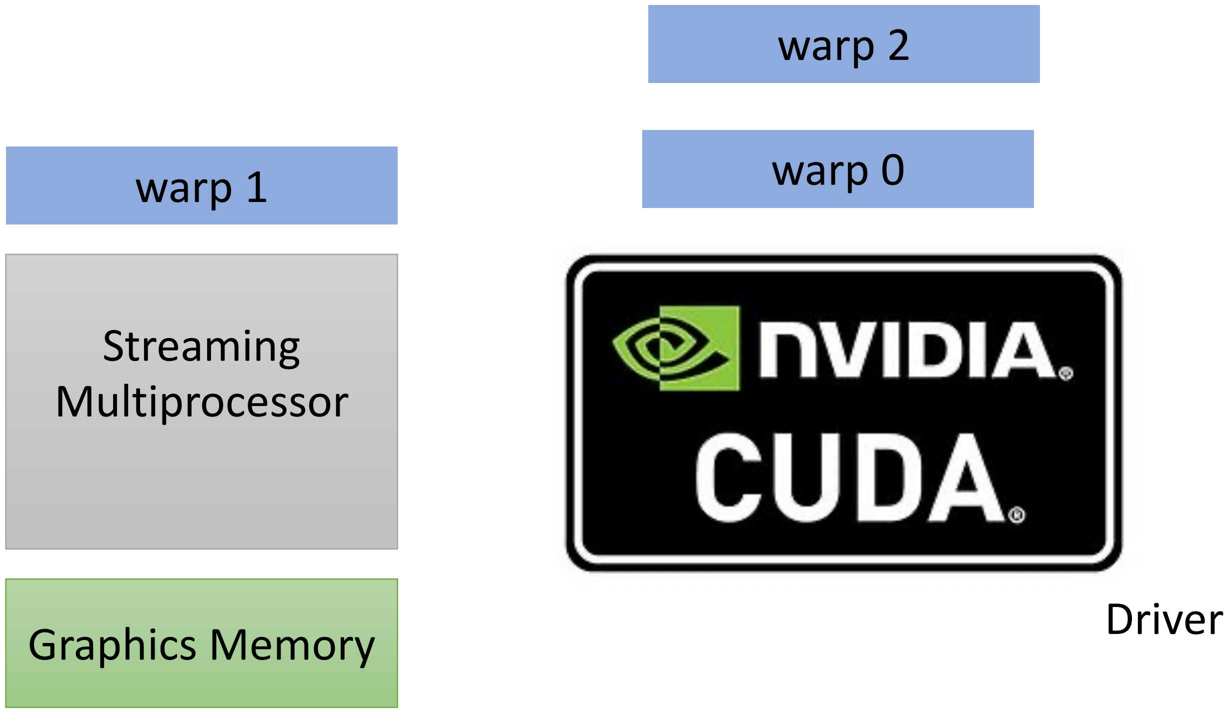


We can hide latency through preemption and concurrency!

Driver

preempt warp 0
and put warp 1 on

Preemption and concurrency?



warp 2

warp 1

warp 0

Streaming Multiprocessor

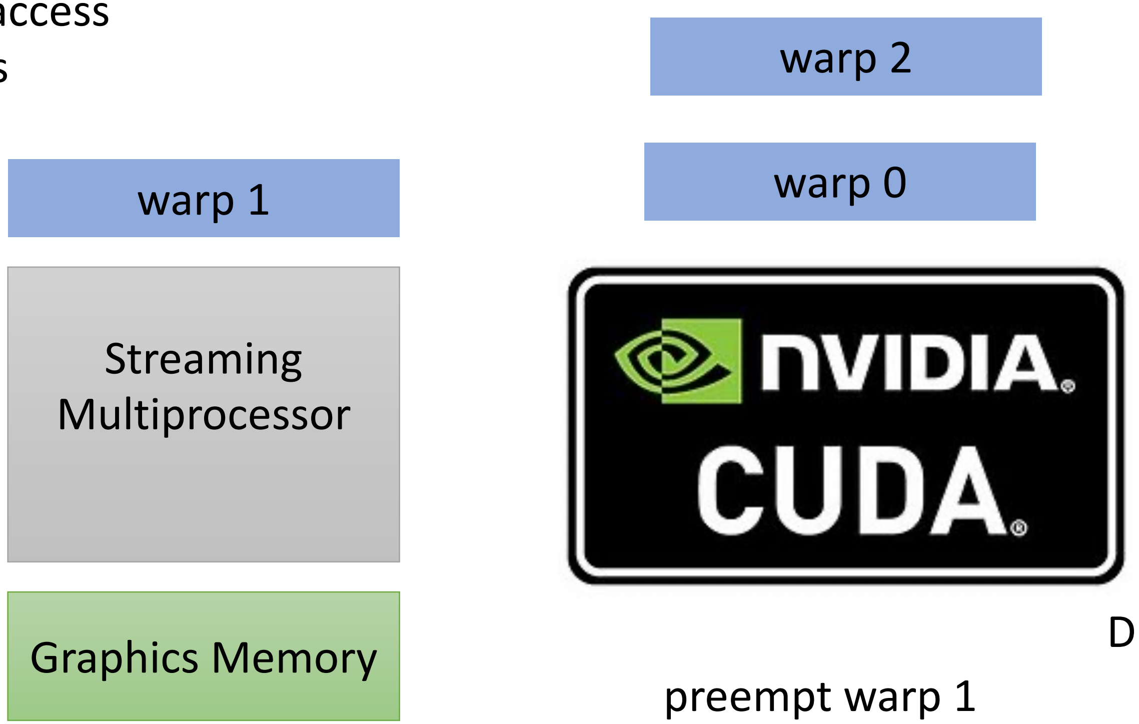
Graphics Memory

Driver

We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

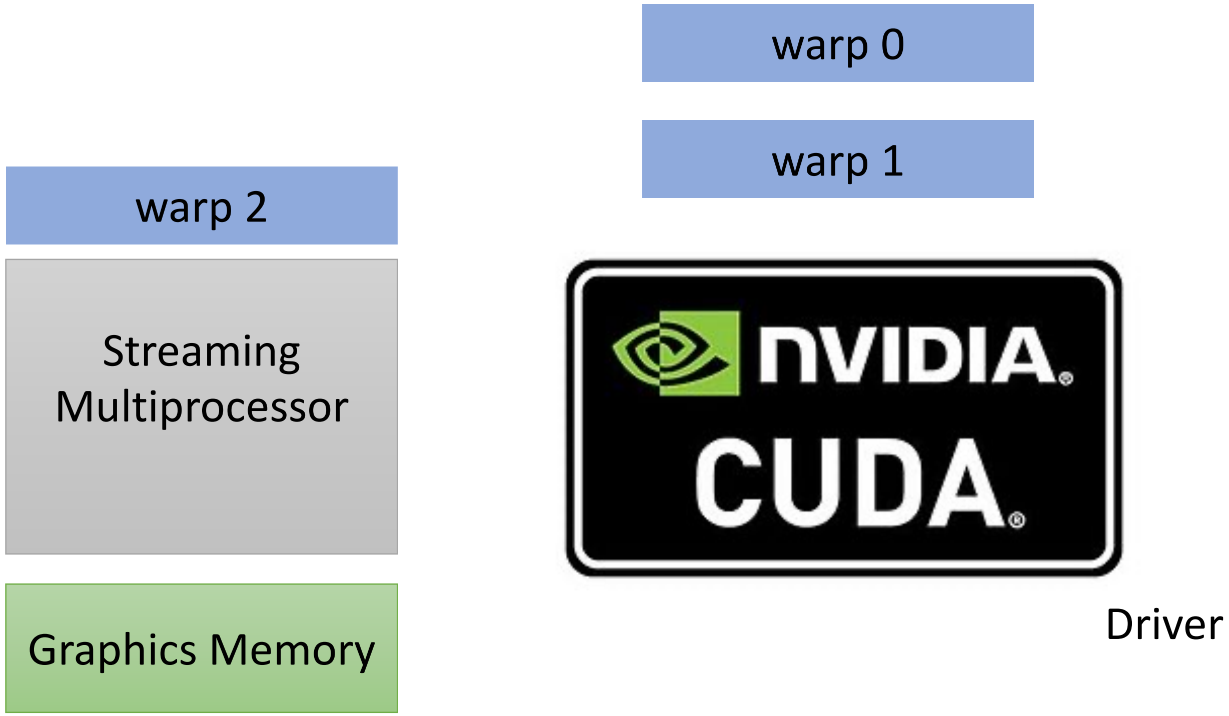


Driver

preempt warp 1
and put warp 2 on

We can hide latency through preemption and concurrency!

Preemption and concurrency?



warp 0

warp 1

warp 2

Streaming Multiprocessor

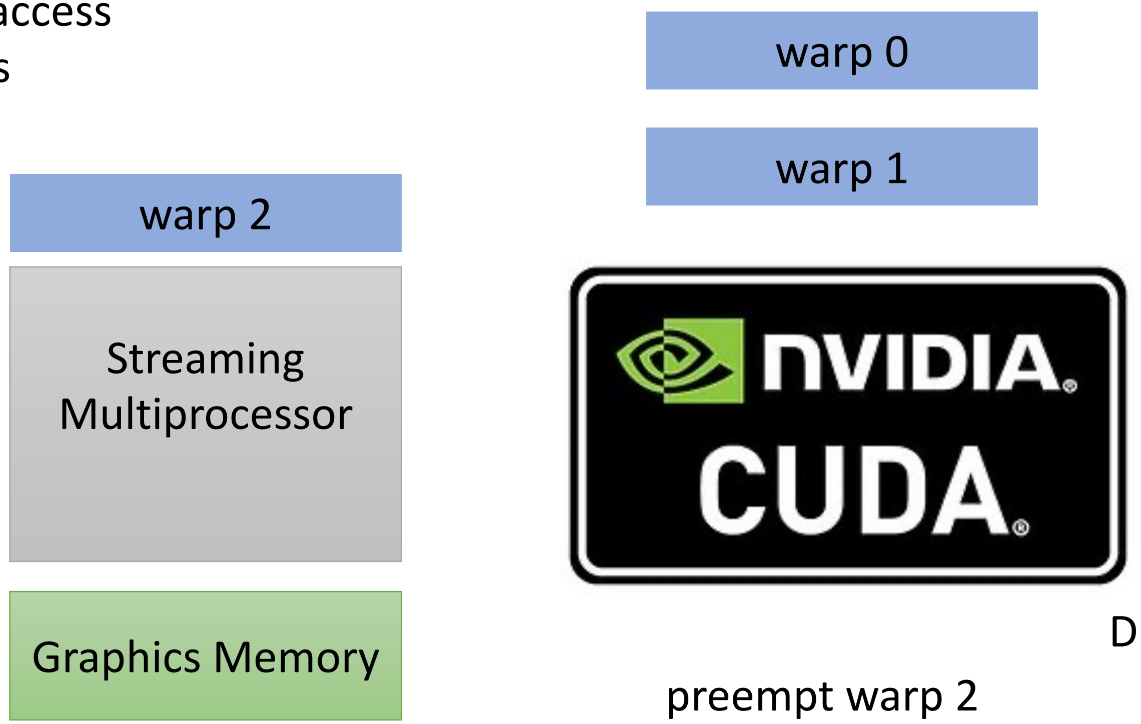
Graphics Memory

Driver

We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles



warp 0

warp 1

warp 2

Streaming Multiprocessor

Graphics Memory

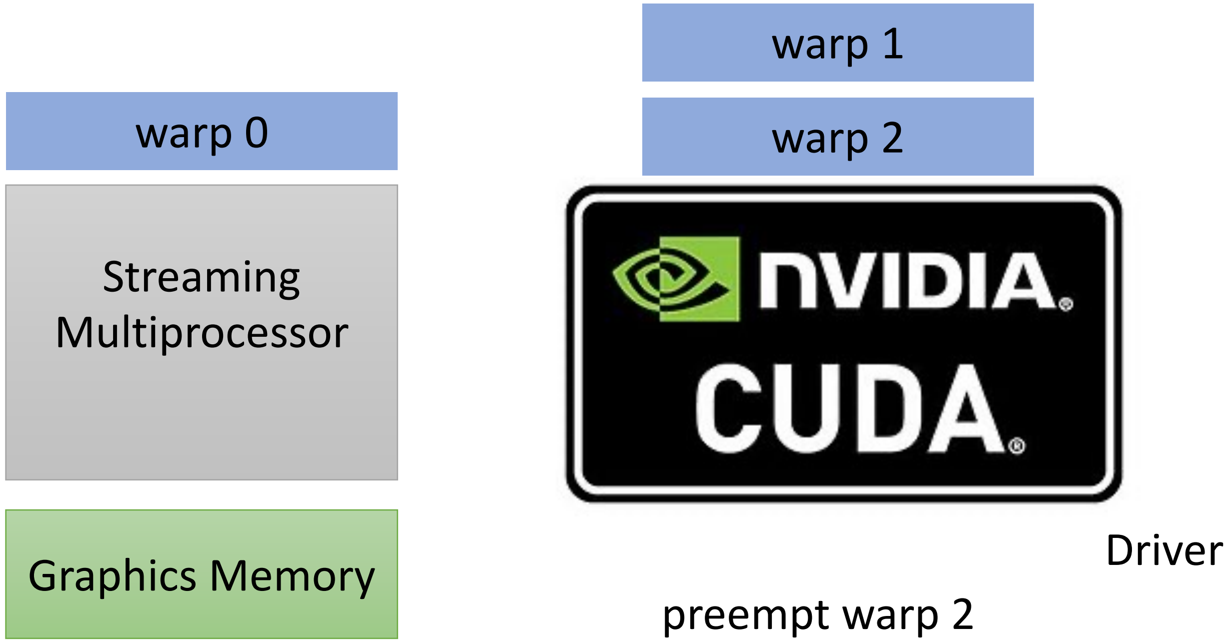
Driver

preempt warp 2
and put warp 0 on

We can hide latency through
preemption and concurrency!

Preemption and concurrency?

Hey, my memory has arrived!



warp 1

warp 2

warp 0

Streaming Multiprocessor

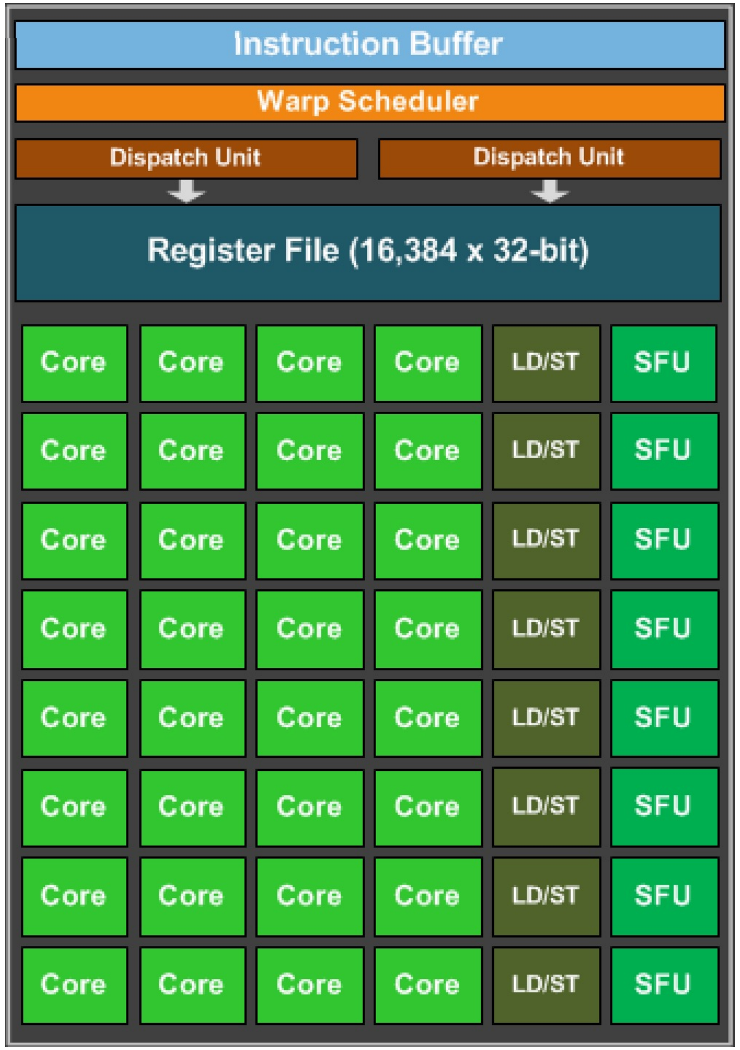
Graphics Memory

Driver

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?

bound on number of warps: 32

Lots of specialized HW to help out
(register files, scheduler, instruction buffer)

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];
   }
 }
```

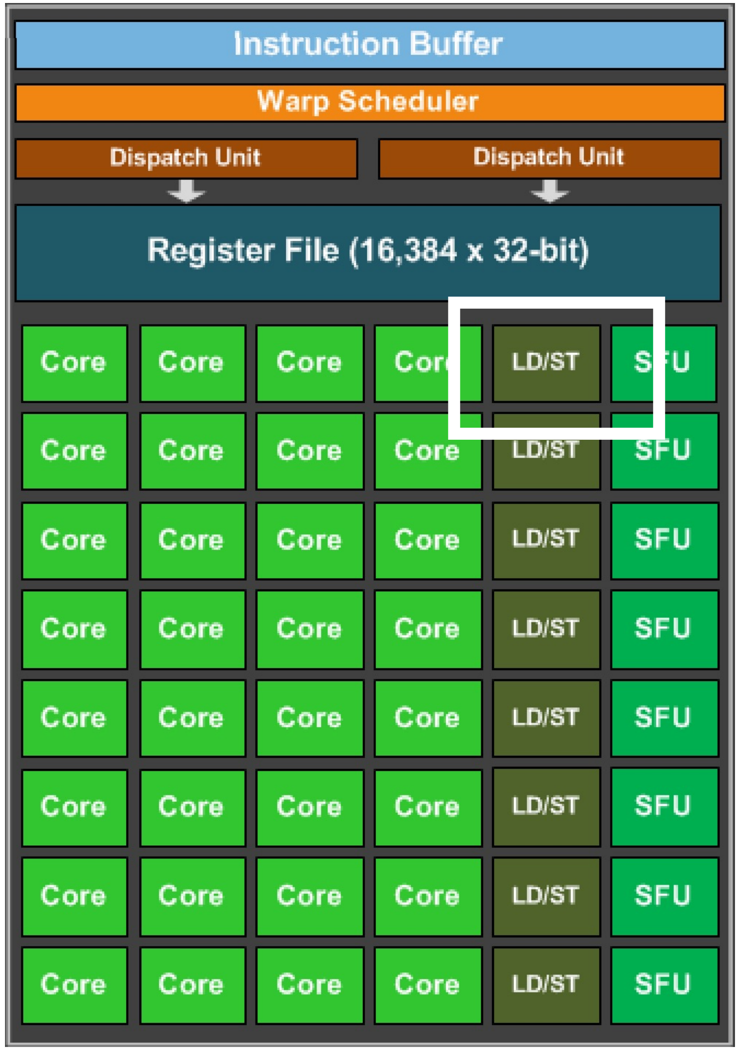
calling the function

Lets launch with 32 warps

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

Memory accesses

Optimizing memory accesses

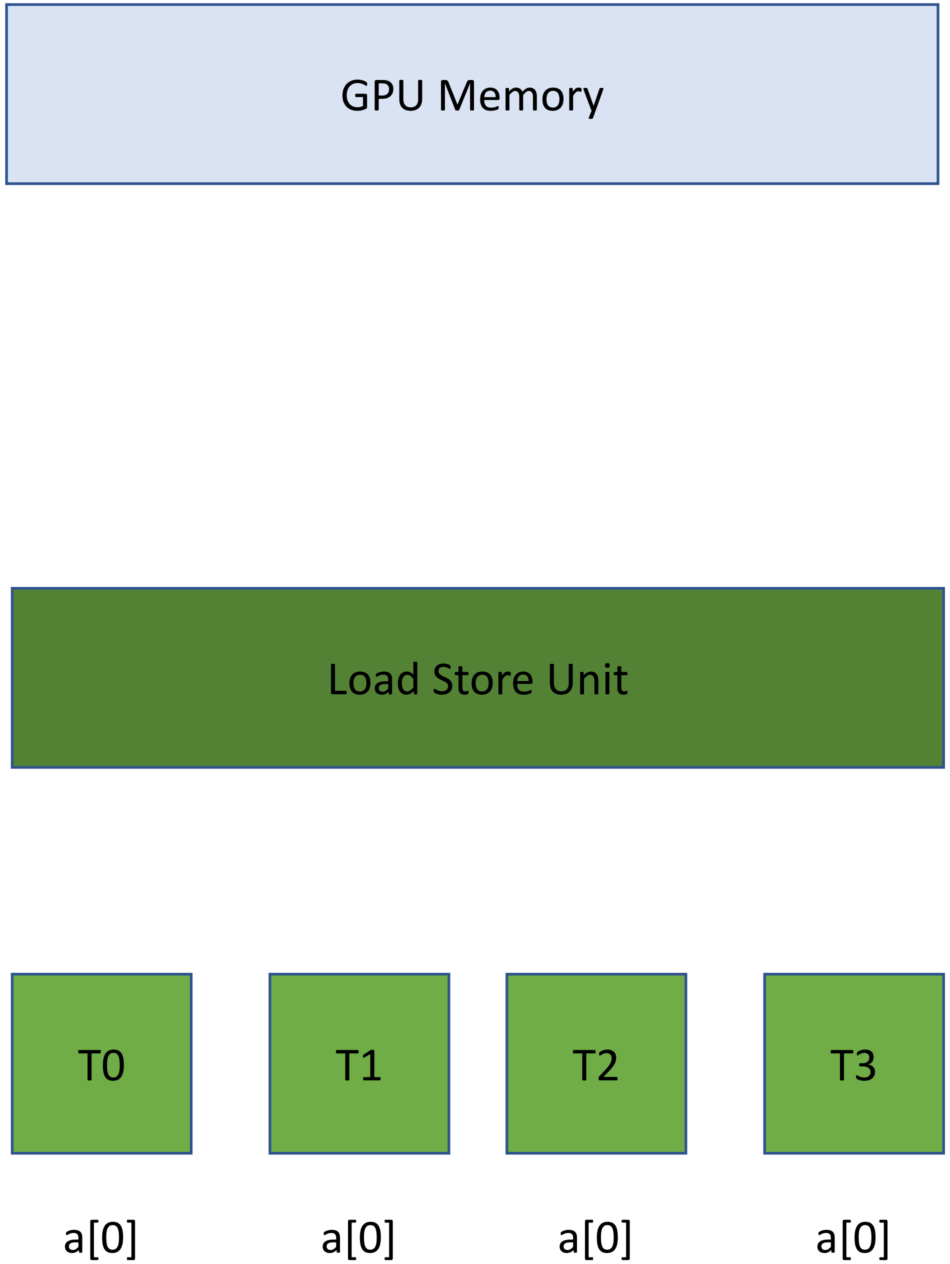


this is the load/store unit. The hardware component responsible for issuing loads and stores.

Why doesn’t every core have one?

4 cores are accessing memory. What can happen

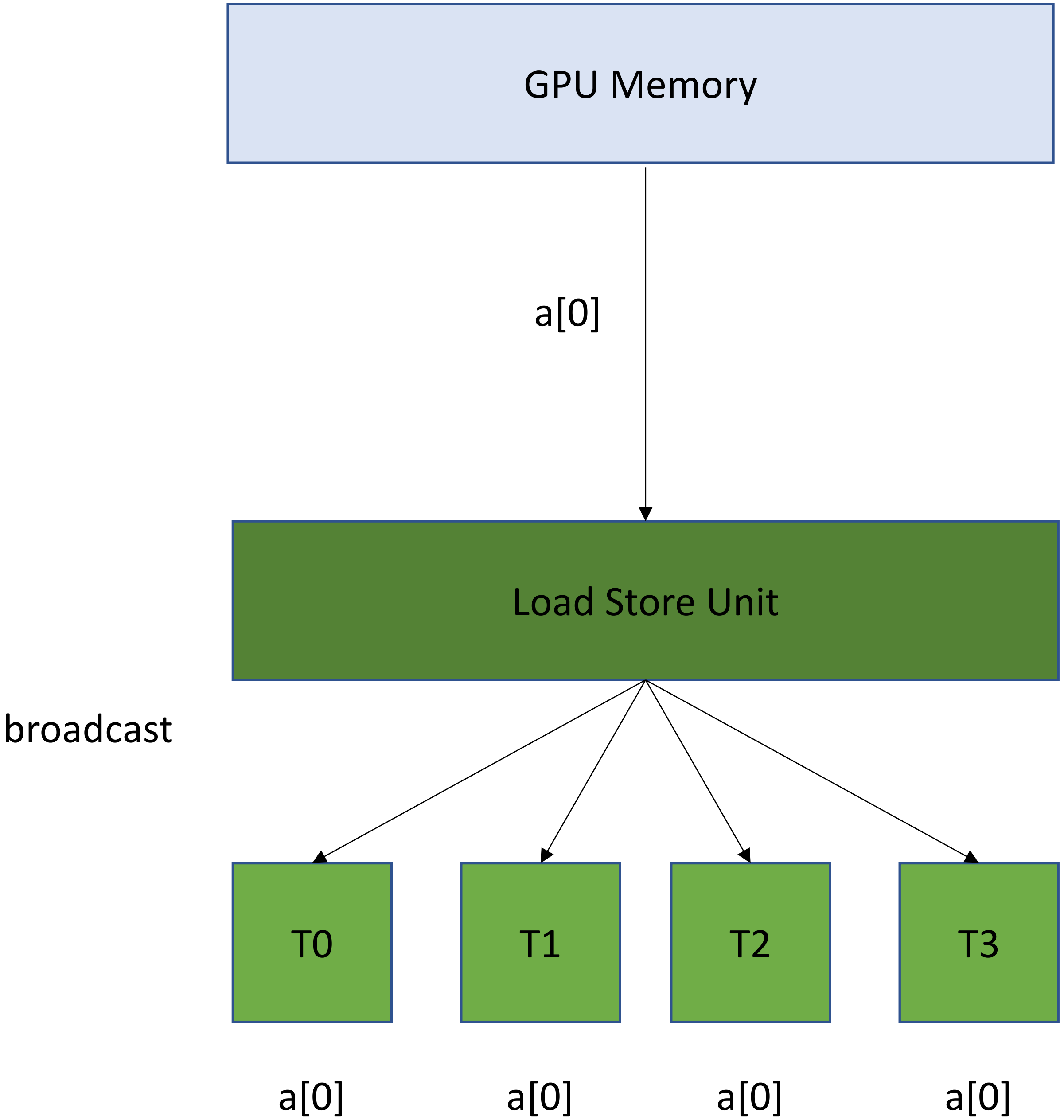
All read the same value



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.

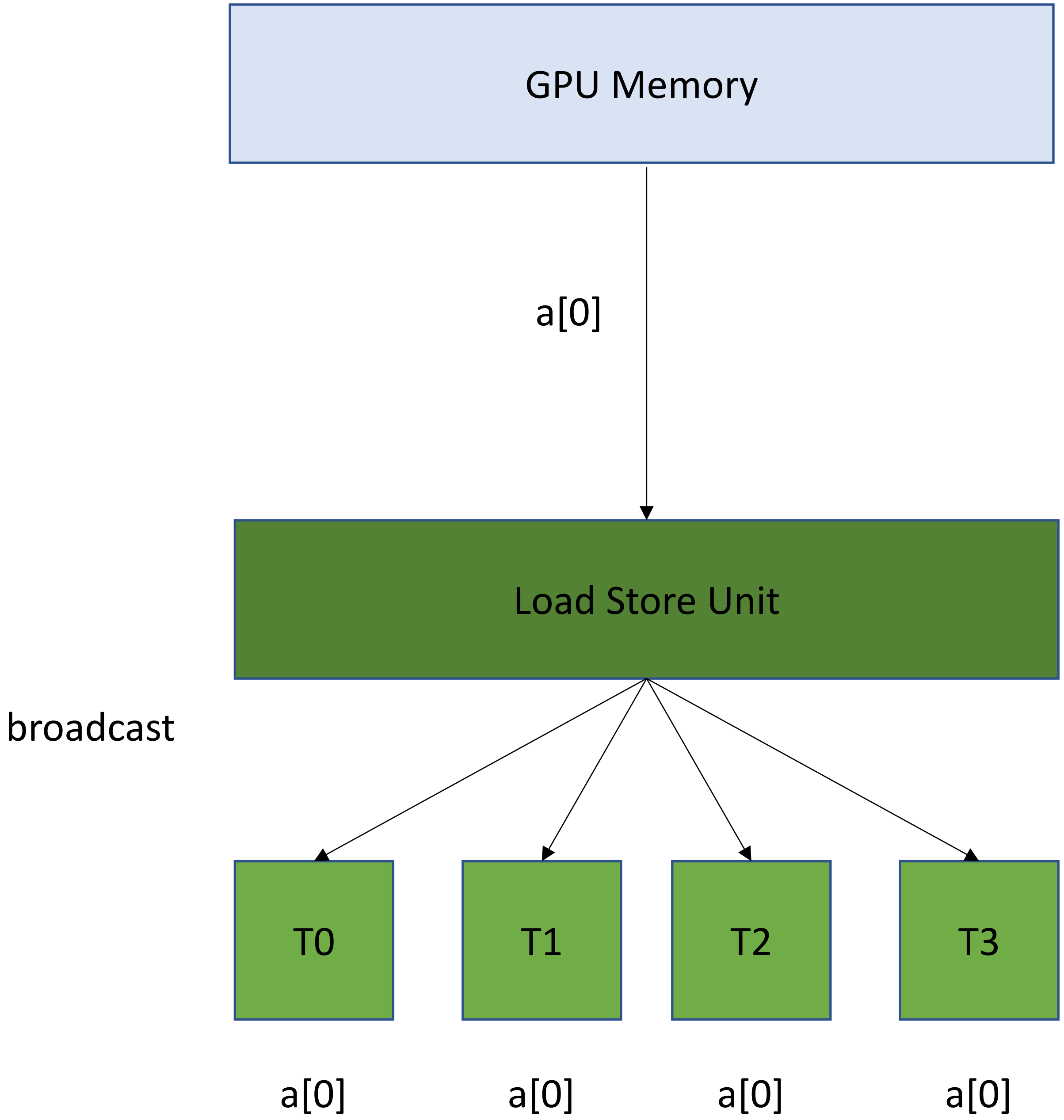


4 cores are accessing memory. What can happen

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

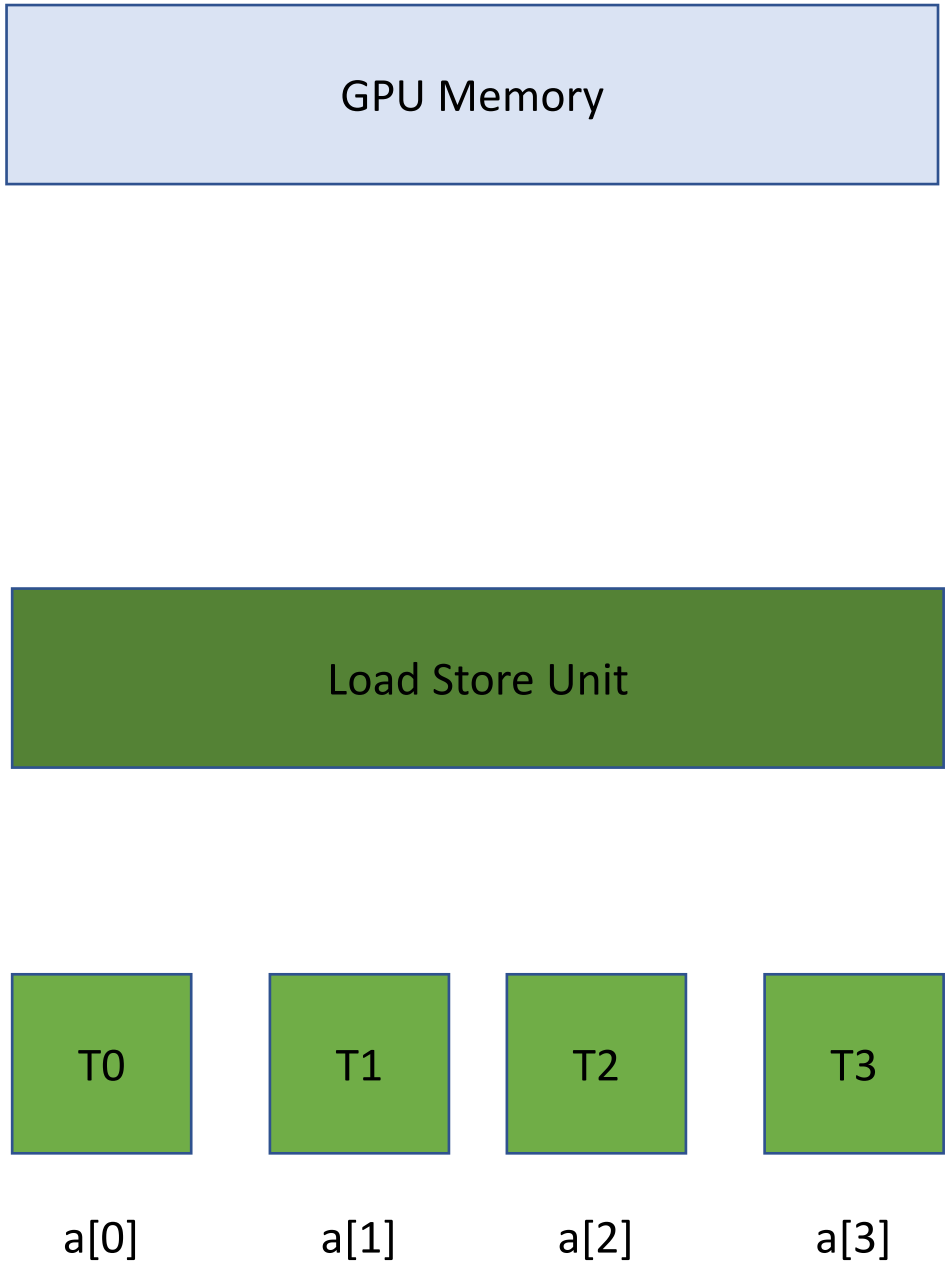
1 request to GPU memory

Efficient, but probably not too common.



4 cores are accessing memory. What can happen

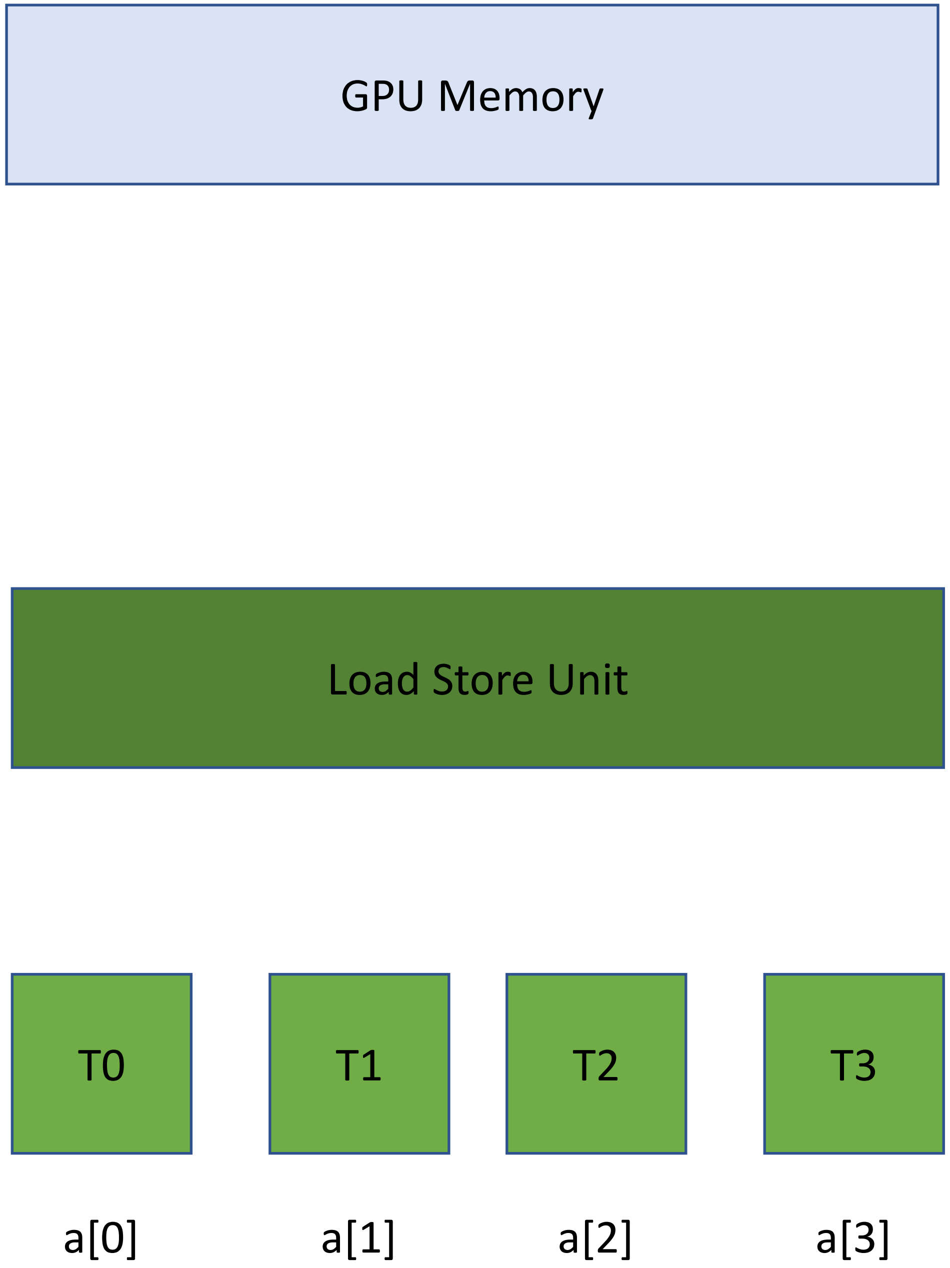
Read contiguous values



4 cores are accessing memory. What can happen

Read contiguous values

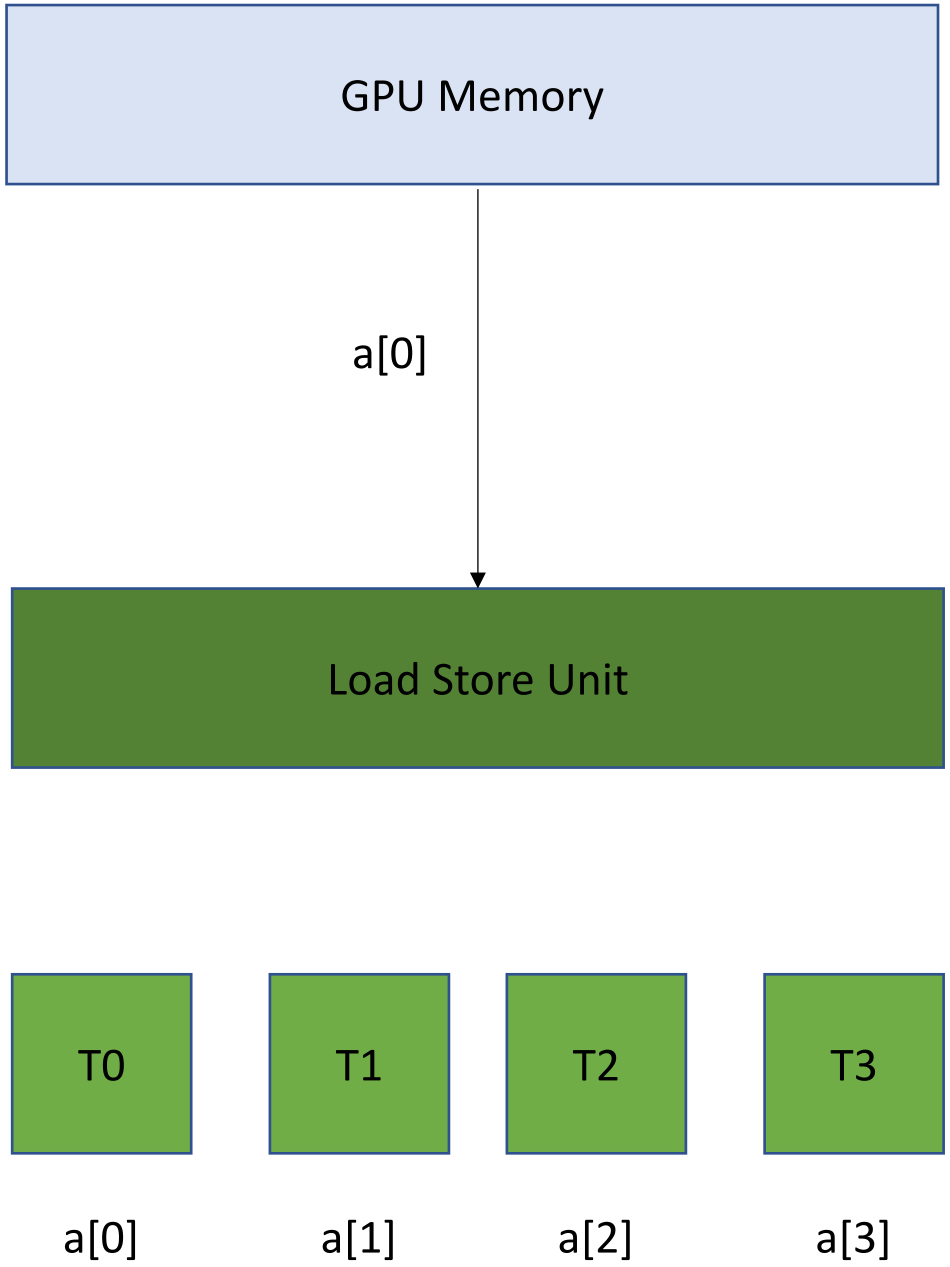
Like the CPU cache, the Load/Store Unit
reads in memory in chunks. 16 bytes



4 cores are accessing memory. What can happen

Read contiguous values

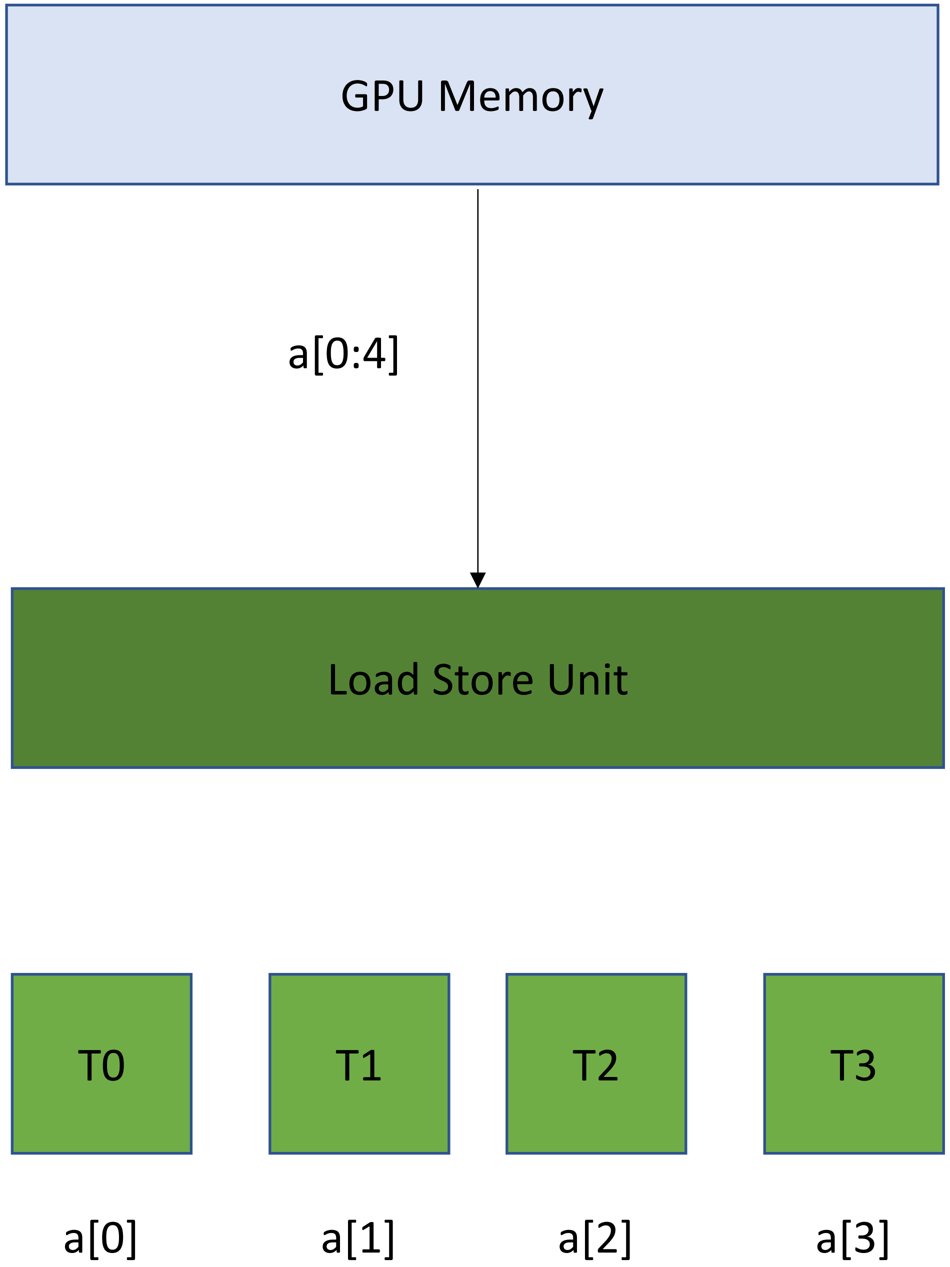
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes



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

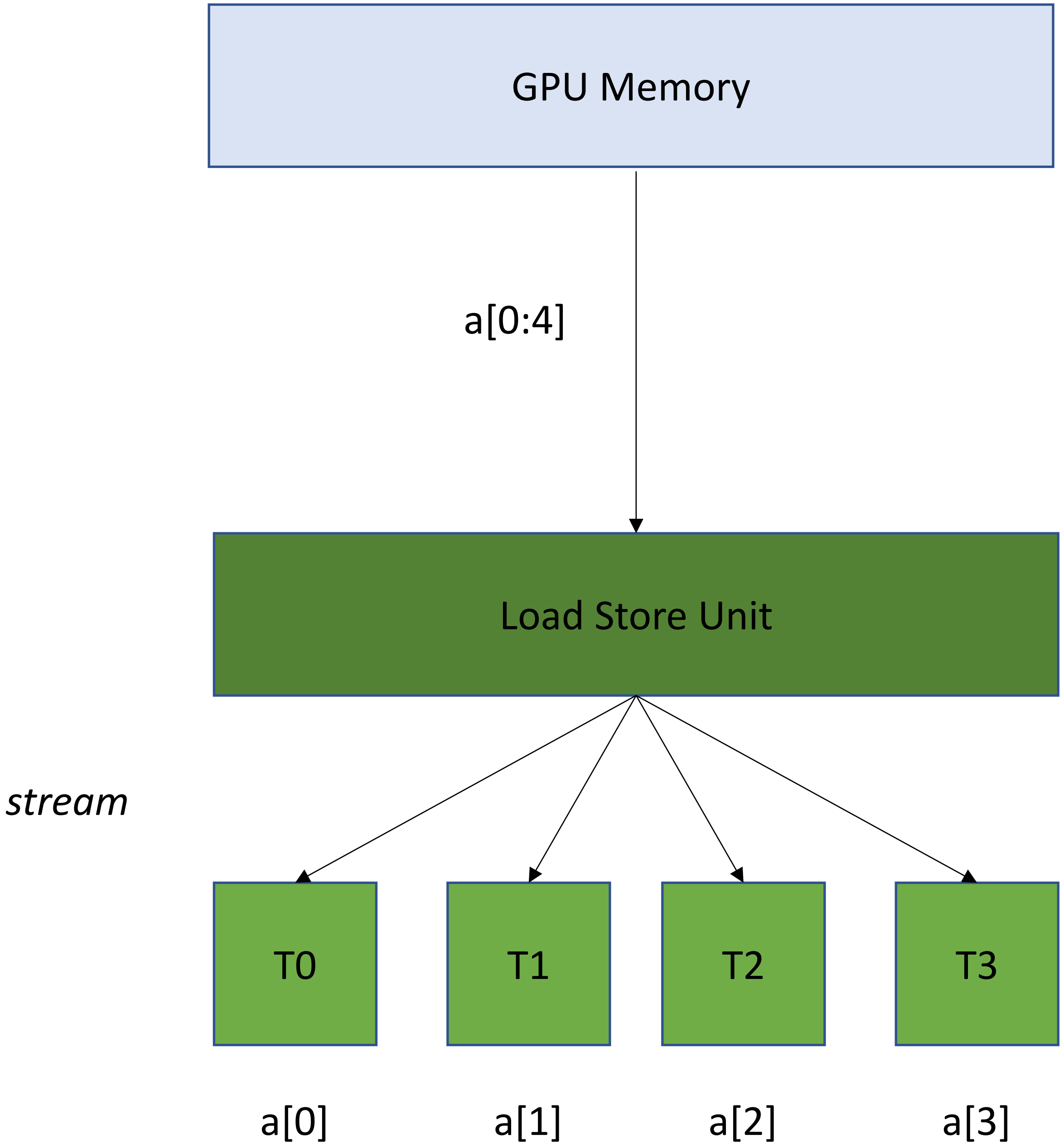


4 cores are accessing memory. What can happen

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

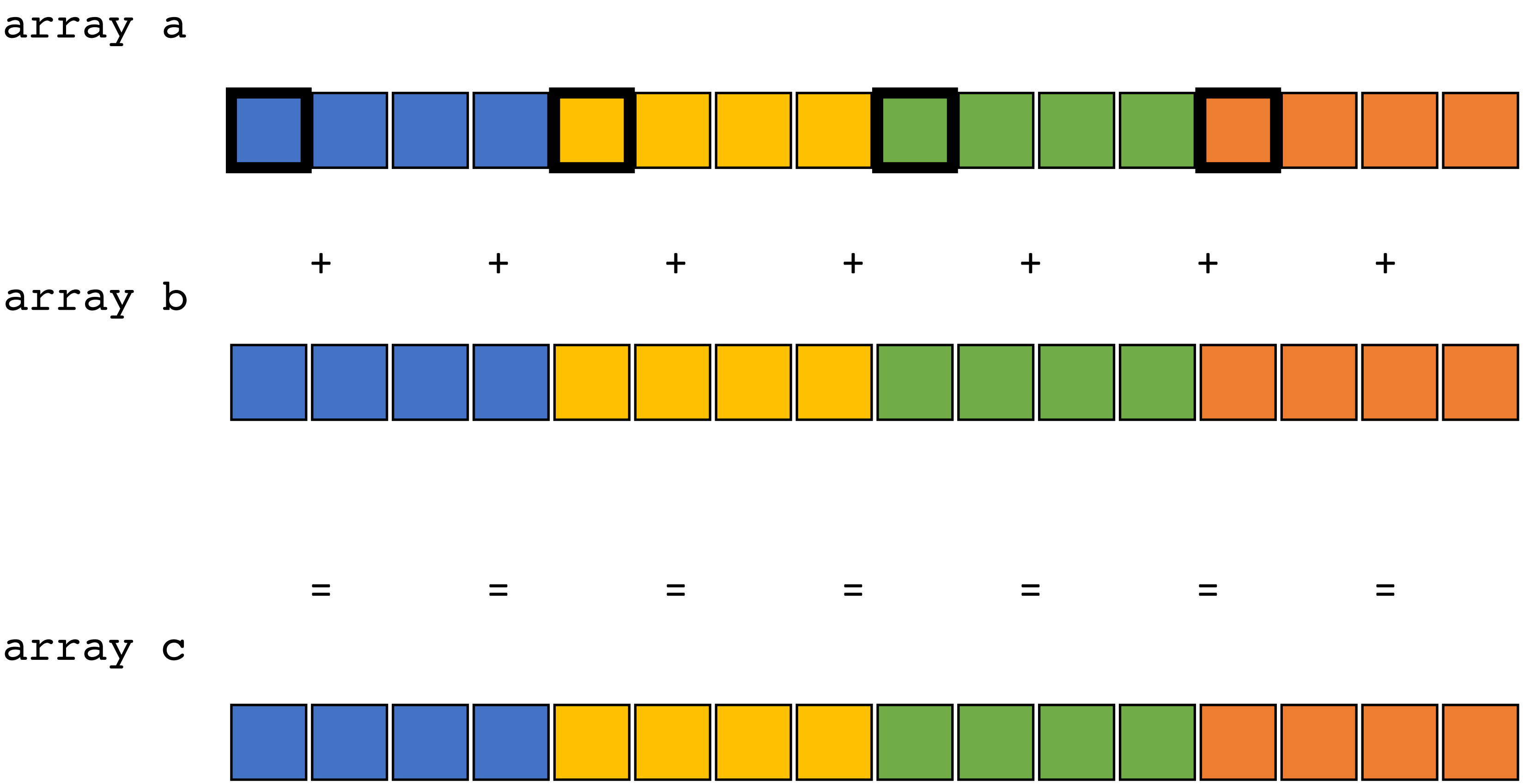


Chunked Pattern

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

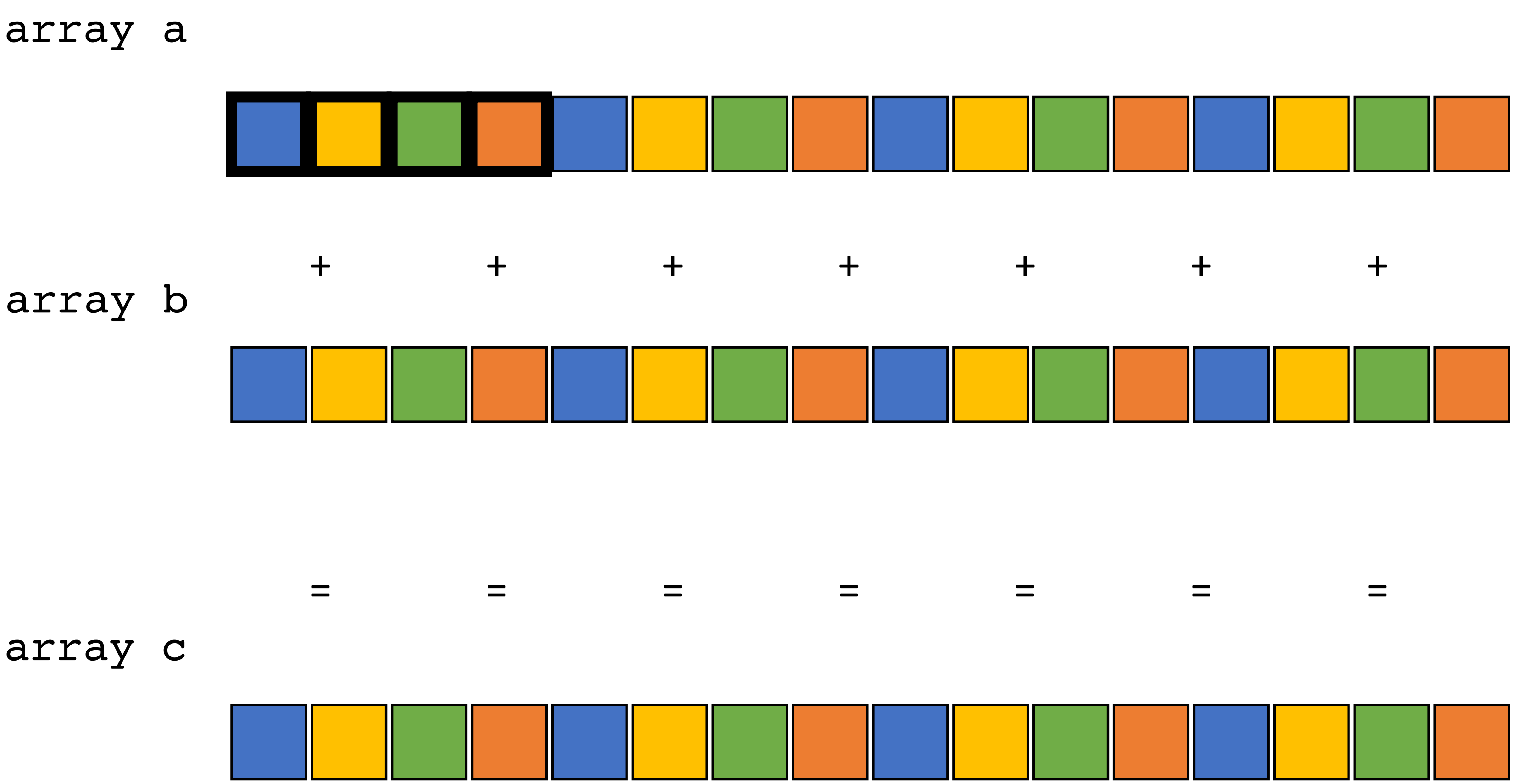


How can we fix this

Stride Pattern

Computation can easily be divided into threads

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



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];
  }
}
```

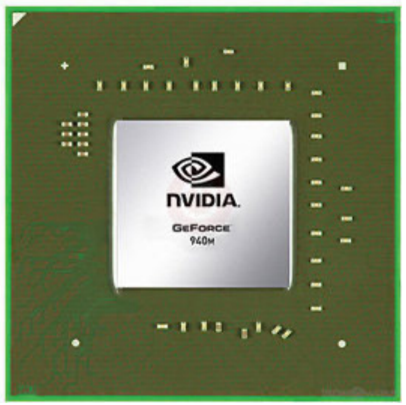
calling the function

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

How far did that get us?

Programming a GPU

The GPU in
my PhD laptop



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

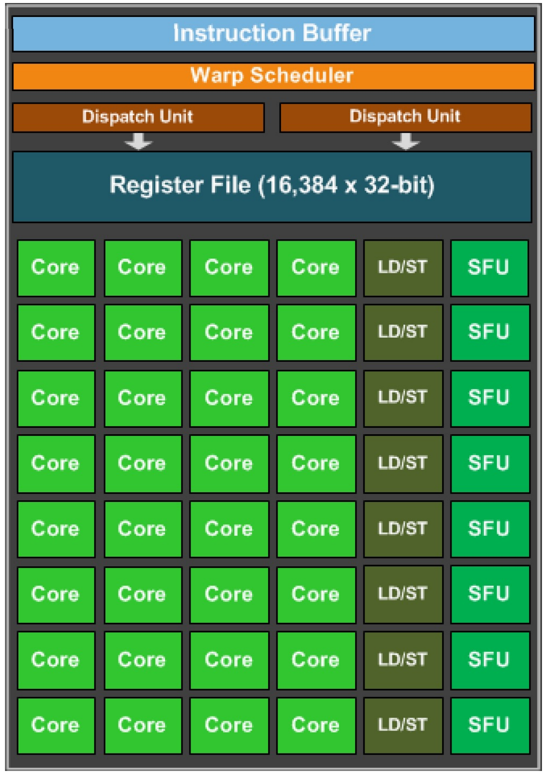
The CPU in
my professor
workstation

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

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

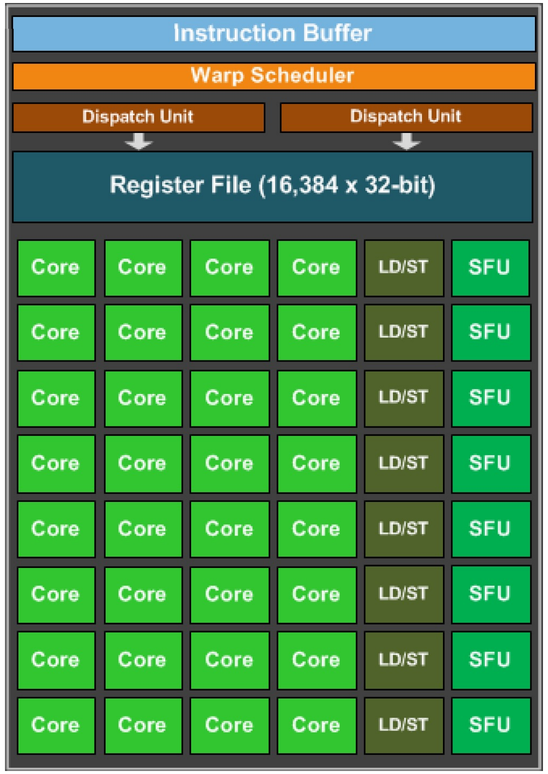
Multiple streaming multiprocessors

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



Multiple streaming multiprocessors

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



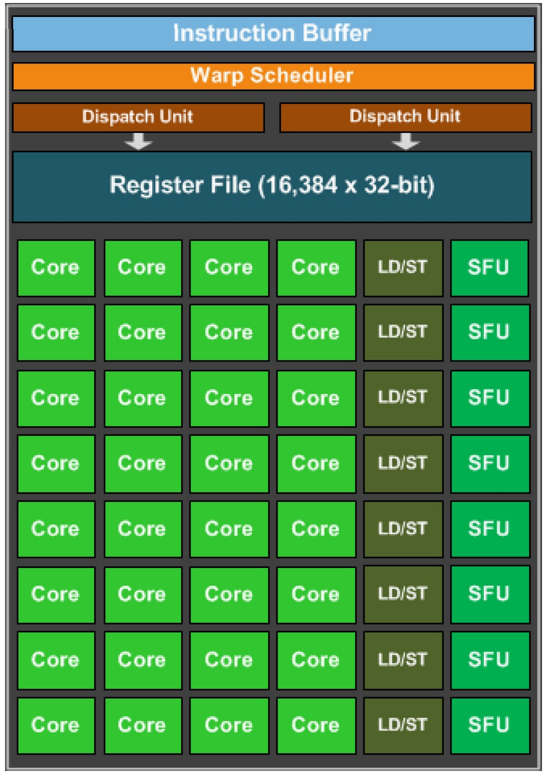
Instruction Buffer

Warp Scheduler

Dispatch Unit | Dispatch Unit

Register File (16,384 x 32-bit)

Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU



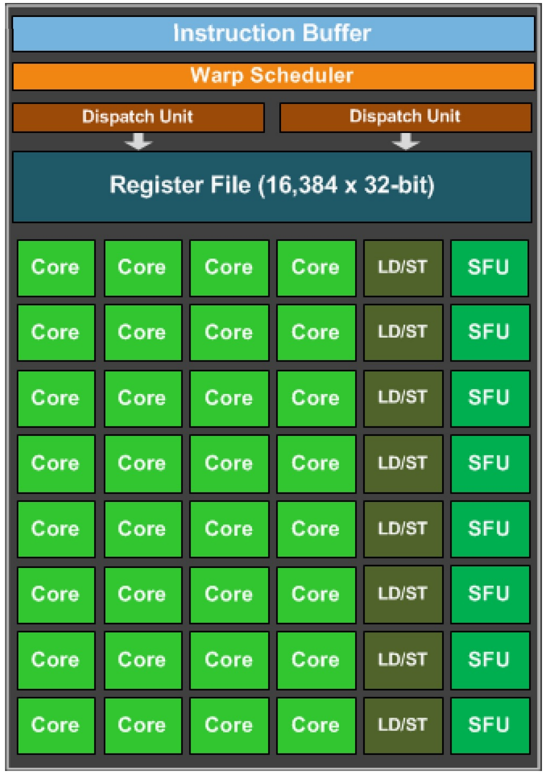
Instruction Buffer

Warp Scheduler

Dispatch Unit | Dispatch Unit

Register File (16,384 x 32-bit)

Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU



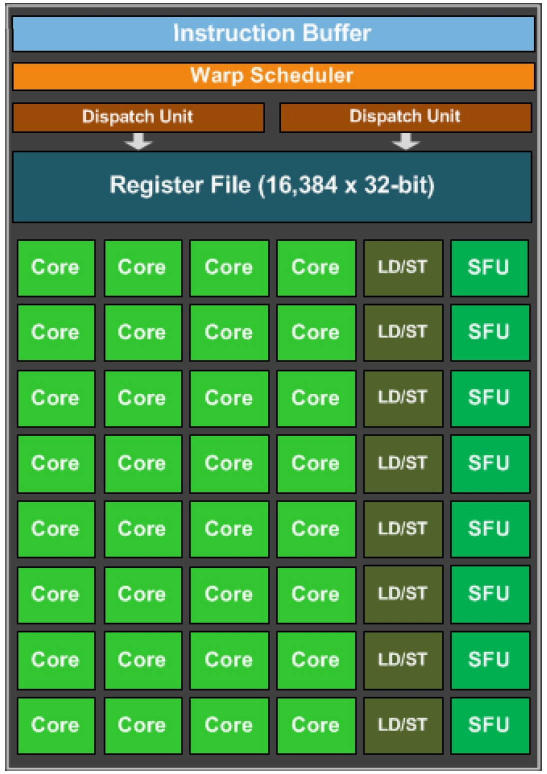
Instruction Buffer

Warp Scheduler

Dispatch Unit | Dispatch Unit

Register File (16,384 x 32-bit)

Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU



Instruction Buffer

Warp Scheduler

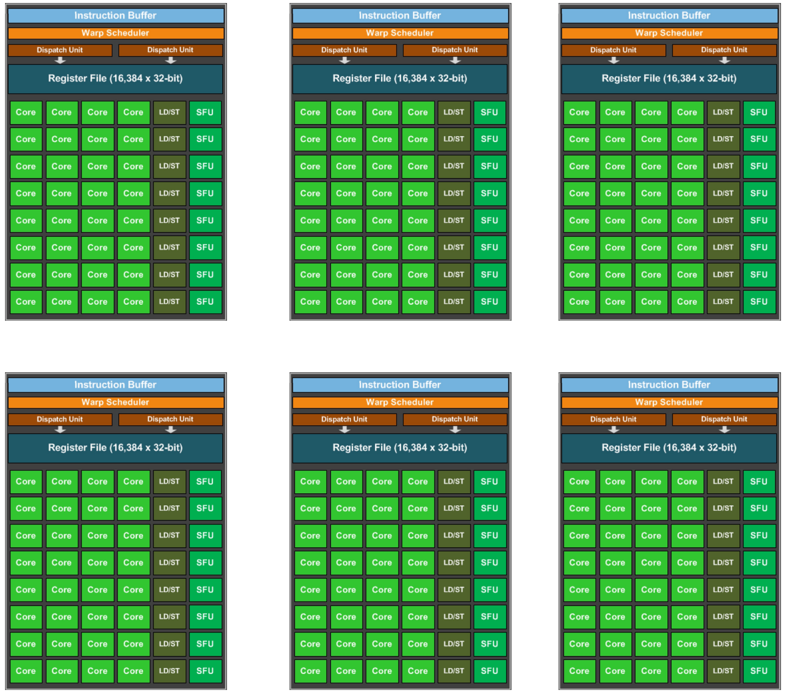
Dispatch Unit | Dispatch Unit

Register File (16,384 x 32-bit)

Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU
Core	Core	Core	Core	LD/ST	SFU

Multiple streaming multiprocessors

CUDA provides virtual streaming multiprocessors called **blocks**
Very efficient at launching and joining **blocks.**
No limit on blocks: launch as many as you need to map 1 thread to 1 data element



Go back to our program

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

calling the function

Launch with many thread blocks

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

Go back to our program

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

Need to recalculate some thread ids.

calling the function

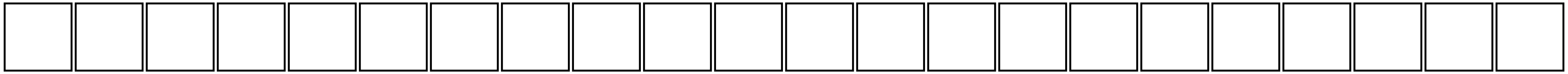
Launch with many thread blocks

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

Now we have 1 thread for each element

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

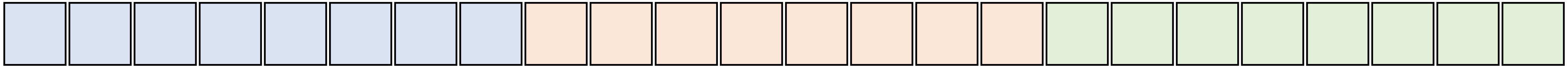
How does this work



Consider thread ids as a flattened array (which is often how they are used to index memory)

How does this work

block 0: 🟦
block 1: 🟧
block 2: 🟩



Consider thread ids as a flattened array (which is often how they are used to index memory)

Say we specify 8 threads per block (this can be up to 1024)

How does this work

block 0: 🟦
block 1: 🟧
block 2: 🟩

global thread ids	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
local thread ids	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7

Consider thread ids as a flattened array (which is often how they are used to index memory)

Say we specify 8 threads per block (this can be up to 1024)

Thread ids are local to a block

Compute global id? `blockIdx.x * blockDim.x + threadIdx.x`

Go back to our program

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

Need to recalculate some thread ids.

calling the function

Launch with many thread blocks

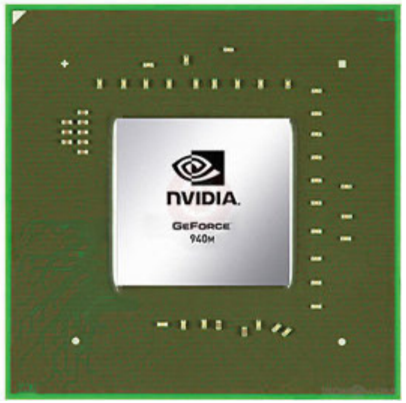
```
vector_add<<<1024,1024>>>(d_a, d_b, d_c, size);
```

Now we have 1 thread for each element

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

Final Round

The GPU in
my PhD laptop



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

The CPU in
my professor
workstation

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

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

WebGPU

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

- Due to canvas scaling: you will need to scale distance values by 1000
  - Step size should be .001
  - cluster distance should be .003

WebGPU

- wgsl is NOT javascript

- Javascript is interpreted: not possible on GPUs

- wgsl is compiled
  - into Vulkan on Linux
  - into Metal on Apple
  - into HLSL on Windows

- No printing (can be difficult to debug)

WebGPU

- variables (optional types):

var <name> = <value>;

```
var cluster_dist = 0.003;
```

```
var <name> : <type> = <value>;
var cluster_dist : f32 = 0.003;
```

WebGPU

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

- types:
  - i32
  - u32
  - f32
  - vec2<f32>
  - array<*type*>

```
struct Particles {
    particles : array<Particle>;
};
```

```
var index_pos : vec2<f32> = particlesA.particles[index].pos;
```

- structures

```
var index : u32 = GlobalInvocationID.x;
```

- Built-ins (global id)

you have one thread for each particle!

Web GPU

- Built in functions:
  - arrayLength
  - sqrt
  - pow
  - distance

Web GPU

For loops:

```
for (var i : u32 = 0u; i < arrayLength(&particlesA.particles); i = i + 1u)
```

Web GPU

- Types can be frustrating

- But compiler errors will help you, and you can do casts.

Wrapping up

Thank you!

- You are now all now experts on parallel programming!
- You’re all going to do great on the final! March 17
  - Available all day
  - Our scheduled time is 4 - 7 PM if you want to schedule time
- Thank you for being such great students during such a hard time. I’m proud of all of you!
- See you around!