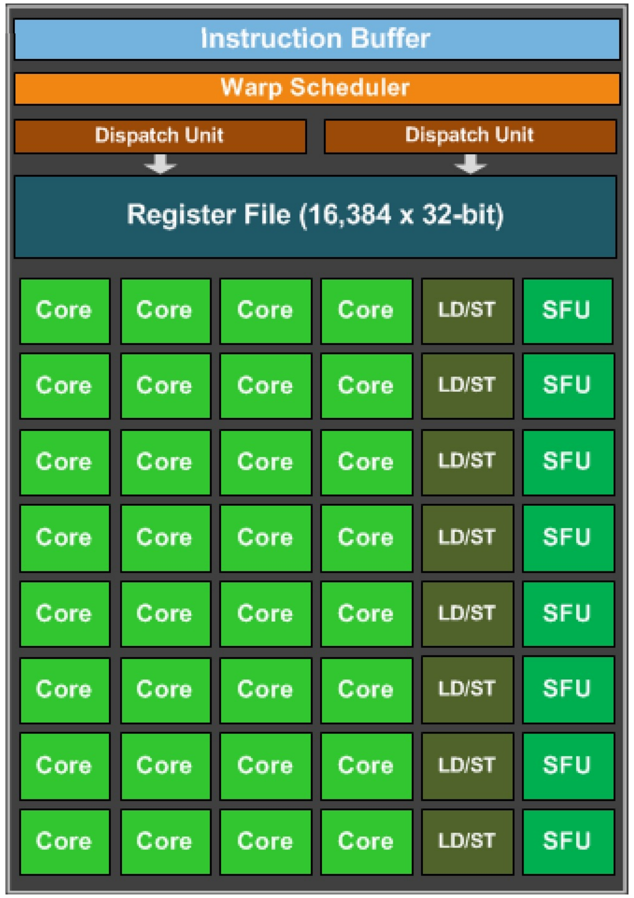
CSE113: Parallel Programming

June 1, 2021

- **Topic**: GPUs 2
  - Review last week optimizations
  - Continue optimizing and have the final round of CPU vs. GPU!
  - Overview of advanced GPU topics



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

Announcements

- HW2 grades posted
  - Talk to us in 1 week if you have questions/issues
- We plan to have HW3 done in ~1 week.
- HW4 is out
  - Please try not to be late on this one!
  - Due on Monday, June 7
  - There is no guarantee that we will check Piazza on the weekend
  - Joint office hours on Wednesday

Few hints on HW4

- If you are having trouble observing relaxed behaviors:
  - Try closing ALL applications
  - Try running natively (e.g. not in Docker)
  - Try running on the unix timeshare
    - Compilation on the time share works
    - I (and other students) have been able to get relaxed behaviors observations
    - run “top” and “who” to make sure the machine is not being heavily used

Announcements

- SETs are out
  - Please do them!

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

Announcements

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

Quiz

Quiz

- Go over answers

Schedule

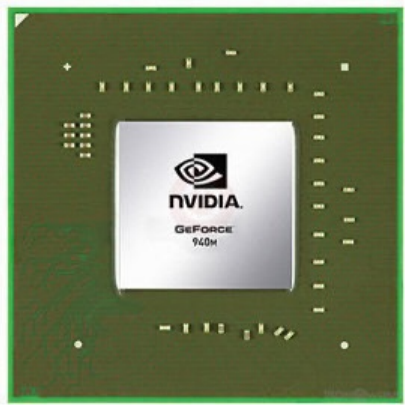
- Review previous optimizations
- New optimizations
- Advanced GPU topics

Schedule

- **Review previous optimizations**
- New optimizations
- Advanced GPU topics

Round 2

The GPU in
my PhD laptop



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

Fight!

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

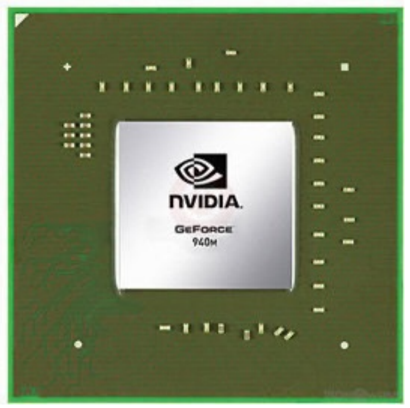
The CPU in
my professor
workstation

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

Where were we?

Round 2

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/

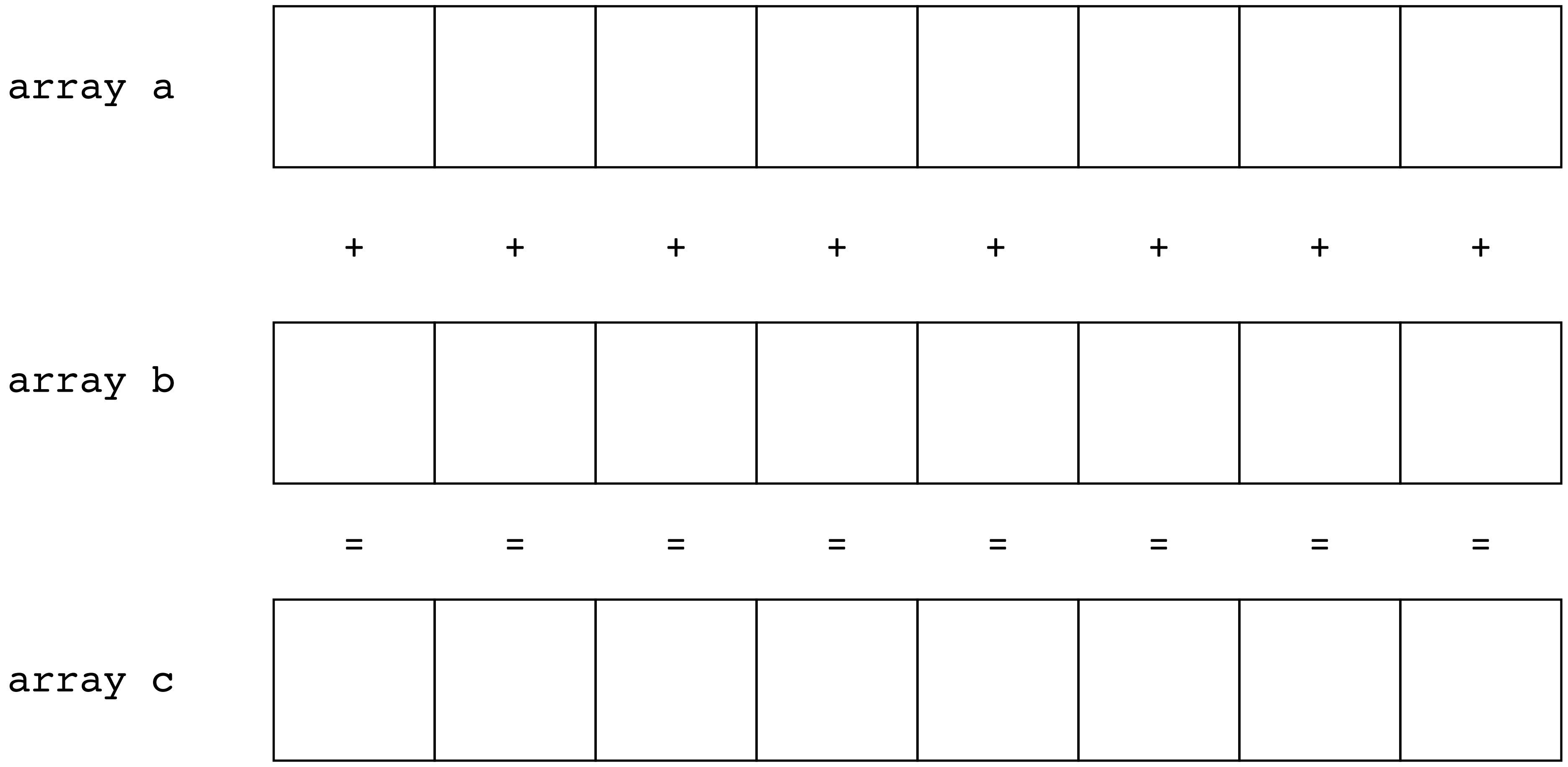
Programming a GPU

- 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

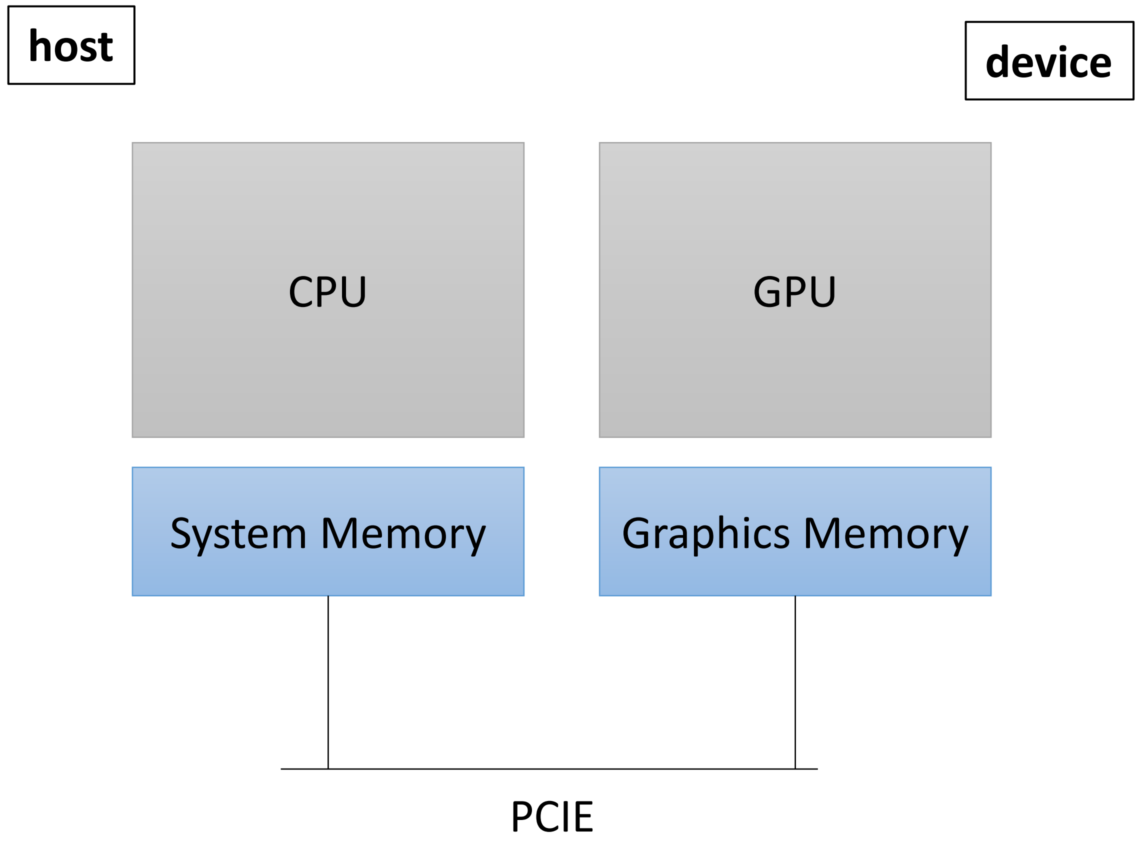
- Why do we access memory like this?

GPU code

- Review:

GPU set up

- Our heterogeneous, parallel, programming model



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

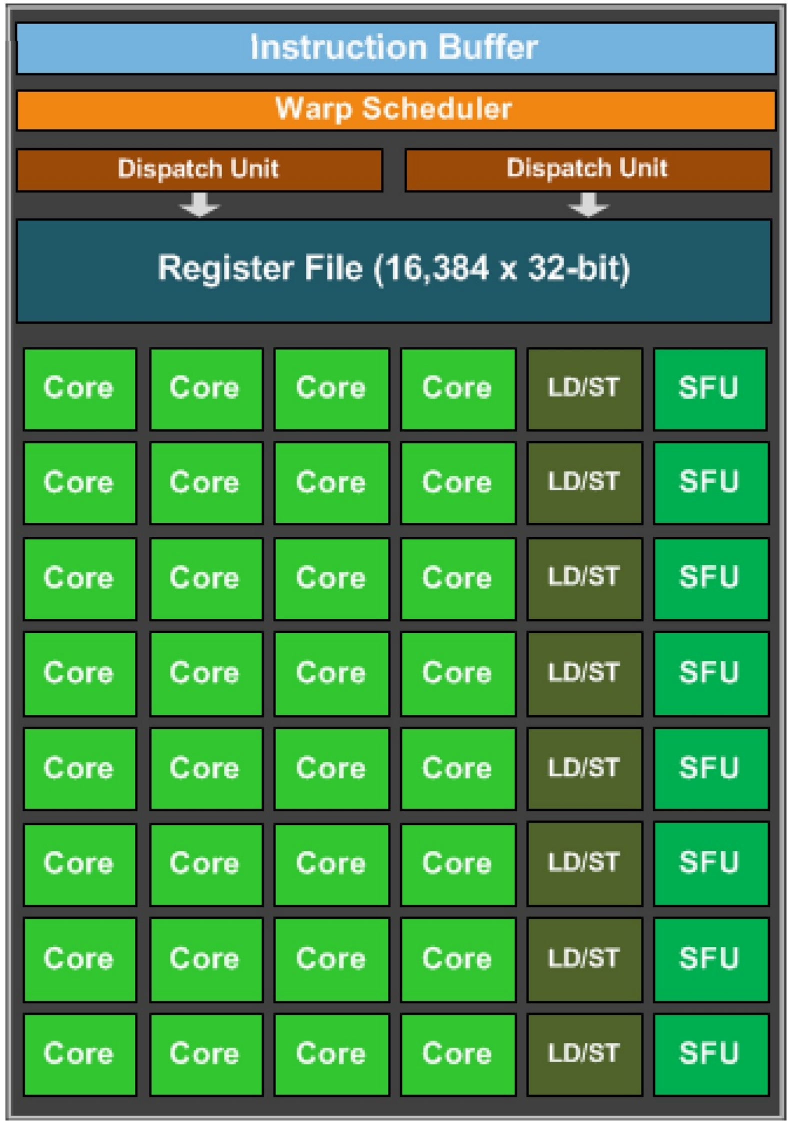
The GPU Program

```
__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];
  }
}
```

calling the function

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

The GPU hardware



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];
   }
}
```

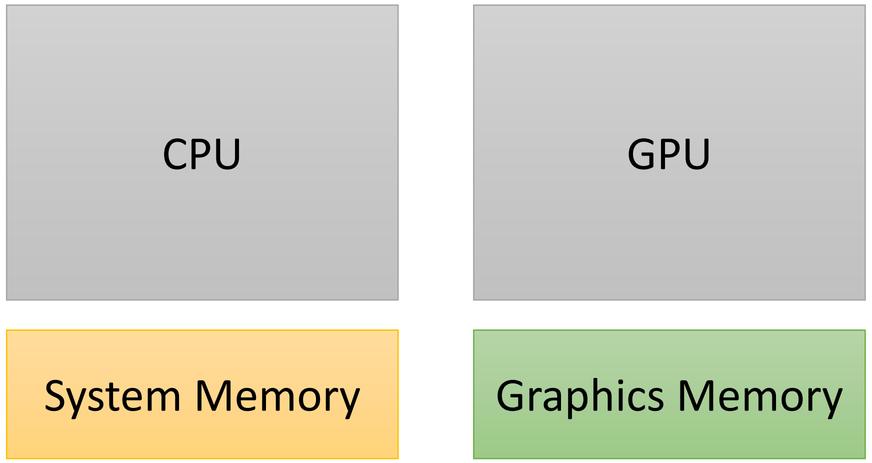
calling the function

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

number of threads
thread id

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

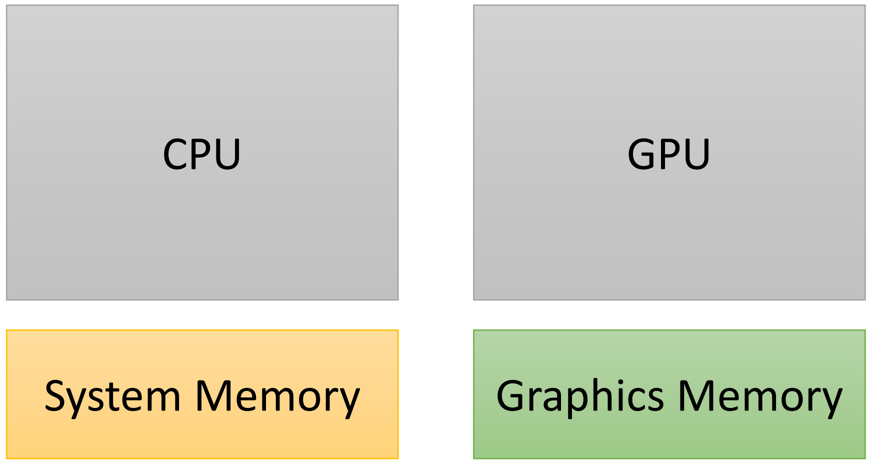
16-lane highway on a windy road driven by semi trucks

GPU Memory

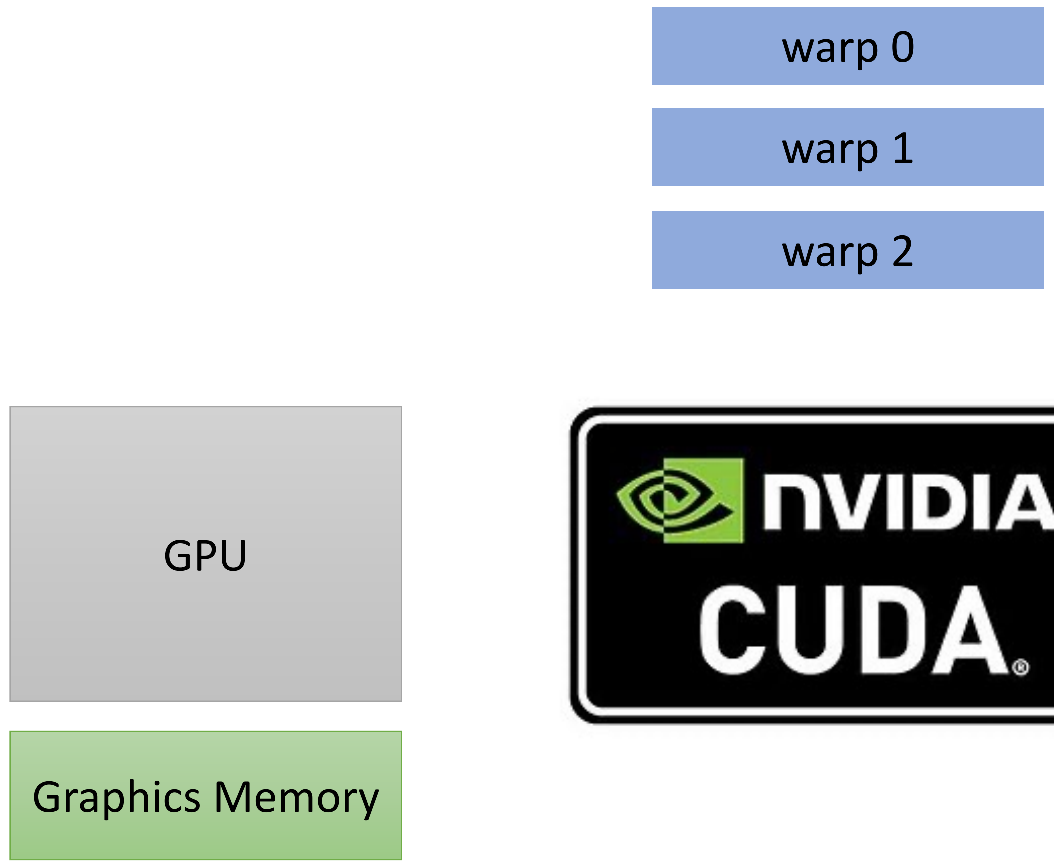
bandwidth:
~**700 GB/s** for GPU
~**50 GB/s** for CPUs

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

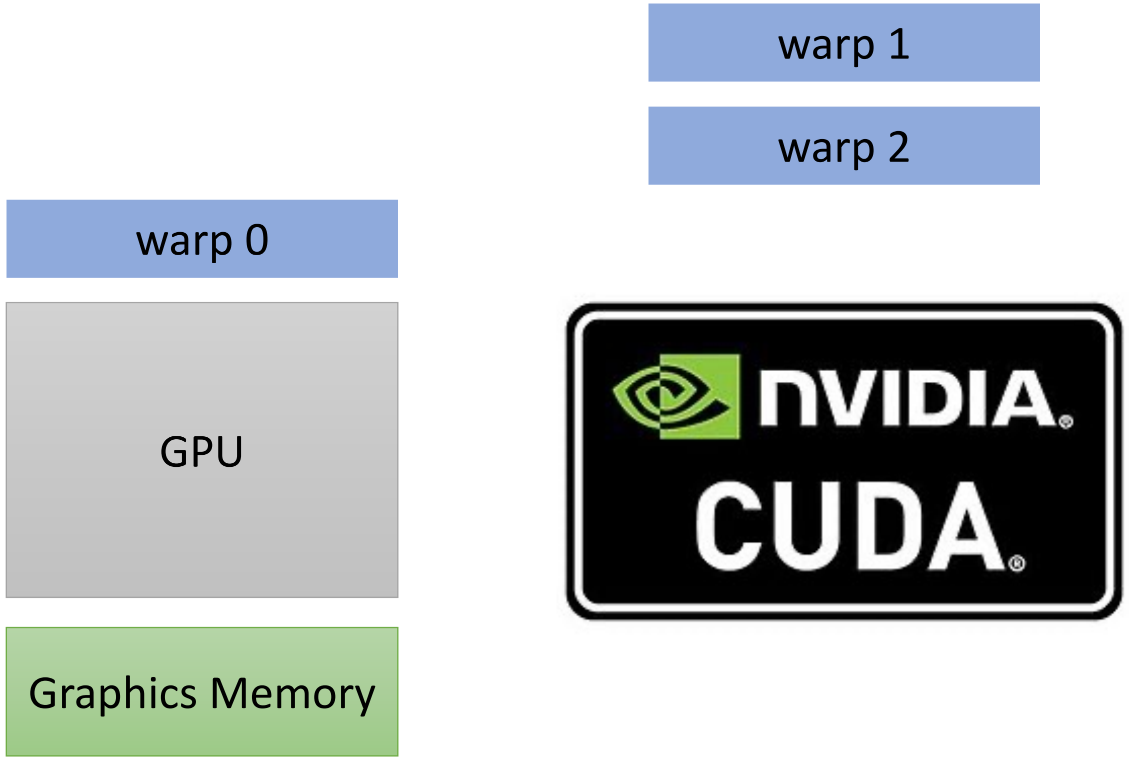


Preemption and concurrency?



We can hide latency through preemption and concurrency!

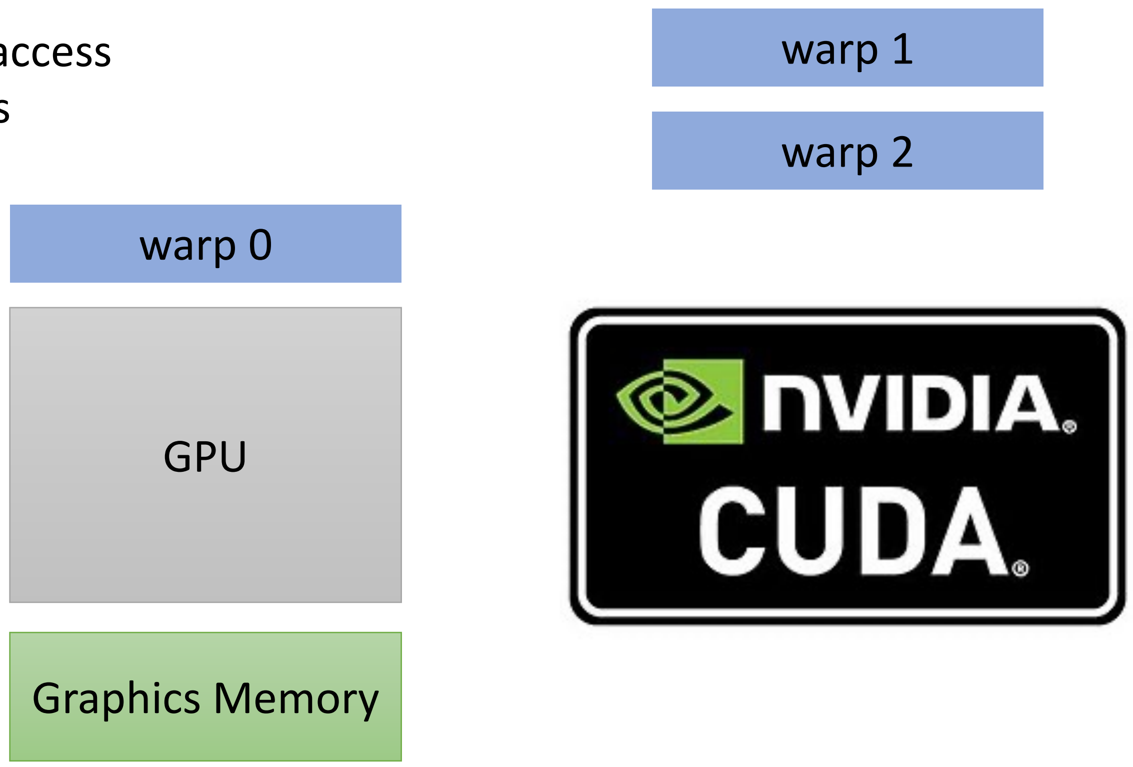
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

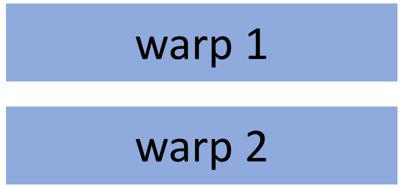
memory access
600 cycles

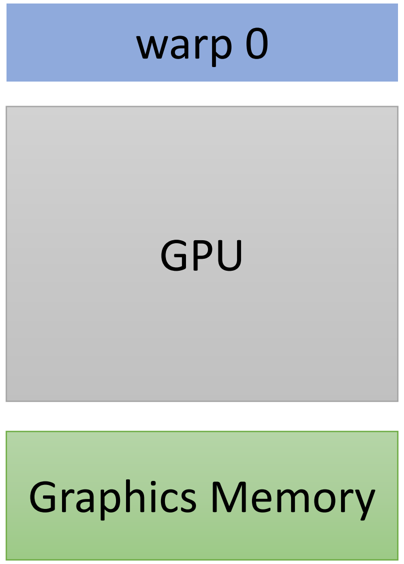


We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

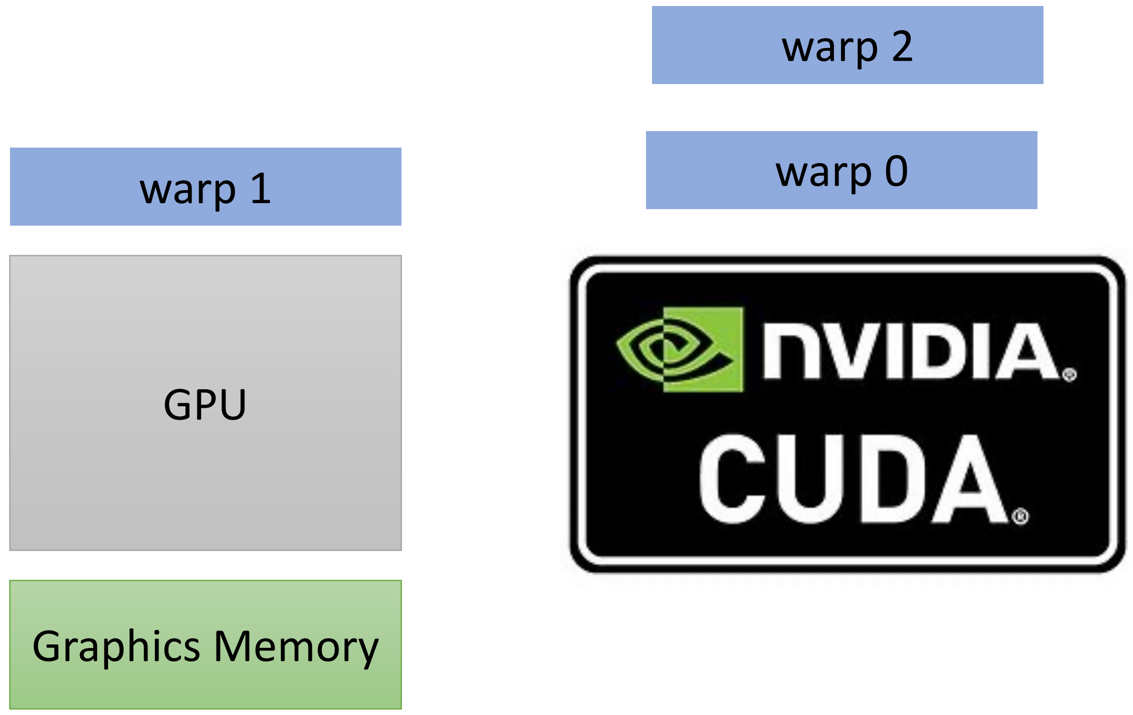




preempt warp 0
and put warp 1 on

We can hide latency through preemption and concurrency!

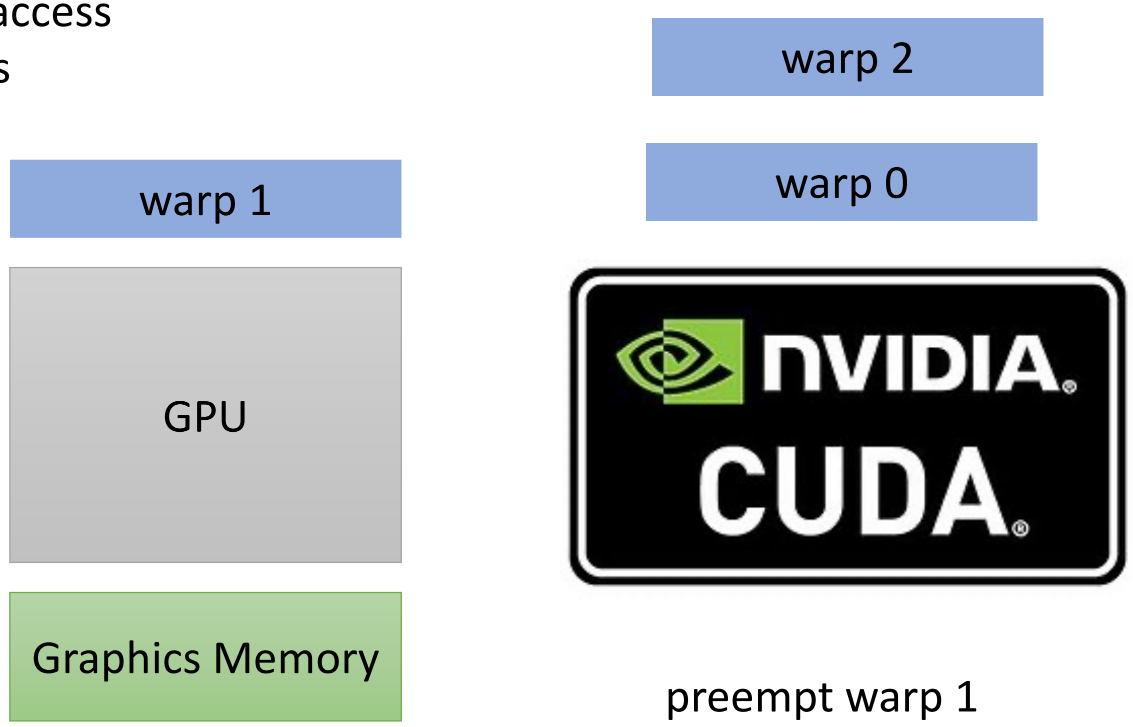
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

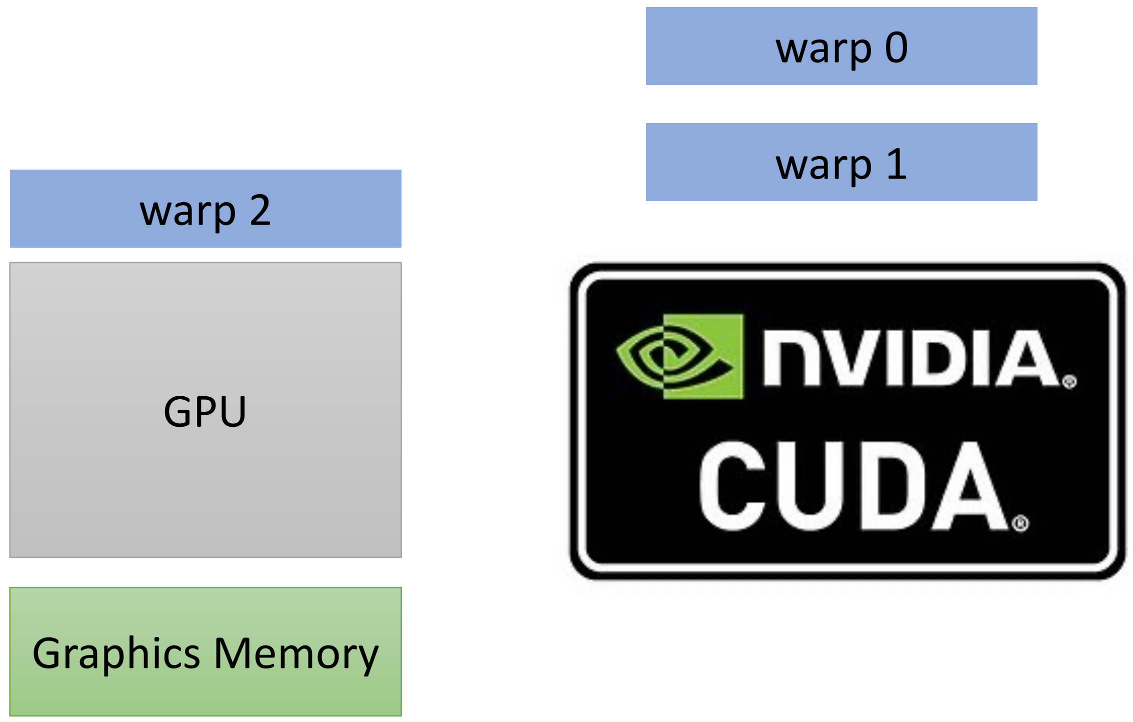
memory access
600 cycles



preempt warp 1
and put warp 2 on

We can hide latency through
preemption and concurrency!

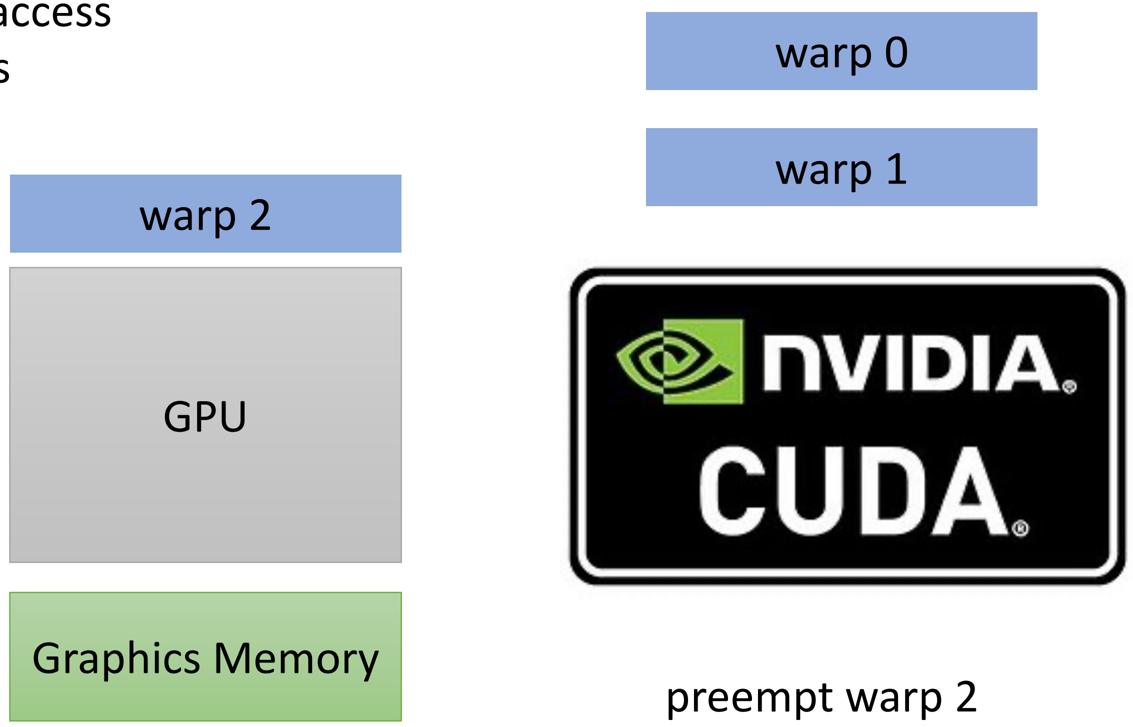
Preemption and concurrency?



We can hide latency through preemption and concurrency!

Preemption and concurrency?

memory access
600 cycles

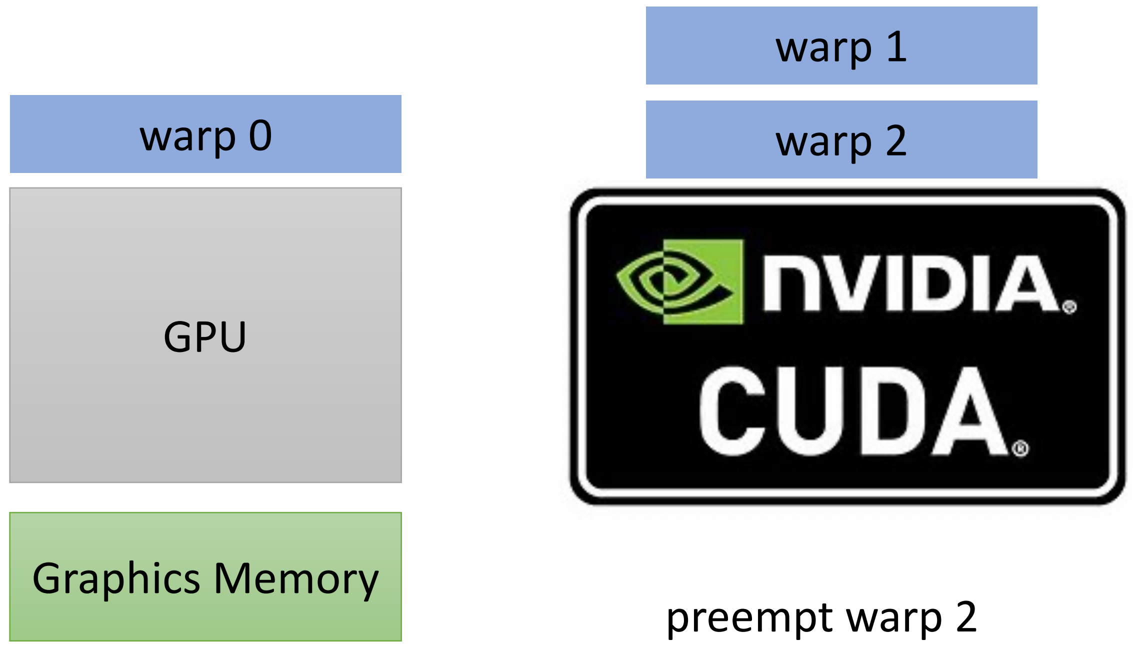


preempt warp 2
and put warp 0 on

We can hide latency through
preemption and concurrency!

Preemption and concurrency?

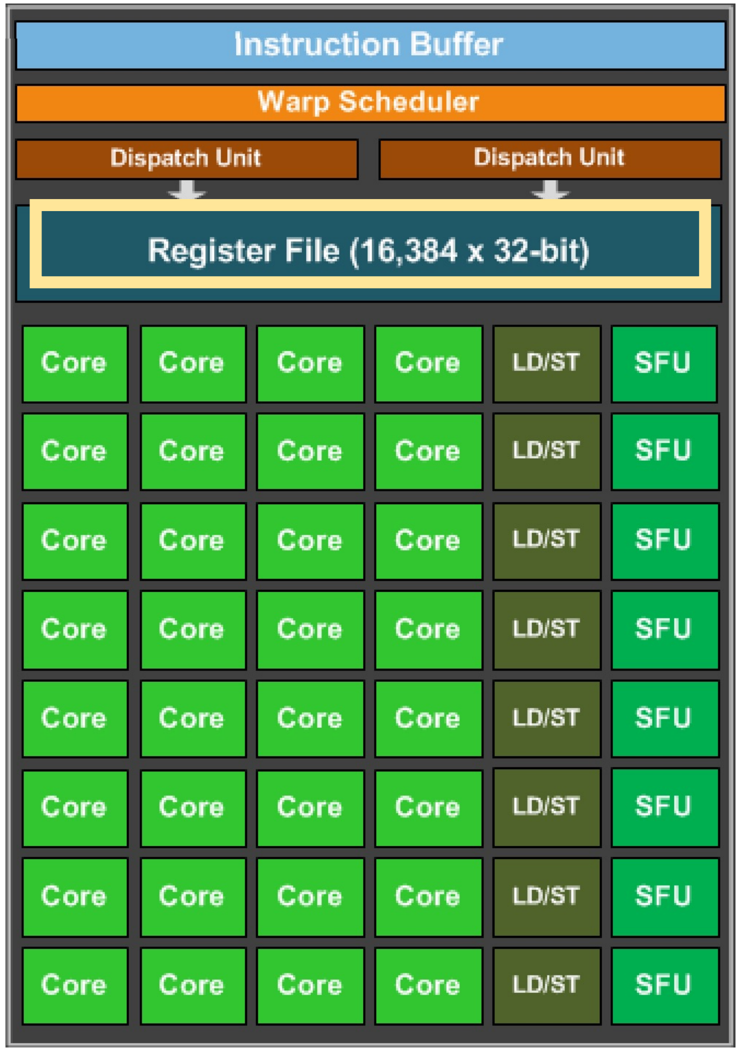
Hey, my memory has arrived!



preempt warp 2
and put warp 0 on

We can hide latency through preemption and concurrency!

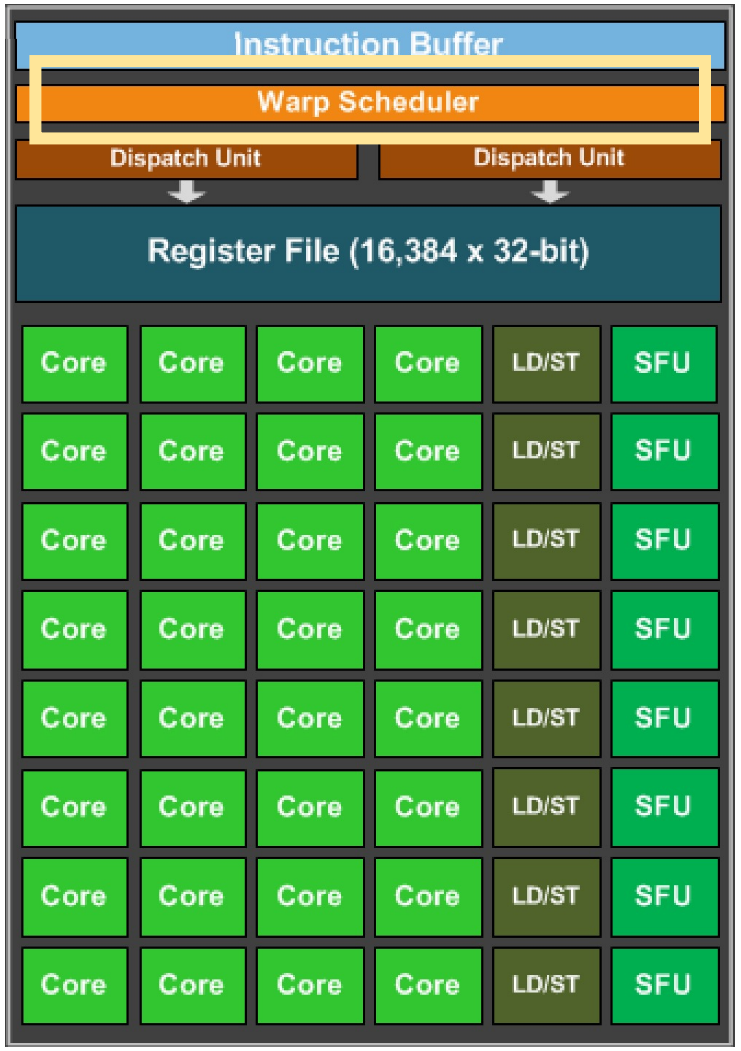
Preemption and concurrency?



But wait, I thought preemption was expensive?

Registers all stay on chip

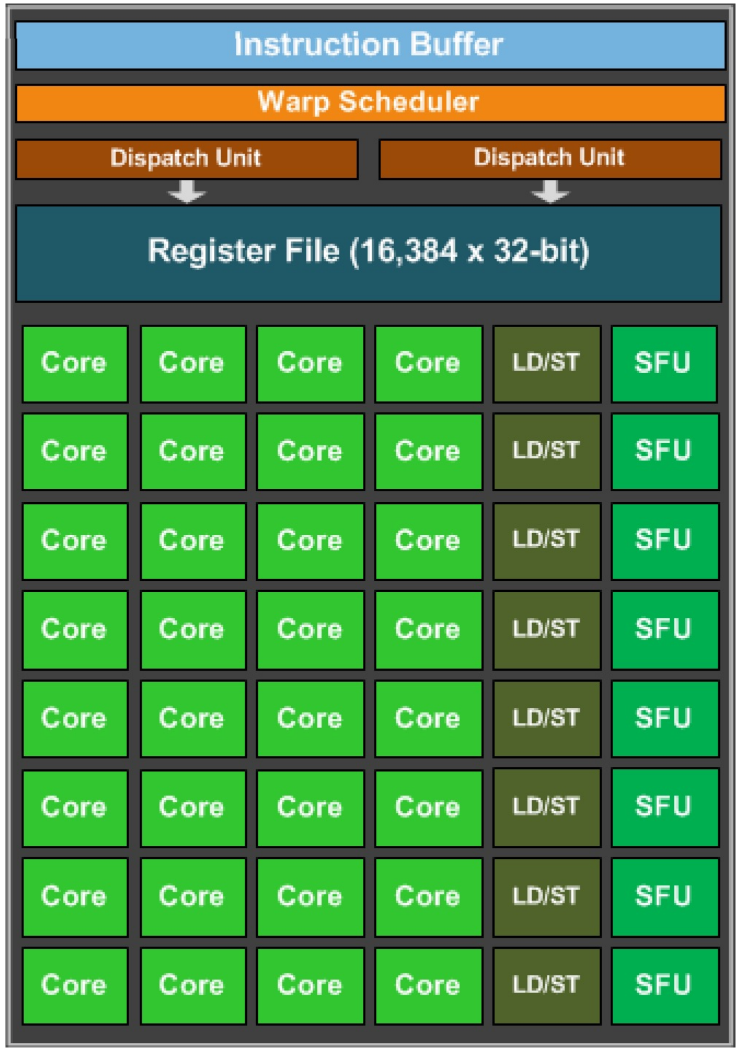
Preemption and concurrency?



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

Go back to our program

- What performance were we at?

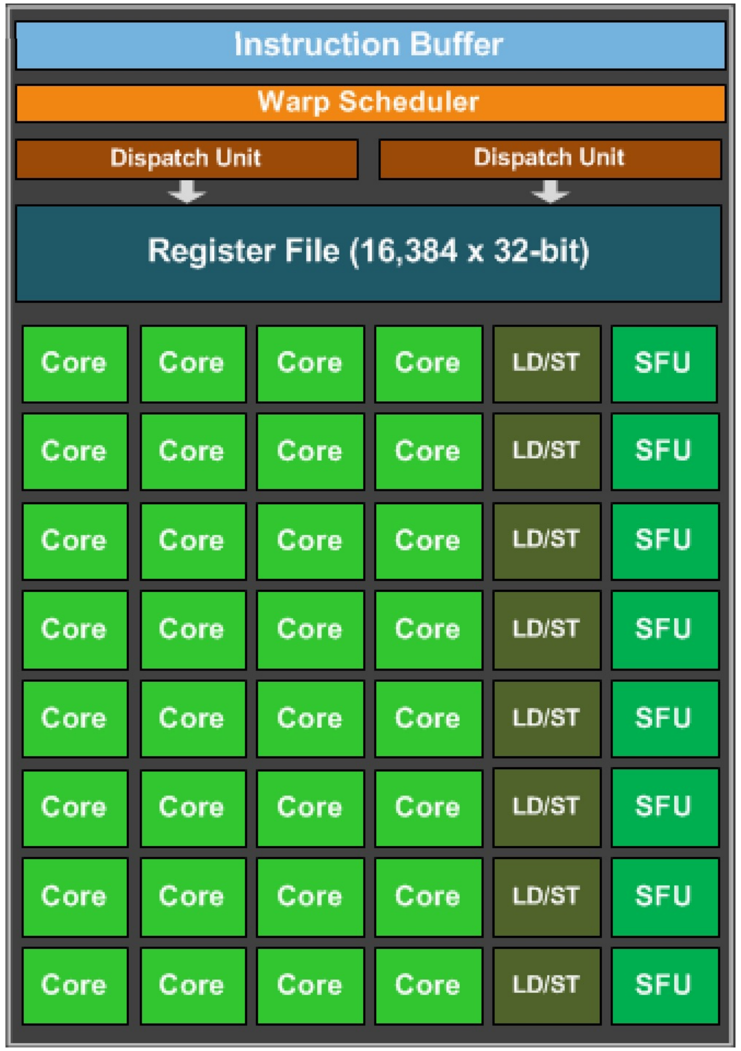
Schedule

- Review previous optimizations
- New optimizations
- Advanced GPU topics

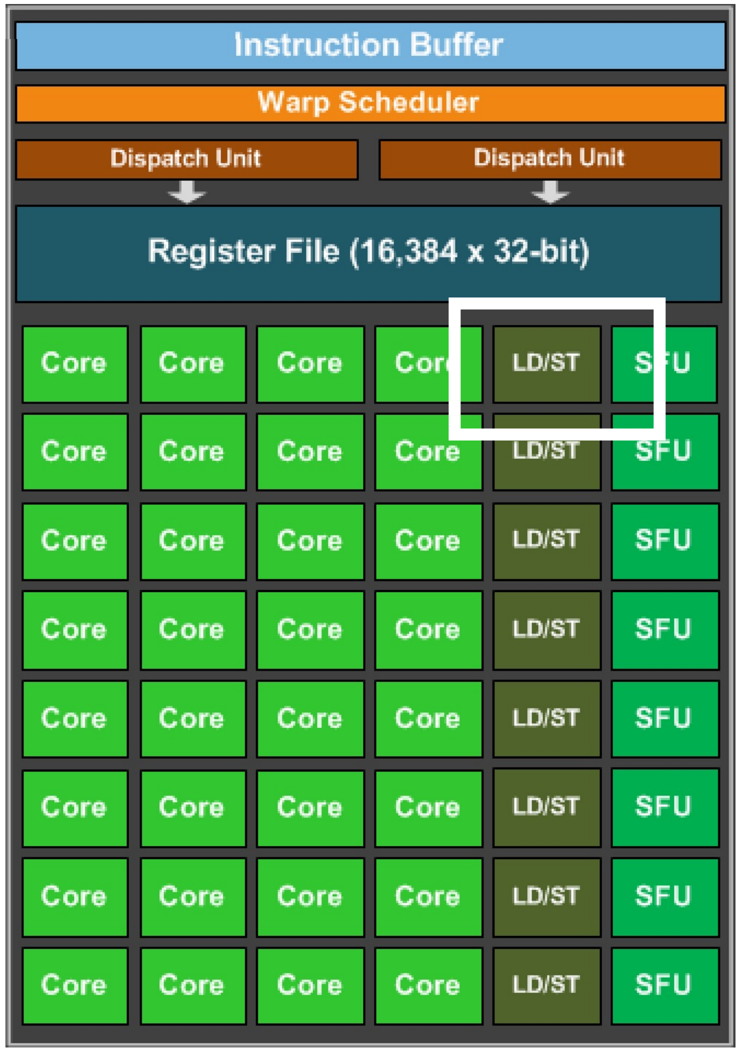
Schedule

- Review previous optimizations
- **New optimizations**
- Advanced GPU topics

Optimizing memory accesses



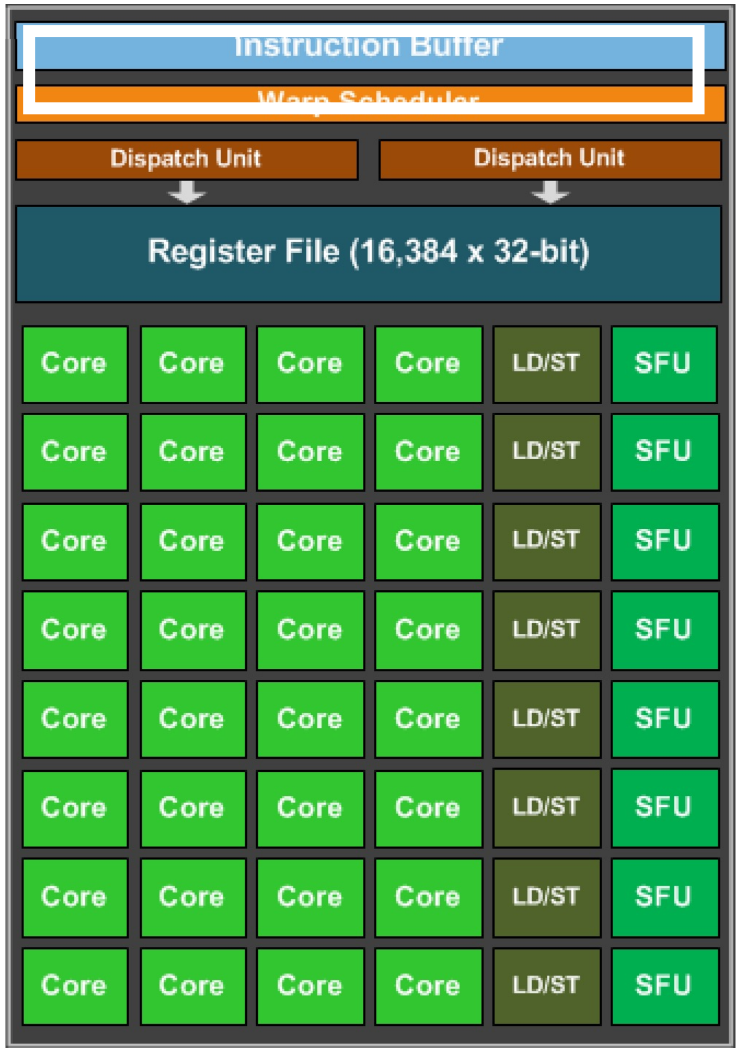
Optimizing memory accesses



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

Why doesn't every core have one?

Optimizing memory accesses

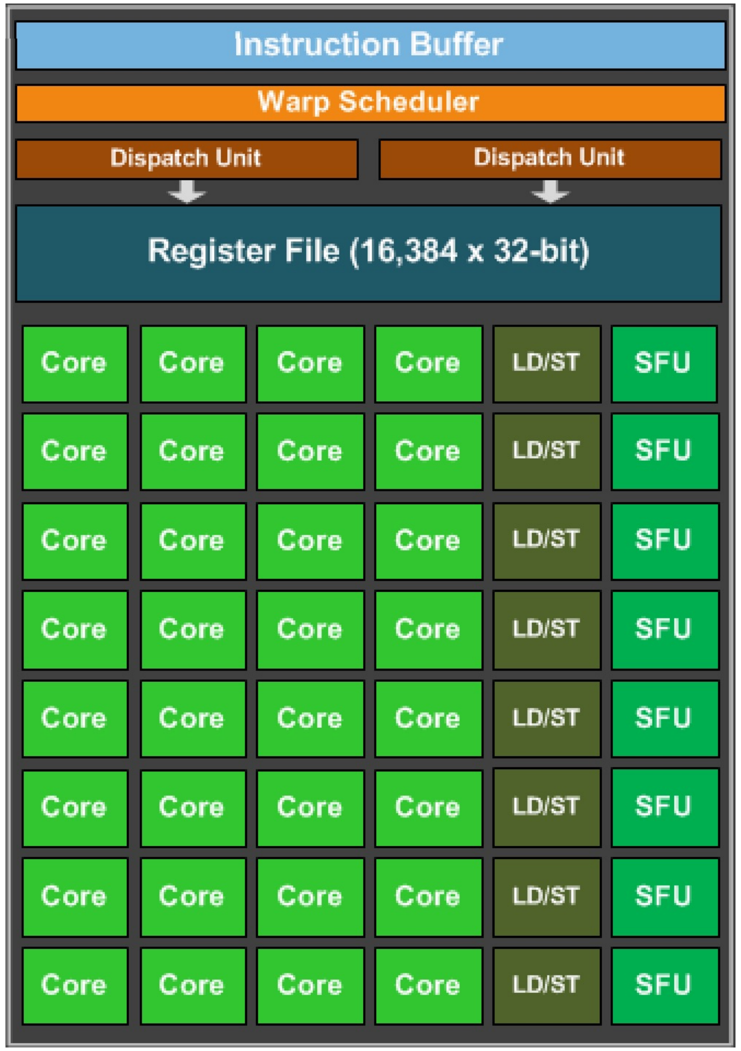


This is the instruction cache... Why doesn't every core have a instruction buffer to keep track of its program?

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

Why doesn't every core have one?

Warp execution



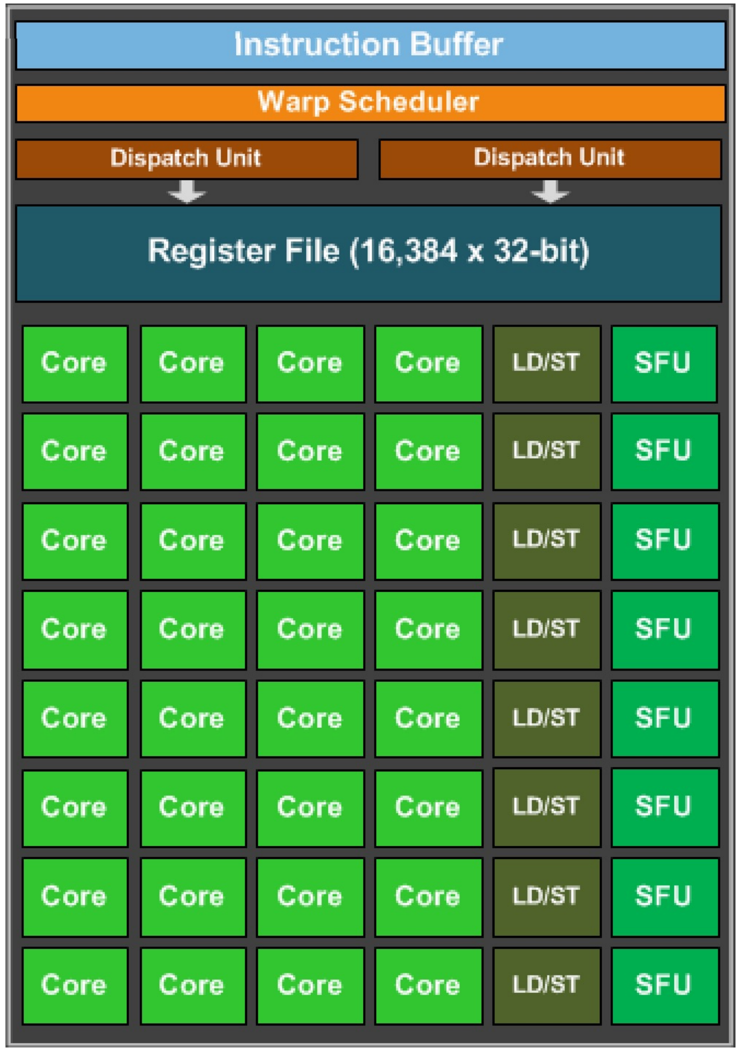
Groups of 32 threads are called a “warp”

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

Warp execution

Groups of 32 threads are called a "warp"

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

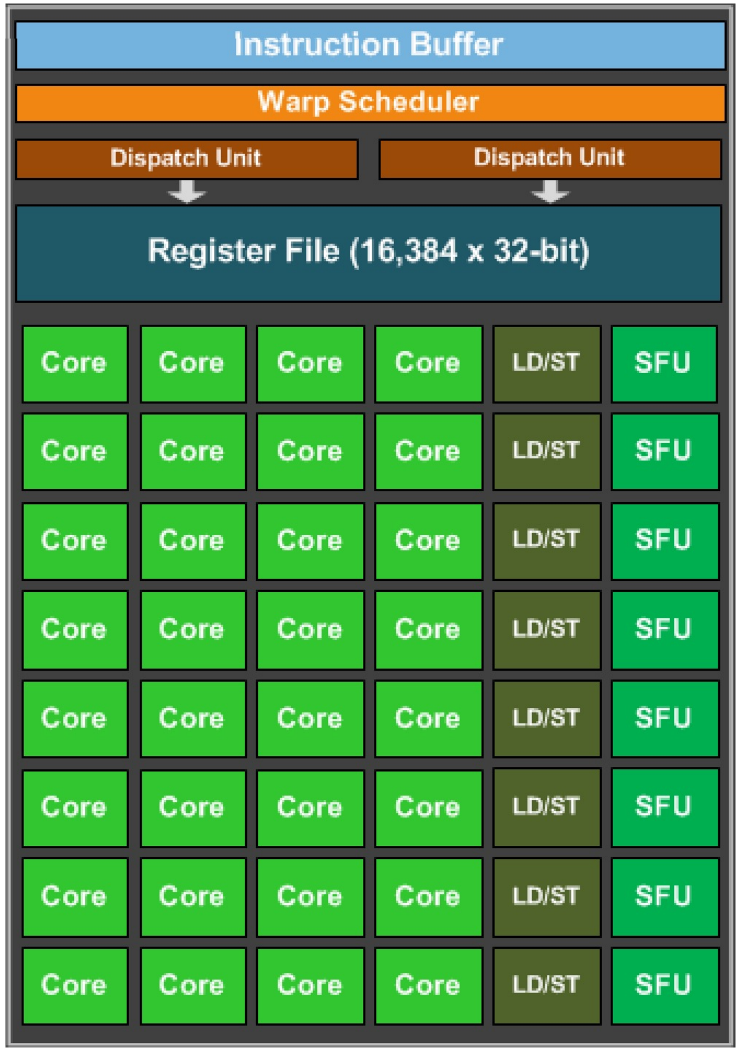


Program:
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Warp execution

Groups of 32 threads are called a "warp"

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



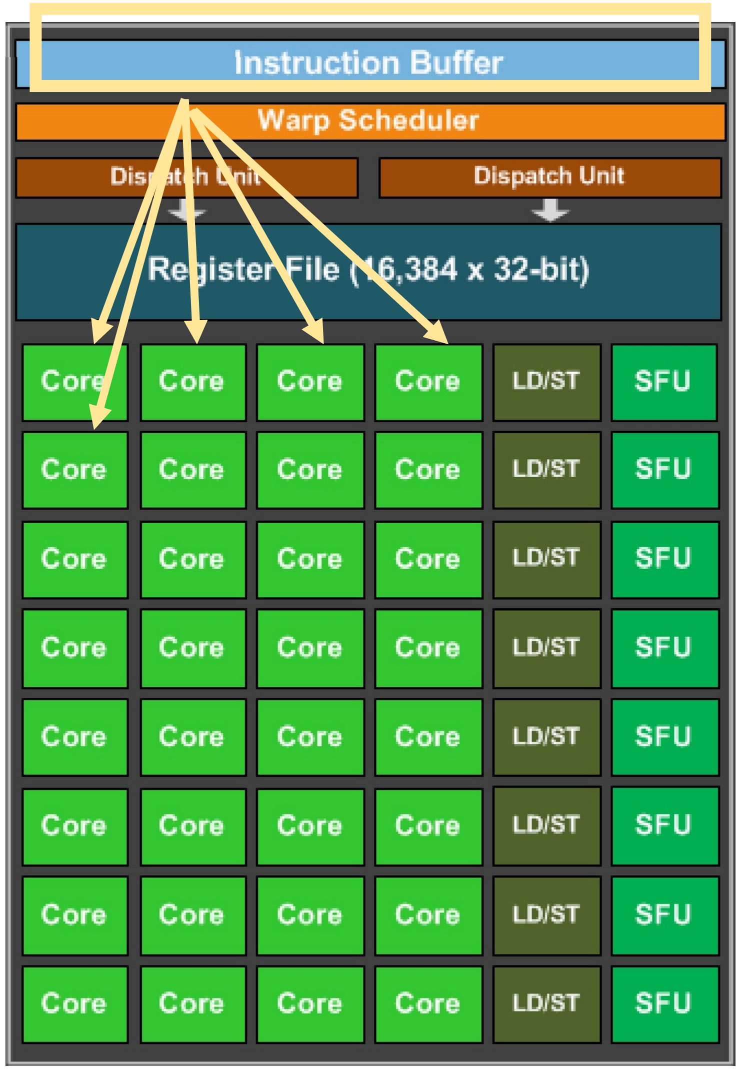
Program:

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

Warp execution

Groups of 32 threads are called a “warp”

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



instruction is fetched from the buffer and distributed to all the cores.

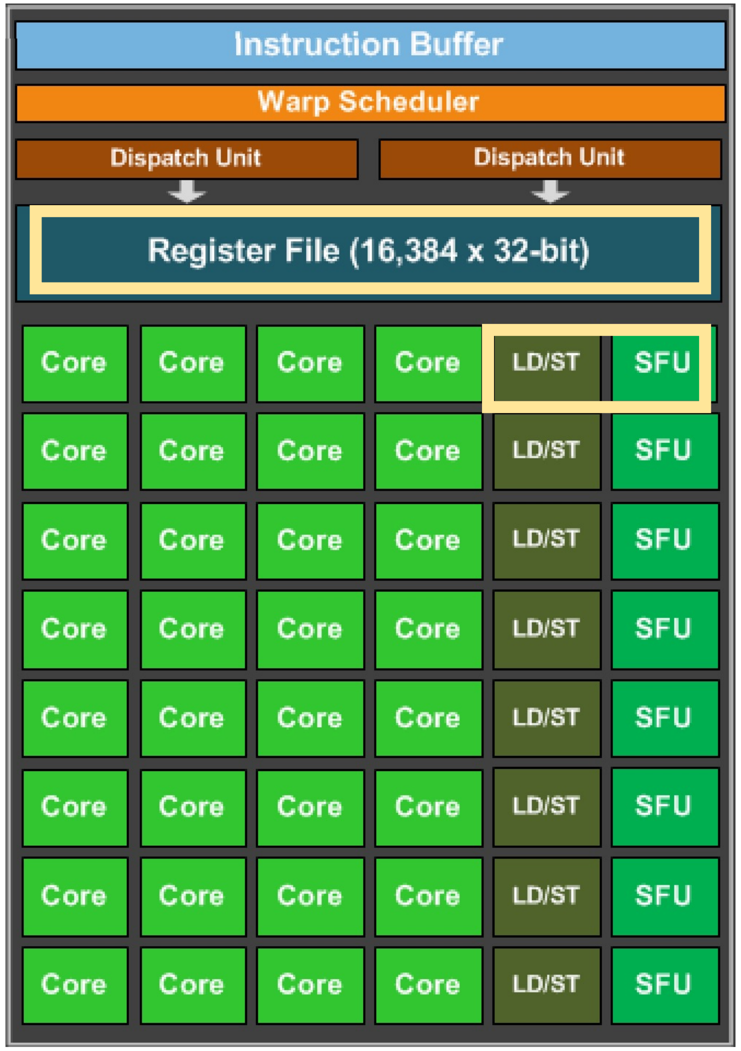
Program:

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

Warp execution

Groups of 32 threads are called a “warp”

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



Cores can a large register file
they share expensive HW units (load/store and special functions)

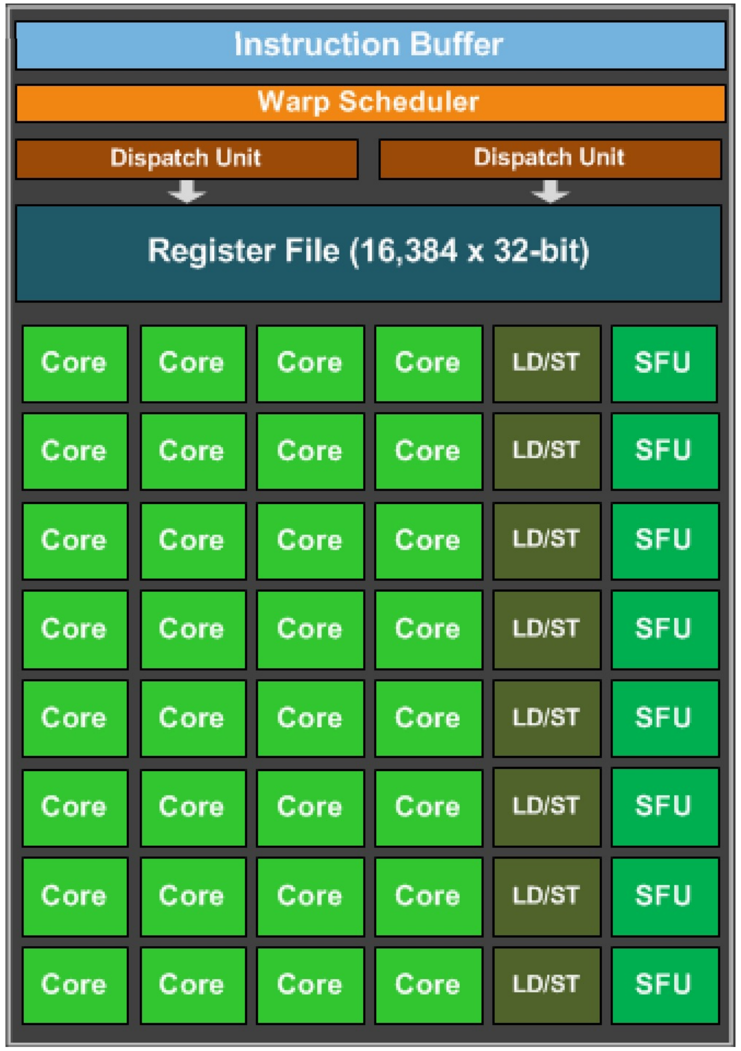
Program:

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

Warp execution

Groups of 32 threads are called a “warp”

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



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

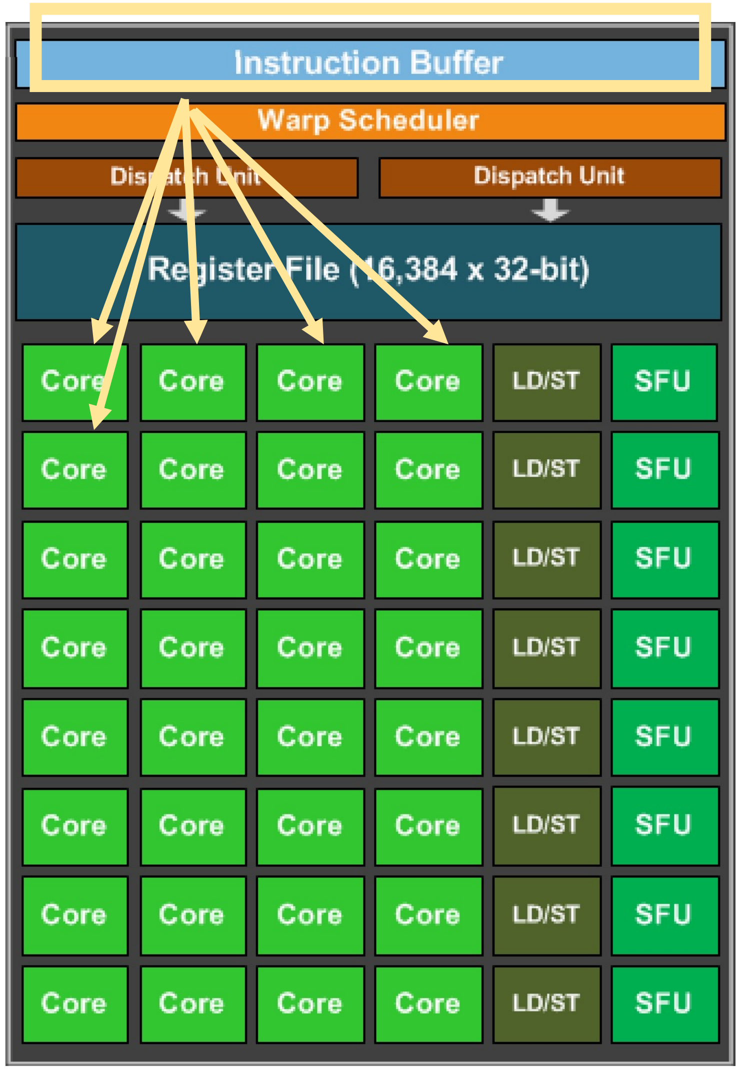
Program:

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

Warp execution

Groups of 32 threads are called a "warp"

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



Start the next instruction.

Program:

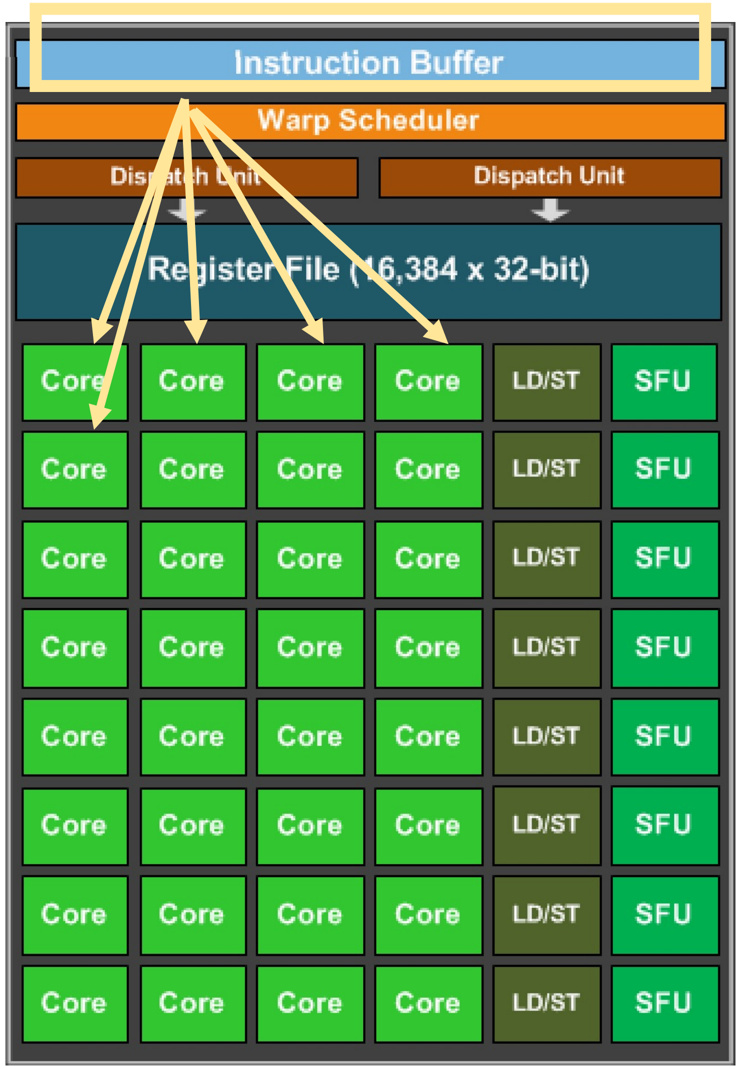
```
int variable1 = b[0];
int variable2 = c[0];
int variable3 = variable1 + variable2;
a[0] = variable3;
```

Why would we have a programming model like this?

Warp execution

Groups of 32 threads are called a “warp”

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



Start the next instruction.

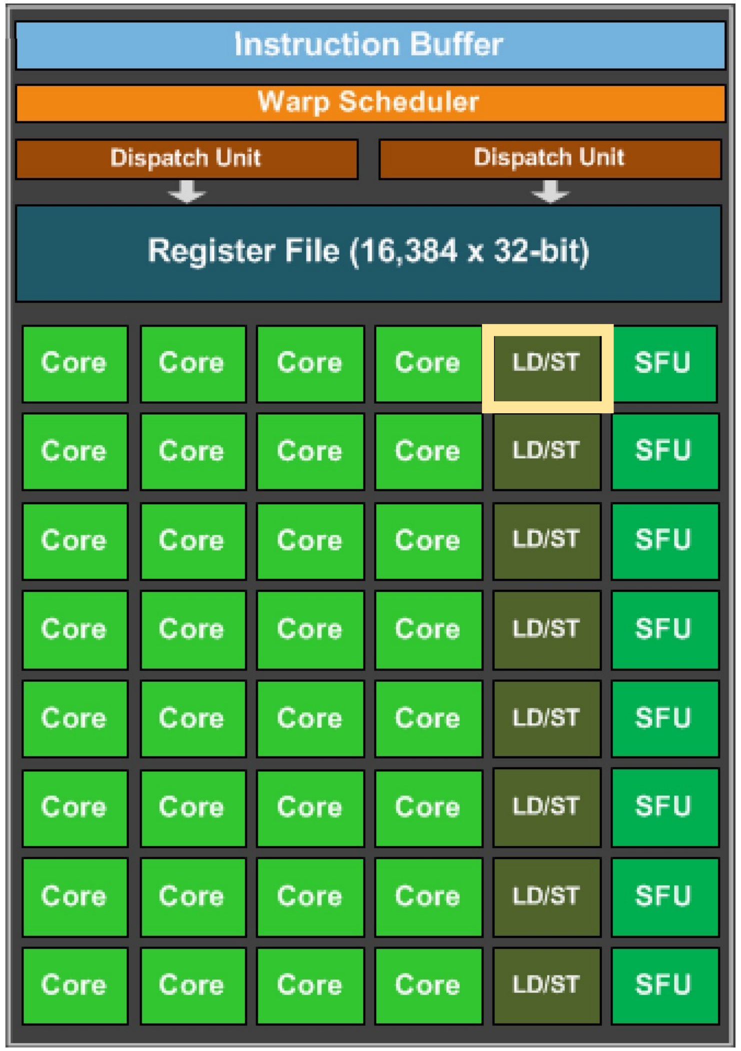
Program:

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

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

Warp execution

Lets look closer at memory

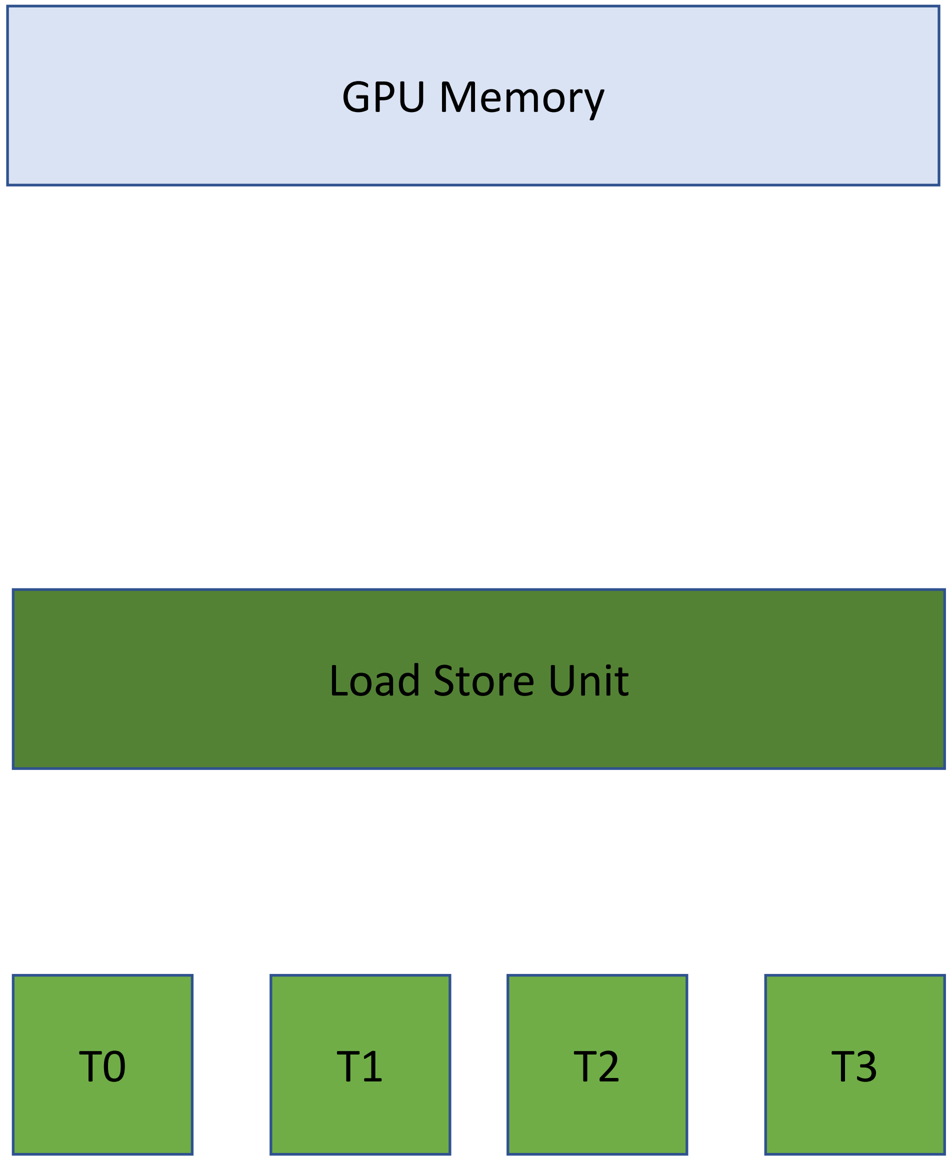


Program:

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

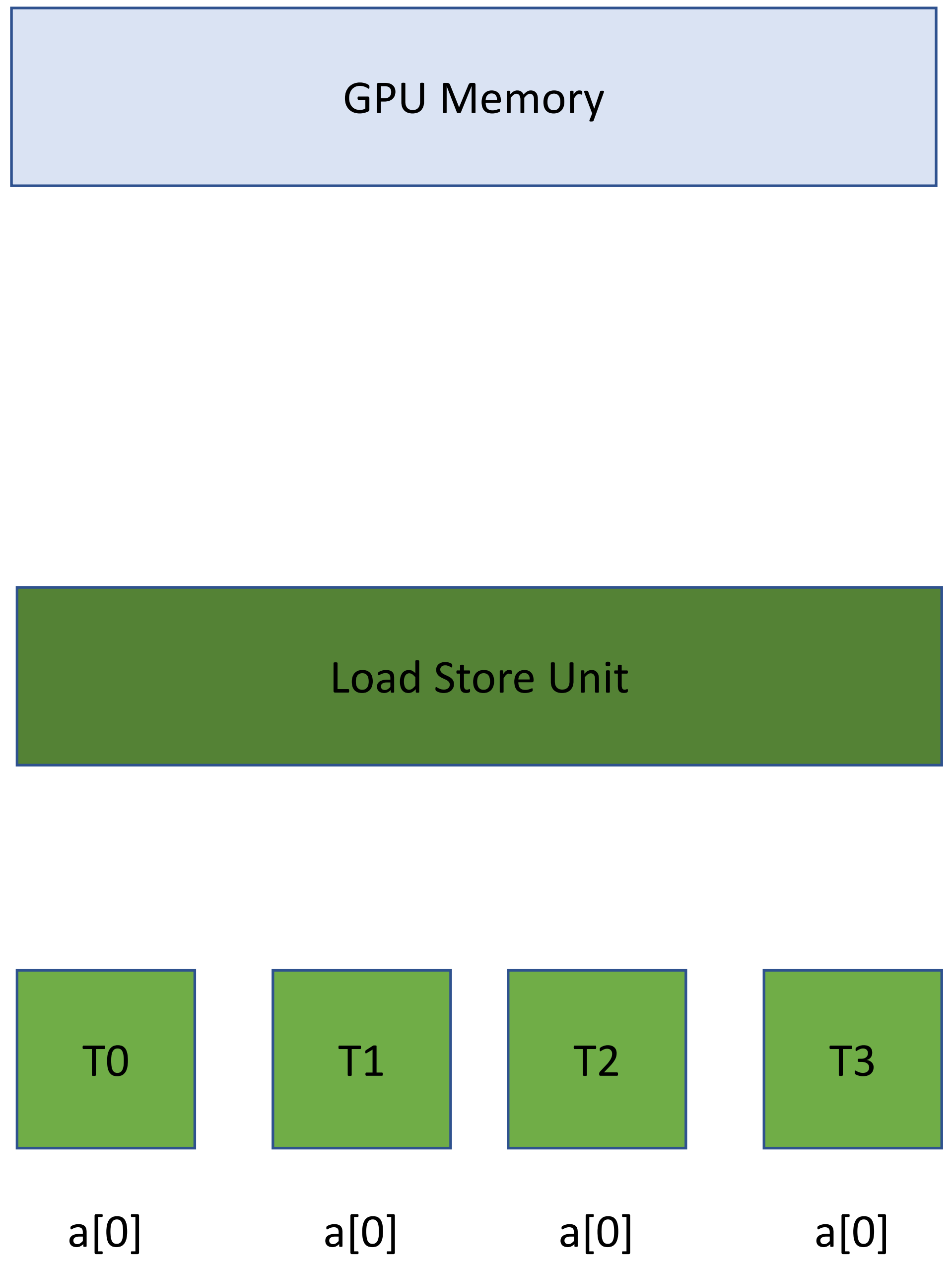
4 cores are accessing memory. what happens if they access the same value?

4 cores are accessing memory. What can happen



4 cores are accessing memory. What can happen

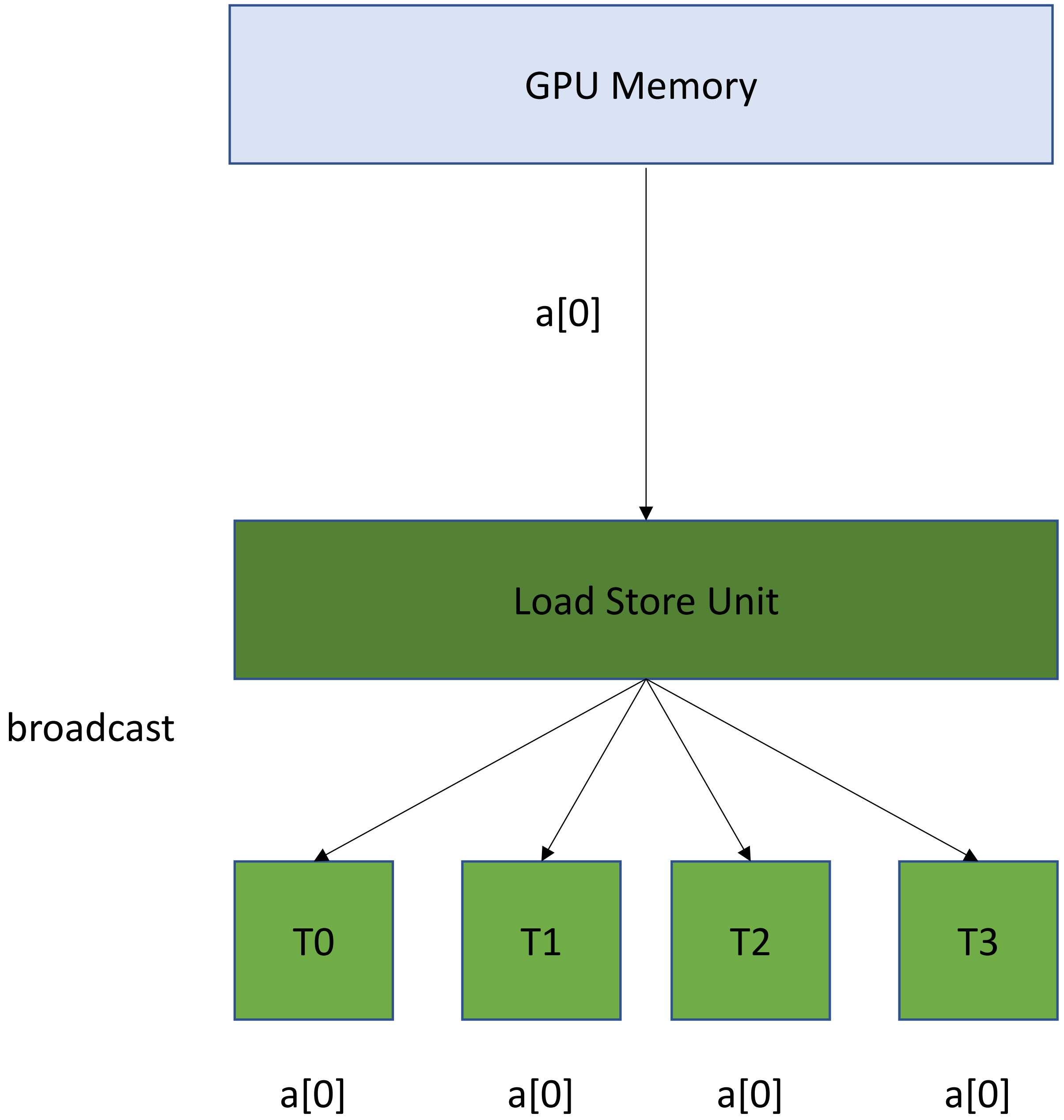
All read the same value



4 cores are accessing memory. What can happen

All read the same value

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

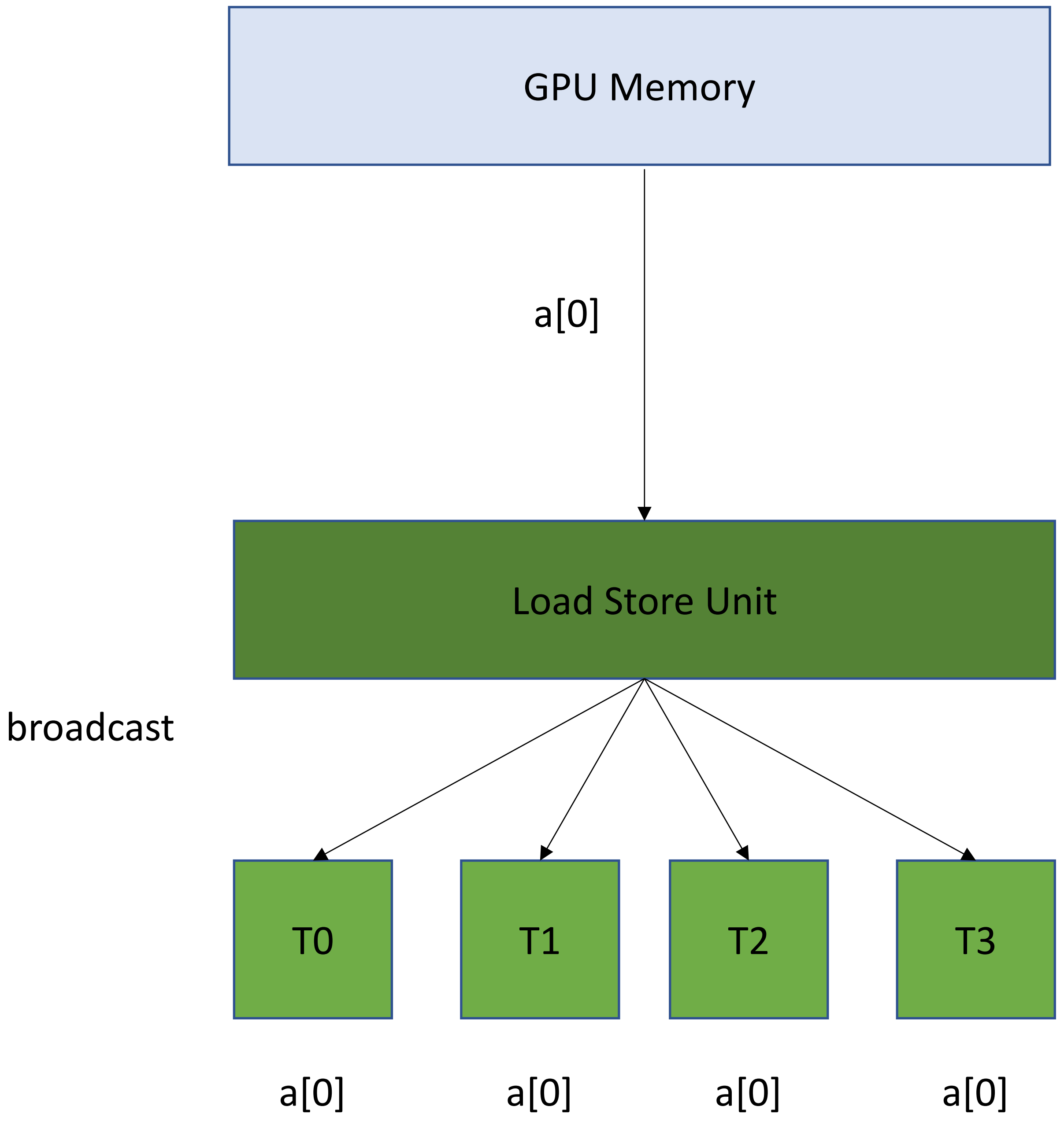


4 cores are accessing memory. What can happen

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

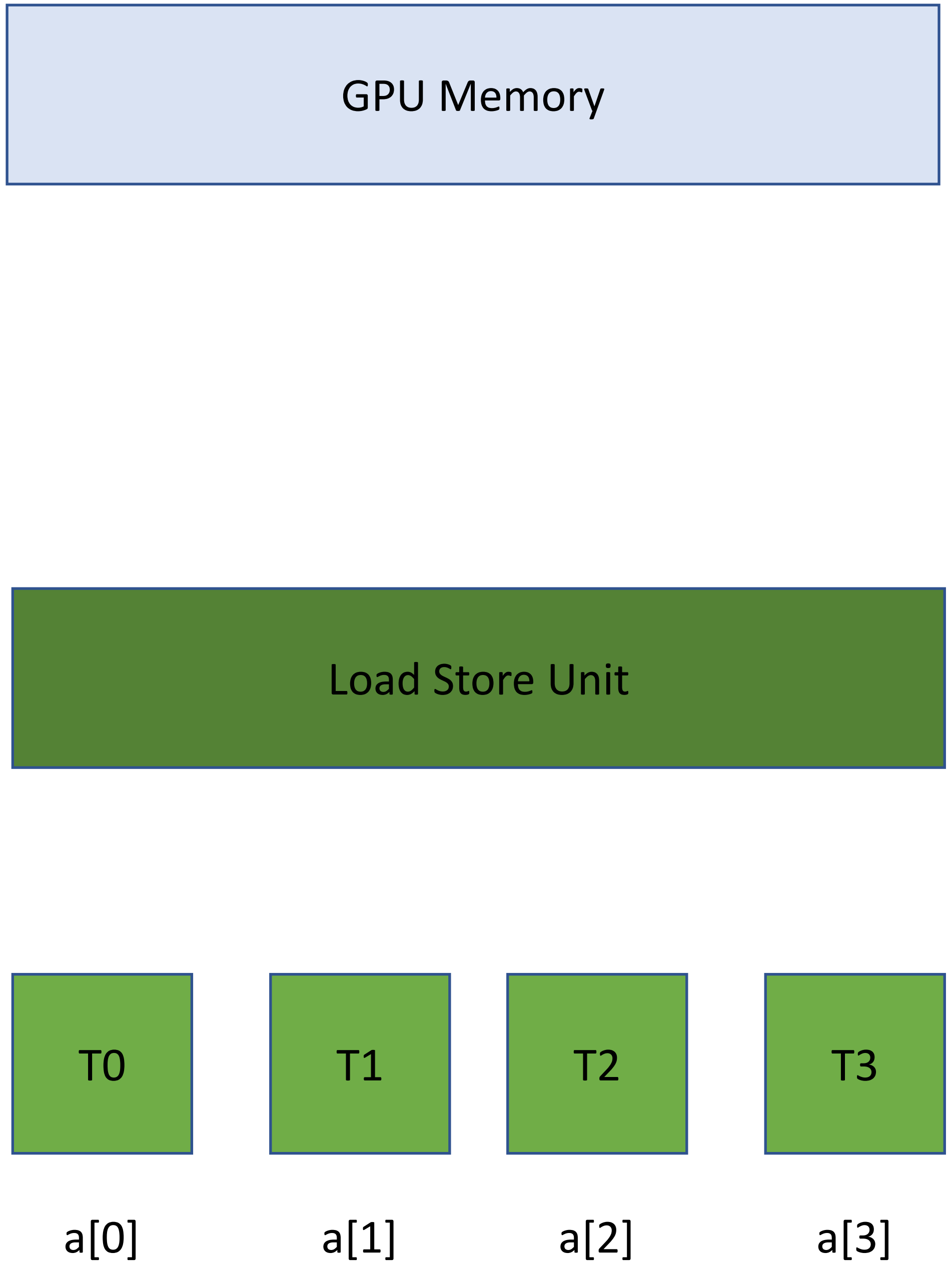
1 request to GPU memory

Efficient, but probably not too common.



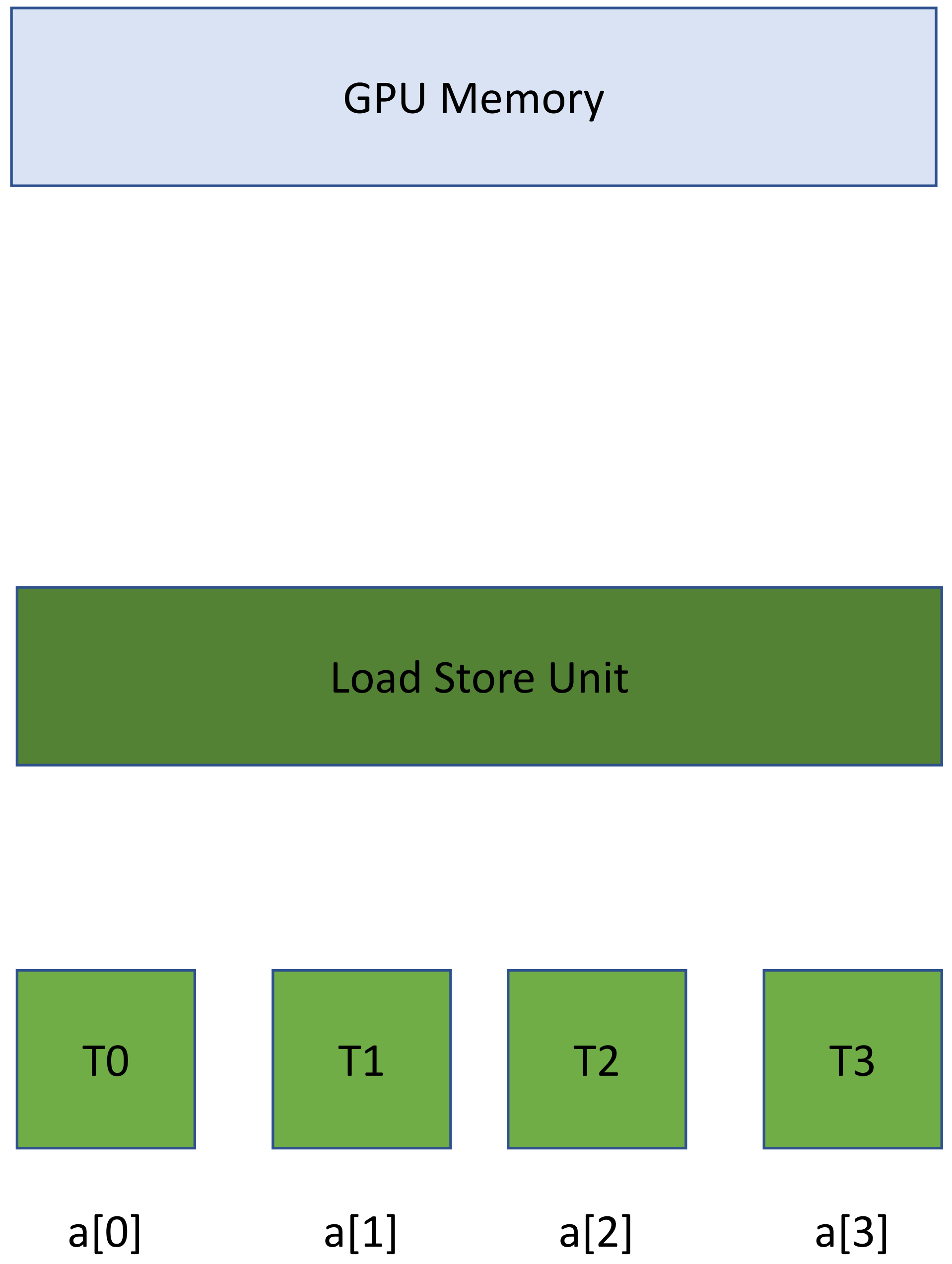
4 cores are accessing memory. What can happen

Read contiguous values



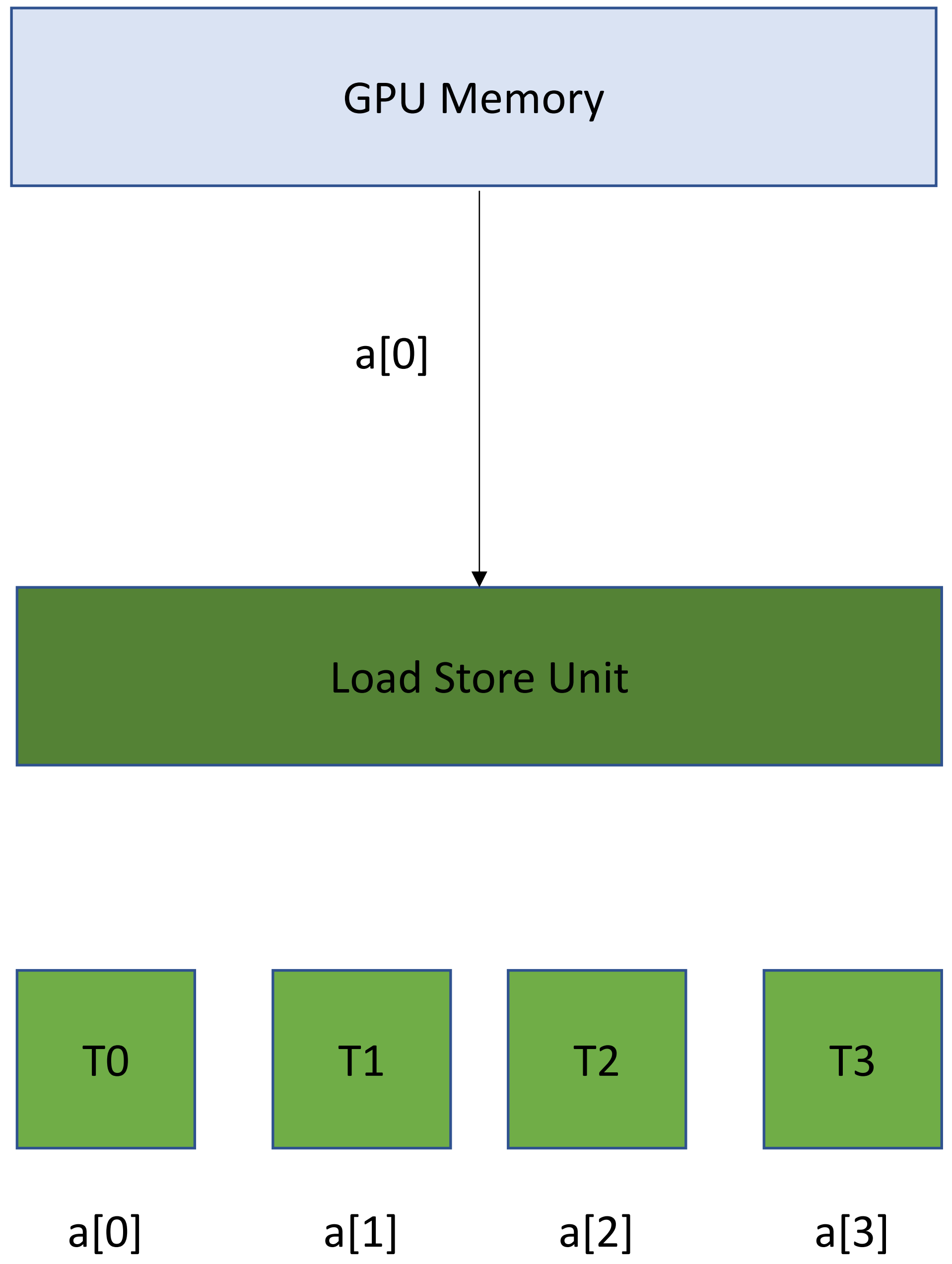
4 cores are accessing memory. What can happen

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



4 cores are accessing memory. What can happen

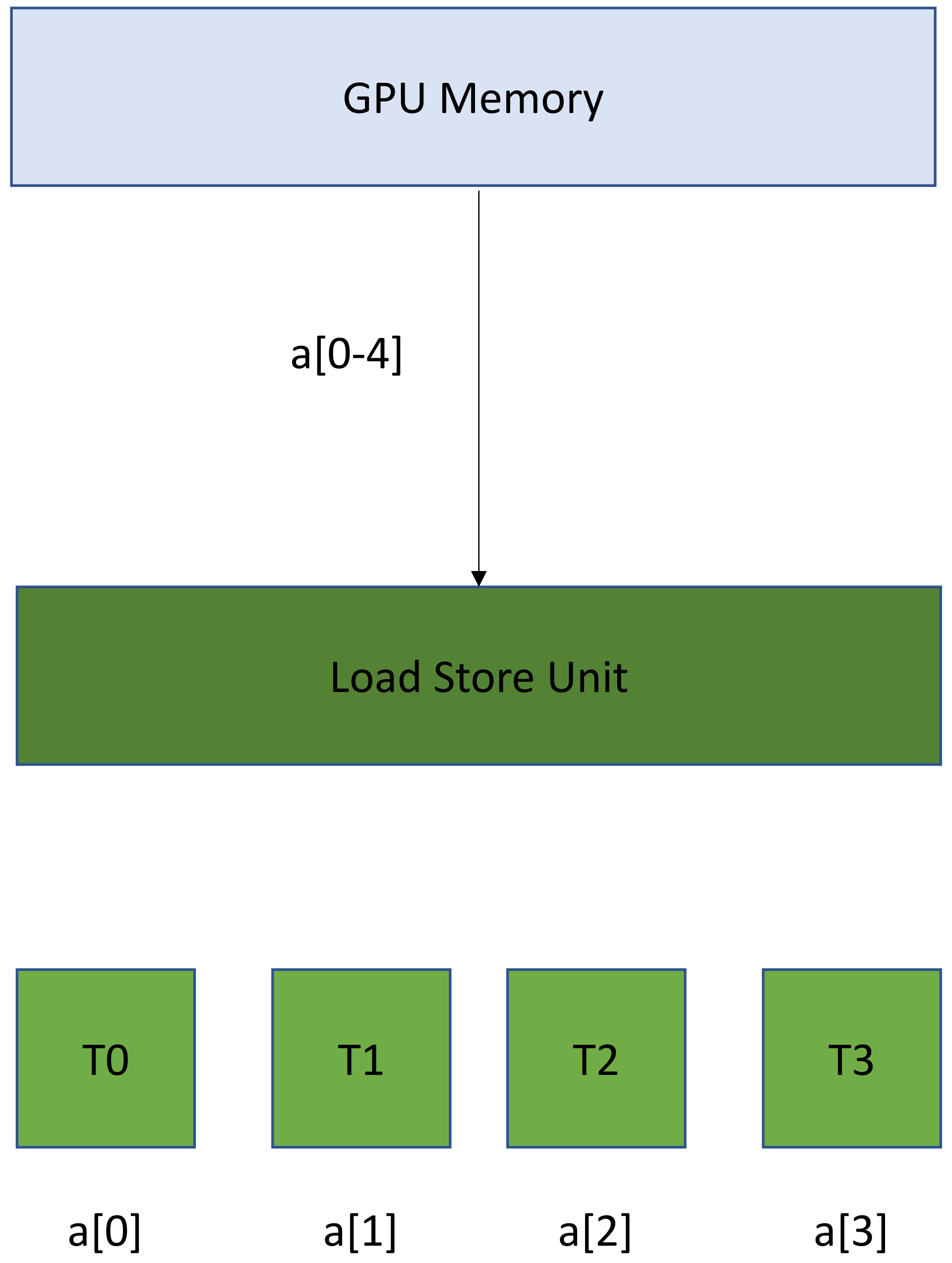
Read contiguous values
Like the CPU cache, the Load/Store Unit reads in memory in chunks. 16 bytes



4 cores are accessing memory. What can happen

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

Can easily distribute the values to the threads

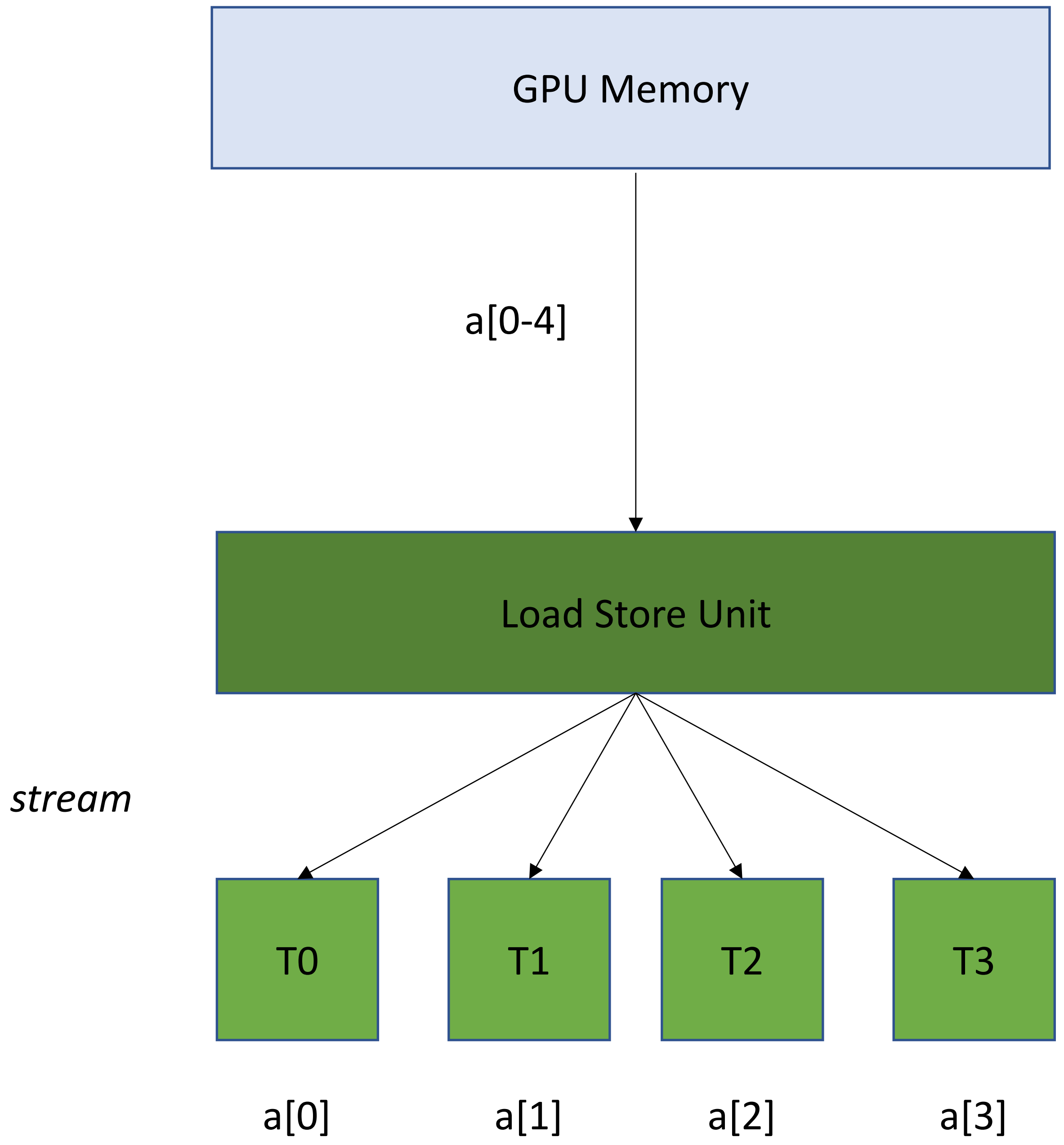


4 cores are accessing memory. What can happen

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

Can easily distribute the values to the threads

1 request to GPU memory

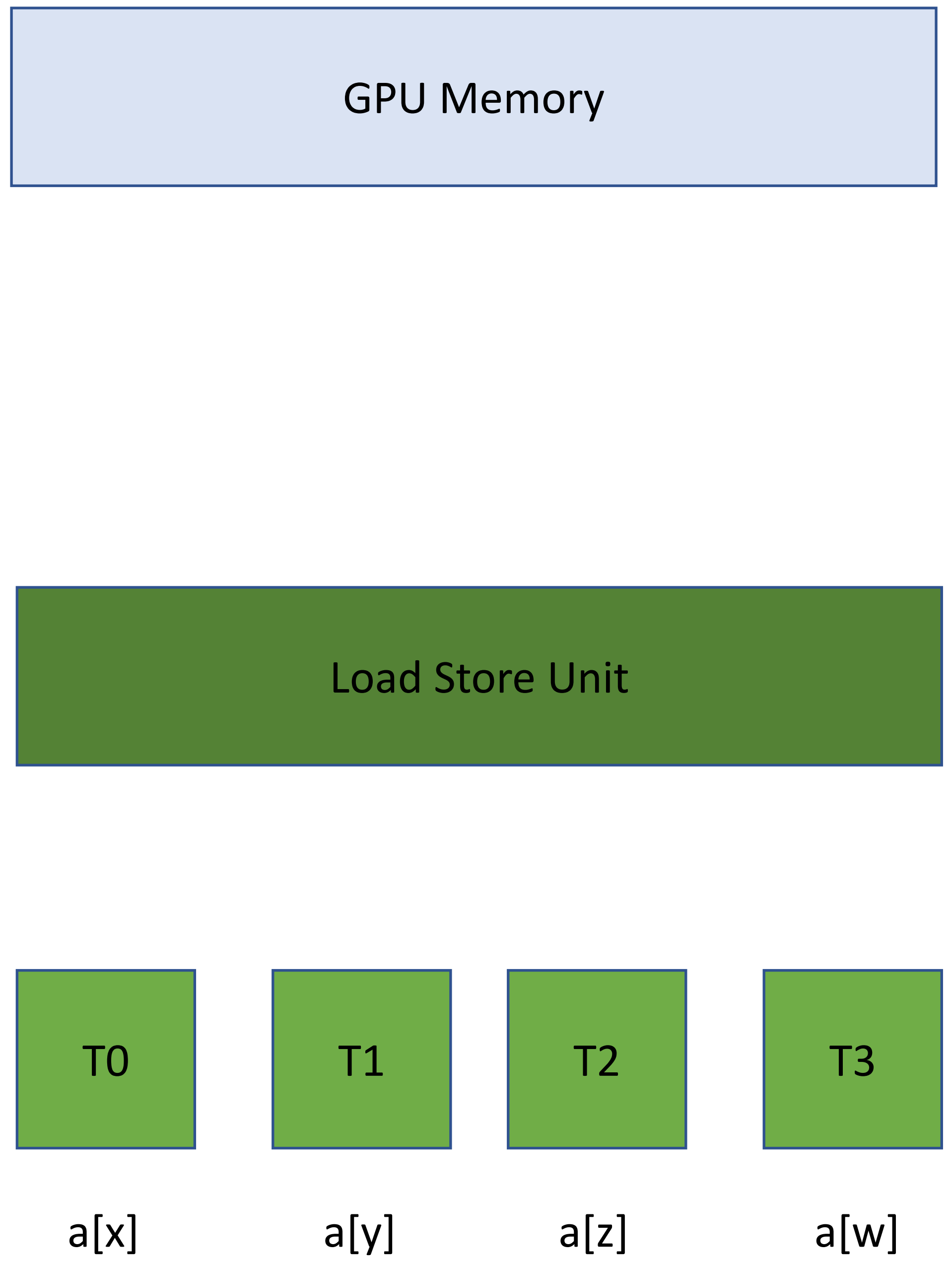


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

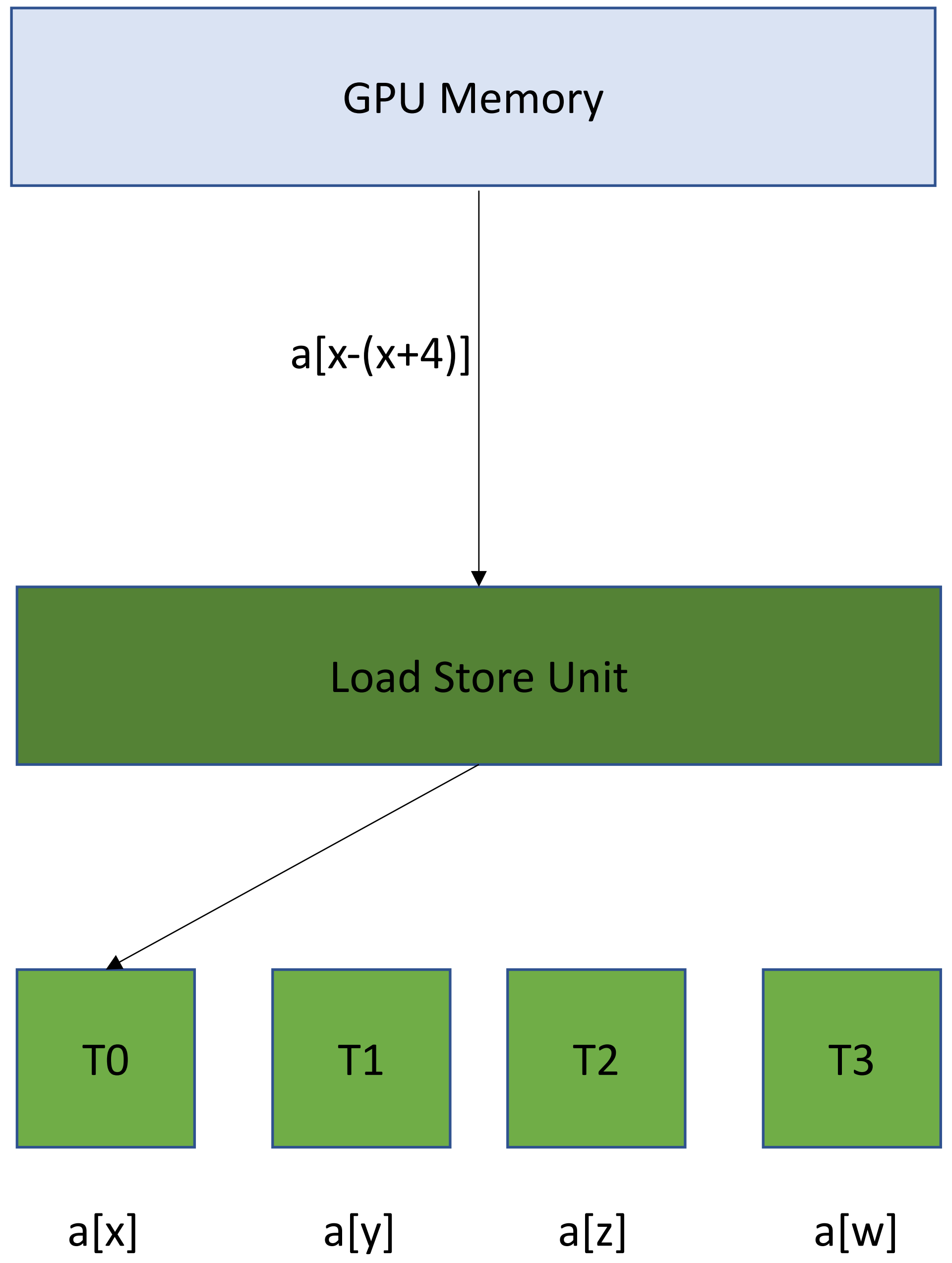


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

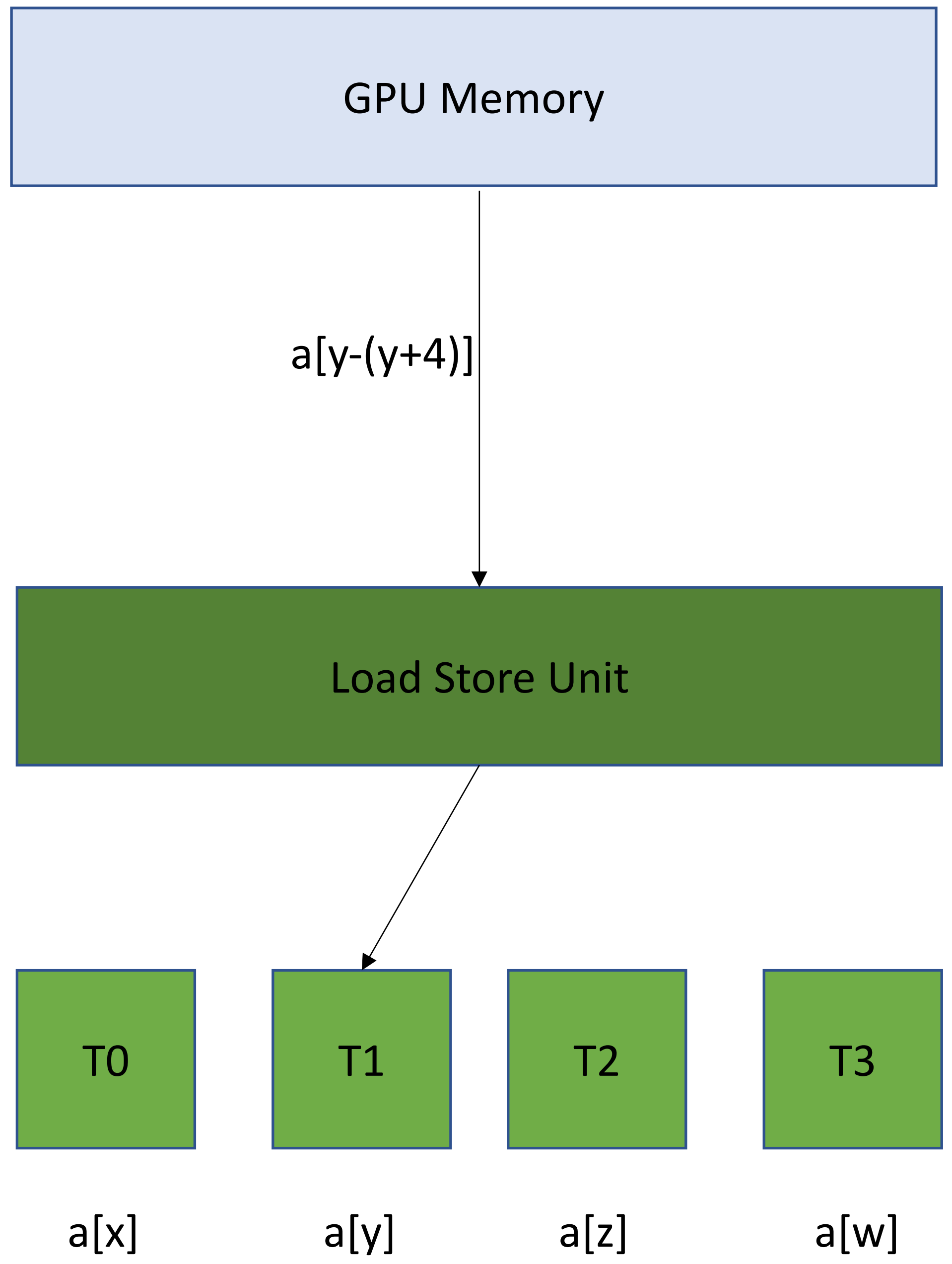


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

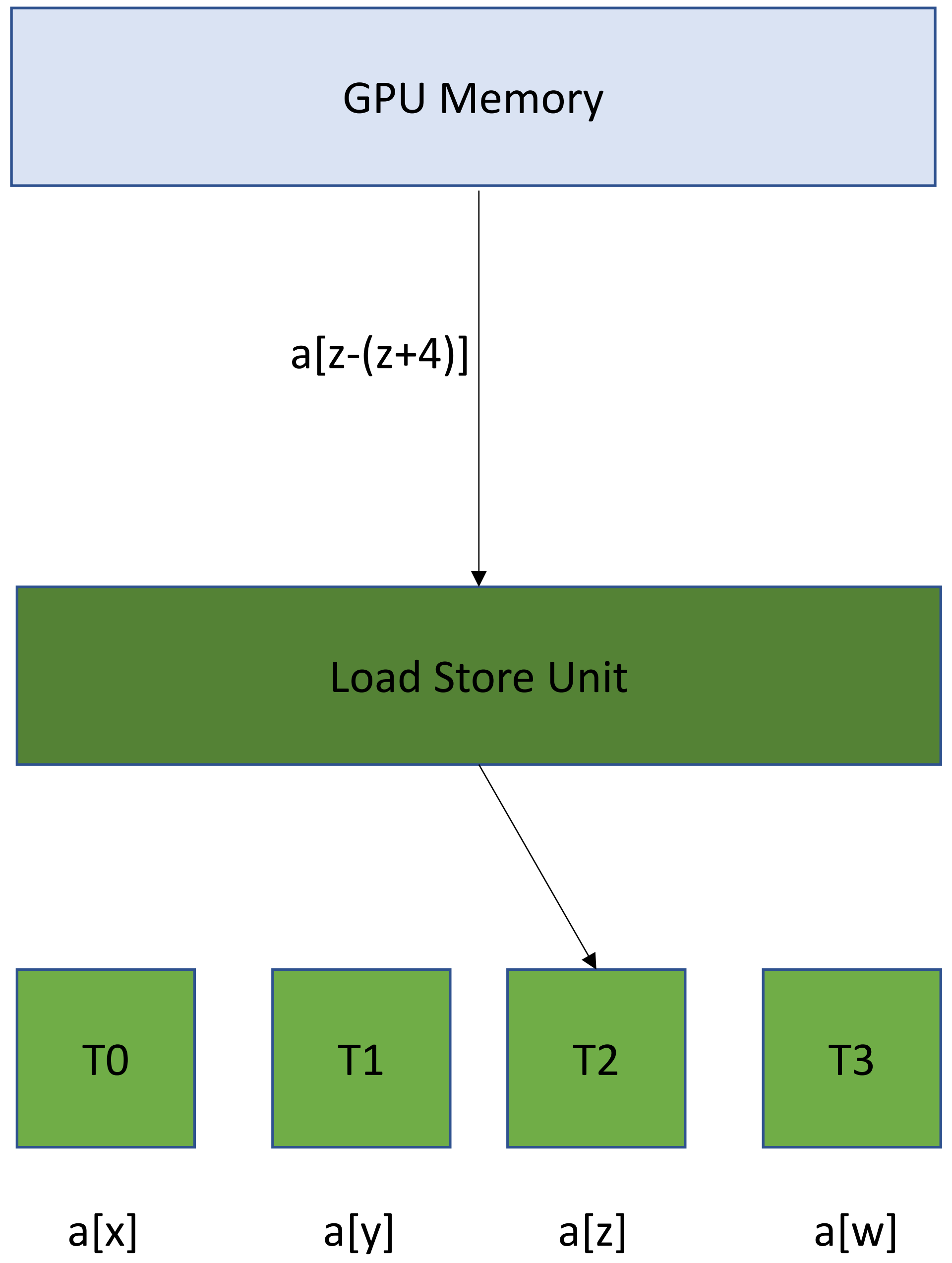


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

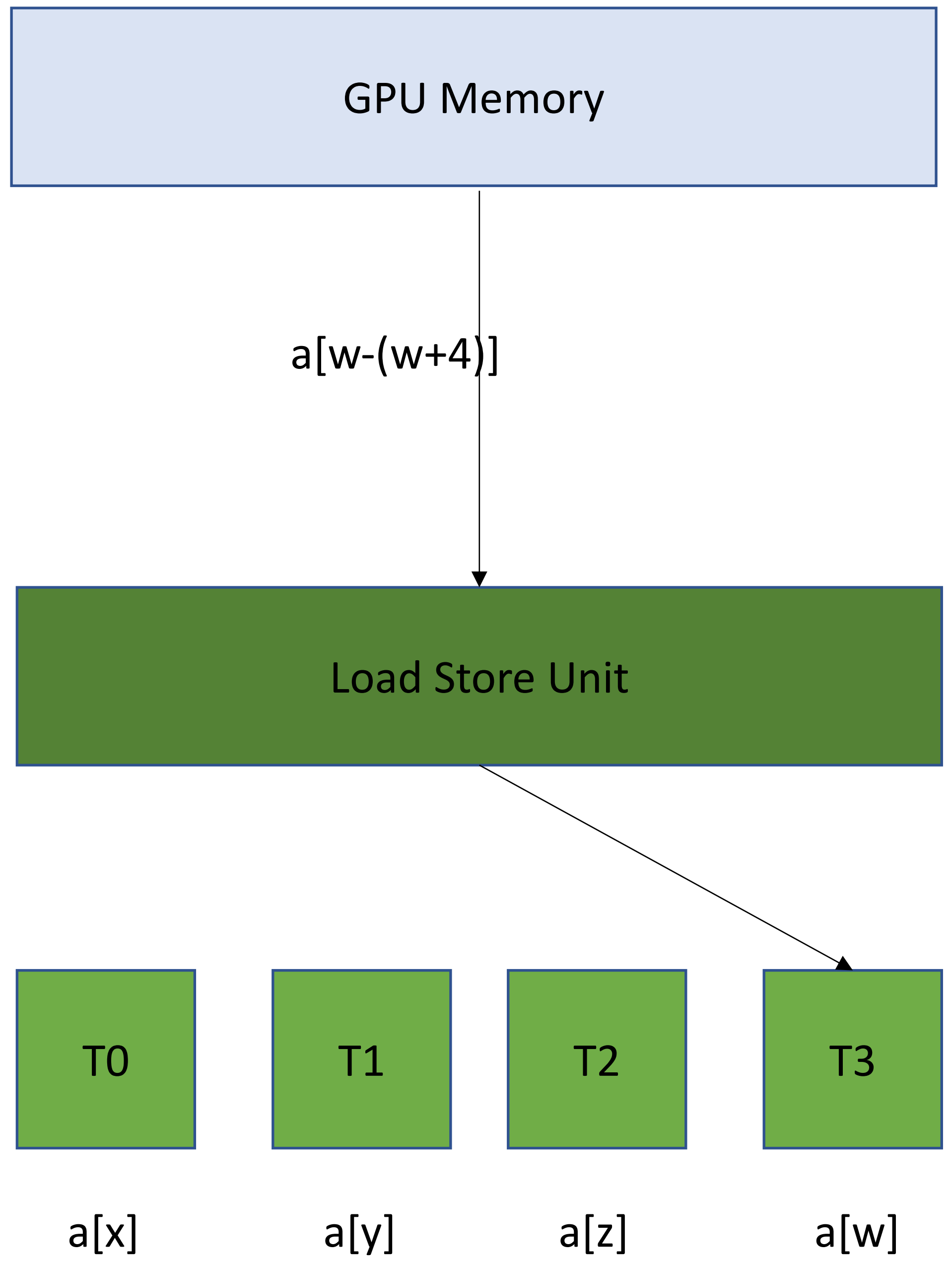


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory



Go back to our program

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

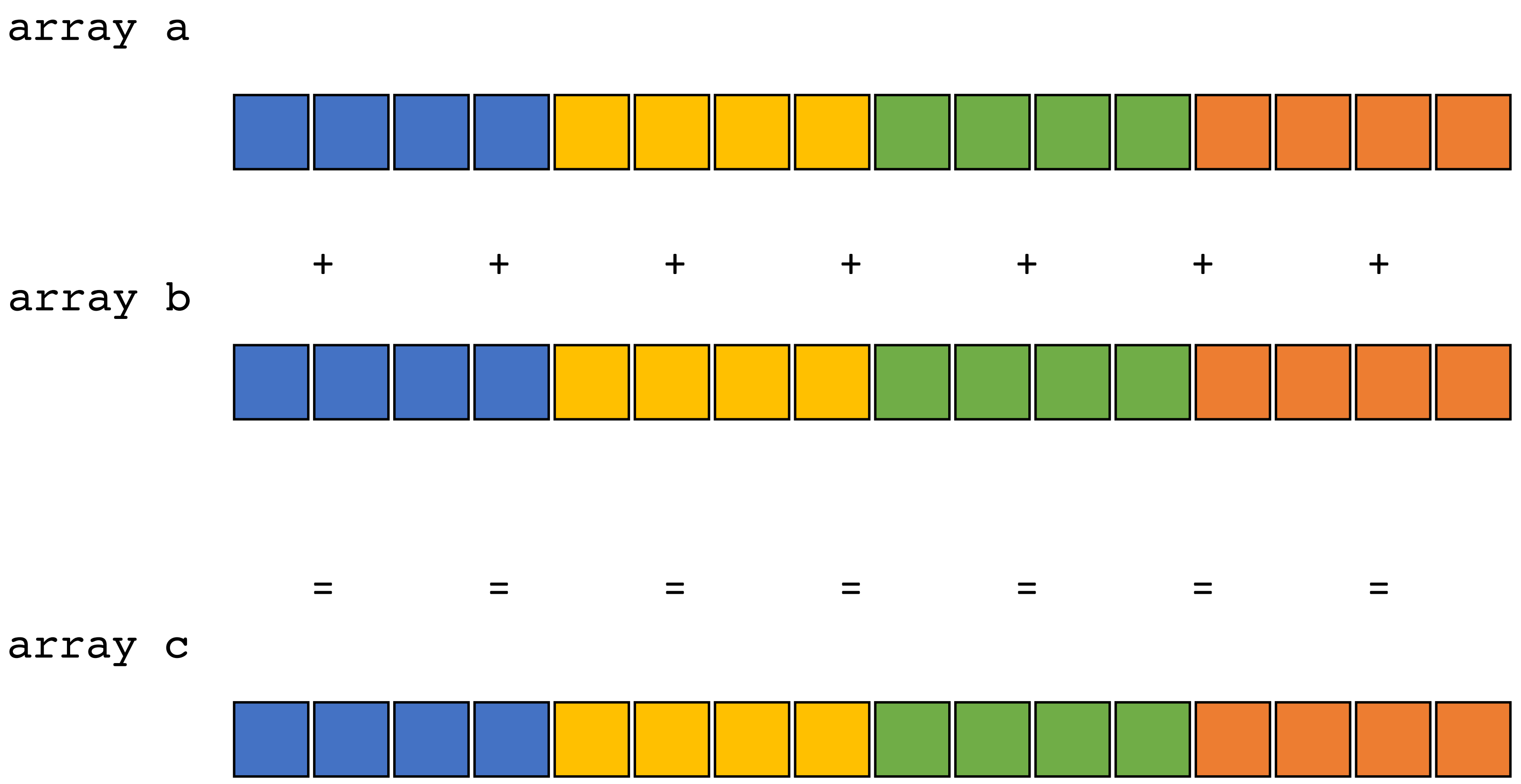
calling the function

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

Chunked Pattern

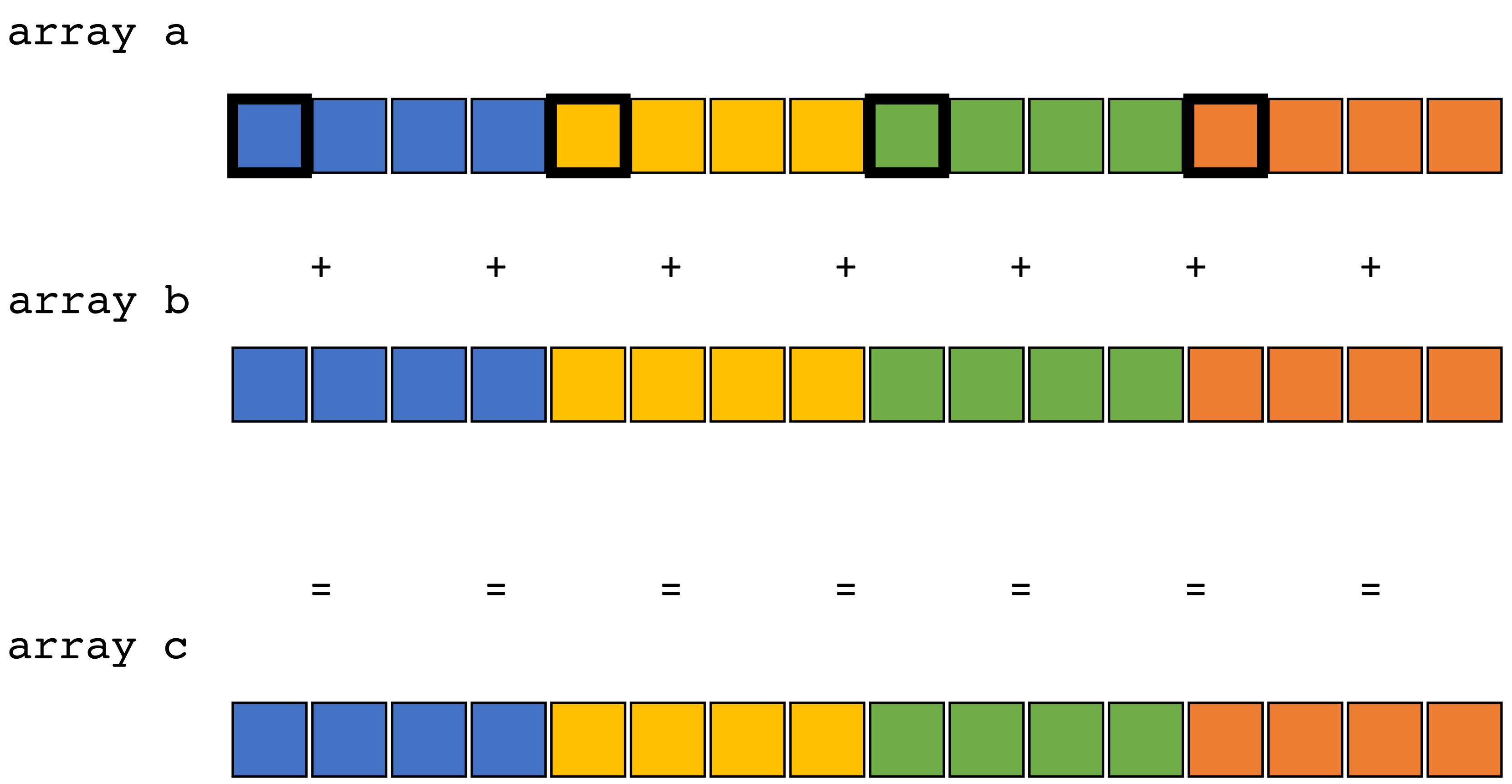
Computation can easily be divided into threads

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 queue. What sort of access is this?

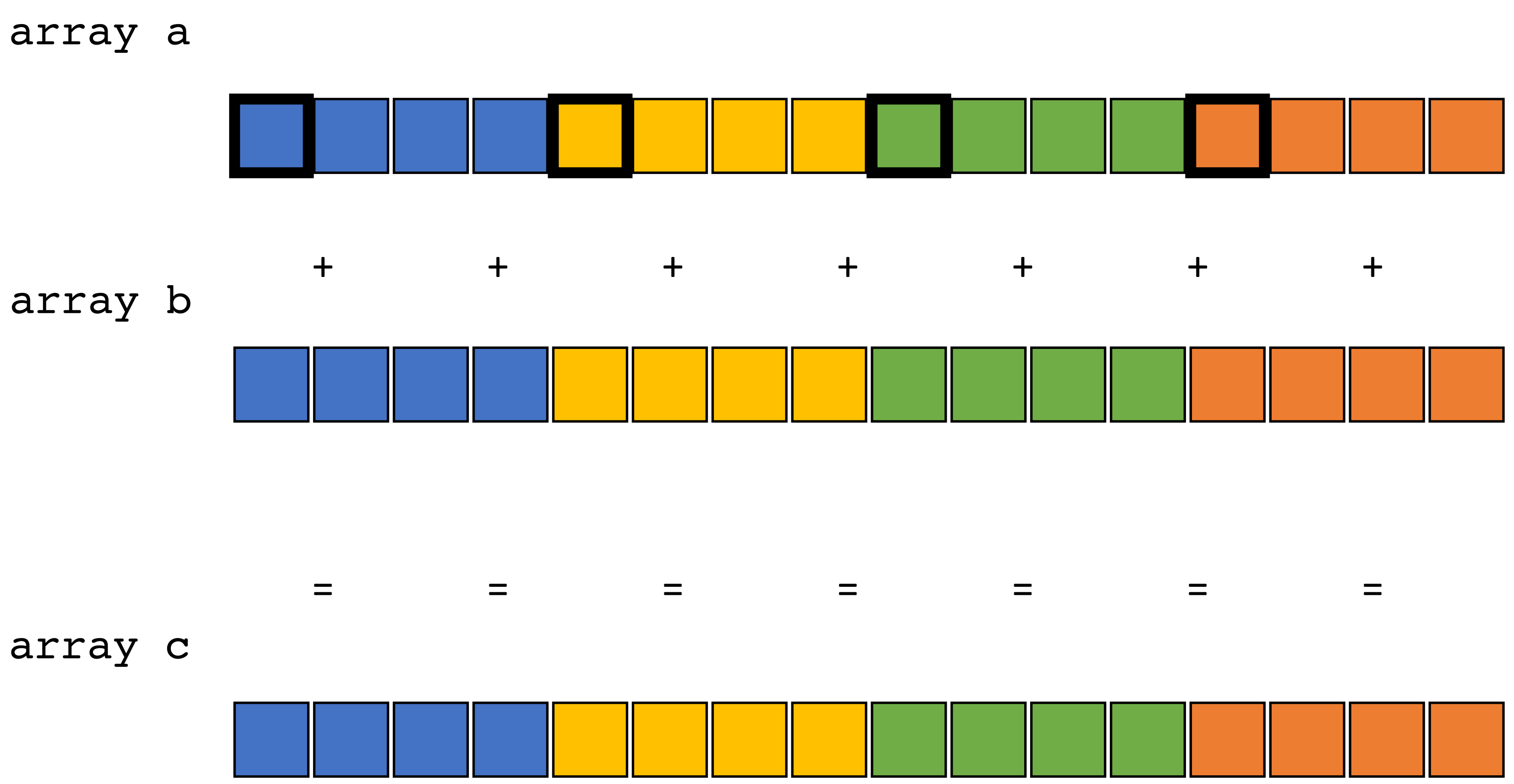


Computation can easily be divided into threads

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

Chunked Pattern

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



Computation can easily be divided into threads

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

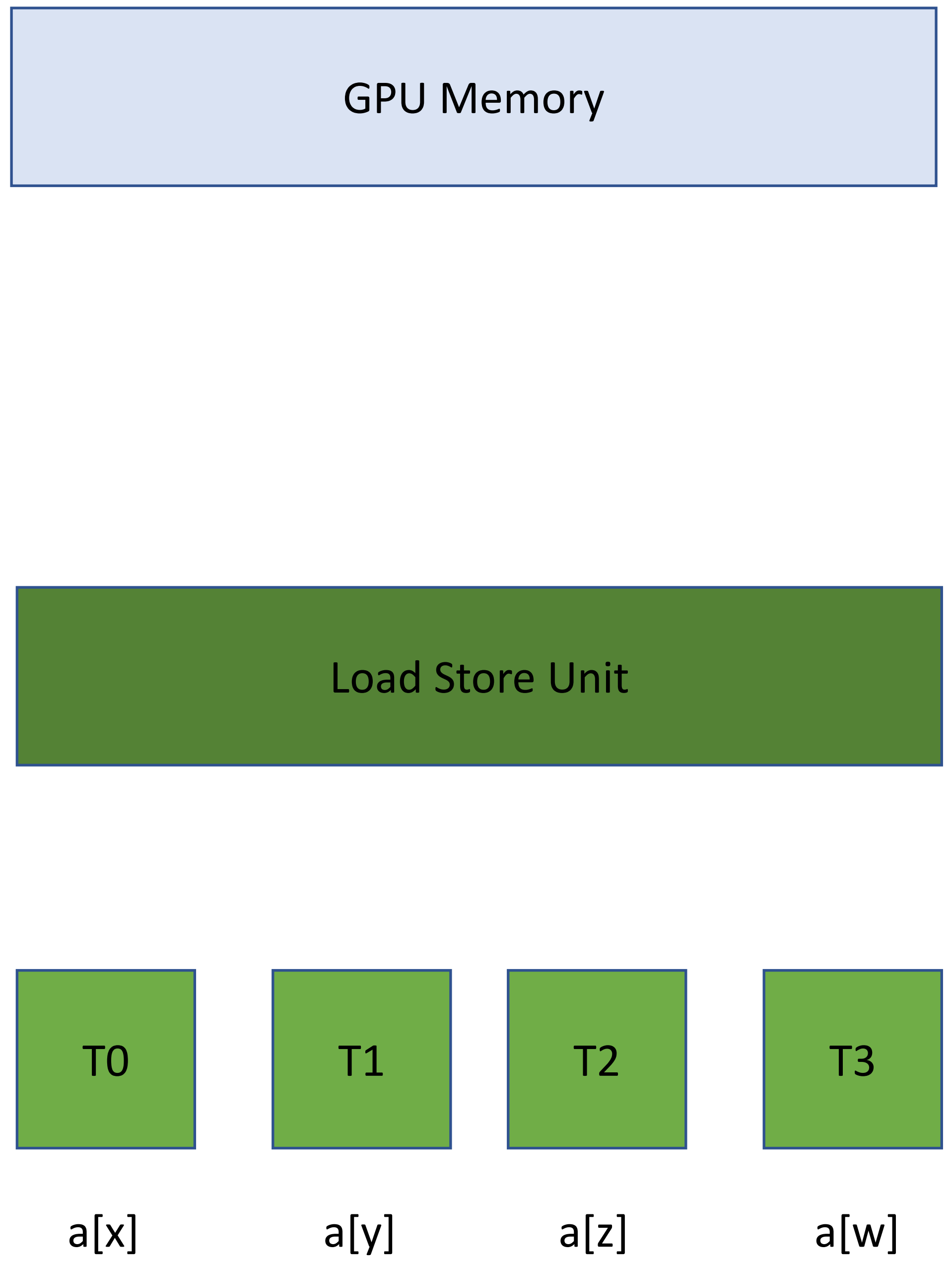
How can we fix this

4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

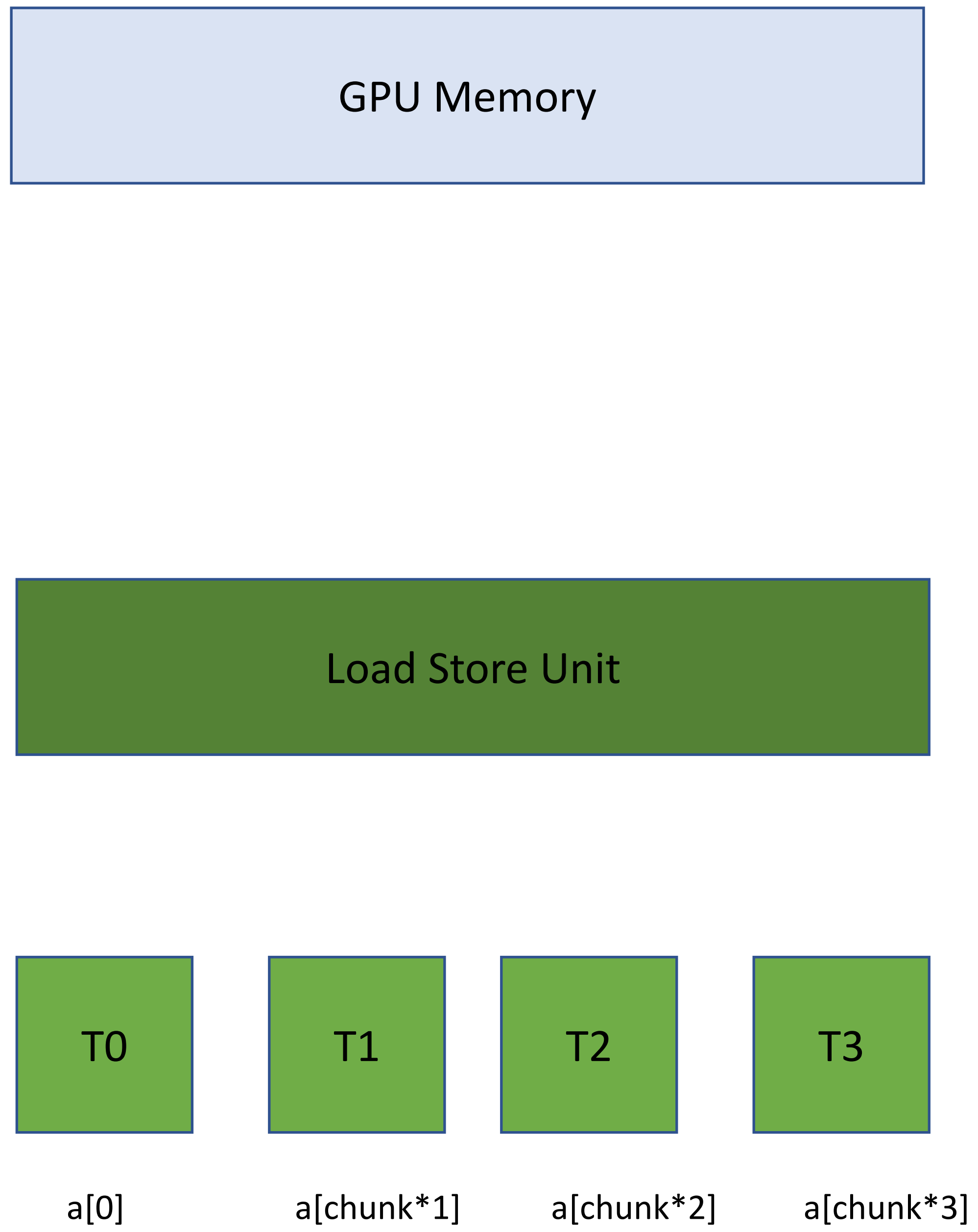


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

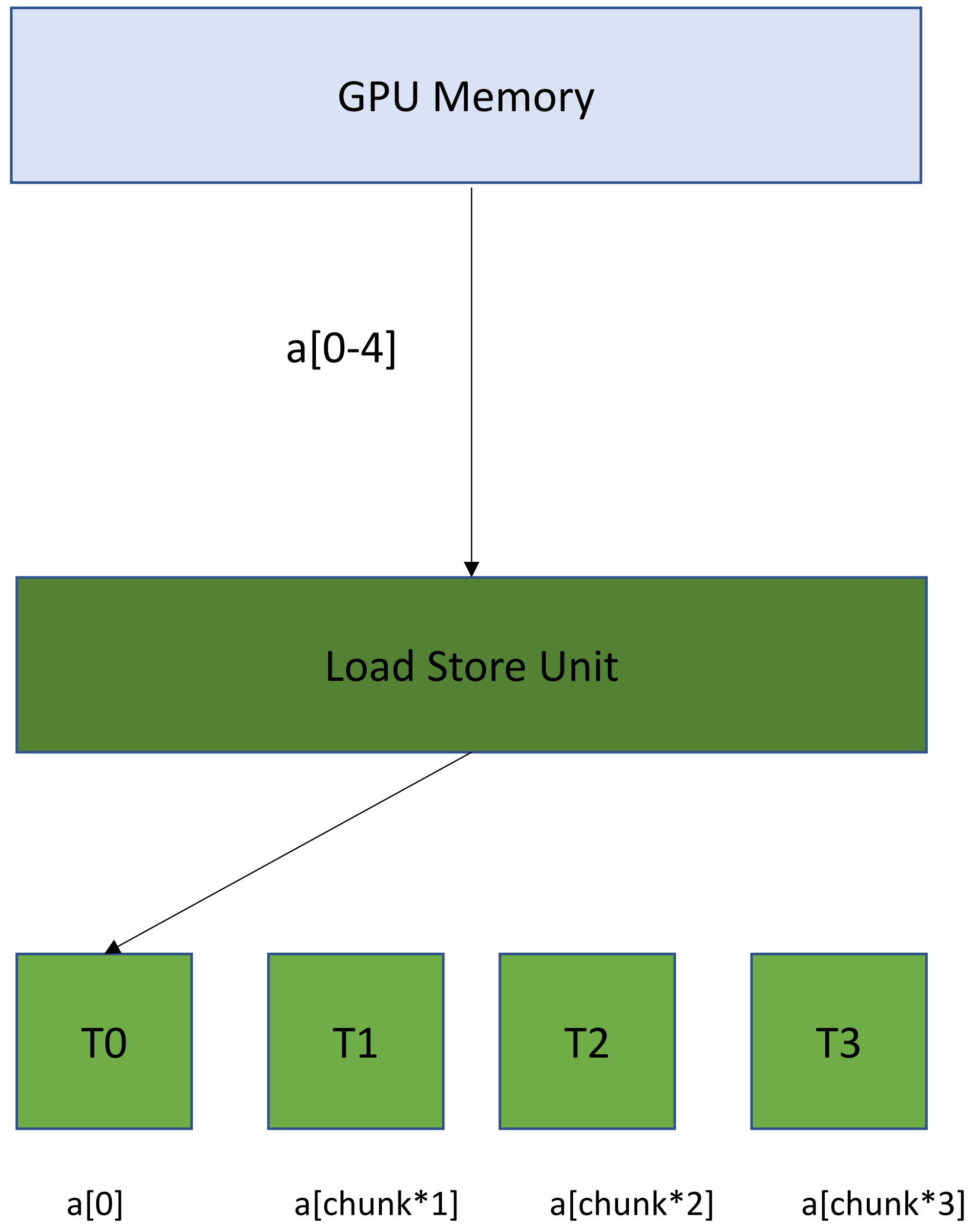


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

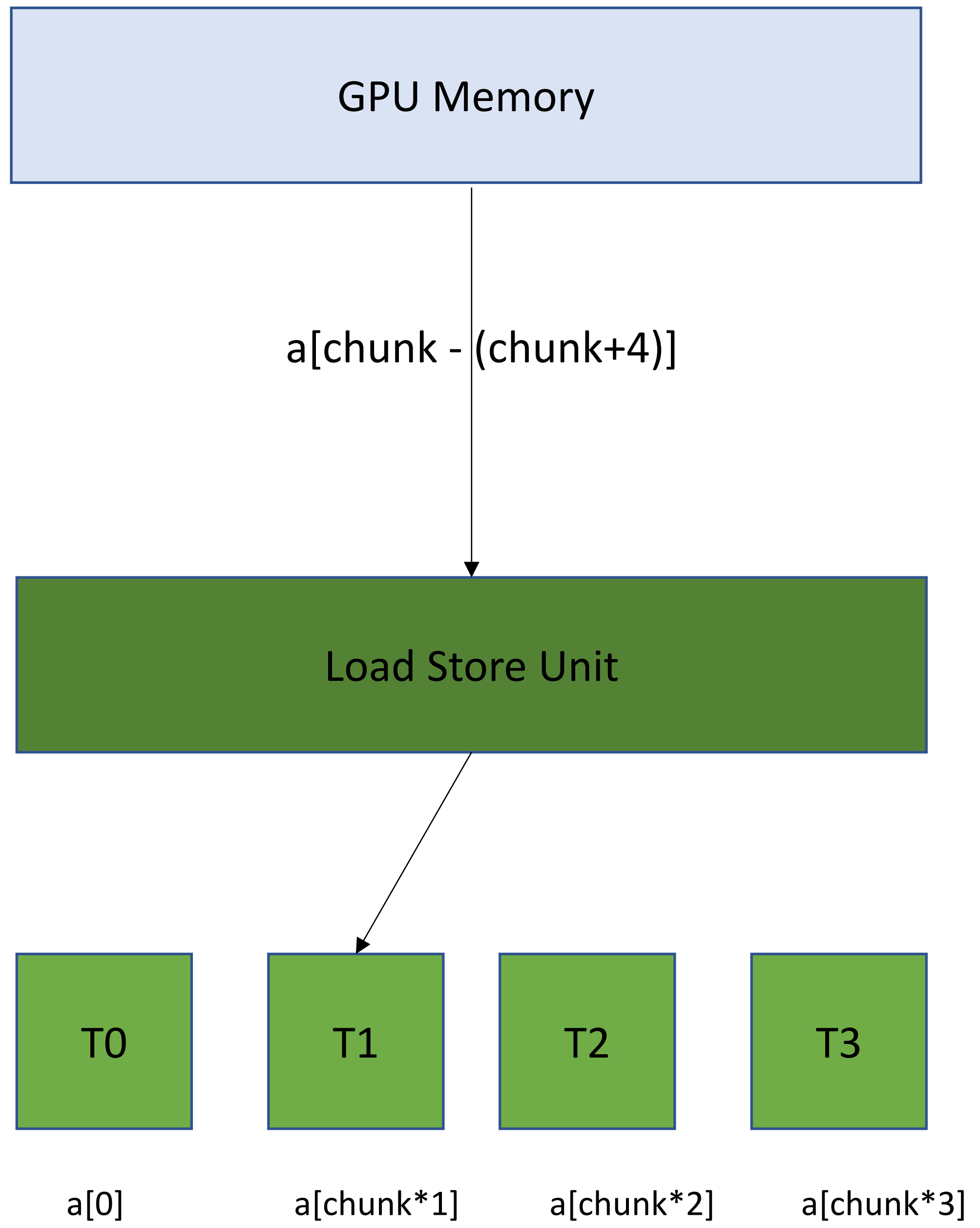


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

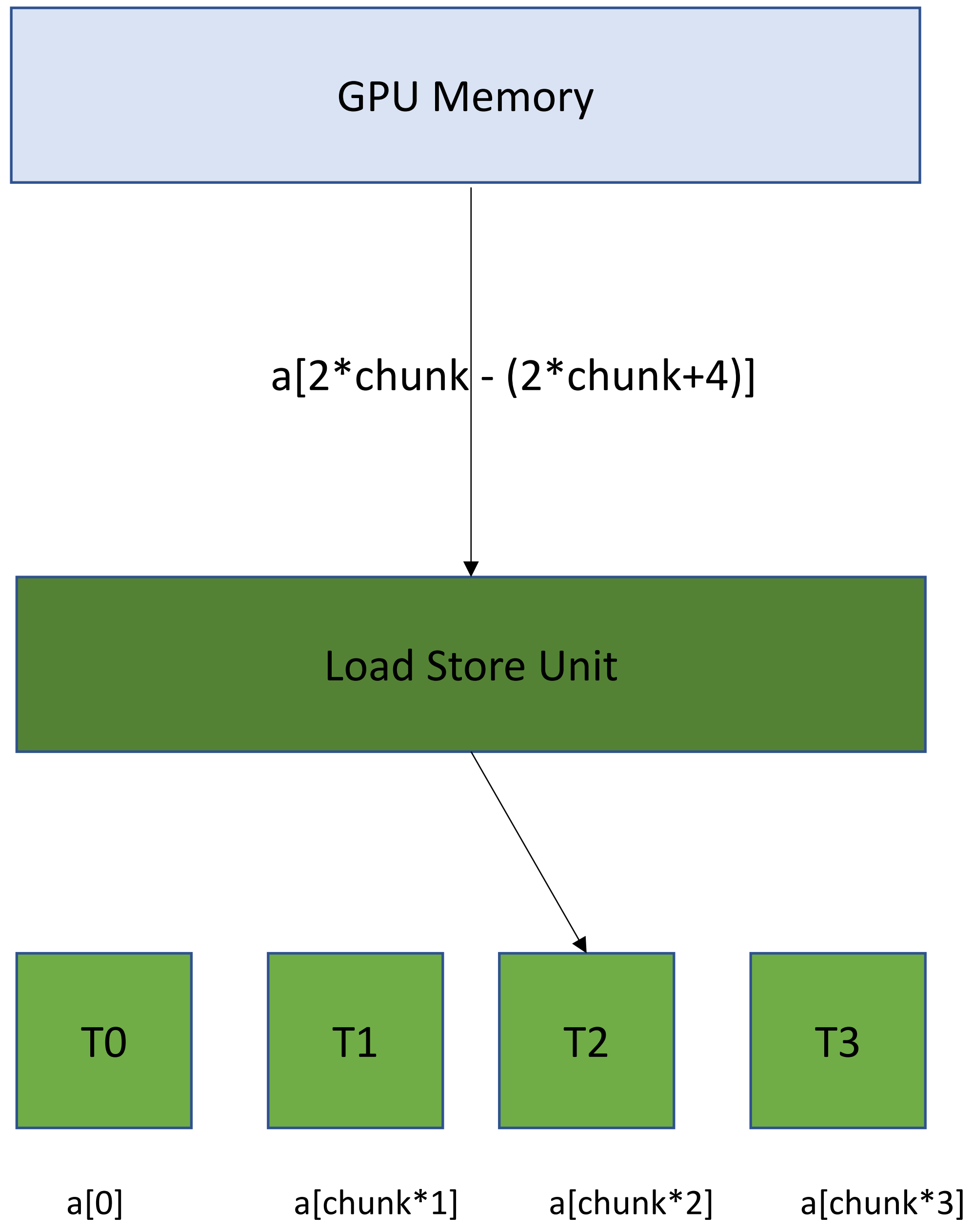


4 cores are accessing memory. What can happen

Read non-contiguous values

Not good!

Accesses are Serialized.
You need 4 requests to GPU memory

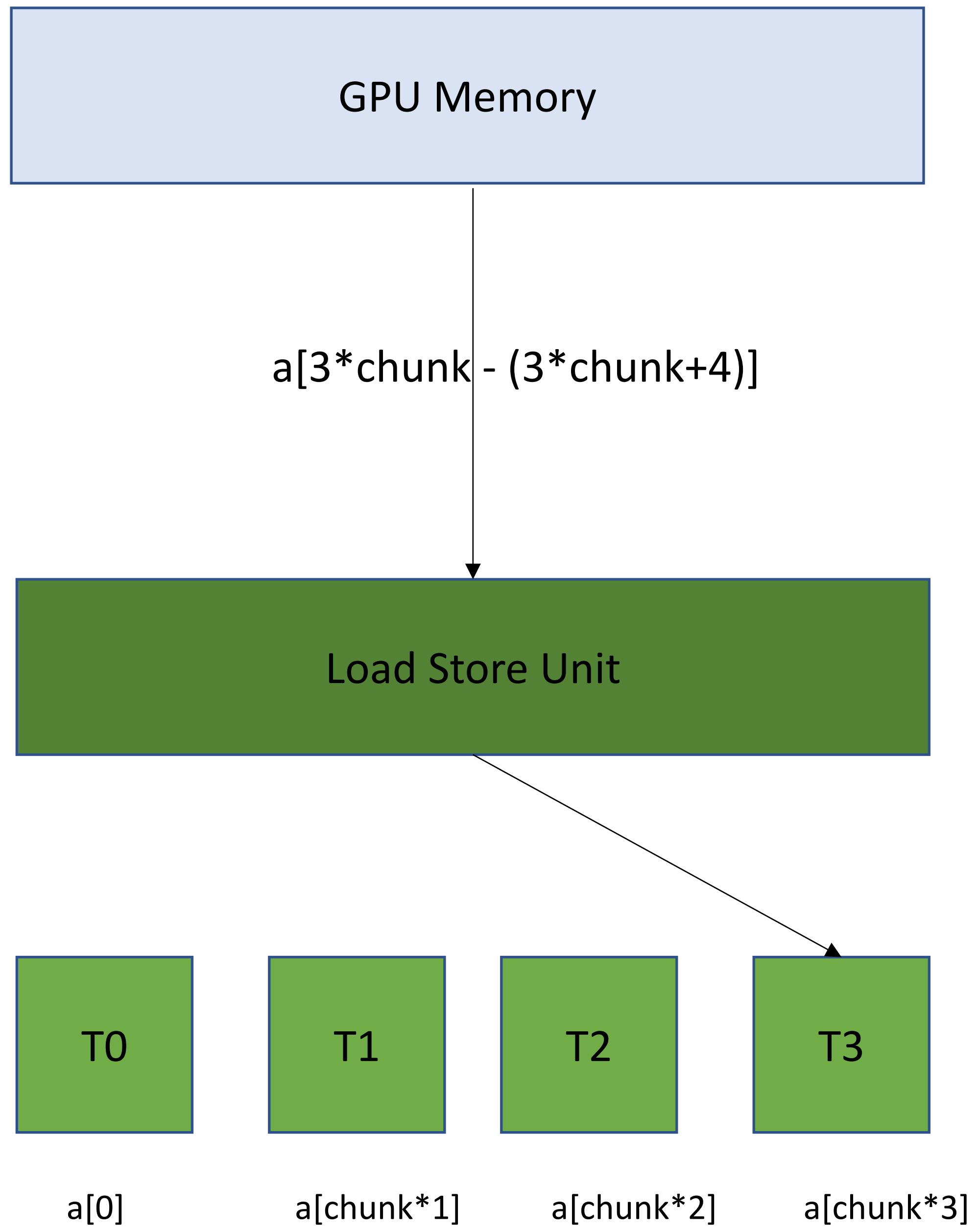


4 cores are accessing memory. What can happen

Read non-contiguous values

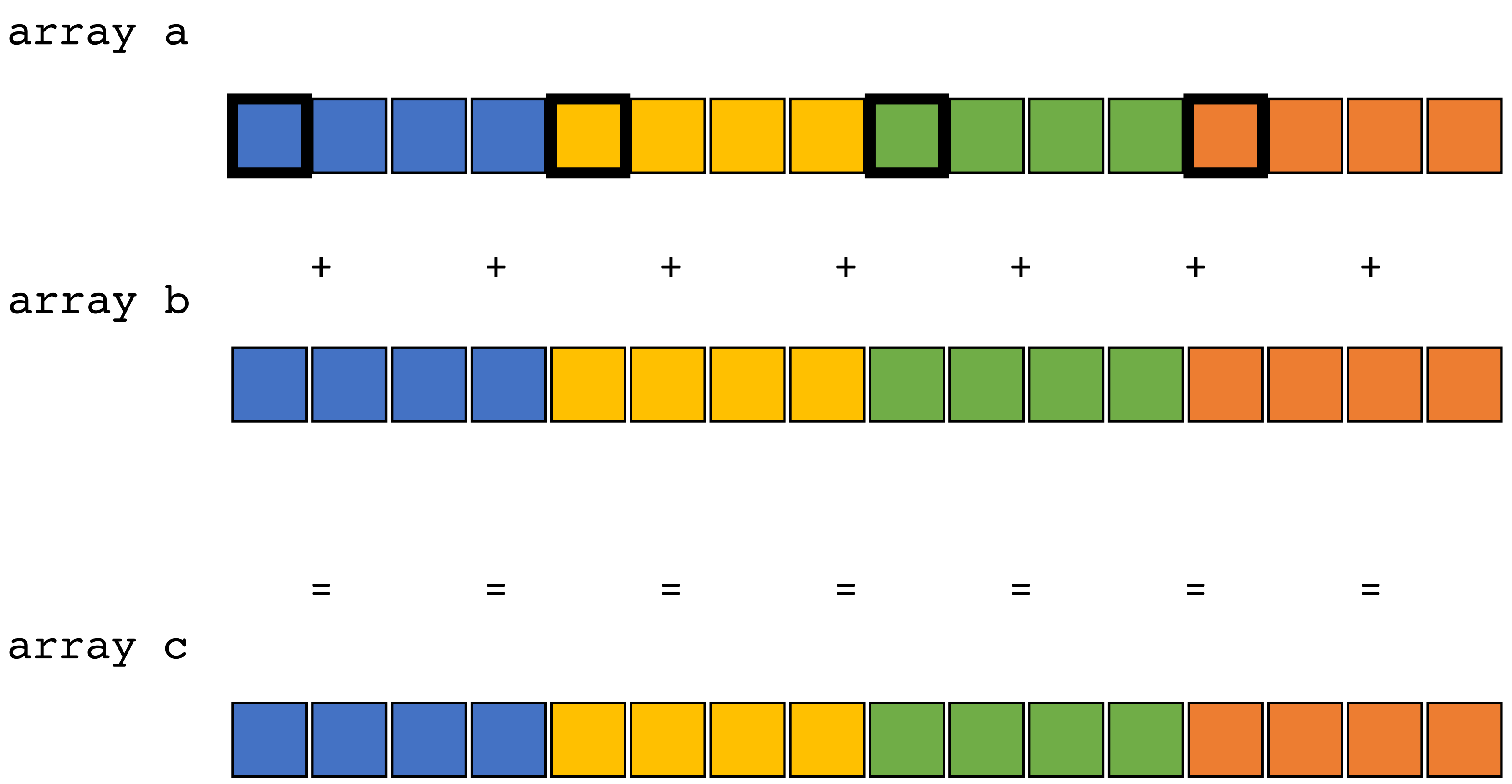
Not good!

Accesses are Serialized.
You need 4 requests to GPU memory



Chunked Pattern

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

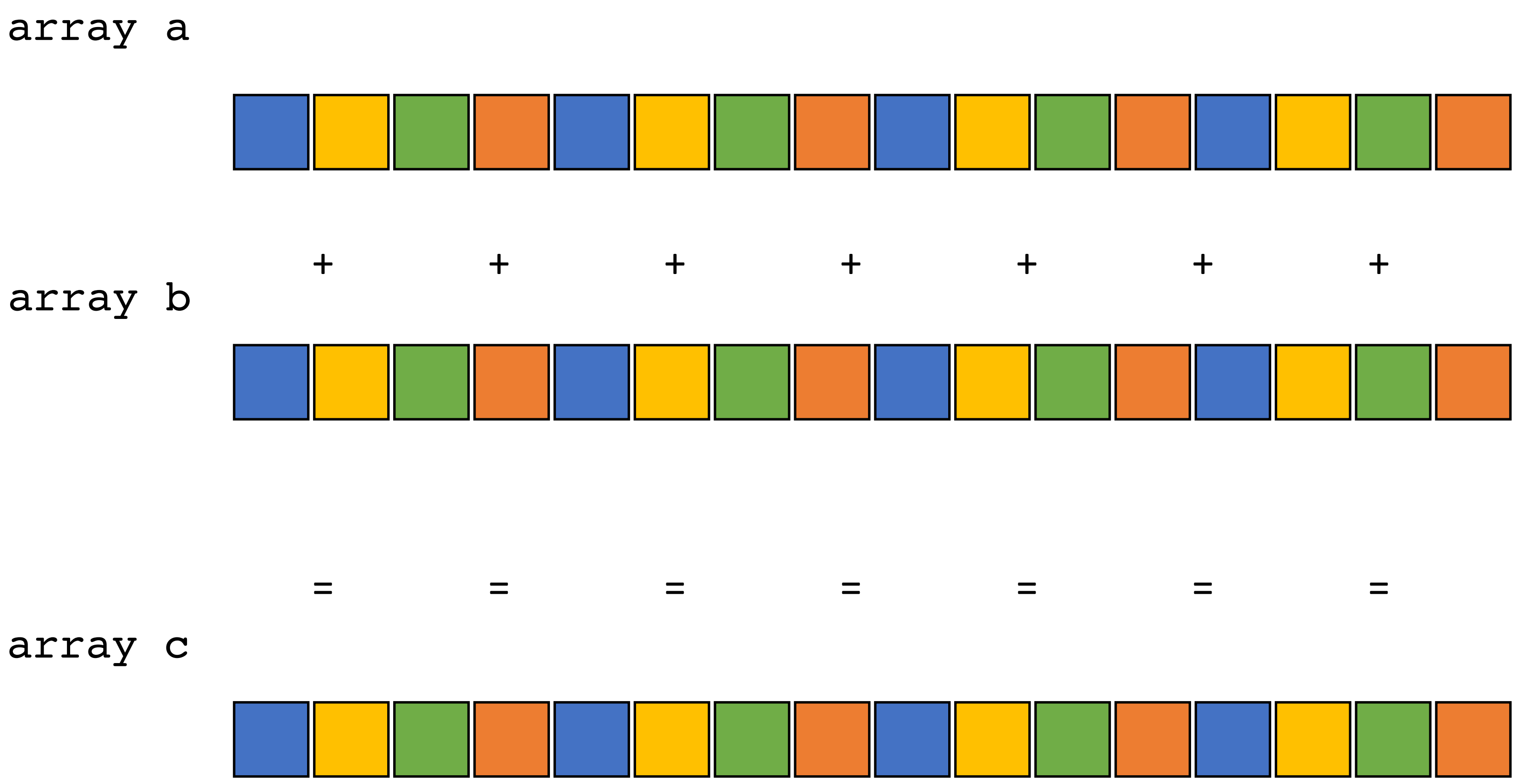


Computation can easily be divided into threads

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

How can we fix this

Stride Pattern



Computation can easily be divided into threads

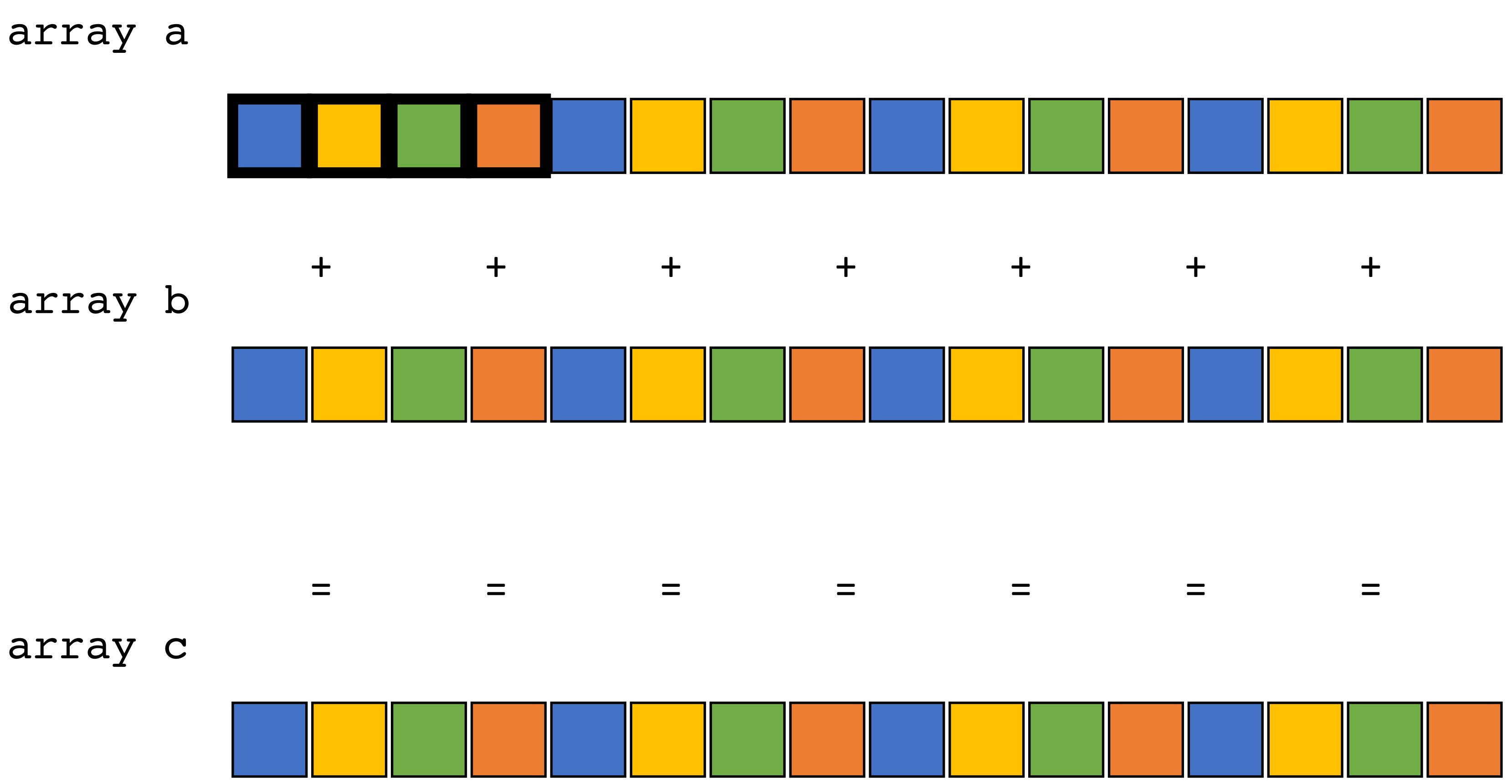
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

Stride Pattern

What sort of pattern is this?

Computation can easily be divided into threads

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



Go back to our program

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

calling the function

Lets change this to a stride pattern

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

Go back to our program

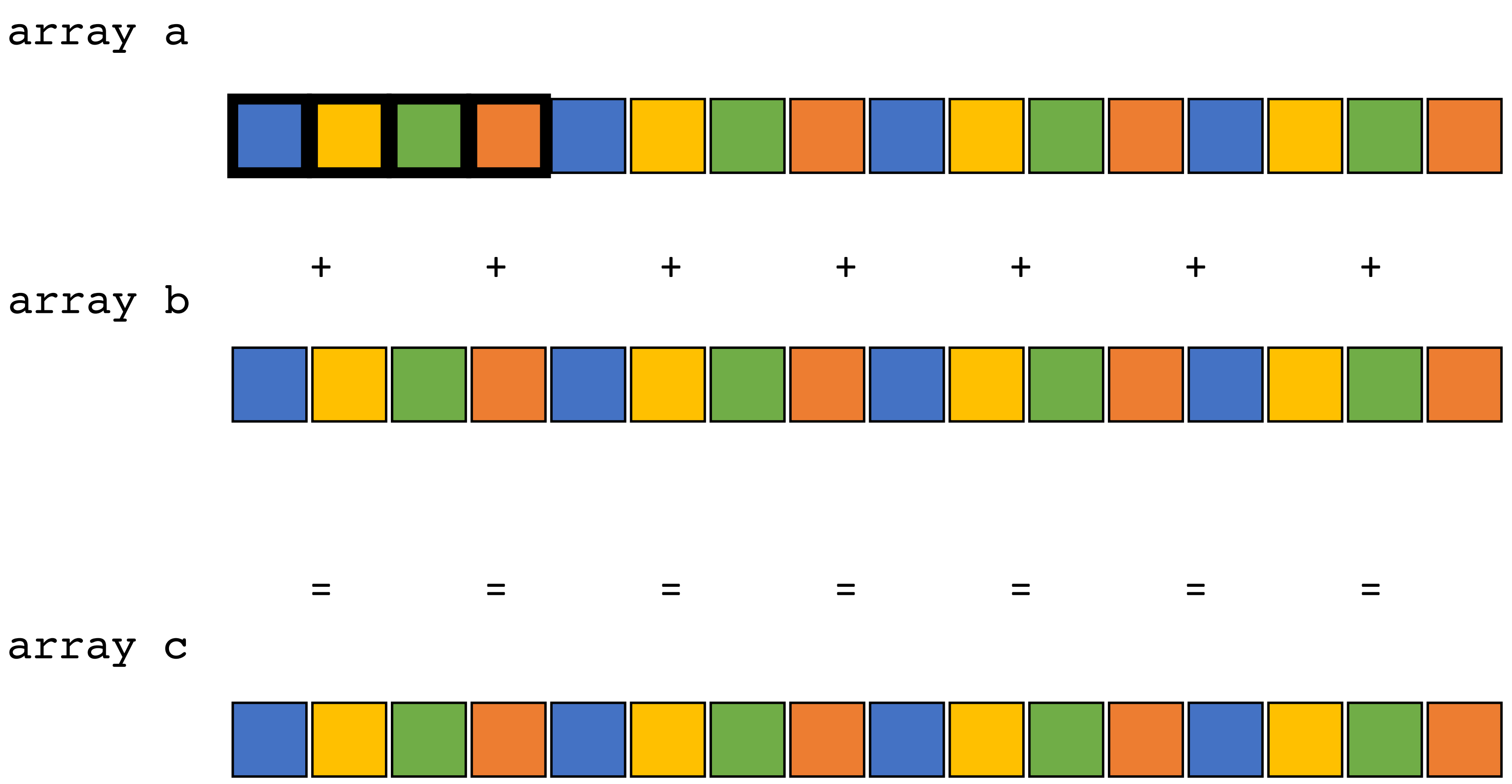
```
__global__ void vector_add(int * d_a, int * d_b, int * d_c, int size) {
  for (int i = threadIdx.x; i < size; i+=blockDim.x) {
     d_a[i] = d_b[i] + d_c[i];
  }
}
```

calling the function

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

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

Coalesced memory accesses

Lets try it! What do we think?

Coalesced memory accesses

Lets try it! What do we think? 😀

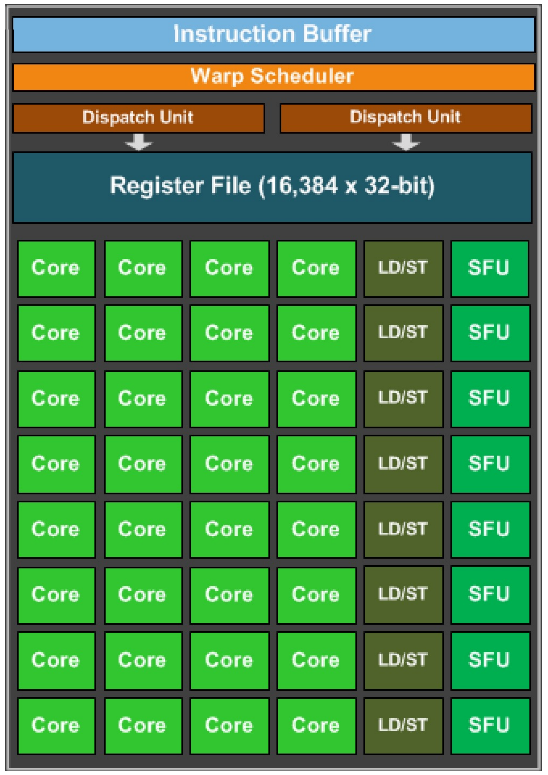
What else can we do?

Good time for a break

- 5 minute break

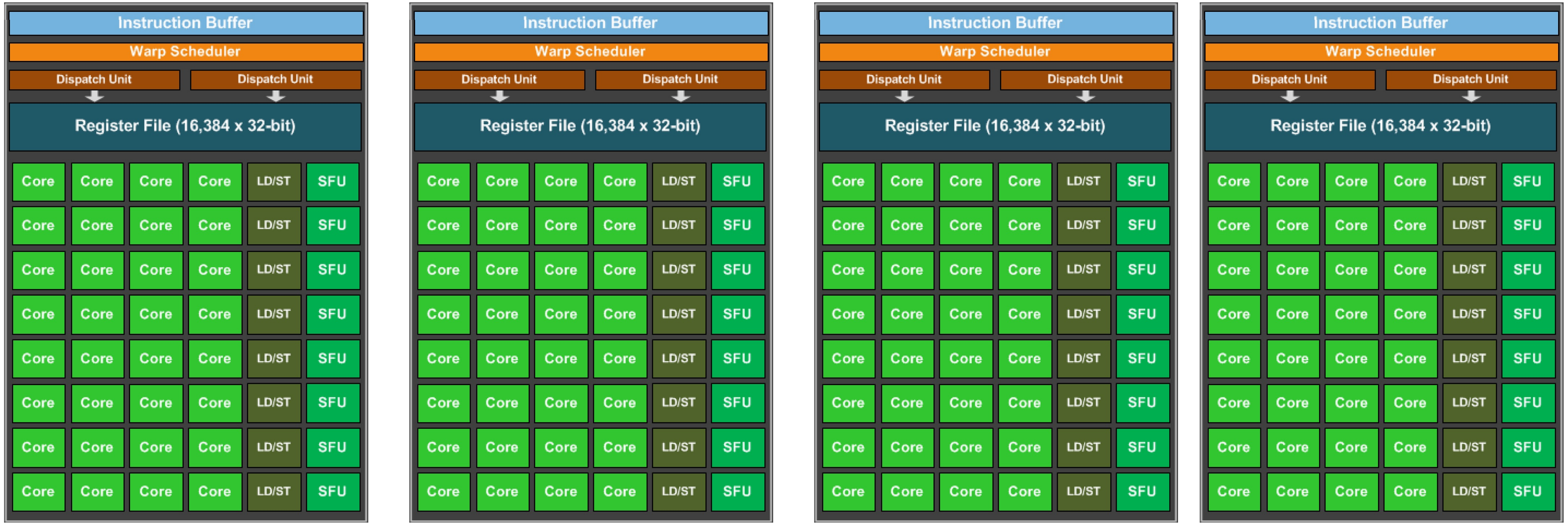
Multiple streaming multiprocessors

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



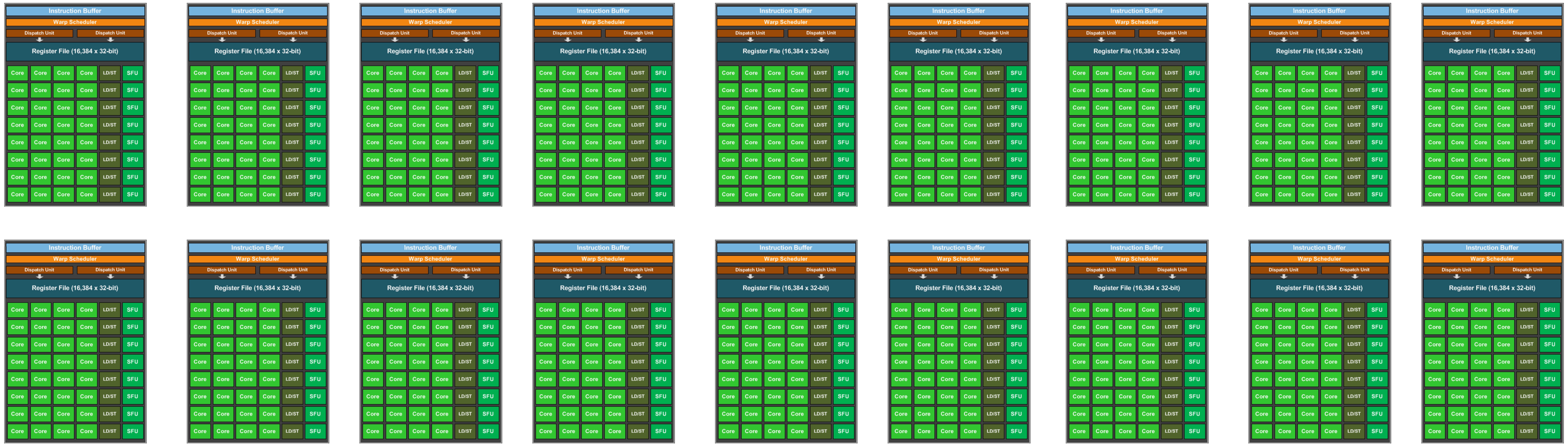
Multiple streaming multiprocessors

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



Multiple streaming multiprocessors

CUDA provides virtual streaming multiprocessors called **blocks**
Very efficient at launching and joining **blocks.**
practically 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];
  }
}
```

id within the block

calling the function

```
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];
  }
}
```

id within the block

threads per block

calling the function

```
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

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);
```

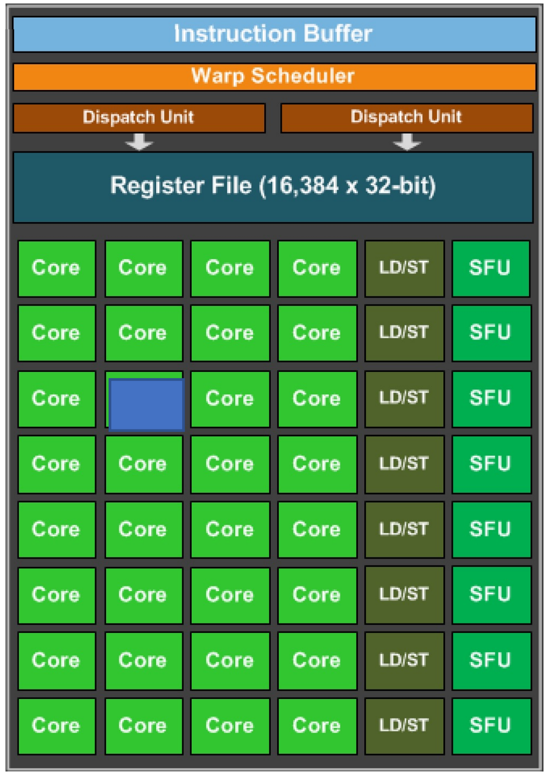
Now we have 1 thread for each element

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

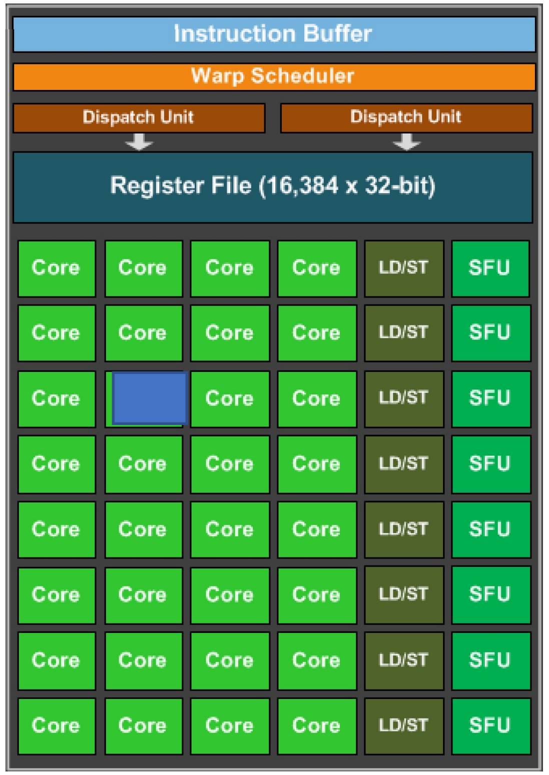
thread ids

threadIdx.x is thread id within a block

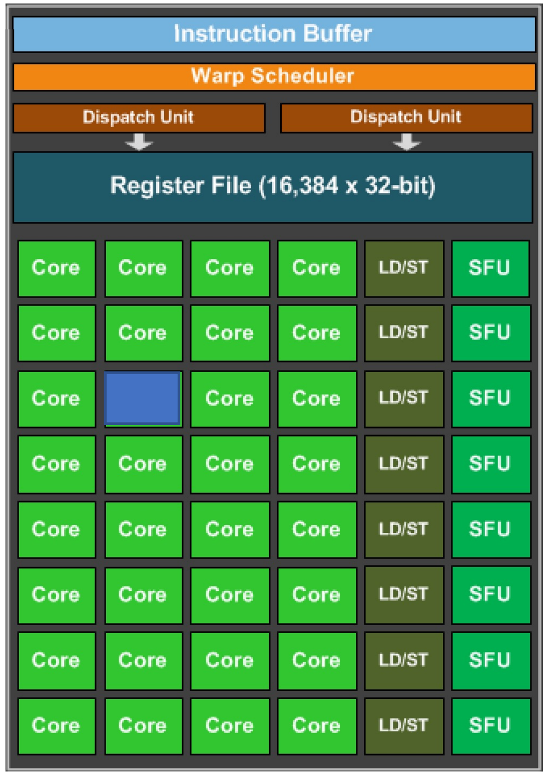
example threadIdx.x == 9



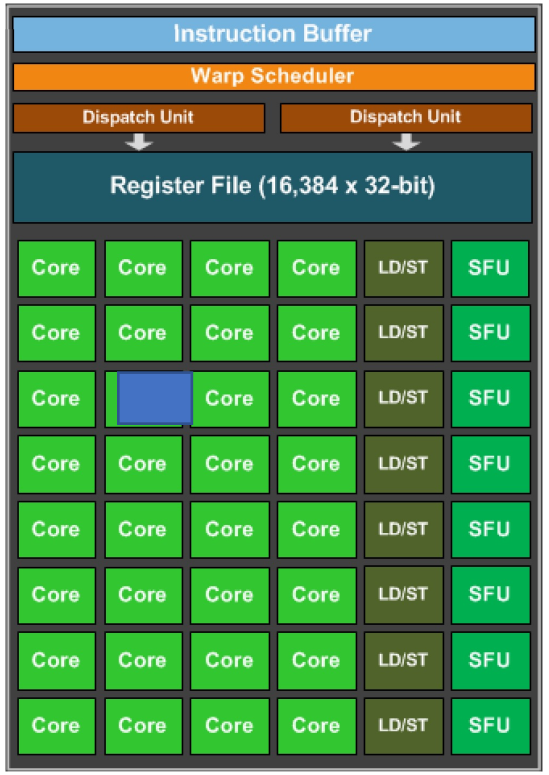
blockIdx.x == 0



blockIdx.x == 1



blockIdx.x == 2



blockIdx.x == 4

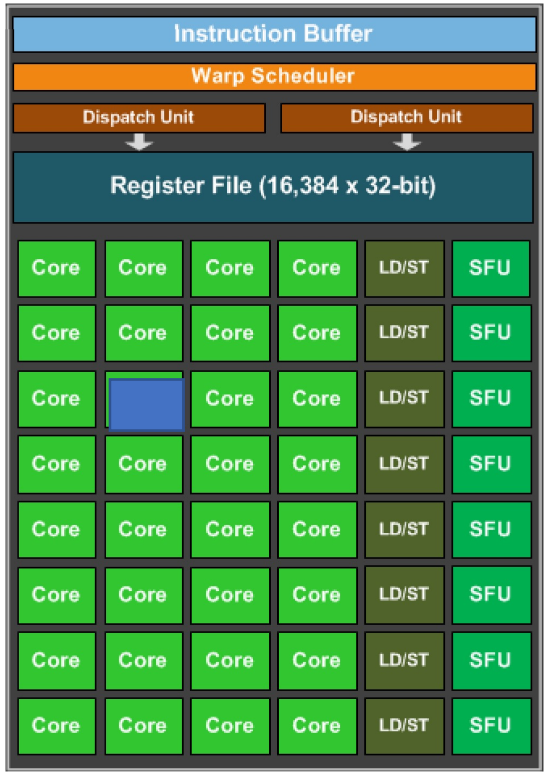
thread ids

How to get a unique id per thread?

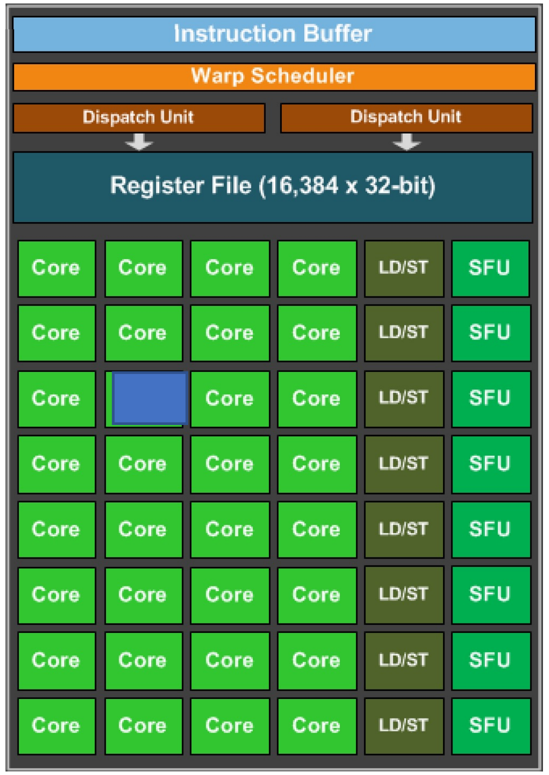
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

threadIdx.x is thread id within a block

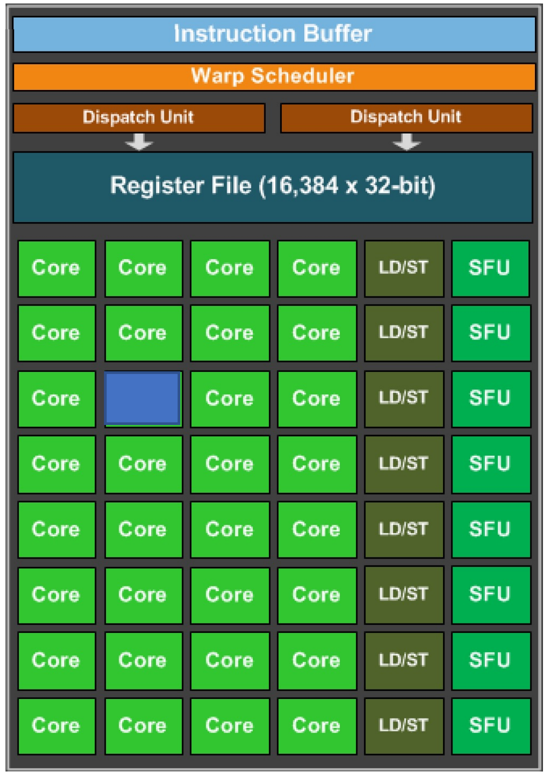
example threadIdx.x == 9



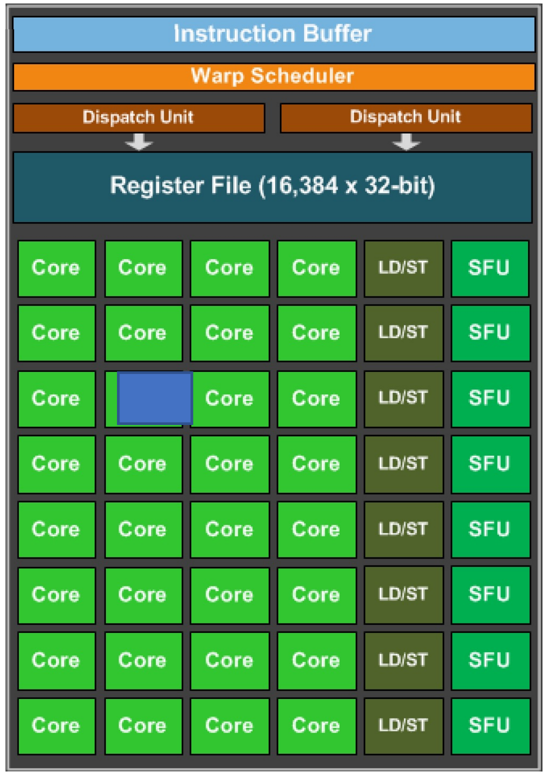
blockIdx.x == 0



blockIdx.x == 1



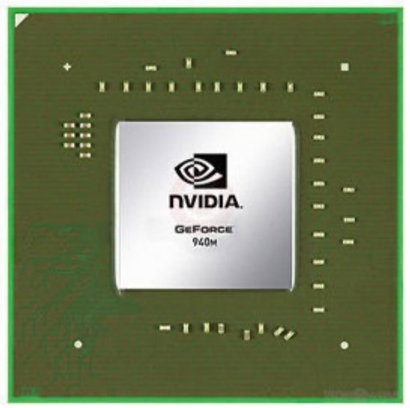
blockIdx.x == 2



blockIdx.x == 4

Final Round

The GPU in
my PhD laptop



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

Fight!

The CPU in
my professor
workstation

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

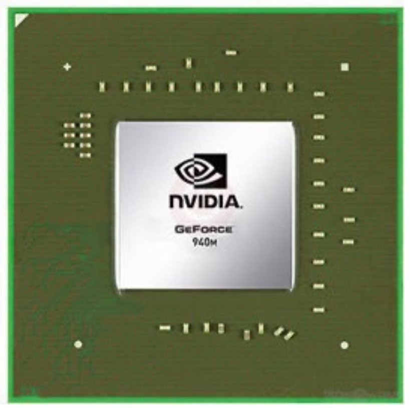
https://www.techpowerup.com/gpu-specs/geforce-940m.c2648
https://www.alibaba.com/product-detail/Intel-Core-i7-9700K-8-Cores_62512430487.html
https://www.prolast.com/prolast-elevated-boxing-rings-22-x-22/

Final Round

Nearly 4x faster!!

Fight!

The GPU in
my PhD laptop



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

The CPU in
my professor
workstation

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

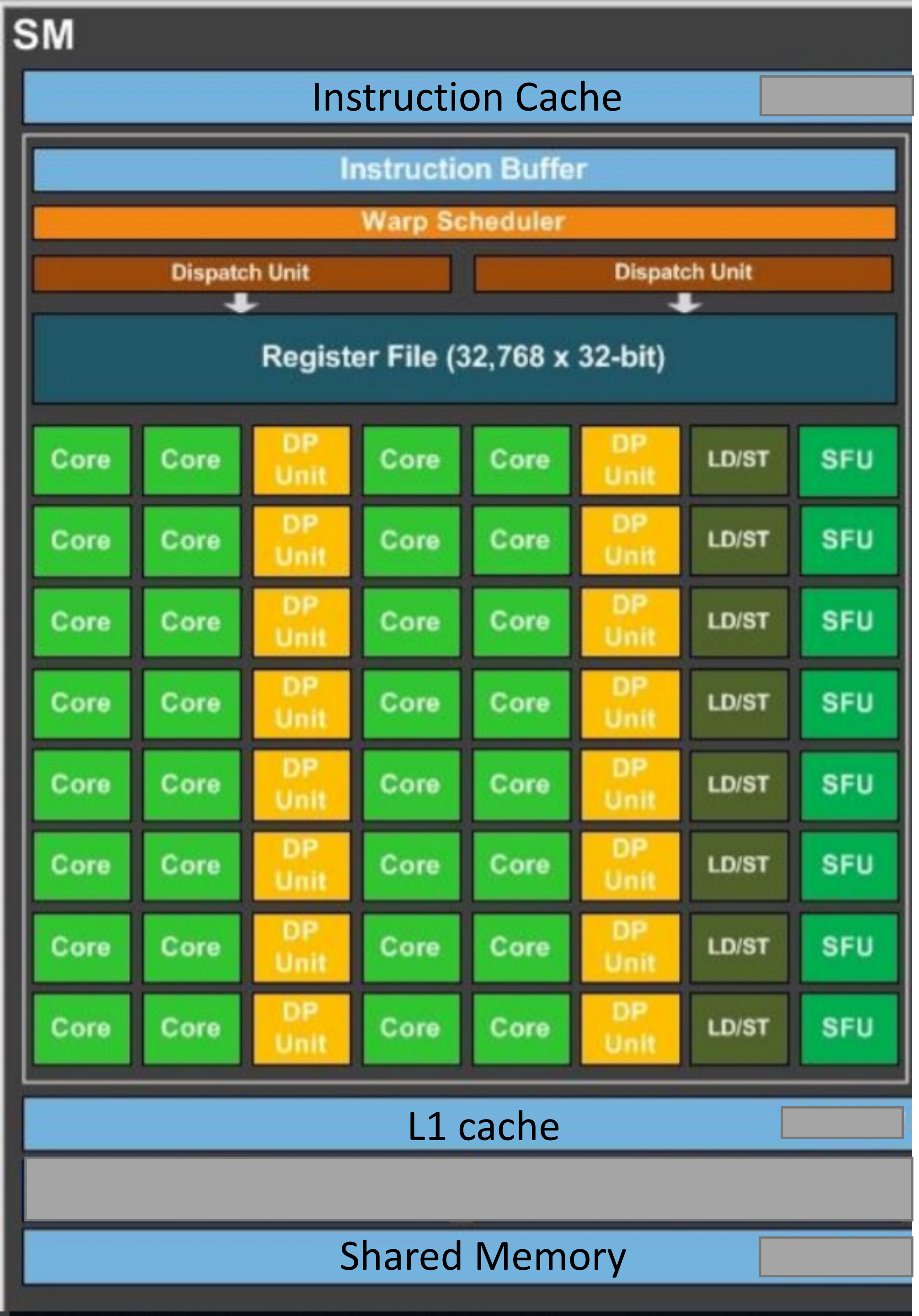
https://www.techpowerup.com/gpu-specs/geforce-940m.c2648
https://www.alibaba.com/product-detail/Intel-Core-i7-9700K-8-Cores_62512430487.html
https://www.prolast.com/prolast-elevated-boxing-rings-22-x-22/

Schedule

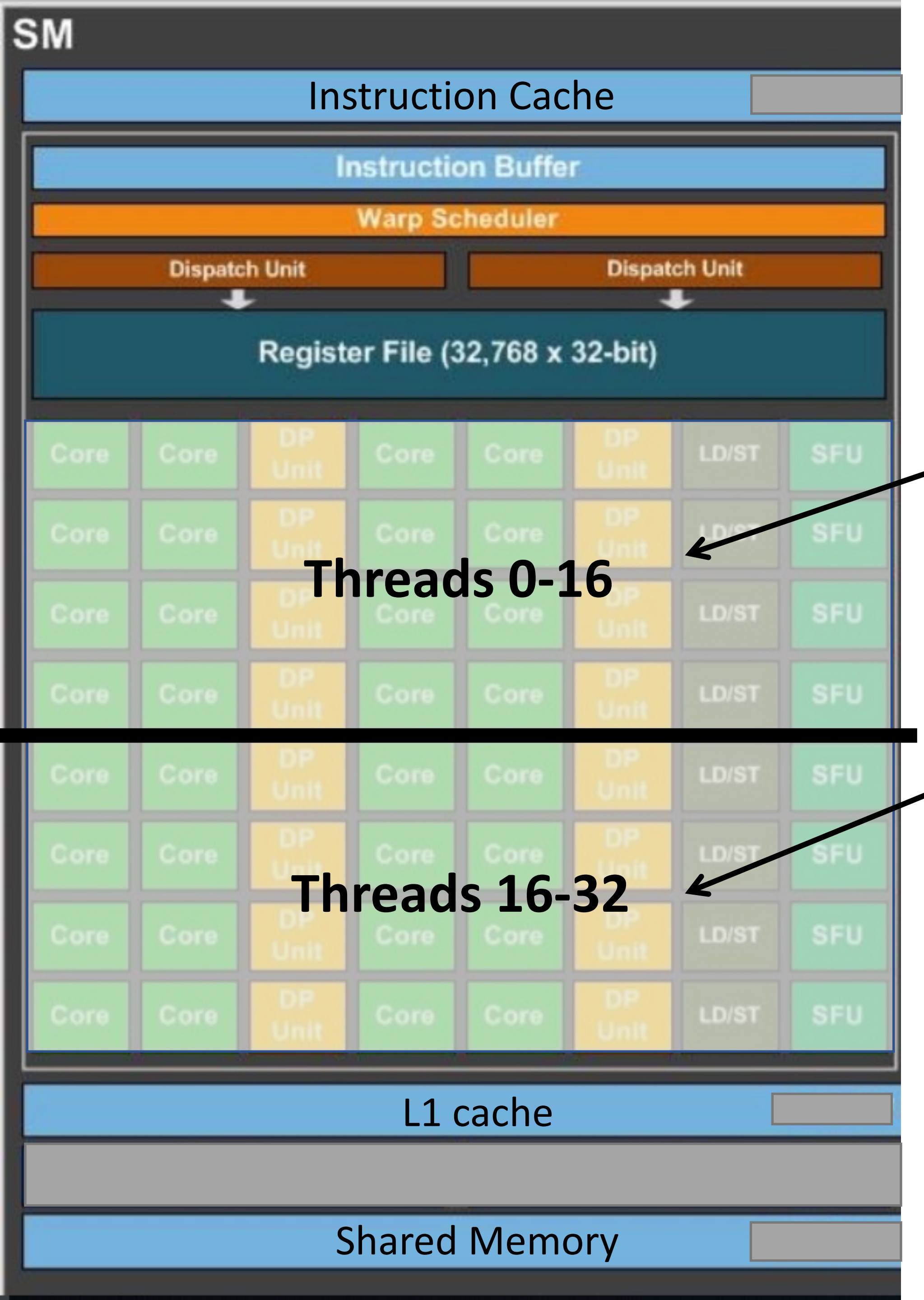
- Review previous optimizations
- New optimizations
- Advanced GPU topics

Schedule

- Review previous optimizations
- New optimizations
- **Advanced GPU topics**

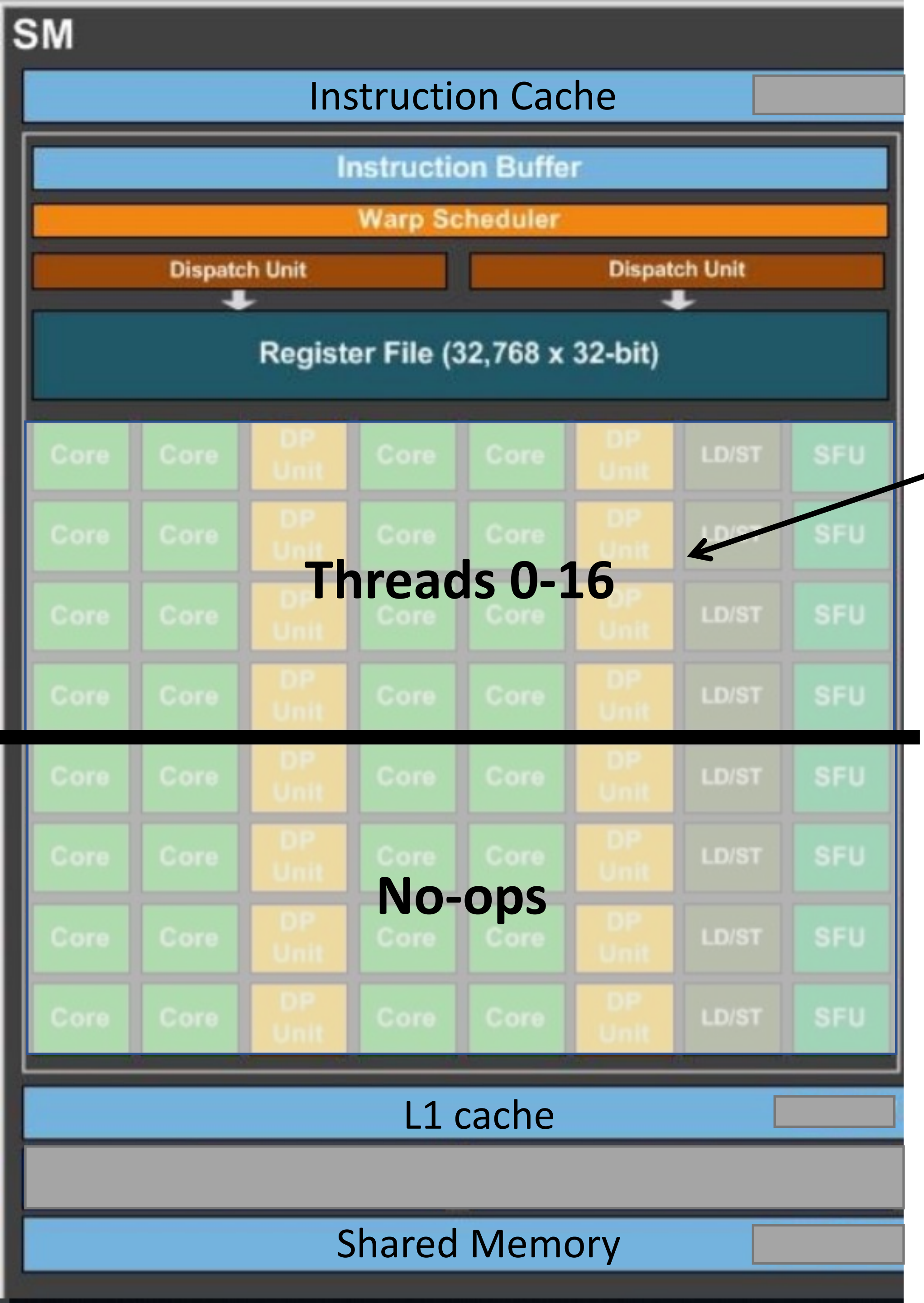


```
__global__
void diverged (...) {
  if (threadIdx.x < 16) {
    // Do some work
  }
  else {
    // Do something else
  }
}
```

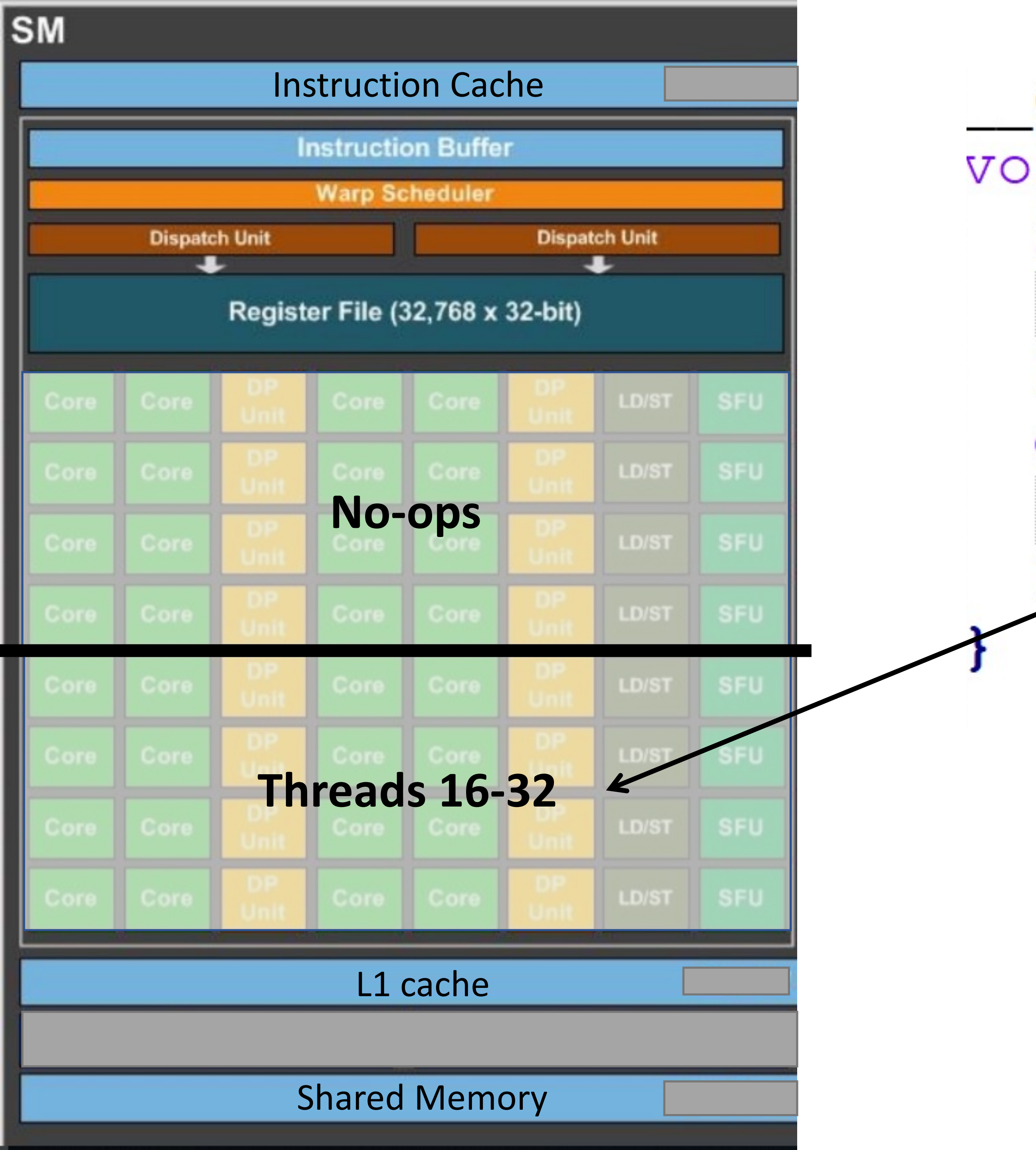


```
__global__
void diverged (...) {
  if (threadIdx.x < 16) {
    // Do some work
  }
  else {
    // Do something else
  }
}
```

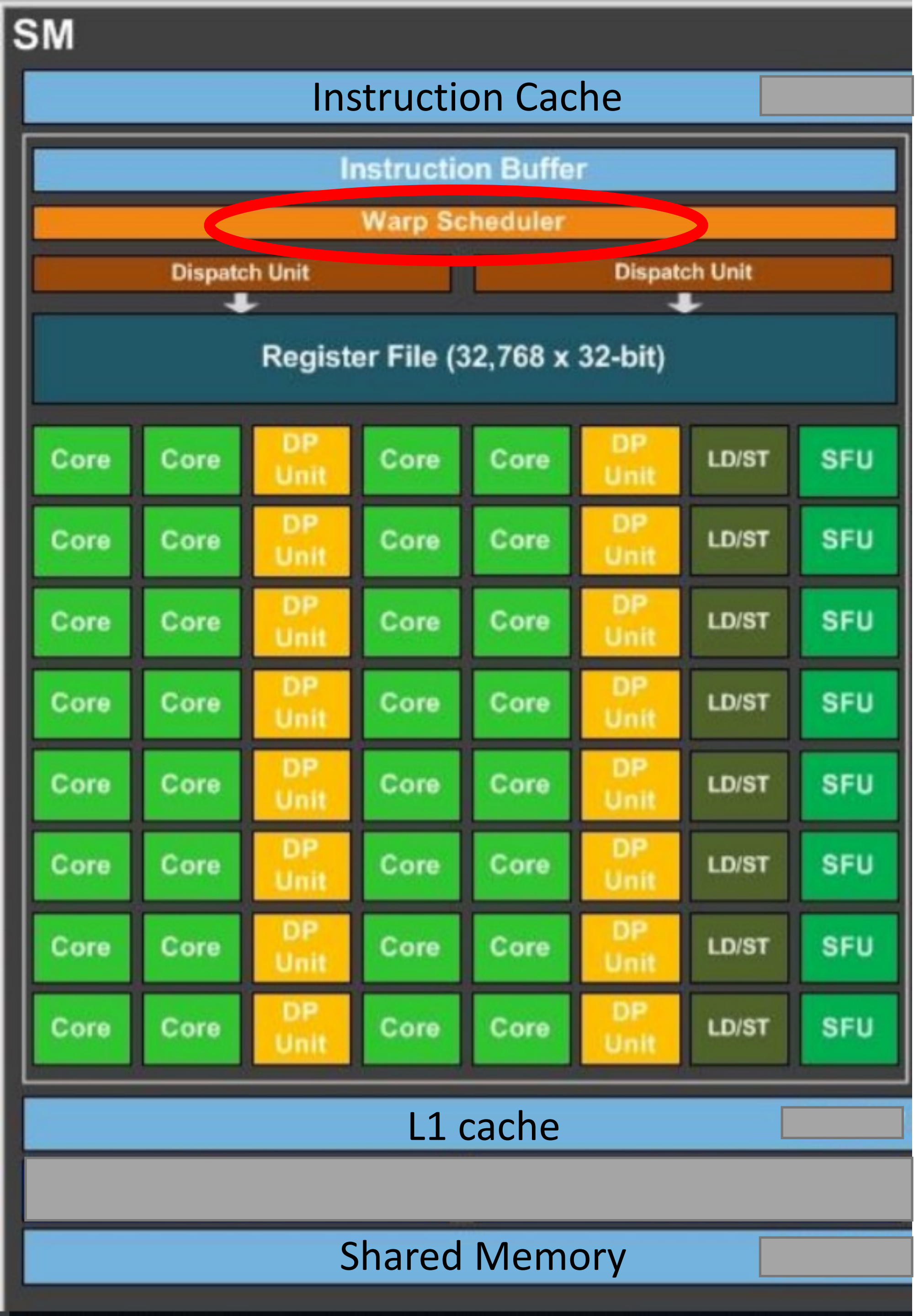
CANNOT execute them in parallel!



```
__global__
void diverged (...) {
  if (threadIdx.x < 16) {
    // Do some work
  }
  else {
    // Do something else
  }
}
```



```
__global__
void diverged (...) {
  if (threadIdx.x < 16) {
    // Do some work
  }
  else {
    // Do something else
  }
}
```



Warp scheduler keeps a bitmask for each warp

Example:

[0 1 1 0 …] -> execute thread 1 and 2, no-op the rest

How is the bit vector initialized?

Do "only" need a bit vector?

Pretend we have a warp size of 16

Bit vector stack for warp 0

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Pretend we have a warp size of 16

Bit vector stack for warp 0

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

[1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0]
```

```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Push on the stack for **if**

Pretend we have a warp size of 16

```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Bit vector stack for warp 0

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

[1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0]

[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
```

Pretend we have a warp size of 16

Bit vector stack for warp 0

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

[1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0]
```

```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Pop the stack

Pretend we have a warp size of 16

Bit vector stack for warp 0

```
[1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]

[0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1]
```

```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Complement the top item on the stack for **else**

Pretend we have a warp size of 16

Bit vector stack for warp 0

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

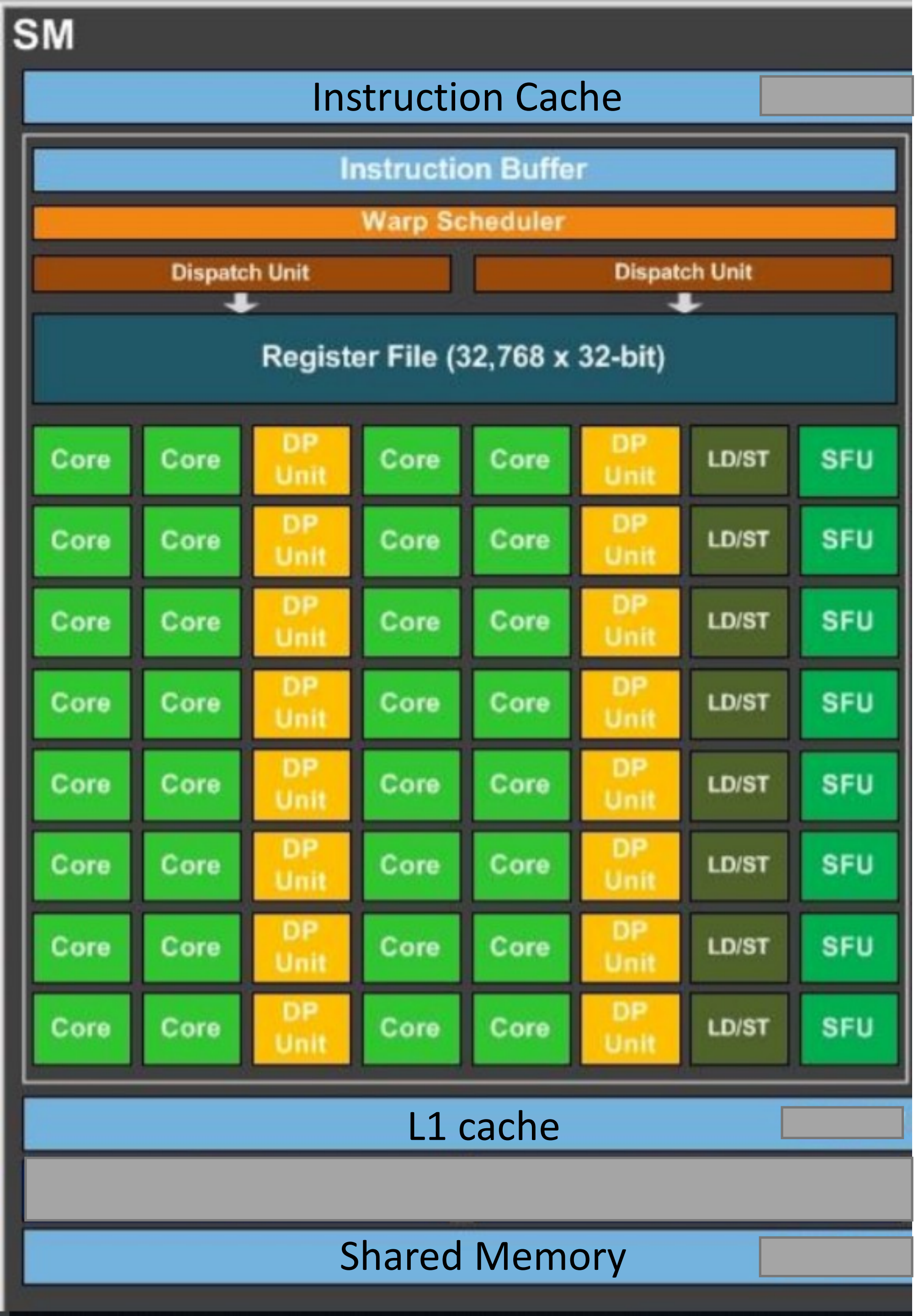
```
__global__
void diverged2 (...) {
  if (threadIdx.x < 8) {
    // Work 0
    if (threadIdx.x == 3) {
      // Work 1
    }
    // Work 2
  }
  else {
    // Work 3
  }
}
```

Pop

What about this?

```
__global__
void waiting(...) {
  if (threadIdx.x == 0){
    while (global_flag != 1); //Wait for thread 1 to set a flag
  }
  if (thread Idx.x == 1) {
    *global_flag = 1;
  }
}
```

Synchronization and Shared Memory

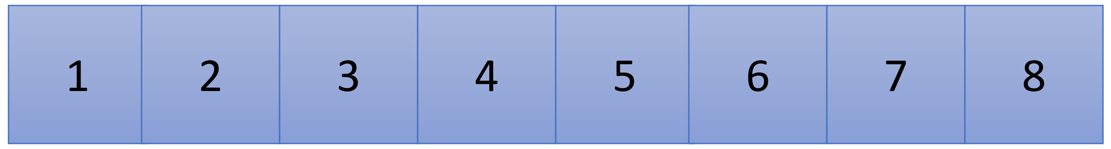


Synchronization and Shared Memory

- Last part of SM architecture to talk about!
- What if program requires thread communication?
- Threads in same warp execute “lock-step”
- BUT threads in different warps can de-sync (execute different instructions)

Case study: Reduction

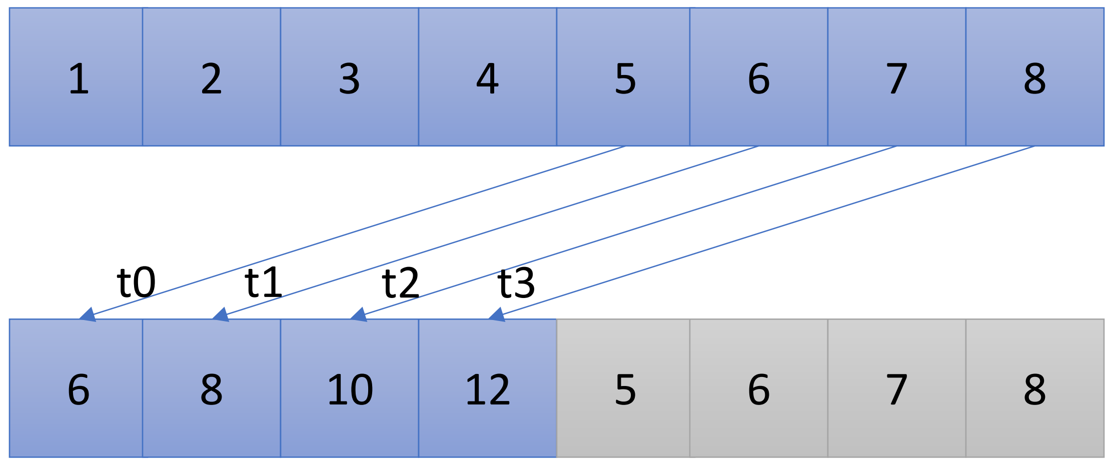
- Vector reduction: Sum up elements in array



= 36

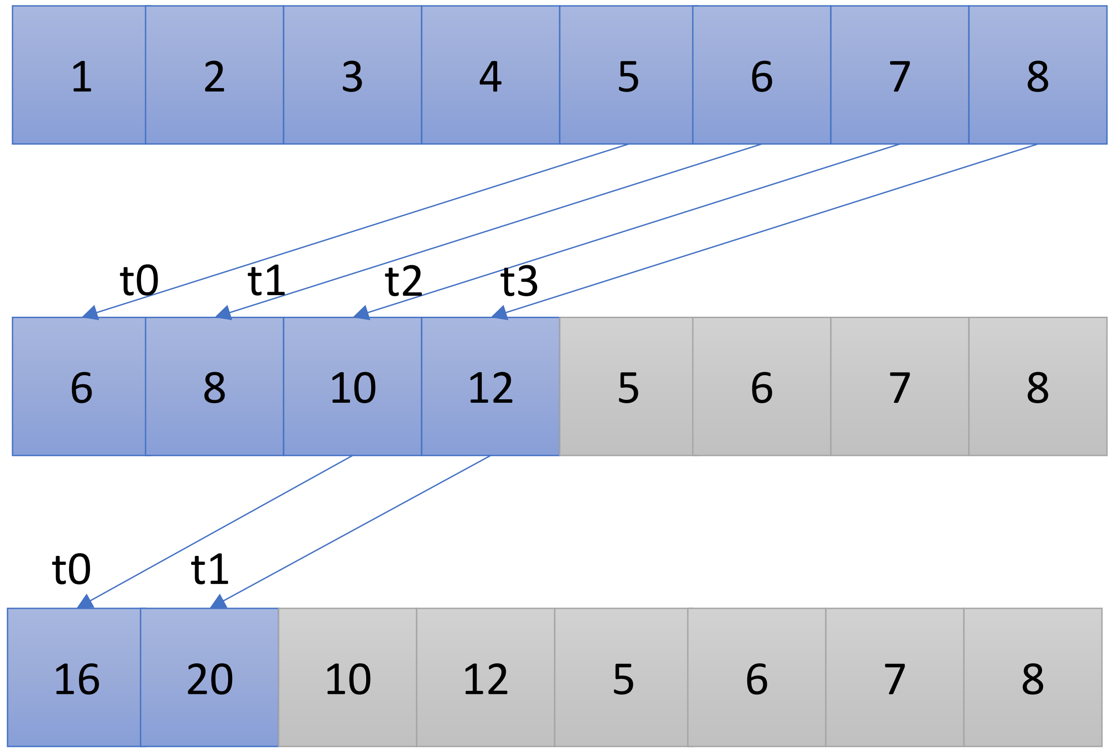
- Parallel algorithm? Consider 4 threads

Case study: Reduction



Reduce to 4 elements by striding

Case study: Reduction

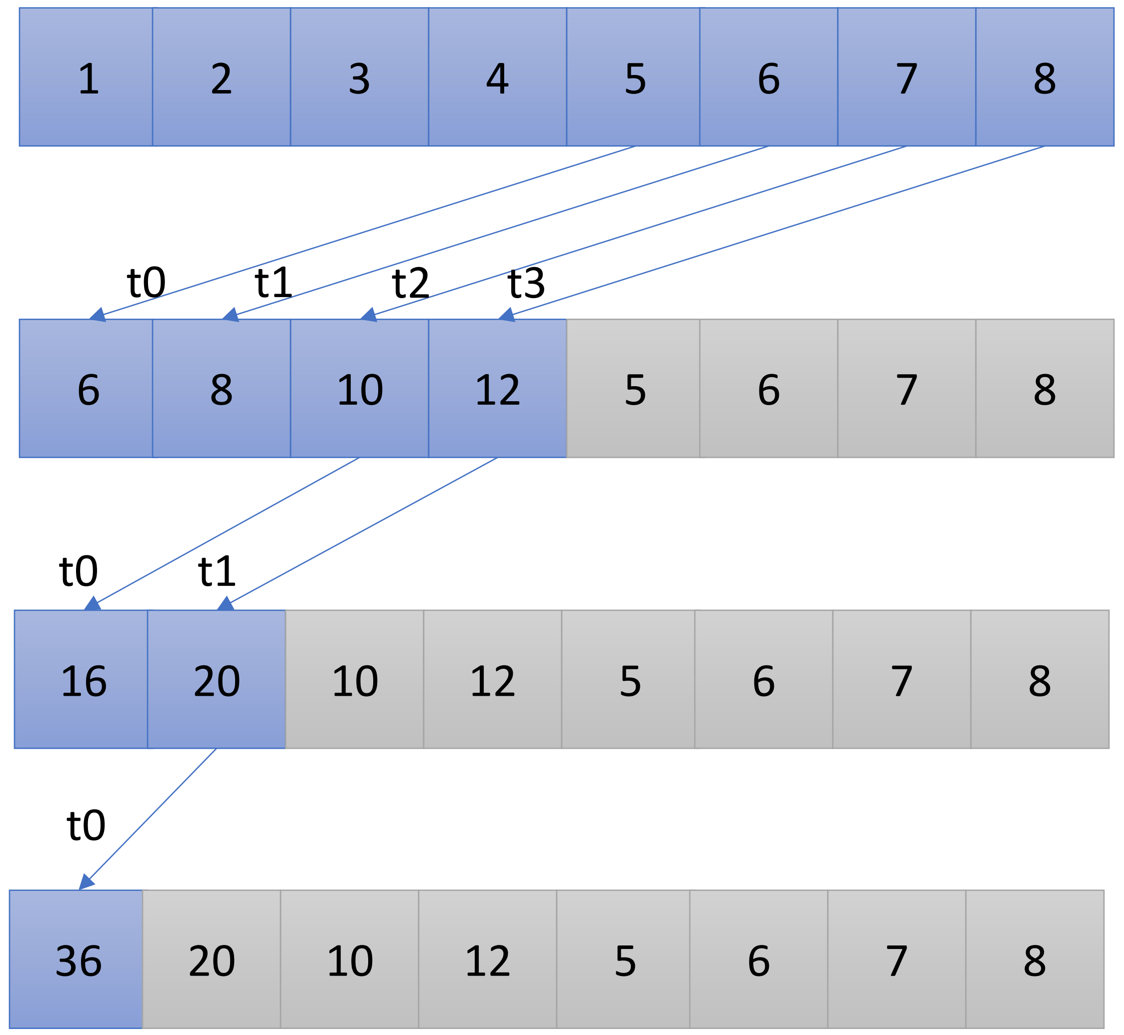


Reduce to 4 elements by striding

Use half as many threads to compute the next round

Case study: Reduction

Requires synchronization between levels!



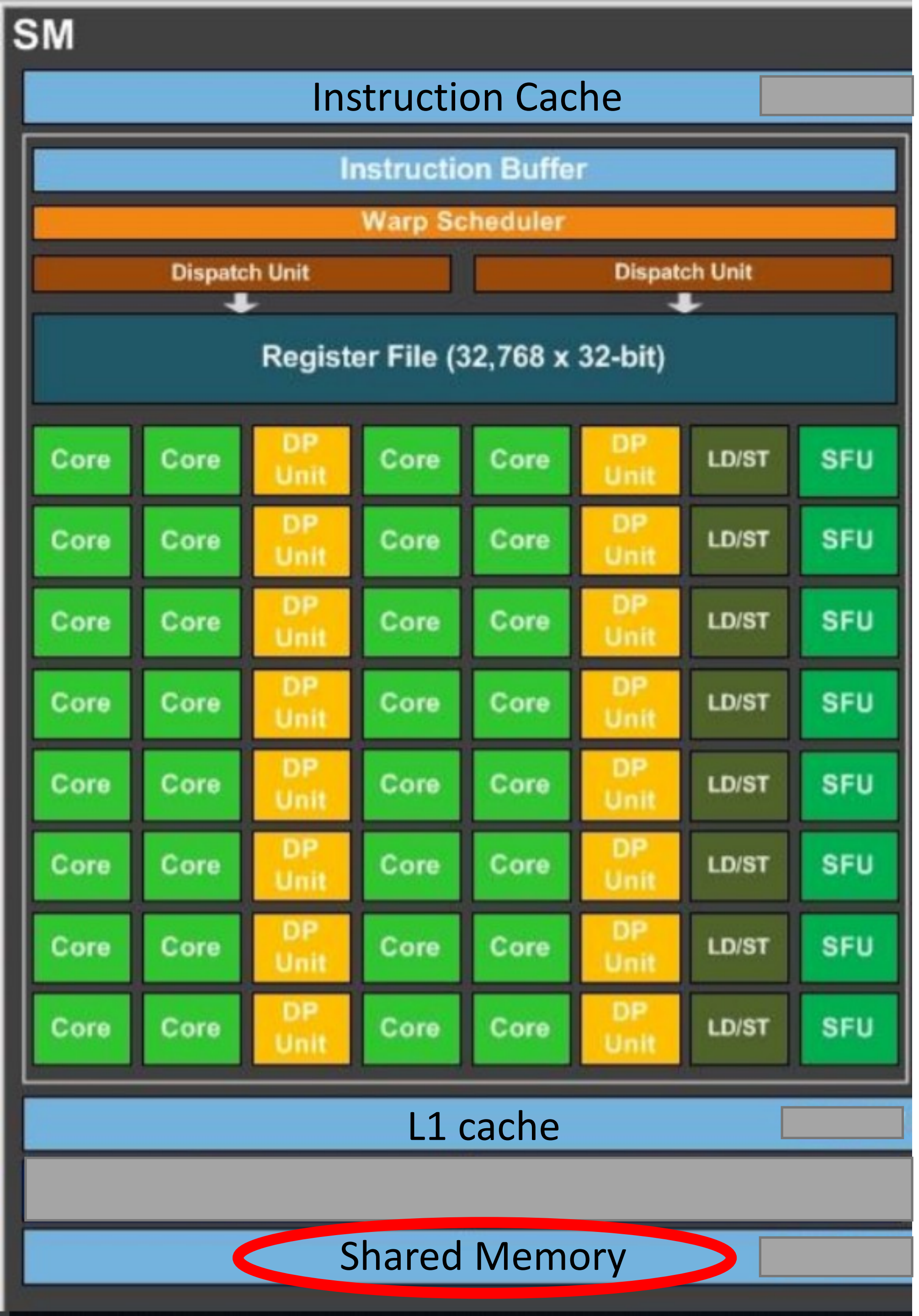
Reduce to 4 elements by striding

Use half as many threads to compute the next round

half as many again, until single result is reached

Demo

- Barrier



Shared memory (if time)

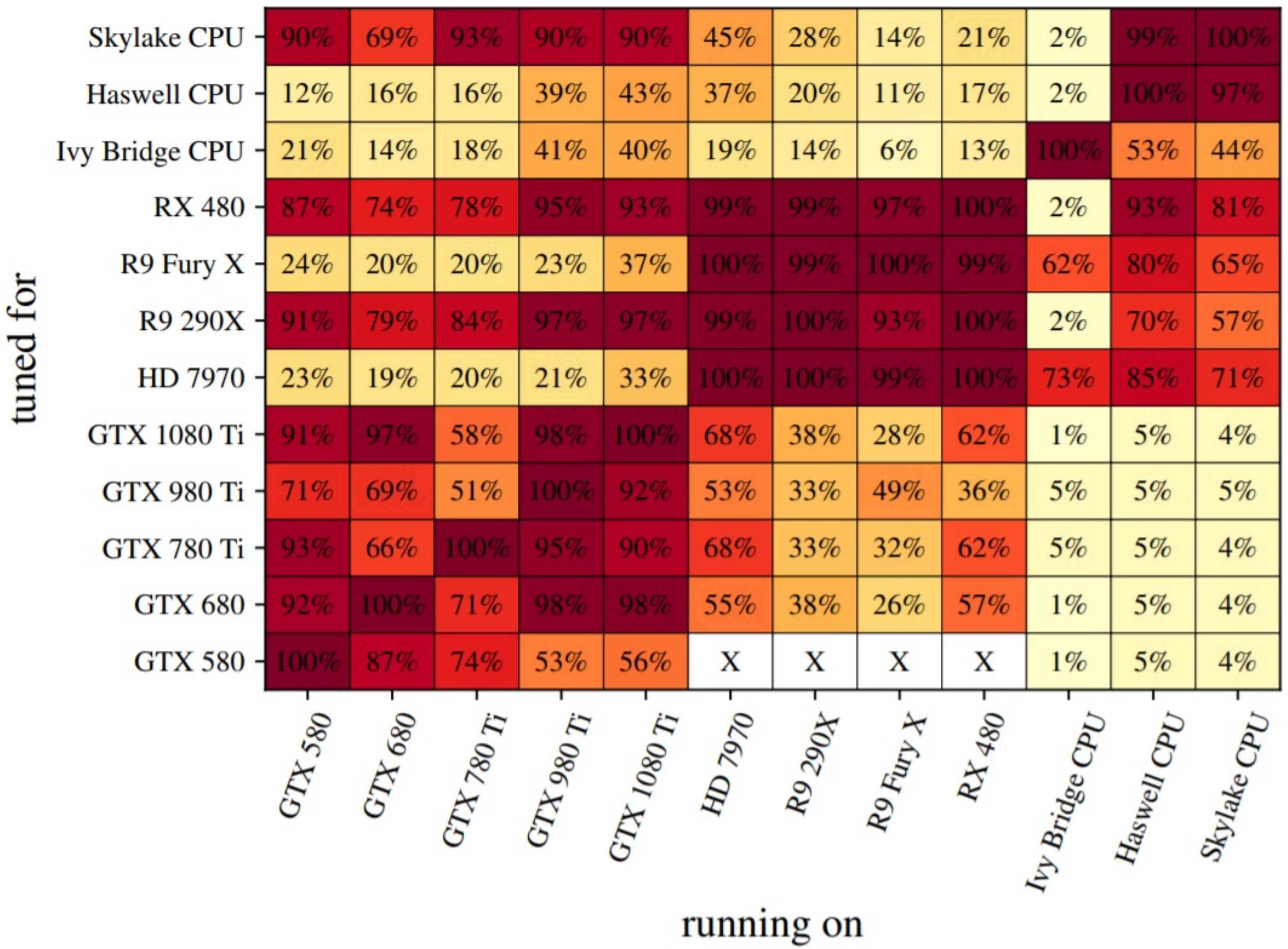
- Software managed cache
- Can be used to share memory between threads in the same block efficiently

Performance Portability

What about different vendors?

- AMD – warp size of 64
- Intel – warp size of 8 OR 16 depending on resource usage
- ARM – old chips do not have warps, new chips have warp size of 8
- All vendors except Nvidia have max block size of 256. ARM supports only 128 in some cases.
- Different memory technologies (DDR, GDDR, HBM)

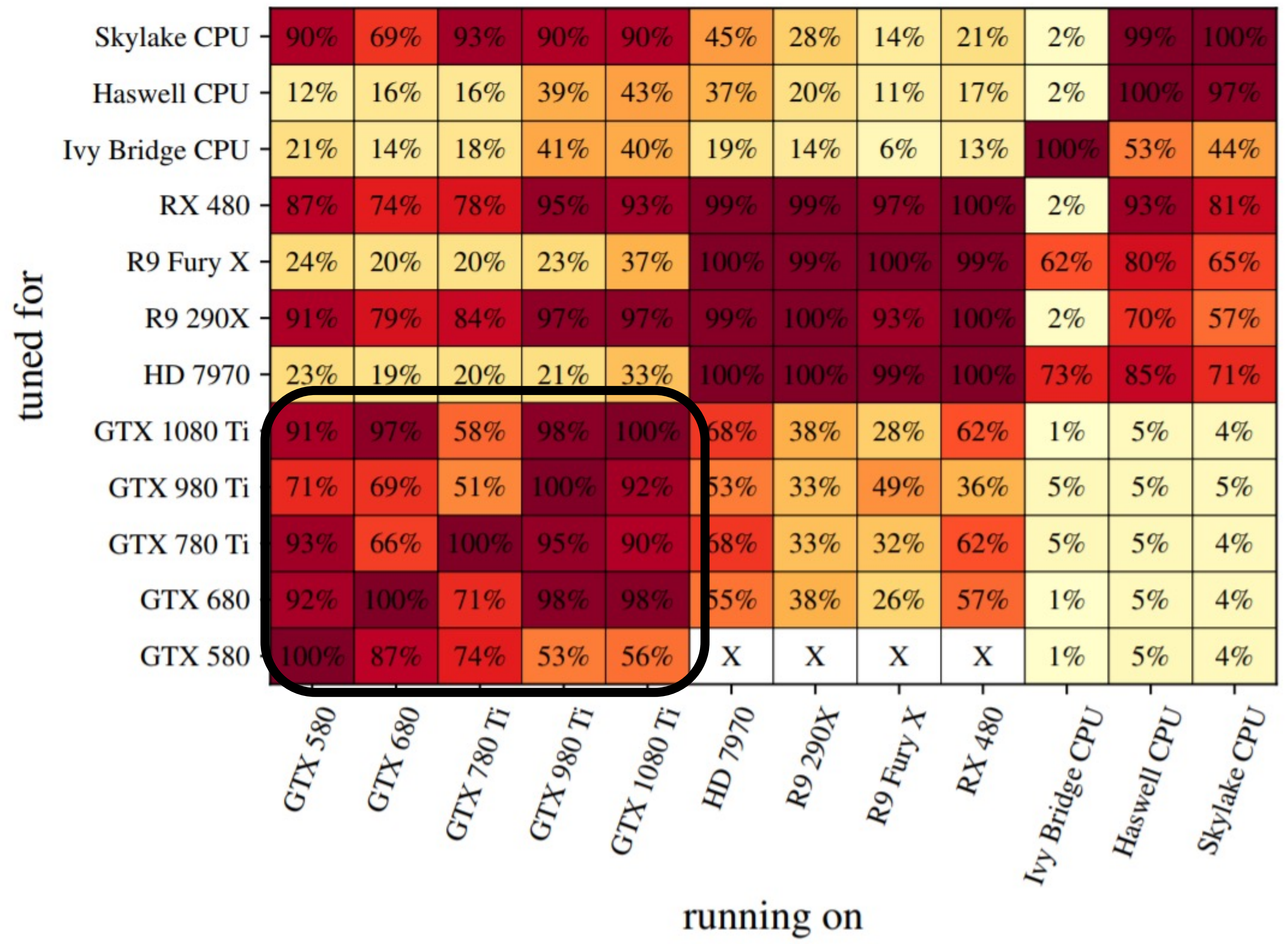
Performance portability is very difficult…



tuned for \ running on	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

From:
“*Analyzing and improving performance portability of OpenCL applications via auto-tuning*” by Price and McIntosh-Smith

Performance portability is very difficult…

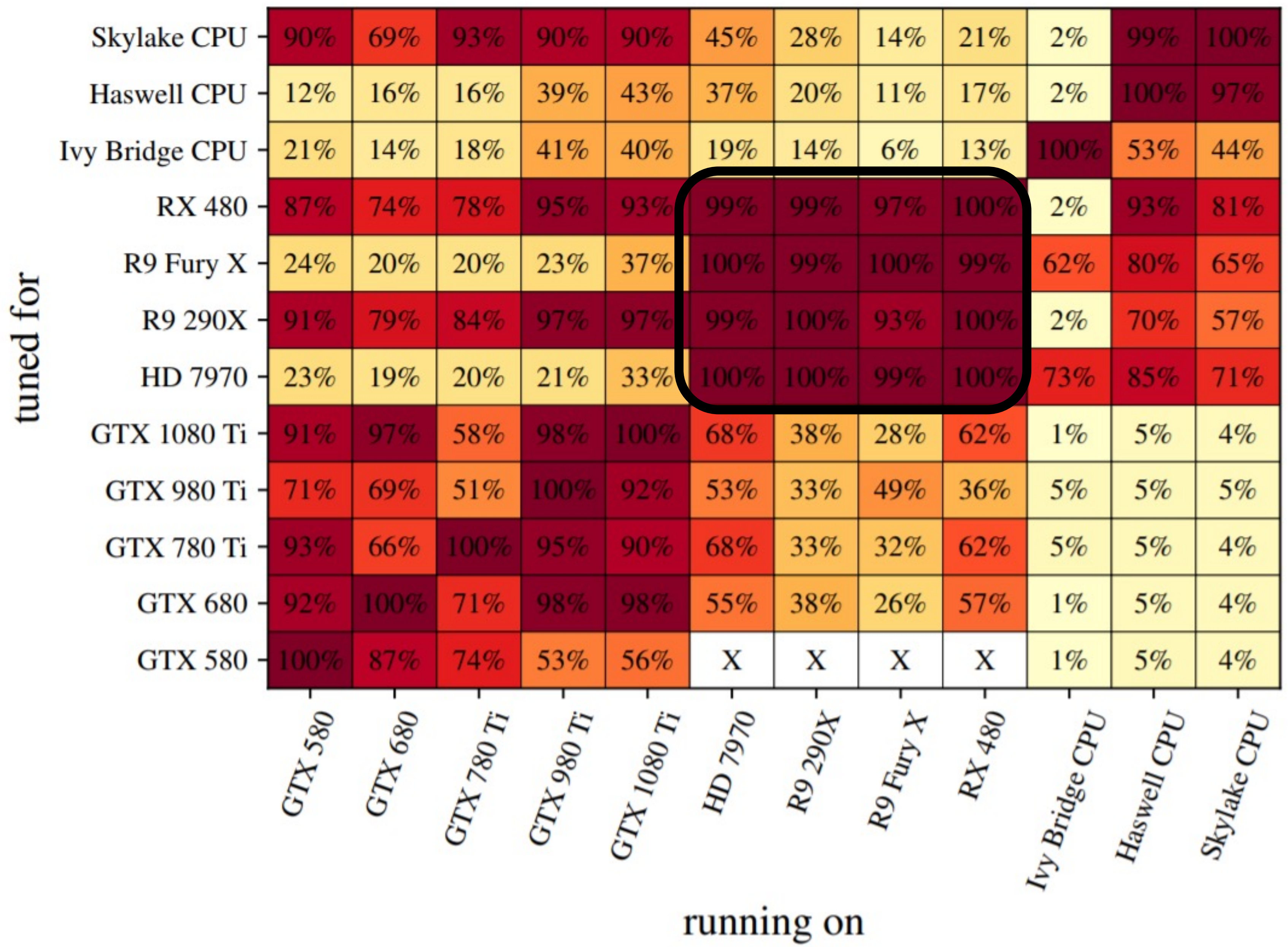


tuned for \ running on	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

nvidia chips tuned for Nvidia gpus

From:
“*Analyzing and improving performance portability of OpenCL applications via auto-tuning*” by Price and McIntosh-Smith

Performance portability is very difficult…

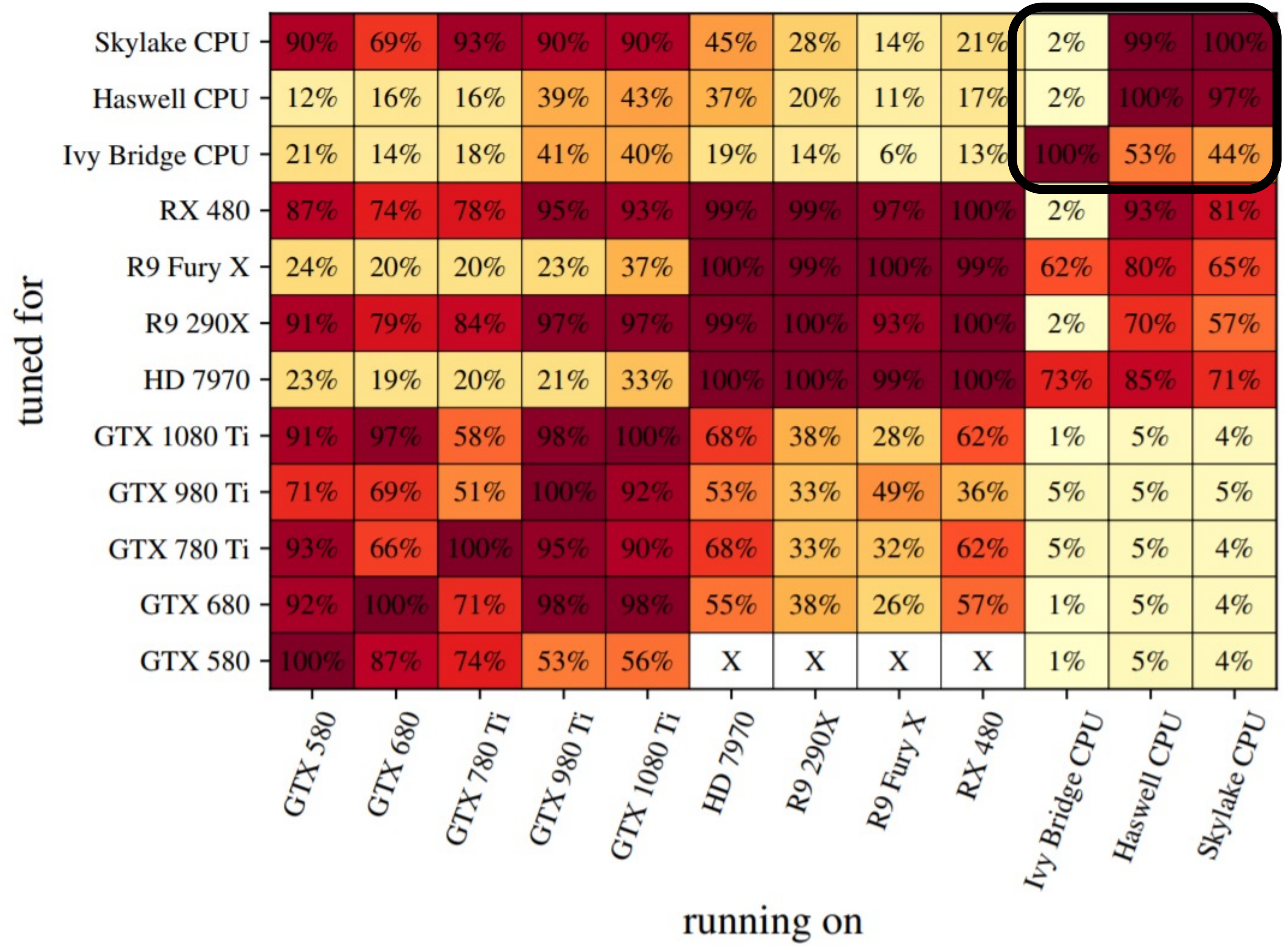


	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

AMD chips tuned for AMD chips

From:
“*Analyzing and improving performance portability of OpenCL applications via auto-tuning*” by Price and McIntosh-Smith

Performance portability is very difficult…

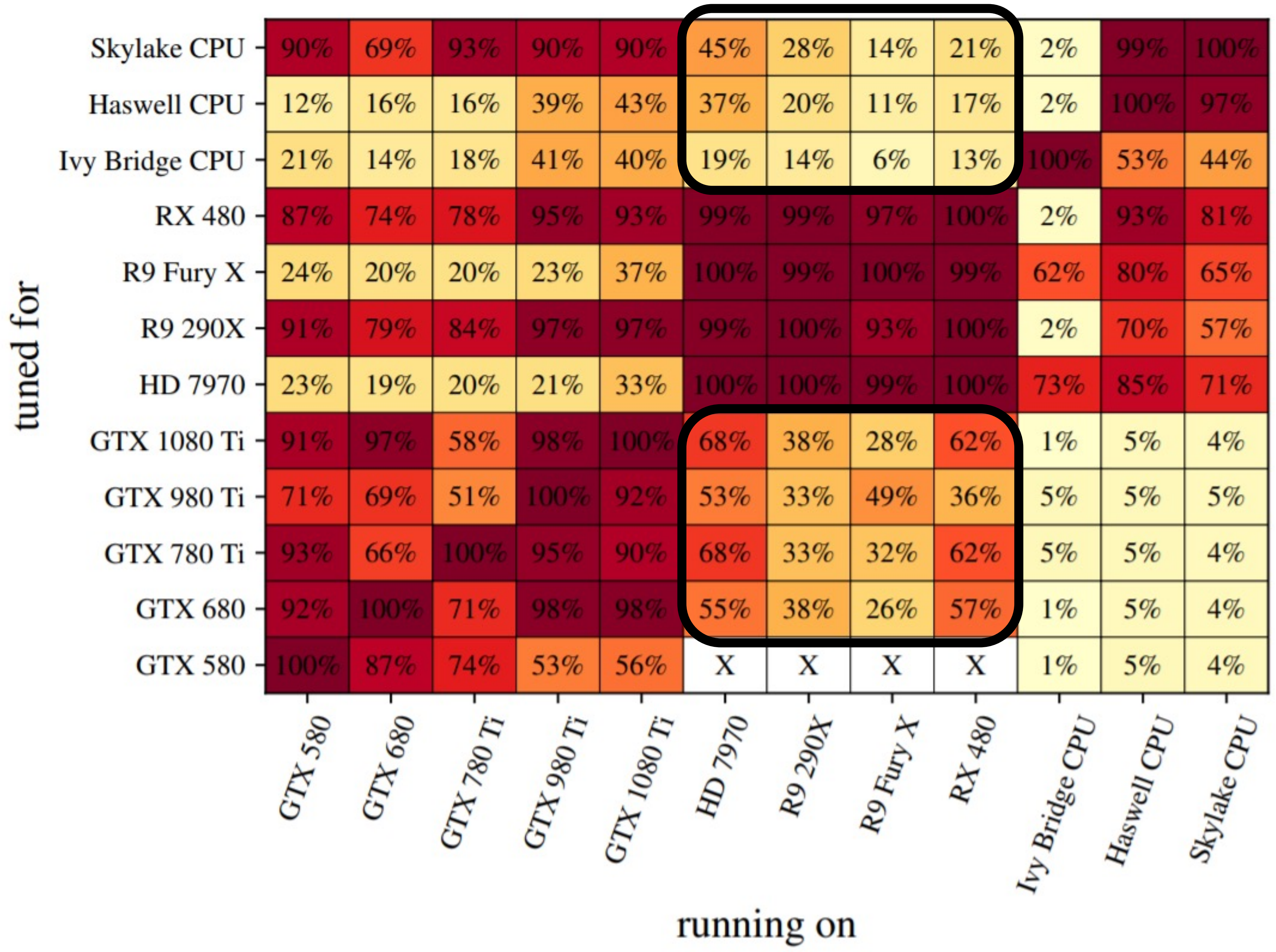


	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

CPU chips tuned for CPUs

From:
"*Analyzing and improving performance portability of OpenCL applications via auto-tuning*" by Price and McIntosh-Smith

Performance portability is very difficult…

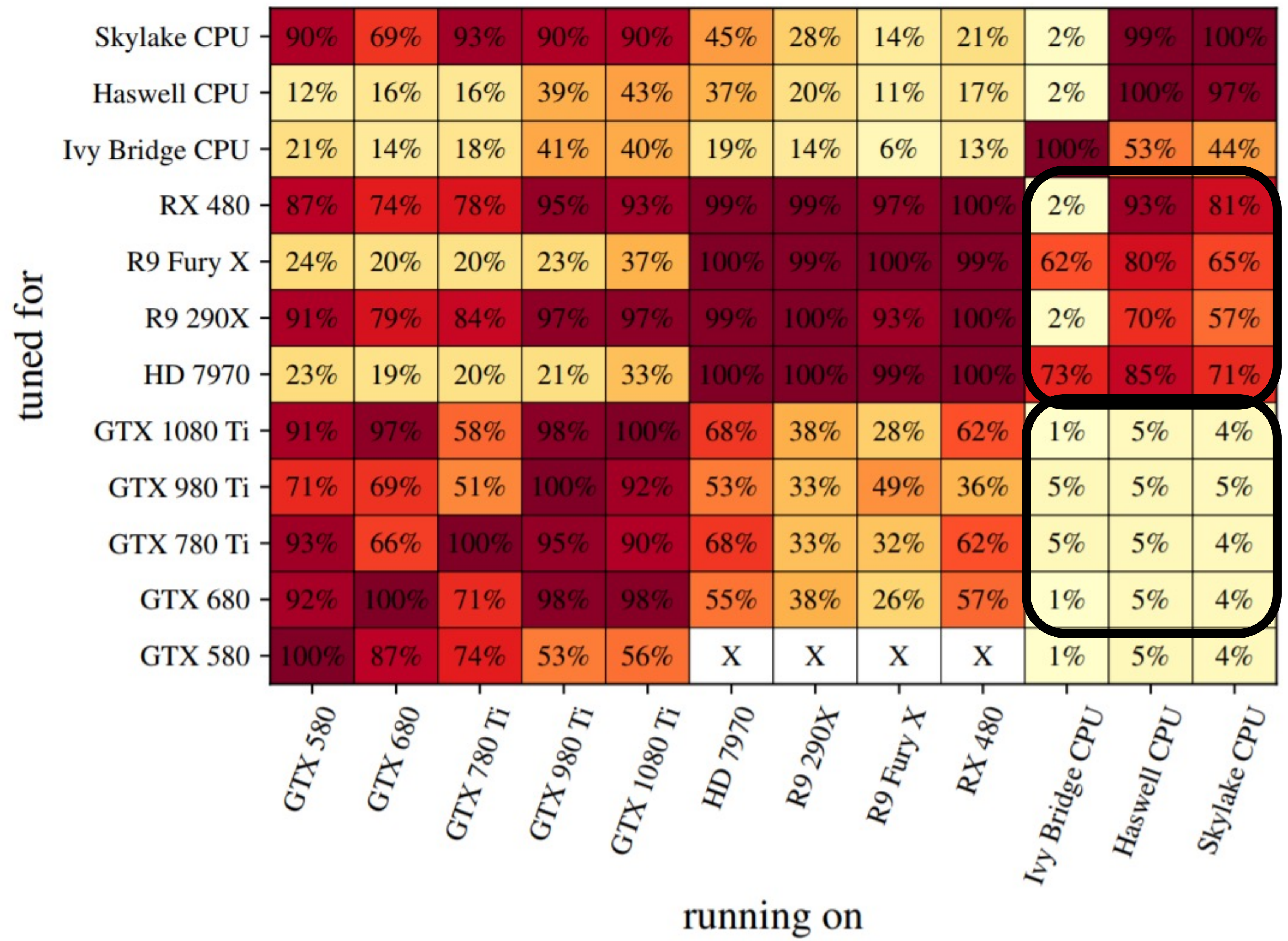


	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

AMD chips tuned for CPUs and Nvidia GPUS

From:
"*Analyzing and improving performance portability of OpenCL applications via auto-tuning*" by Price and McIntosh-Smith

Performance portability is very difficult…



tuned for \ running on	GTX 580	GTX 680	GTX 780 Ti	GTX 980 Ti	GTX 1080 Ti	HD 7970	R9 290X	R9 Fury X	RX 480	Ivy Bridge CPU	Haswell CPU	Skylake CPU
Skylake CPU	90%	69%	93%	90%	90%	45%	28%	14%	21%	2%	99%	100%
Haswell CPU	12%	16%	16%	39%	43%	37%	20%	11%	17%	2%	100%	97%
Ivy Bridge CPU	21%	14%	18%	41%	40%	19%	14%	6%	13%	100%	53%	44%
RX 480	87%	74%	78%	95%	93%	99%	99%	97%	100%	2%	93%	81%
R9 Fury X	24%	20%	20%	23%	37%	100%	99%	100%	99%	62%	80%	65%
R9 290X	91%	79%	84%	97%	97%	99%	100%	93%	100%	2%	70%	57%
HD 7970	23%	19%	20%	21%	33%	100%	100%	99%	100%	73%	85%	71%
GTX 1080 Ti	91%	97%	58%	98%	100%	68%	38%	28%	62%	1%	5%	4%
GTX 980 Ti	71%	69%	51%	100%	92%	53%	33%	49%	36%	5%	5%	5%
GTX 780 Ti	93%	66%	100%	95%	90%	68%	33%	32%	62%	5%	5%	4%
GTX 680	92%	100%	71%	98%	98%	55%	38%	26%	57%	1%	5%	4%
GTX 580	100%	87%	74%	53%	56%	X	X	X	X	1%	5%	4%

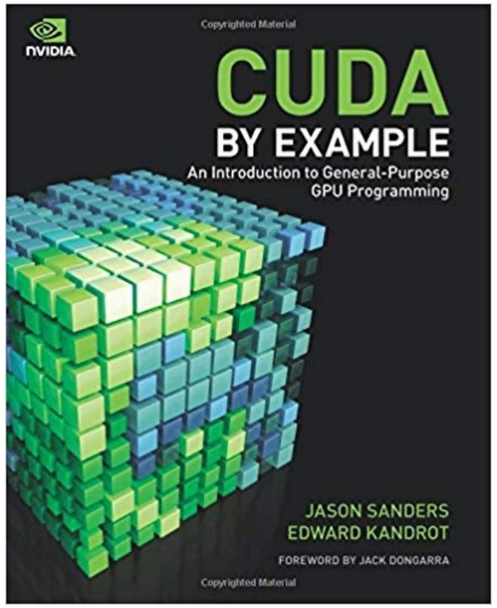
CPU chips tuned for Nvidia and AMD GPUs

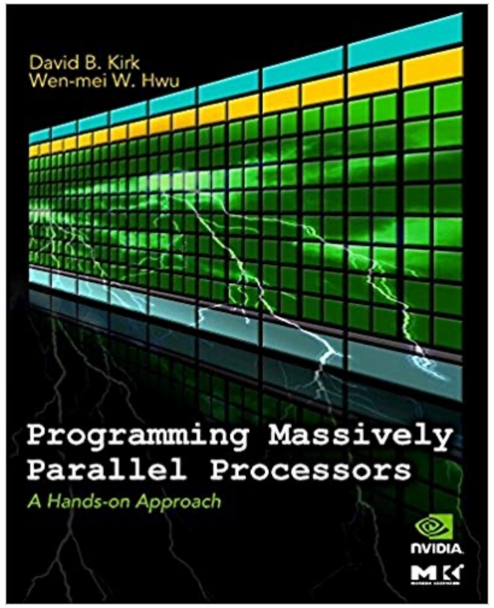
From:
"*Analyzing and improving performance portability of OpenCL applications via auto-tuning*" by Price and McIntosh-Smith

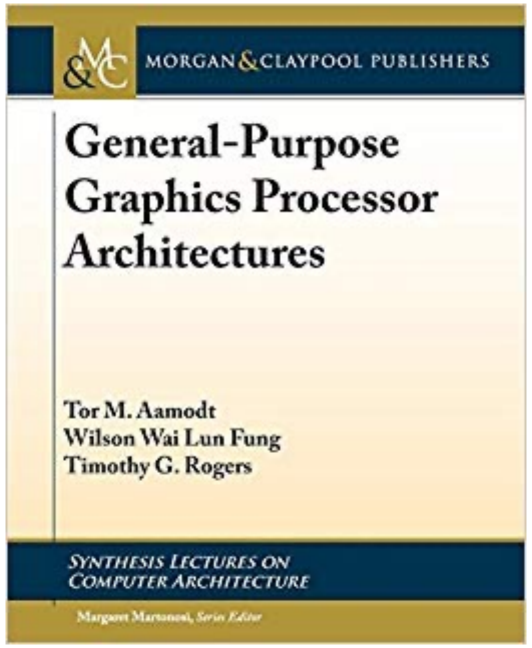
Putting it all together:

- GPUs are programmed as external accelerators: Host must manage memory!
- threads execute in groups of 32 (or 1,8,16,64) called a warp.
- Parallelism across warps hides latency
- Access memory in strided patterns
- Use many blocks!
- Synchronization available across threads in the same block

Further readings






MORGAN & CLAYPOOL PUBLISHERS

General-Purpose Graphics Processor Architectures

Tor M. Aamodt
Wilson Wai Lun Fung
Timothy G. Rogers

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, Series Editor

On Thursday

- Reese will talk about distributed computing!
- Office hours on Wednesday will be joint (go over homework questions)