CUDA, Part One

CS 3220 / CS 5220 Lecture 4-C

Jason Hibbeler
University of Vermont

Spring 2024

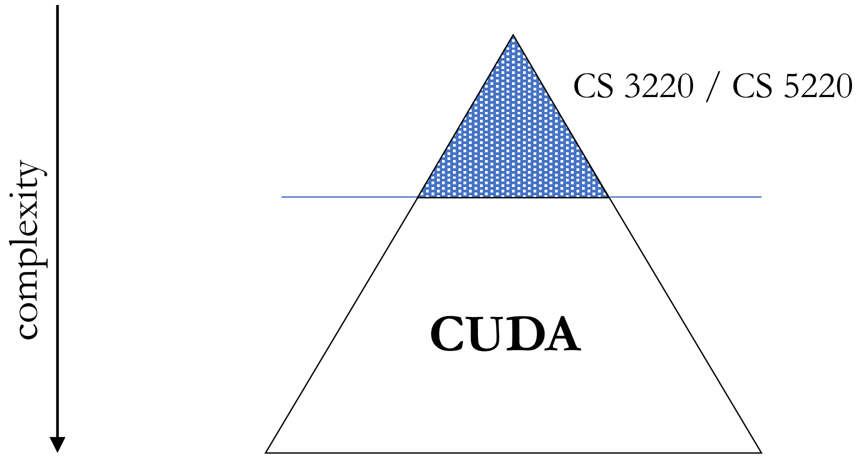
Topics

- Thread blocks
- Kernel
- Allocating memory
- Vector addition
- Grid-stride loop

Goal/Plan

Enable you to write basic CUDA programs

- we'll look at the basic concepts and mechanisms
- you'll then be in position to understand and write more sophisticated GPU algorithms



Goal for the User

Primary wish: “Make my program run faster.”

Recall:

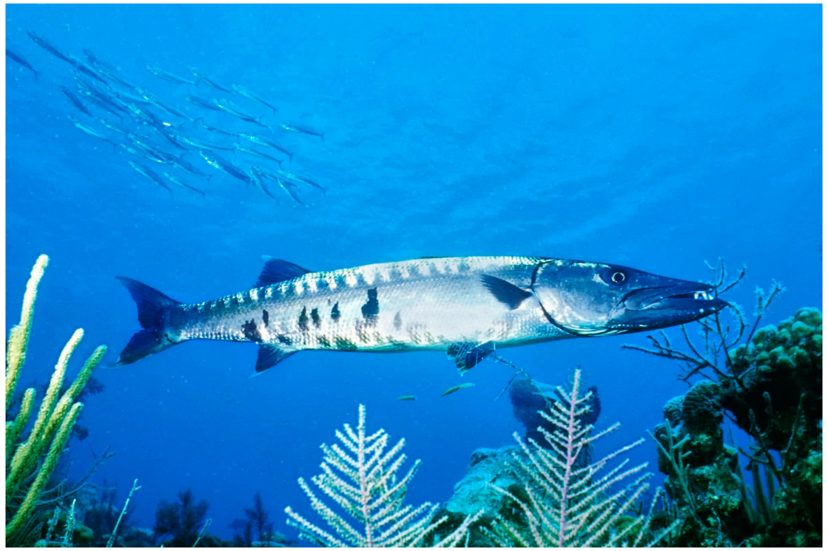
$$CPU\ time = \frac{instructions}{program} \times \frac{clock\ cycles}{instruction} \times \frac{seconds}{clock\ cycle}$$

So to get speedup:

- we can try to make each instruction take less time (primarily through ILP)
- or, we can reduce the #instructions, by having each instruction process more data

(We’ll assume that we can’t increase the clock frequency)

Barracuda vs. 'Cuda vs. CUDA


Barracuda

NVIDIA

CUDA

Nvidia CUDA

1970 Plymouth 'Cuda

Intro to CUDA

The part of a program that runs on a GPU is called a *kernel*

The CPU is called the *host,* and the GPU is called the *device*

Intro to CUDA

The basic structure of a CUDA program:

- set up the problem on the host, using C or C++
- transfer data, as appropriate, to the GPU
- call the kernel, which transfers the program code to the GPU
- let the GPU do the intensive computation on its many cores
- copy results, as appropriate, back to the host's memory

The computation will be in one or more kernels

- each kernel will be executed N times in parallel by N different CUDA threads
- and N can potentially be large value (in the thousands)

First Example

Suppose we want to add together two integer vectors of length N

- here's a serial implementation:

```
#include <stdio.h>

#define N 16

int main() {
  int X[N], Y[N], Z[N];
  int i, fail;

  // initialization
  for (i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  // computation
  for (i=0; i<N; ++i) {
    Z[i] = X[i] + Y[i];
  }
```

```
  // sanity check: are the results correct?
  fail = 0;
  for (i=0; i<N; ++i) {
    if (Z[i] != X[i] + Y[i]) // integer comparison is OK
      fail = 1;
  }

  if (fail)
    printf("error!\n");
  else
    printf("success!\n");

  return 0;
}
```

First Example

Why check the results explicitly?

Suppose we're using a complex algorithm to perform the calculation
- it's a sophisticated algorithm that ultimately will give us the results we expect
- but it will run much much faster (which is why we're developing it)

Then as we develop, we should check that the algorithm is giving us what we expect
- and in our production code, we can comment out (or ifdef out) the checking

The GPU kernel will be the complex algorithm that we're using

First Example

Obviously, the actual addition can be parallelized

- this problem is *embarrassingly parallel*

```
#include <stdio.h>

#define N 16

int main() {
  int X[N], Y[N], Z[N];
  int i, fail;

  // initialization
  for (i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  // computation
  for (i=0; i<N; ++i) {
    Z[i] = X[i] + Y[i];
  }
```

```
  // sanity check: are the results correct?
  fail = 0;
  for (i=0; i<N; ++i) {
    if (Z[i] != X[i] + Y[i]) // integer comparison is OK
      fail = 1;
  }

  if (fail)
    printf("error!\n");
  else
    printf("success!\n");

  return 0;
}
```

I can compute each `Z[i]` in parallel!

Adding Two Vectors: The Kernel

Now let's parallelize this

- we'll use N different threads, and each will add two corresponding entries in the vectors
- we'll put this addition into a function
- and we'll have each of the N threads call this function
- and we'll assume that the global variable `myThreadId` will be set to each thread's number
- and we'll use the thread id as the index in the vectors (one-to-one correspondence btw threads and elements)

Now, here's what the add function would look like:

```
void add(int *v1, int *v2, int *v3, int N) { // note: we're not using N yet
  int i = myThreadId;
  v3[i] = v1[i] + v2[i]
}
```

Adding Two Vectors: The Kernel

We can make this a little more general

- separate the hardware (the number of threads) and the problem (the value of N)
- and assume that we might launch more that N threads
- this will let us use the same code for many problems (independent of the problem size)

Then, here's what the add function would like:

```
void add(int *v1, int *v2, int *v3, int N) {
  int i = myThreadId;
  if (i < N) // this will handle the situation that #threads is independent of N
    v3[i] = v1[i] + v2[i]
}
```

Adding Two Vectors

The data (the arrays) will start in one place

The computation will take place in a different place

- and so the data will have to be transferred from the first place to the second place

When will this scheme work well?

Adding Two Vectors

When will this scheme work well?

If the cost of transferring data and launching each thread is not too high

- relative to the amount of processing that each thread is performing

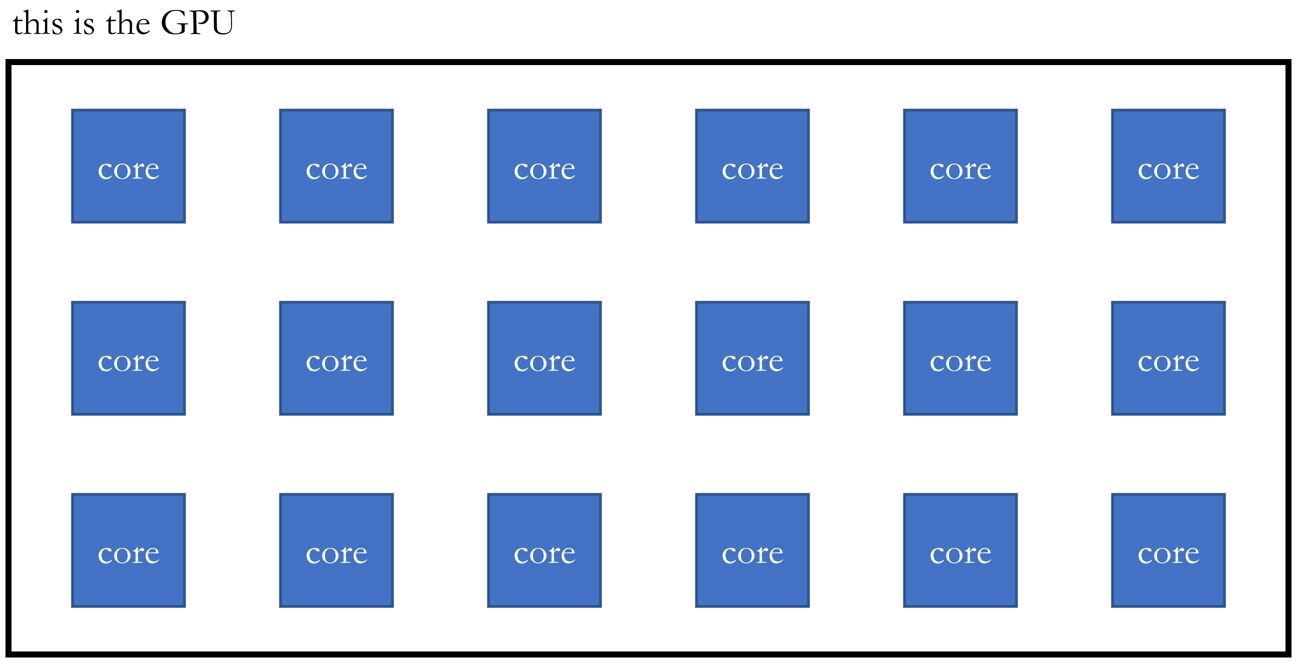
We should be suspicious about this example

- a single integer addition (per thread) just isn't very much work!

However, this is still useful to get us thinking about parallelization

Blocks

Here's the simplified structure of a GPU:



A *thread block* is the work assigned to each core, in the form of one or more threads

- in the beginning, we'll assume one thread per block

Each thread can then ask "what block am I in?"

- this is an analogous question to "what is my threadID?"

Nvidia calls a core a *streaming multiprocessor* (SM)

Blocks

CUDA makes a global variable available to the kernels: `blockIdx`

- `blockIdx.x` is the block that I am in
- here, we'll assume a 1-D structure of blocks—we'll talk about 2-D structures soon
- and we’ll assume one thread per block
- so that each block computes one element of the vector sum

kernel: the code that is running on the GPU

Here's the actual CUDA add kernel:

```
__global__
void add(int *v1, int *v2, int *v3, int N) {
  int tid = blockIdx.x;
  if (tid < N)
    v3[tid] = v2[tid] + v1[tid];
}
```

Kernel Function

The `__global__` annotation marks this as code that will run on the device (the GPU)

The mapping of threads to cores can actually be done in three dimensions

- which means that there is a `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`

But again, we're doing a one-dimensional example here

- since the structure of the data (the 1-D vectors) is inherently one dimensional
- and so the mapping of blocks to data elements will be as a one-dimensional array

We'll look at higher-dimensional problems later

Memory

The CPU has its own memory

- and we use the standard `malloc()` to allocate memory on the CPU ("the host”)

The GPU has its own local memory

- and we have to use a special CUDA library call to allocate memory on the GPU ("the device")
- the call is `cudaMalloc()`
- and we have to use a CUDA library call to copy data back and forth between host memory and device memory: `cudaMemcpy()`

The Kernel

This is the code that will run on the GPU

```
__global__
void add(int *X, int *Y, int *Z, int  N) {
  int tid = blockIdx.x; // blockIdx is a global variable; tells which block I am in
  if (tid < N)
    Z[tid] = X[tid] + Y[tid];
}
```

The `main()`

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int X[N], Y[N], Z[N];
  int *dev_X, *dev_Y, *dev_Z;

  // allocate memory on the GPU
  cudaMalloc( (void **) &dev_X, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Y, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Z, N*sizeof(int) );

  // we'll do a simple calculation: Z[i] = i + i*i
  for (int i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  cudaMemcpy( dev_X, X, N*sizeof(int), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_Y, Y, N*sizeof(int), cudaMemcpyHostToDevice );

  add<<<N, 1>>>( dev_X, dev_Y, dev_Z, N );

  cudaDeviceSynchronize(); // block until GPU finishes
  cudaMemcpy( Z, dev_Z, N*sizeof(int), cudaMemcpyDeviceToHost );
```

```
  //sanity check
  int fail = 0;
  for (int i=0; i<N; ++i) {
    if (Z[i] != X[i] + Y[i])
      fail = 1;
  }

  if (fail)
    printf("error!\n");
  else
    printf("success\n");

  cudaFree( dev_X );
  cudaFree( dev_Y );
  cudaFree( dev_Z );

  return 0;
}
```

The `dev_` prefix is a convention; some people use `d_` for device (GPU) memory and `h_` for host (CPU) memory.
Pick a convention and use it consistently.

cudaDeviceSynchronize()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int X[N], Y[N], Z[N];
  int *dev_X, *dev_Y, *dev_Z;

  // allocate memory on the GPU
  cudaMalloc( (void **) &dev_X, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Y, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Z, N*sizeof(int) );

  // we'll do a simple calculation: Z[i] = i + i*i
  for (int i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  cudaMemcpy( dev_X, X, N*sizeof(int), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_Y, Y, N*sizeof(int), cudaMemcpyHostToDevice );

  add<<<N, 1>>>( dev_X, dev_Y, dev_Z, N );

  cudaDeviceSynchronize(); // block until GPU finishes
  cudaMemcpy( Z, dev_Z, N*sizeof(int), cudaMemcpyDeviceToHost );
```

The GPU does its work asynchronously

Here, the add() kernel will run at some point after the code that the CPU is running calls the add() function--but we don't know when.

This means that any subsequent code that depends on having the results of the add() function must explicilty wait for the add() to complete.

The way to do this is by calling `cudaDeviceSynchronize()`

But there's an exception to this rule (next slide)

cudaDeviceSynchronize()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int X[N], Y[N], Z[N];
  int *dev_X, *dev_Y, *dev_Z;

  // allocate memory on the GPU
  cudaMalloc( (void **) &dev_X, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Y, N*sizeof(int) );
  cudaMalloc( (void **) &dev_Z, N*sizeof(int) );

  // we'll do a simple calculation: Z[i] = i + i*i
  for (int i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  cudaMemcpy( dev_X, X, N*sizeof(int), cudaMemcpyHostToDevice );
  cudaMemcpy( dev_Y, Y, N*sizeof(int), cudaMemcpyHostToDevice );

  add<<<N, 1>>>( dev_X, dev_Y, dev_Z, N );

  cudaDeviceSynchronize(); // block until GPU finishes
  cudaMemcpy( Z, dev_Z, N*sizeof(int), cudaMemcpyDeviceToHost );
```

Exception to the previous comment

CUDA functions that are called by the CPU will be serialized: they will be executed in the order in which they appear in the program.

In particular, the `cudaMemcpy()` will only execute after the `add()` has completed; this means that the `cudaDeviceSynchronize()` call in this code is not actually necessary.

But it cause any incorrect behavior--it's just a no-op in this situation.

Launching the Kernel

The key statement is:

```
add<<<N, 1>>>( dev_X, dev_Y, dev_Z, N );
```

This launches *N* blocks, each of which will run the `add()` function

The value `1` means "one thread per block"

- for simplicity, that's what we're doing first
- we'll talk about this soon

Allocating Memory on the GPU

The function to allocate memory on the GPU is `cudaMalloc()`

```
// allocate memory on the GPU
cudaMalloc( (void **) &dev_X, N*sizeof(int) );
cudaMalloc( (void **) &dev_Y, N*sizeof(int) );
cudaMalloc( (void **) &dev_Z, N*sizeof(int) );
```

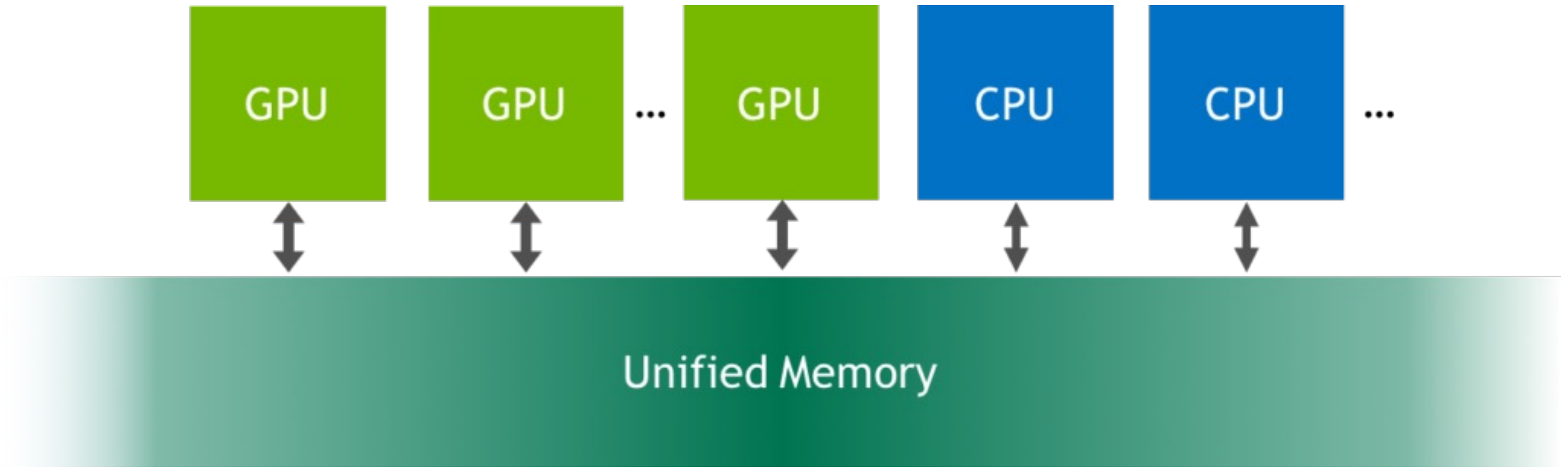
The cast to `void**` is not necessary

- it's kind of a convention
- use it if you like

Allocating Memory: Simplified

Nvidia has simplified the way that the host and the device allocate memory

- they've introduced a structure called “unified memory”



from https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Vector Add, Using Unified Memory

```
int main() {
  int *X, *Y, *Z;

  // allocate memory
  cudaMallocManaged( &X, N*sizeof(int) );
  cudaMallocManaged( &Y, N*sizeof(int) );
  cudaMallocManaged( &Z, N*sizeof(int) );

  for (int i=0; i<N; ++i) {
    X[i] = i;
    Y[i] = i*i;
  }

  add<<<N,1>>>( X, Y, Z, N );

  // here, the cudaDeviceSynchronize() call
  // IS necessary!!!
  cudaDeviceSynchronize();
```

```
  int fail = 0;
  for (int i=0; i<N; ++i) {
    if (Z[i] != X[i] + Y[i])
      fail = 1;
  }

  if (fail)
    printf("error!\n");
  else
    printf("success\n");

  cudaFree( X );
  cudaFree( Y );
  cudaFree( Z );

  return 0;
}
```

`cudaDeviceSynchronize()`, Again

In the preceding example, with unified memory, the synchronization call IS necessary

The next statement after the `add()` call is executed on the CPU: it's the accessing of the unified memory

- unless we are sure that the GPU has finished its work (in the `add()` kernel), then we can't be sure that the result will be available for the CPU to use

Conclusion/rule:

- if you use `cudaMalloc()` and then call a kernel and then call `cudaMemcpy()`, then it's not necessary to synchronize
- if you use unified memory and call a kernel and then access the unified memory to get results that the kernel has produced, then it is necessary to synchronize

Unified Memory

Unified memory

- a single logical address space that is accessible from all CPUs and GPUs in the system
- the goal is to enable easier development (by simplifying and unifying the memory allocation)
- and more efficient use of memory (by including hardware that can manage the memory automatically)

But the use of unified memory requires a sophisticated knowledge of how each GPU—and each GPU SM—will use memory

- SM: streaming multiprocessor—a GPU "core"

For CS 3220 / CS 5220:

- understand and use one of the techniques for allocating and using GPU memory
- either the old-fashioned way or the new way (unified memory)

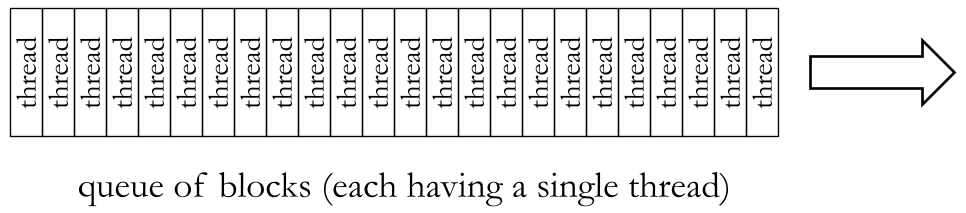
A GPU Core (Streaming Multiprocessor)

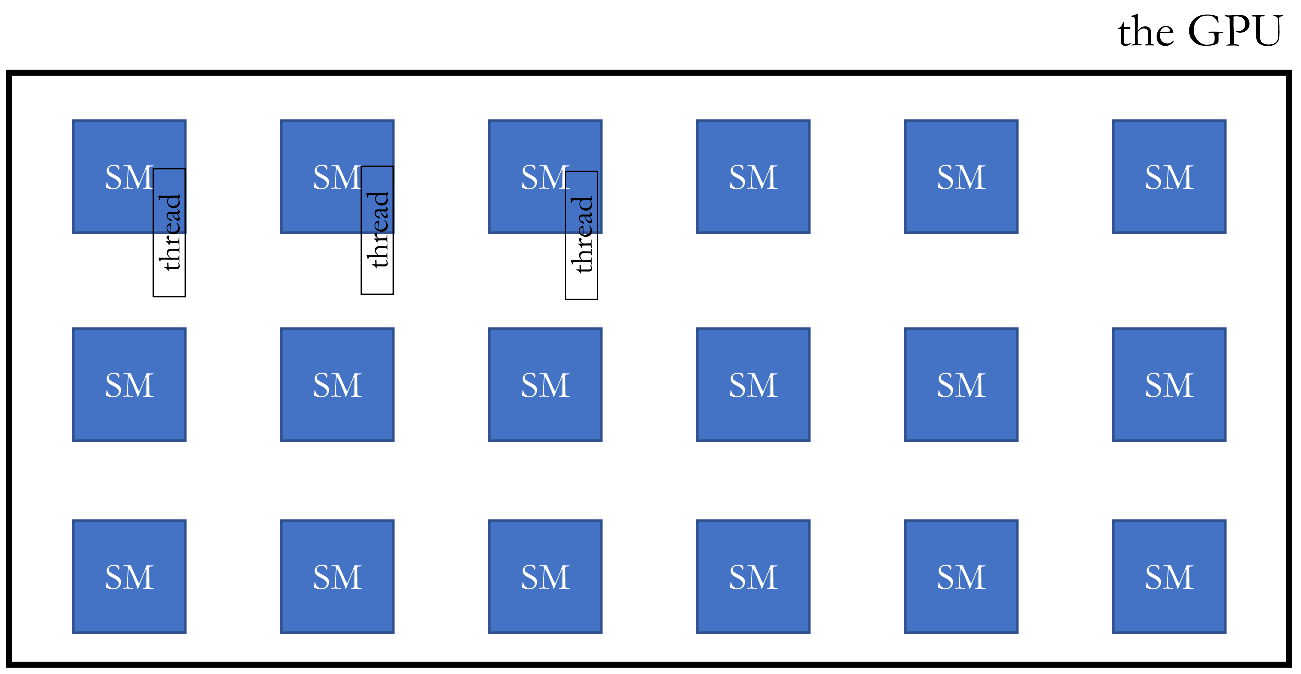
The previous example launched `N` separate threads

- **one** thread per SM
- this is the `1` in `add<<<N, 1>>>( dev_X, dev_Y, dev_Z, N );`

So conceptually, here's what this looks like in the GPU:

- there's a queue of *N* blocks, each containing a single thread
- and each block is assigned to an SM and then executes the kernel code
- the thread-block scheduler creates this queue





A GPU Core (Streaming Multiprocessor)

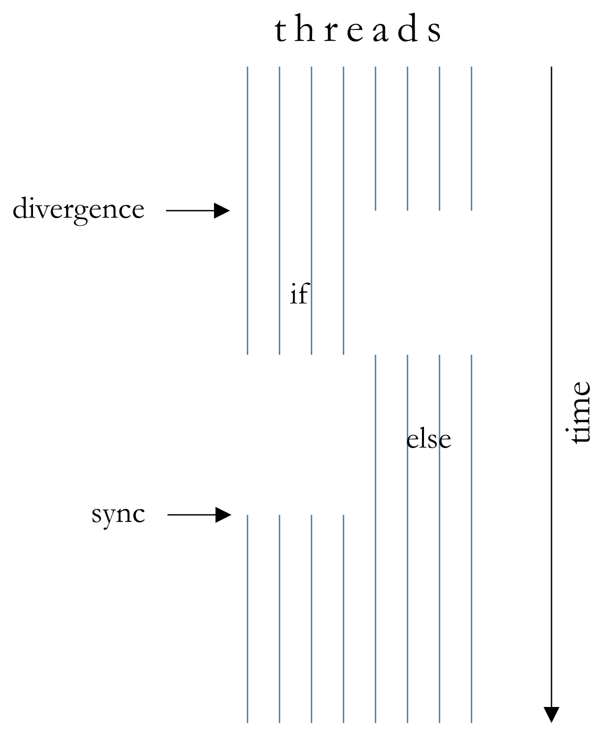
But in fact, more than one block is assigned to each SM

- and the SM can run these all together, one block at a time
- all threads in a block will execute the same instruction
- but across all blocks assigned to an SM, there could be for example 512 concurrent threads per SM
- and some number of these (say, 32) can actually execute <u>in parallel</u>
- note that with 50 SMs, this gives us something like 25000 simultaneously executing threads
- they don't all run in parallel, but potentially 50 * 32 ≈ 1500 could literally run simultaneously
- and for the latest Nvidia GPUs, the block size is even larger (1024) and the #SMs is larger (say, 80)

So for the vector addition:

- we could increase the degree of parallelism tremendously
- use `n` blocks, and give each block `m` threads

Predication



Basic idea: conditional execution through predication

- in predicate registers

The compiler optimizes a simple outer-level if-then-else

- by predication
- the threads that take a branch diverge
- branches converge, through synchronization, after the different execution paths complete and the threads are back in sync
- a simple way to do this is to have some threads in the "if", with the "else" threads idle; and then vice versa
- the key thing is that in any particular clock cycle, two threads must either execute the same instruction or else one or both must be idle: they can't do different operations

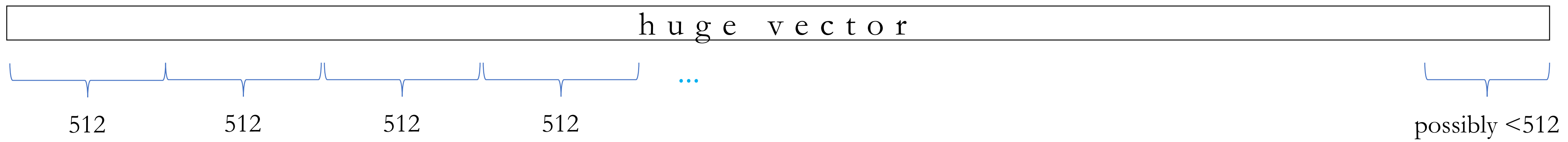
Predication

The key structural fact that constrains CUDA threads:

- in any particular clock cycle, two threads must either execute the same instruction or else one or both must be idle
- remember, these are not POSIX threads (which can do arbitrary computation!)
- these threads are managed by the Nvidia hardware
- this makes it easier to exploit massive parallelism
- but the speed of a program will be constrained to how many threads are active at any cycle

This is how strip mining of vector loops is handled

- divide n into chunks of size 512; the last chunk could have fewer than 512 elements in it



Divergence and Convergence

Predication is efficient for small fragments of conditional code

- especially for an if statement without a corresponding else

Divergence is a hardware technique to handle the more general case of condition-dependent code execution by the threads of a warp

- experts have surmised from Nvidia's published information that code containing a divergence is actually executed twice
- once for the threads for which the condition was true
- and again, for the threads for which the condition was false

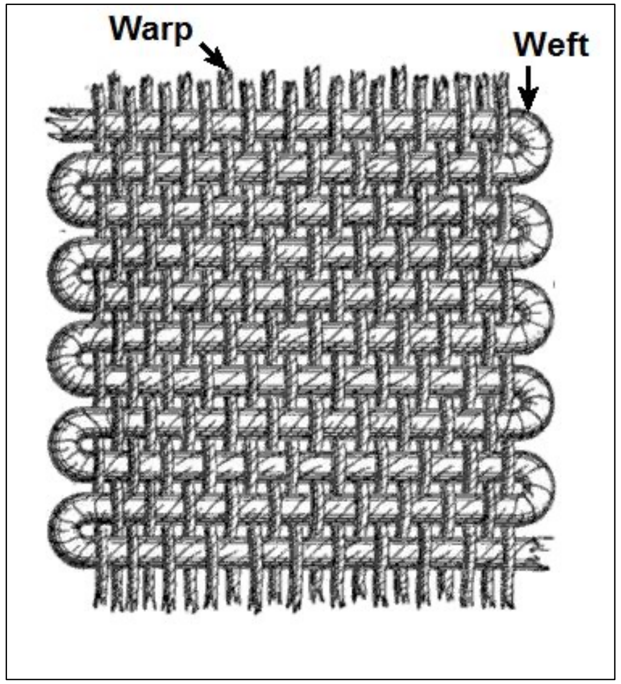
Streaming Multiprocessor

Each SM is an *SIMT* core:

- single instruction multiple thread
- the threads are managed by the GPU hardware
- there is no context switching between threads

All cores execute the same instructions

- but on different data
- each core can run 32 threads simultaneously
- each thread executes the same instruction in parallel
- a group of 32 threads is called a *warp*



warp and weft in weaving

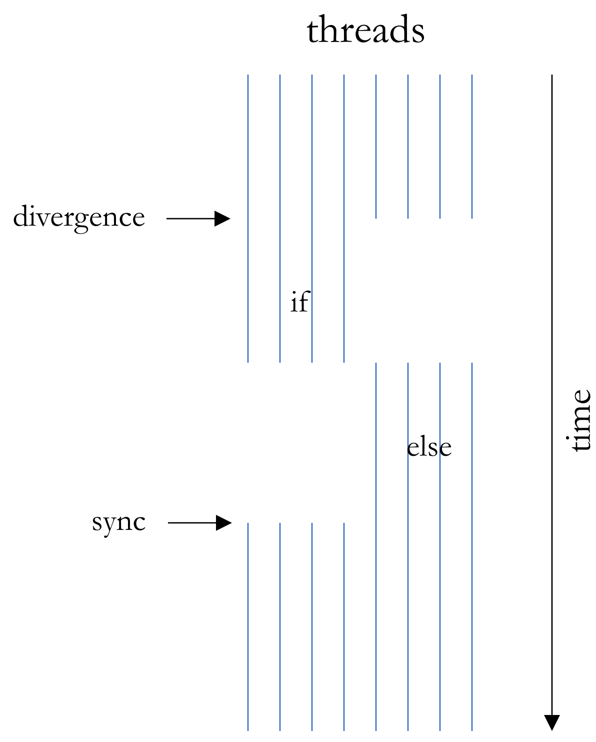
"The term *warp* comes from weaving, the first parallel thread technology." (says Nvidia)

Thread Warp

From Nvidia:

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.



Thread Warp

But also:

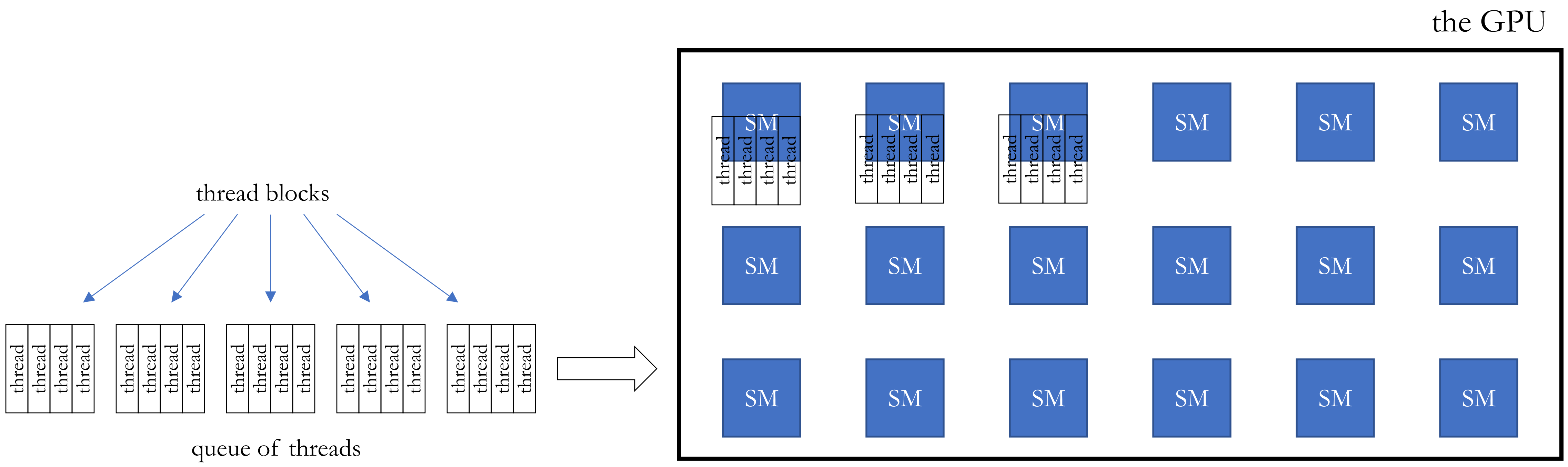
Prior to Volta, warps used a single program counter shared amongst all 32 threads in the warp together with an active mask specifying the active threads of the warp. As a result, threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data, and algorithms requiring fine-grained sharing of data guarded by locks or mutexes can easily lead to deadlock, depending on which warp the contending threads come from.

Starting with the Volta architecture, *Independent Thread Scheduling* allows full concurrency between threads, regardless of warp. With Independent Thread Scheduling, the GPU maintains execution state per thread, including a program counter and call stack, and can yield execution at a per-thread granularity, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. A schedule optimizer determines how to group active threads from the same warp together into SIMT units. This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: threads can now diverge and reconverge at sub-warp granularity.

A GPU Core (Streaming Multiprocessor)

Now, we'll group threads into "blocks"

- and each block will be assigned to an SM
- contrast this with the simple processing model from before (one thread per SM)



Terminology

Each SM processes groups of threads (for example, 32) in parallel

The thread scheduler (= "Warp Scheduler") schedules the threads of each thread block

- Nvidia recommends that each thread block have 256 or 512 threads

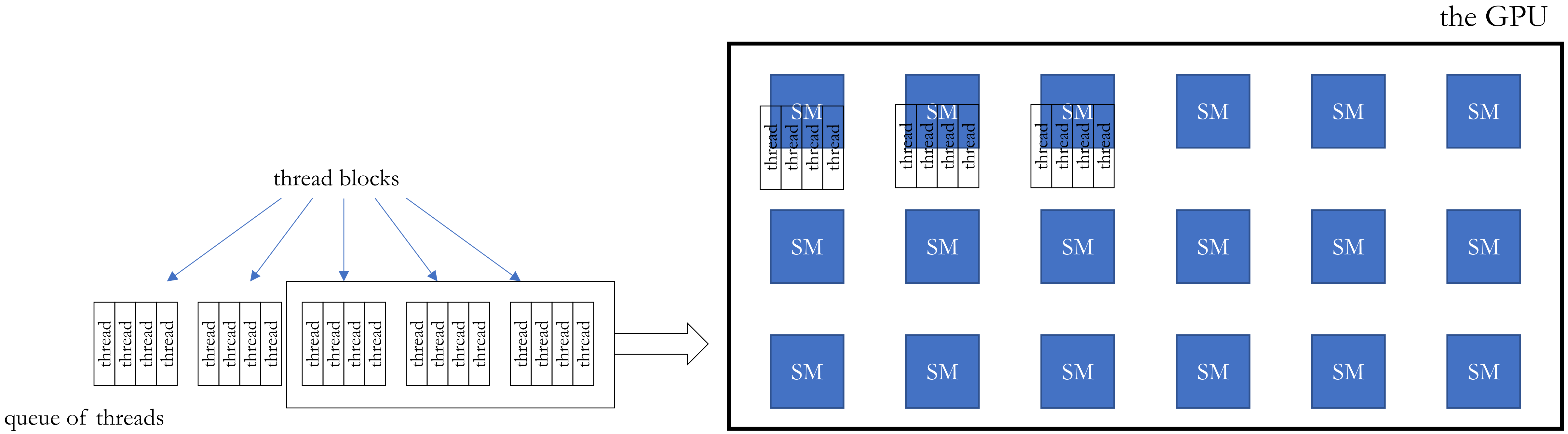
The thread-block scheduler (= "Giga Thread Engine") assigns a thread block to an SM

- this means that each thread block must be independent
- and that a thread block can execute in any order relative to other thread blocks
- thread blocks might execute in parallel or serially—this is determined by the thread-block scheduler
- and so again, we must not create dependences <u>between</u> thread blocks
- because we have no way of knowing the order in which different thread blocks will execute

Thread Blocks

Each block is scheduled as a unit

- but we can't make any assumptions about when a block will run



these three thread blocks (warps) might run at the same time; but the others not; or something totally different—we can't predict when each thread block will run; only that it will run

Terminology

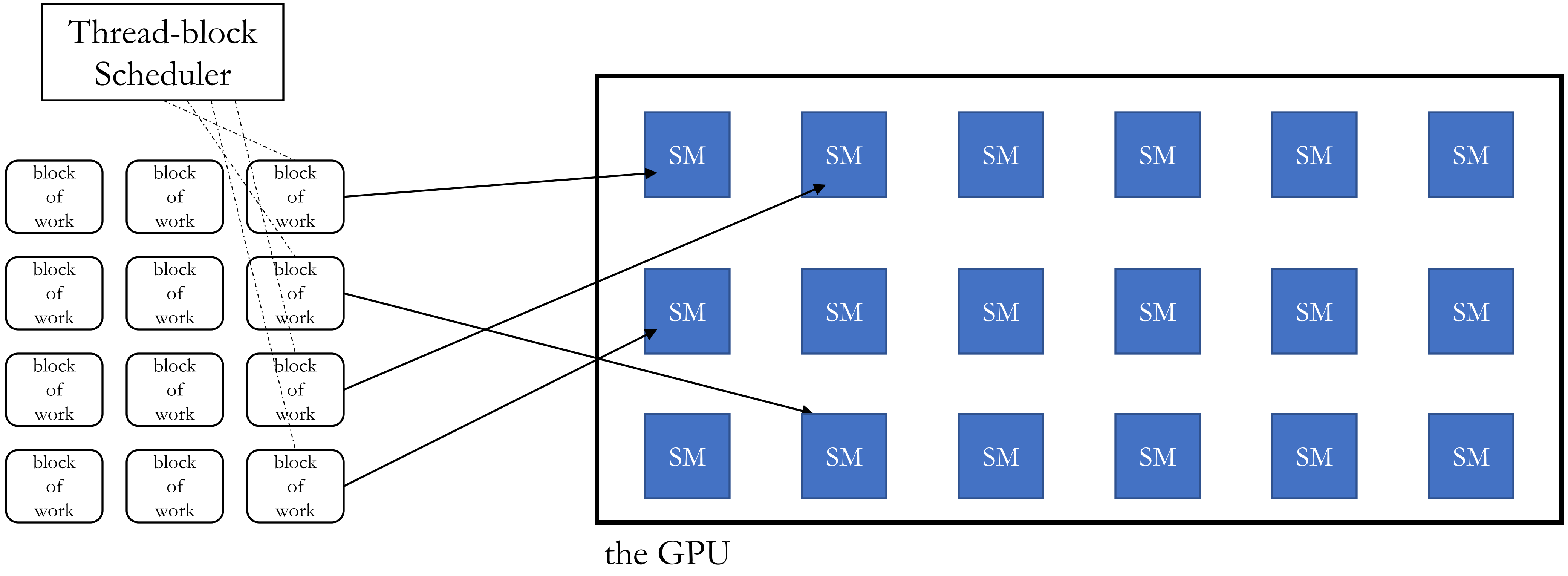
Thread blocks are independent

But the threads in a thread block <u>can</u> cooperate

- through shared memory
- and through synchronization

Thread-Block Scheduler

Assigns each thread block to an SM



The Thread Block

The second parameter in the call to the kernel specifies the size of the thread block

- i.e., the number of threads in each thread block

So this will be the idea:

```
add<<<numBlocks, threadsPerBlock>>>( dev_X, dev_Y, dev_Z, N );
```

And we'll pick appropriate values for `numBlocks` and `threadsPerBlock`

- all threads in a block will reside in the same SM
- the SM has limited memory resources
- “on current GPUs, a thread block may contain up to 1024 threads”

Example, with Large Vectors

Now, the <u>total</u> number of threads will be `numBlocks` * `threadsPerBlock`

- this could potentially be a large number

```
add<<<numBlocks, threadsPerBlock>>>( dev_X, dev_Y, dev_Z, N );
```

Structure of the Computation

Let's say that each thread block will have 256 threads

- then for the vector-addition loop, with vectors of length N, we'll need $N/256$ blocks

But there's a slight gotcha

- we want to make this as general as possible
- if $N < 256$, then $N/256$ would be zero, since this is integer division
- what we really want is $max(1, N/256)$
- and a clever trick to get this is to use $(N + 255)/256$ (using integer division)

So, we'll use this:

```
numBlocks = (N+255) / 256; // so if N < 256, numBlocks will be 1
```

Structure of the Computation

Then, we'll launch the `add` function this way:

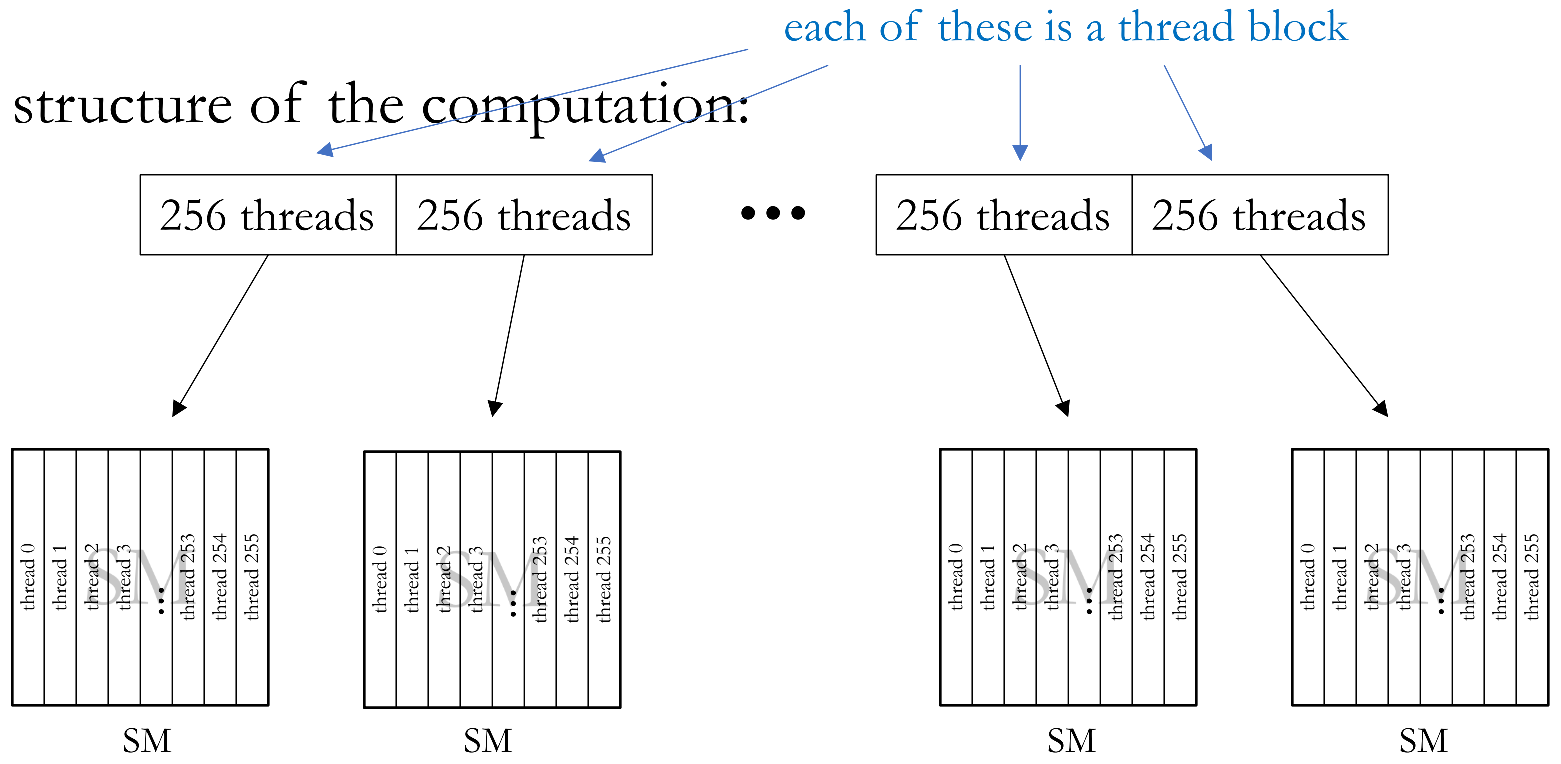
```
threadsPerBlock = 256;
numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;
add<<<numBlocks, threadsPerBlock>>>( dev_X, dev_Y, dev_Z, N );
```

Structure of the Computation

Now, it's not as direct to get each individual thread’s position in the vector

- since whole groups of threads will be in a thread block
- and each thread block will be put on an SM

Here's the structure of the computation:



Thread Position

So now, the position of each thread in the whole problem depends on two things:

- the position of the thread inside of its block
- and the position of that block inside of the set of all blocks

Additional CUDA variables:

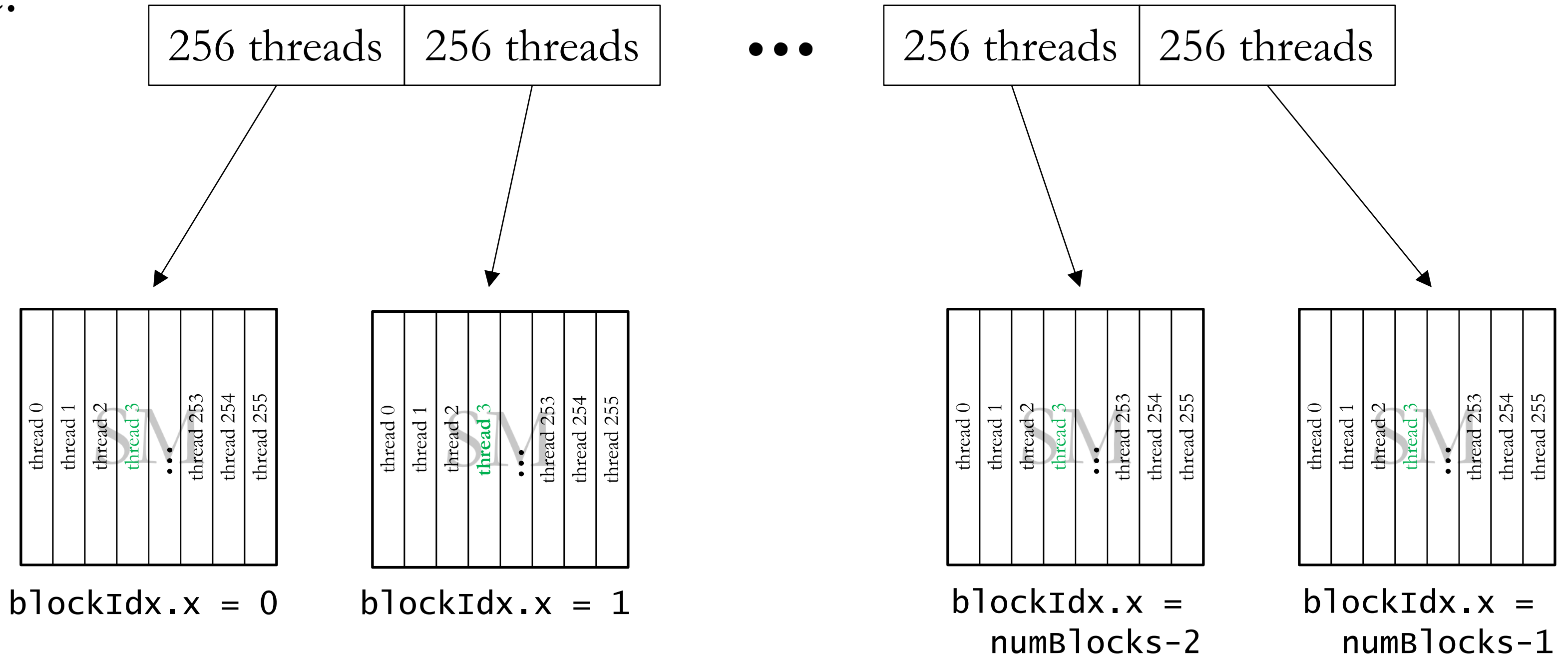
- a thread's position inside of its thread block is given by `threadIdx.x`
- the number of threads in each block is given by `blockDim.x`
- the position of a thread block inside the set of all blocks is given by `blockIdx.x`
- and the total number of blocks is given by `gridDim.x`

We're using just the x component of each of these for now

- since the vector addition is a 1-D problem

Thread Position

So for example:

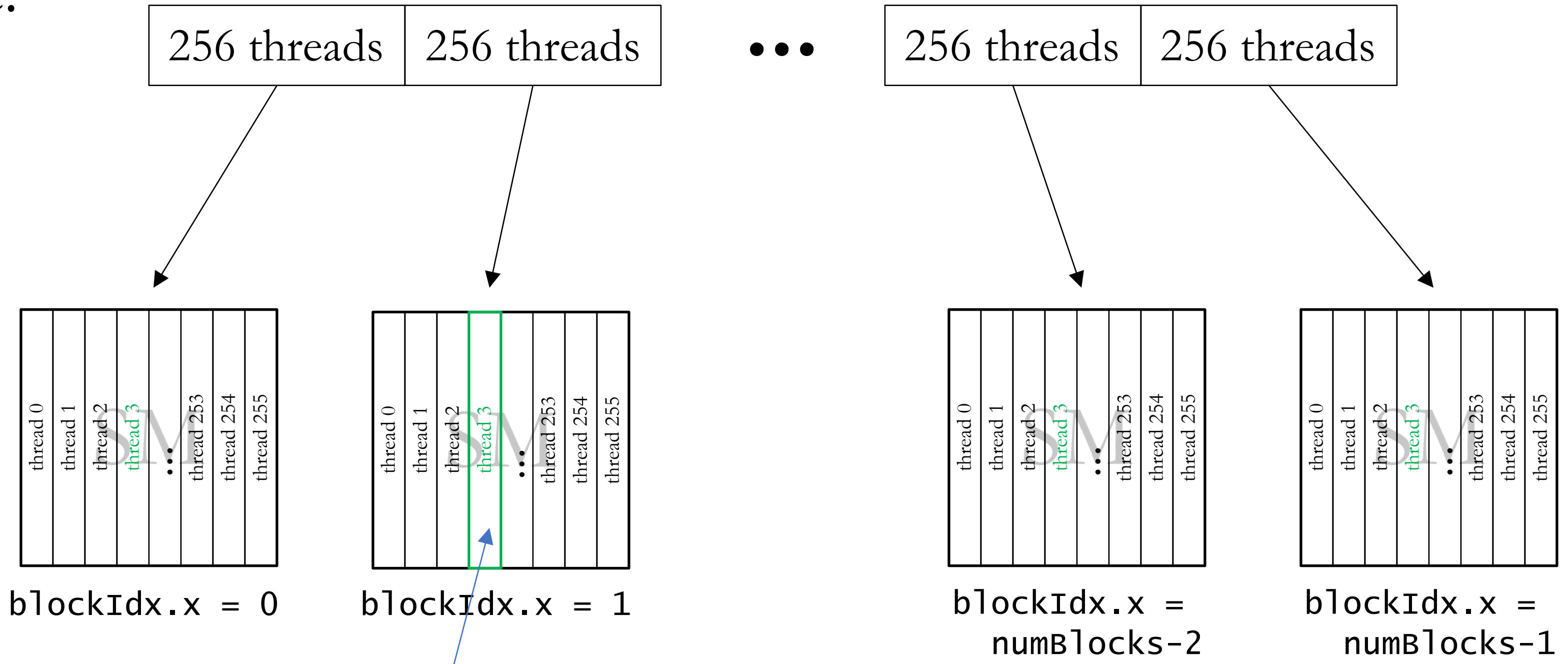


And for the threads in green, `threadIdx.x = 3`

- and in general, the global position of a thread is `blockIdx.x * blockDim.x + threadIdx.x`

Thread Position

So for example:



And for the threads in green, `threadIdx.x = 3`

- and in general, the global position of a thread is `blockIdx.x * blockDim.x + threadIdx.x`
- thus, the global position of the thread with the green outline is 1 * 256 + 3

Thread Position

So for the first thread:

- `blockIdx.x` will be 0
- `threadIdx.x` will be 0
- and so the global position of the thread will be `blockIdx.x * blockDim.x + threadIdx.x = 0`

And for the last thread:

- `blockIdx.x` will be the `gridDim.x - 1`
- `threadIdx.x` will be `blockDim.x - 1`
- and so the global position of the thread will be `(gridDim.x - 1) * blockDim.x + blockDim.x - 1`
- which we can simplify to `gridDim.x * blockDim.x - 1`

This makes sense.

Thread Position

And so the global position of a thread

- `blockIdx.x * blockDim.x + threadIdx.x`

This is:

- where am I in my block? (`threadIdx.x`)
- how many threads are in a block? (`blockDim.x`)
- which block am I in? (`blockIdx.x`)

Back to the Kernel

Now that we know the global position of each thread, we can have each thread do one of the scalar additions in the vector addition:

```
__global__ void add( int *X, int *Y, int *Z, int N ) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  Z[tid] = X[tid] + Y[tid];
}
```

And so in this way:

- each thread will have a unique `tid` value representing its position in the computation
- and the threads in a thread block will access adjacent array elements: this will give us efficient cache use by the SMs (we'll talk about the memory hierarchy in more detail soon)

Generalizing the Computation

The global position of each thread is given by:

```
tid = blockIdx.x * blockDim.x + threadIdx.x
```

Two natural questions:

[what are the two natural questions?]

Generalizing the Computation

The global position of each thread is given by:

```
tid = blockIdx.x * blockDim.x + threadIdx.x
```

Two natural questions:

1. what if there are more threads than there are vector elements?
2. what if there are more vector elements than there are threads?

Generalizing the Computation

Question #1: What if there are more threads than there are vector elements?

- this one is easy to take care of
- in the kernel loop, we can add a check:

```
__global__ void add( int *X, int *Y, int *Z, int N ) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < N)
    Z[tid] = X[tid] + Y[tid];
}
```

And the predicate registers will still enable parallel computation

Same Example, with Very Large Vectors

Suppose that N is huge; say $2^{26} = 67108864$

With 256 threads per thread block, we would need $2^{26}/2^{8} = 2^{18} = 262144$ blocks

But there is a limitation of the GPU hardware:

- the maximum number of thread blocks is 65535[1]

Implication?

- each thread will have to process more than a single element

[1]at some point in time; again, Nvidia is constantly iterating on their hardware and improving it

Grid-Stride Loop

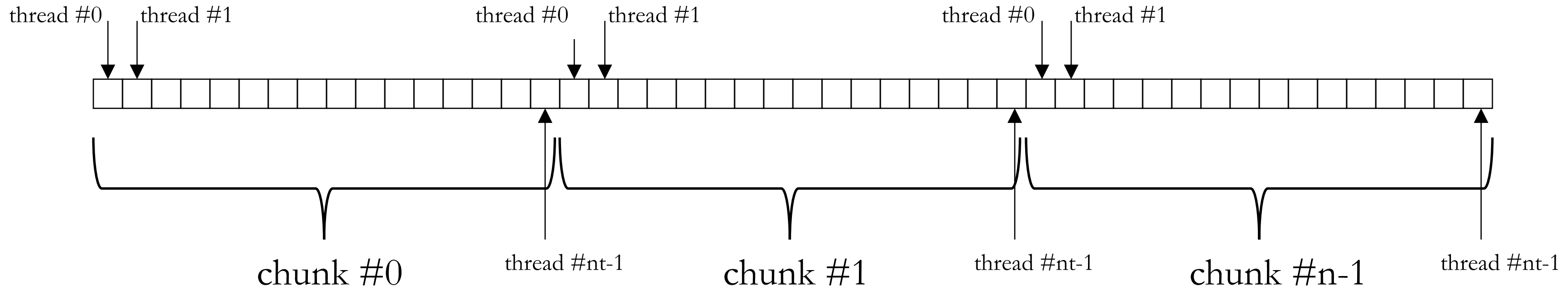
Question #2: what if there are more vector elements that threads?

- this one is a little trickier: each thread will have to process more than one element

The straightforward way to do this is to divide the vector into uniform chunks

- and the size each chunk will be given by the total number of threads (call it `nt` here)
- the total number of threads is `gridDim.x * blockDim.x`: this is the size of a chunk—it's actually the grid size

Then, I have each thread stride through the chunks



Grid-Stride Loop

The stride through the vector is then given by:

```
stride = gridDim.x * blockDim.x; // this is the total #threads
```

And having each thread stride through in this way is called a **grid-stride loop**

```
__global__
void add( int *X, int *Y, int *Z, int N ) {
  int stride = blockDim.x * gridDim.x;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i = i + stride)
    Z[i] = X[i] + Y[i];
}
```

Grid-Stride Loop

This is the most general and efficient way to organize the threads to do a large, parallel computation

- scalability: this supports any problem size and lets us change the number of blocks without changing the code
- makes debugging easy—we can launch one block with one thread (`<<<1, 1>>>`) without changing the code
- increases portability and readability—doesn't rely on the specific configuration of a particular machine

Here's a good blog:

- https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

Grid-Stride Loop: Example

An example where the #elements is greater than the #threads

Suppose that:

- #blocks is 4 (`gridDim.x` = 4)
- #threads per block is 16 (`blockDim.x` = 16)
- N = 512

So here, the chunk size (the grid size) here will be 64: `gridDim.x * blockDim.x`

Grid-Stride Loop: Example

An example where the #elements is greater than the #threads

Suppose that:

- #blocks is 4 (`gridDim.x` = 4)
- #threads per block is 16 (`blockDim.x` = 16)
- N = 512

So there will be 64 total threads, and:

- thread #0 will hit elements 0, 64, 128, 192, 256, 320, 384, 448
- thread #1 will hit elements 1, 65, 129, 193, 257, 321, 385, 449
- thread #2 will hit elements 2, 66, 130, 194, 258, 322, 386, 450
- ...
- thread #63 will hit elements 63, 127, 191, 255, 319, 383, 447, 511