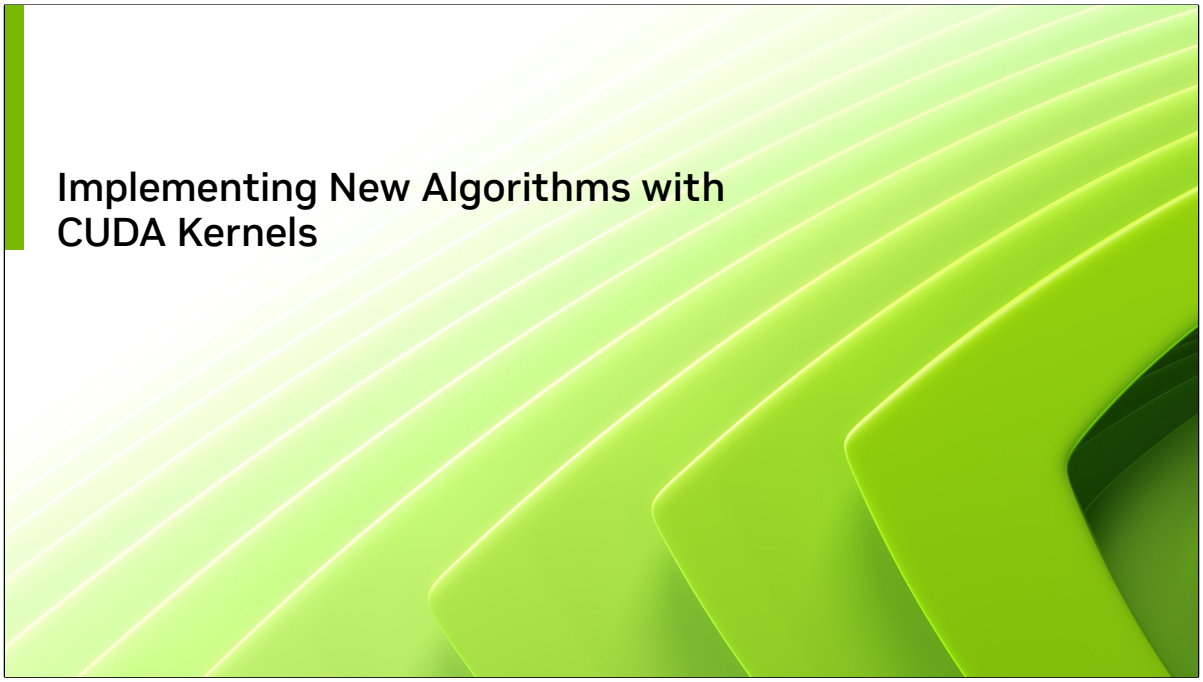
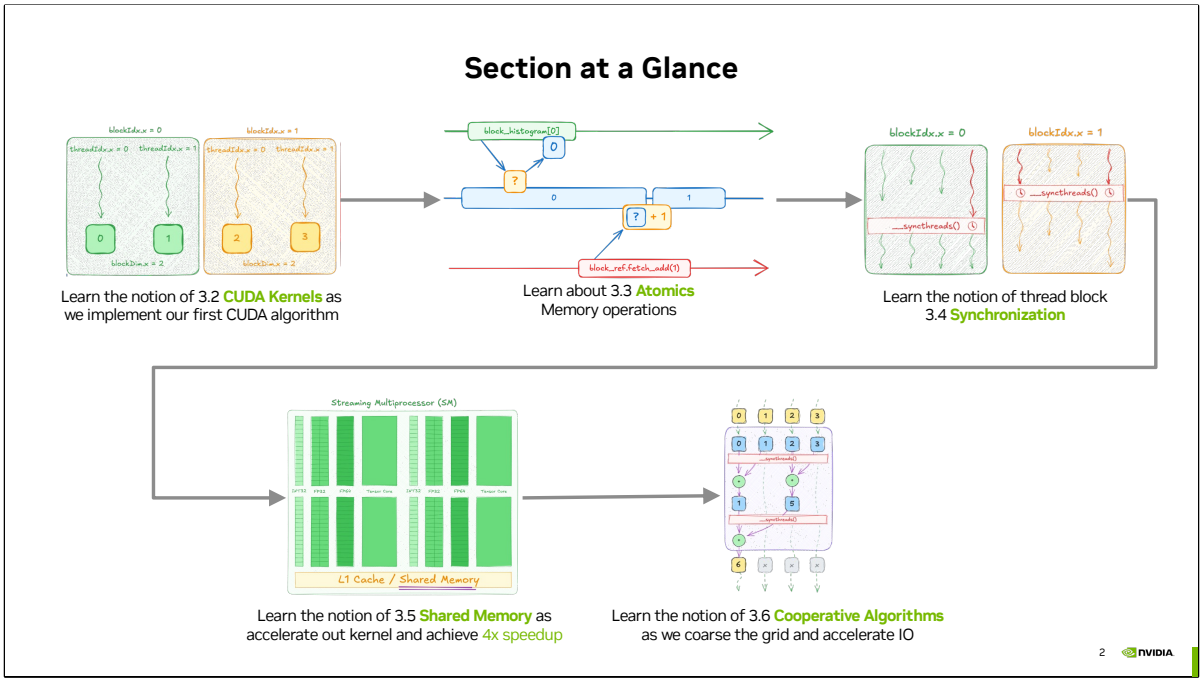


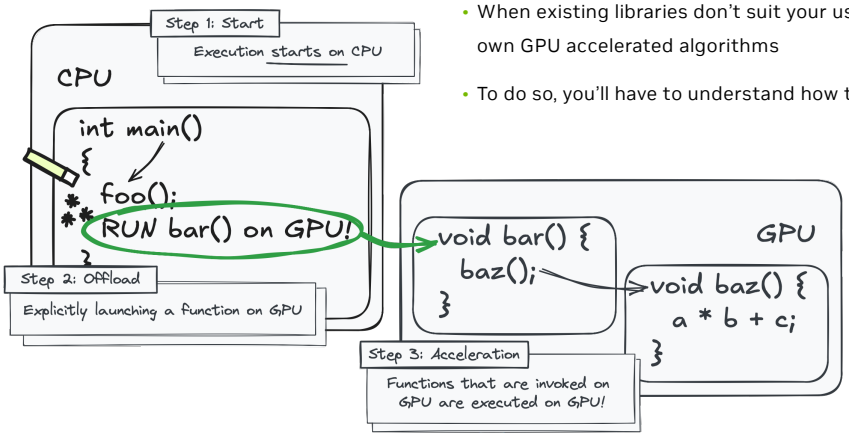
In the previous section, we explored how asynchronous parallel algorithms can further enhance performance. But what if there's no parallel algorithm for your specific use case, or we can't adapt existing ones with fancy iterators? In this section, we'll discuss your options for handling exactly that scenario.



We'll begin by exploring how to program GPUs directly, without relying on parallel algorithms. Next, we'll look at a few specialized vocabulary types that help us write correct and efficient concurrent programs. After that, we'll explore some hardware features that can significantly improve performance of our custom code. Finally, we'll discuss libraries that make low-level GPU programming much simpler.

CUDA Kernels

- CUDA libraries do not have magic powers to invoke code on GPU
- When existing libraries don't suit your use case, you'll have to write your own GPU accelerated algorithms
- To do so, you'll have to understand how to launch work on GPU directly

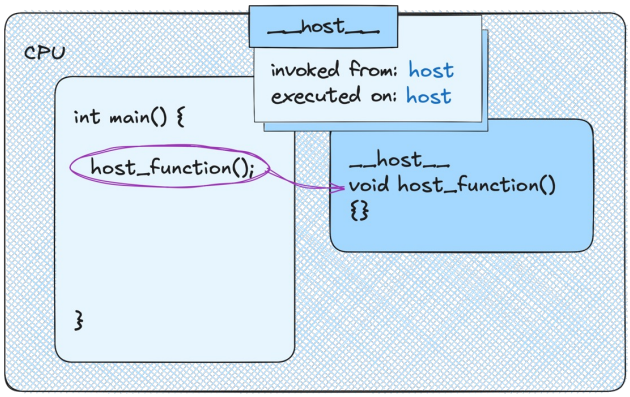


3 NVIDIA

Remember, the CPU typically calls host functions, while the GPU calls device functions. Invoking GPU work from the CPU requires a special mechanism. So far, parallel algorithms handled this 'magic' for us, seamlessly calling GPU code behind the scenes. Now that we want to implement our own algorithms without relying on those libraries, we need to understand how to launch GPU code ourselves.

NVIDIA

CUDA Kernels

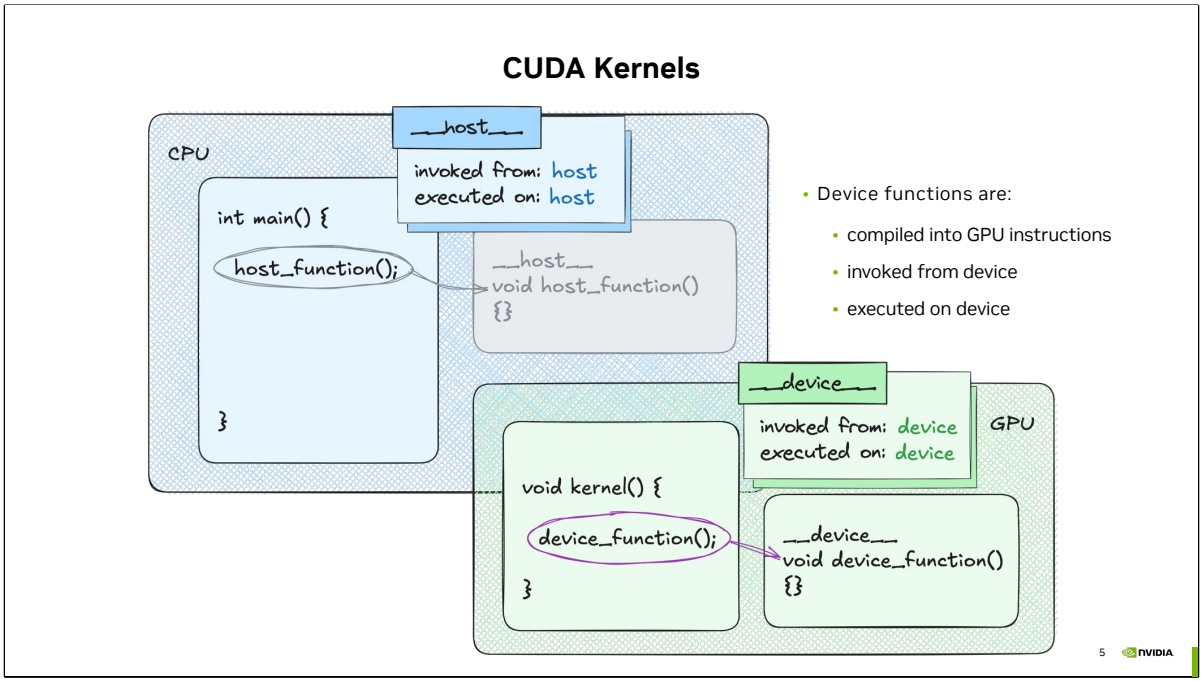


- All functions are implicitly `__host__`
- Host functions are:
  - compiled into CPU instructions
  - invoked from host
  - executed on host

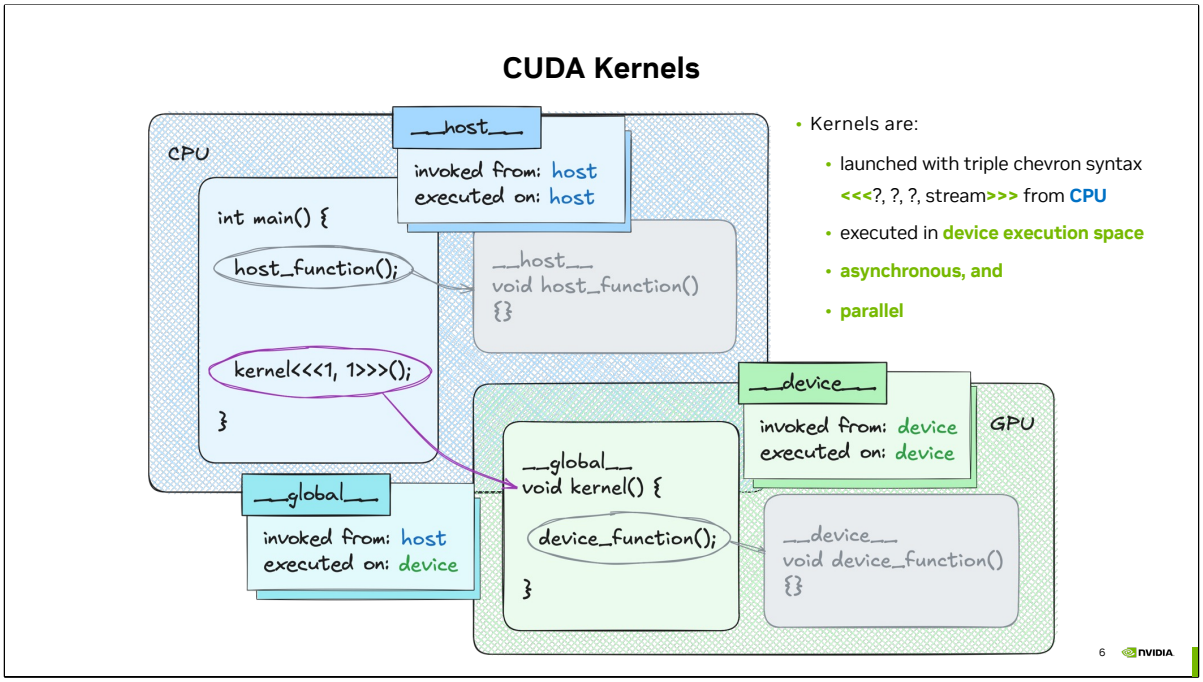
4 NVIDIA

But before diving in, let's quickly recap the execution space specifiers we've encountered so far. Host functions are compiled into CPU instructions, called from the host, and executed on the host.

NVIDIA



Similarly, device functions are compiled into GPU instructions, called from the device, and executed on the device.



There's another annotation we haven't discussed yet. When you mark a function with __global__, it's compiled into GPU instructions and executed on the GPU - but it's called from the CPU! These functions are known as CUDA kernels. Calling a CUDA kernel from the CPU looks unusual because of the triple angle brackets with some numbers inside. We'll talk about what those numbers mean later in the course. Another important detail is that kernel calls are asynchronous: much like calling a CUB algorithm. The CPU doesn't wait for the kernel to finish on the GPU.

Current Code

```
using temperature_grid_f =
    cuda::std::mdspan<float, cuda::std::dextents<int, 2>>;

void simulate(temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  auto ids = thrust::make_counting_iterator(0);
  cub::DeviceTransform::Transform(
    ids, out, in.size(),
    [in] __host__ __device__(int cell_id) {
      return ach::compute(cell_id, in);
    }, stream);
}
```

- Currently, the simulator is:
  - using `cuda::std::mdspan` to view data as 2D
  - Using CUB to transform cell indices into the next temperatures
- Under the hood, CUB:
  - invokes kernels on provided stream
  - doesn't synchronize the stream,
  - which makes CUB asynchronous
- Let's try implementing simulator without CUB!

7 NVIDIA

Before we start writing our own CUDA kernels, let's recap how our simulator currently works. We use an mdspan for a 2D view of the temperature grid. Then, we transform cell indices into new temperatures with CUB device transform. Under the hood, CUB implements device transform using CUDA kernels, which is why they run asynchronously. To see how CUDA kernels actually work, it's best to start with code we're already familiar with. So, let's try rewriting our simulator to launch kernels directly, rather than relying on CUB.

NVIDIA

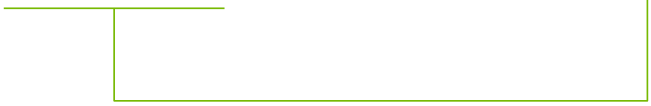
CUDA Kernels

```
__global__
void single_thread_kernel(ach::temperature_grid_f in, float *out)
{
  for (int id = 0; id < in.size(); id++)
  {
    out[id] = ach::compute(id, in);
  }
}
```

Triple chevron accepts four parameters:

- You know what stream is at this point
- we'll go over remaining parameters one at a time

```
void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  single_thread_kernel<<<1, 1, 0, stream>>>(in, out);
}
```



8 NVIDIA

So, let's do exactly that. On this slide, you'll see a simple CUDA kernel that iterates over all cells and transforms their indices into new temperatures. Notice the triple-chevron syntax when launching it: the fourth parameter is a CUDA stream. By passing a stream here, we can run this computation asynchronously on the specified stream.

CUDA Kernels

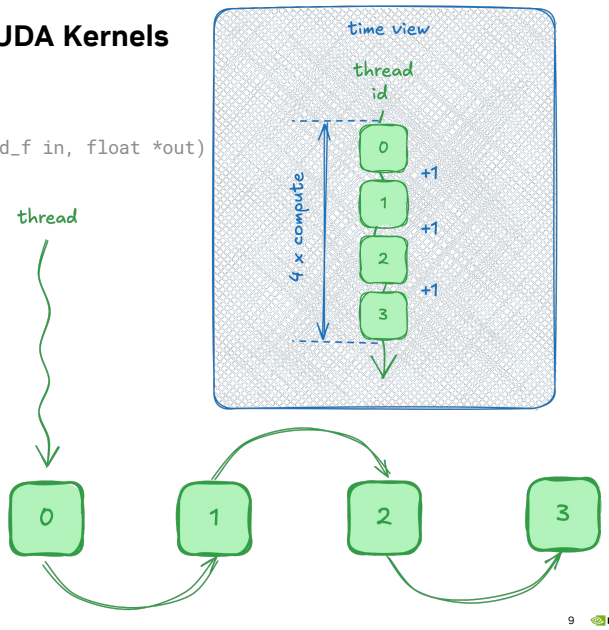
```
__global__
void single_thread_kernel(ach::temperature_grid_f in, float *out)
{
  for (int id = 0; id < in.size(); id++)
  {
    out[id] = ach::compute(id, in);
  }
}
```

As we saw previously, GPUs do not auto-magically parallelize your code:

- Single-threaded code remains serial
- This makes our kernel incredibly slow

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15



9 NVIDIA

But as you already know, the GPU doesn't magically parallelize code on its own! This means, that a single GPU thread is computing each cell in sequence. Overall, this makes our kernel roughly ten thousand times slower than CUB.

NVIDIA

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}

void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  block_kernel<<<1, number_of_threads, 0, stream>>>(in, out);
}
```

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15
- 2 threads: 2.6

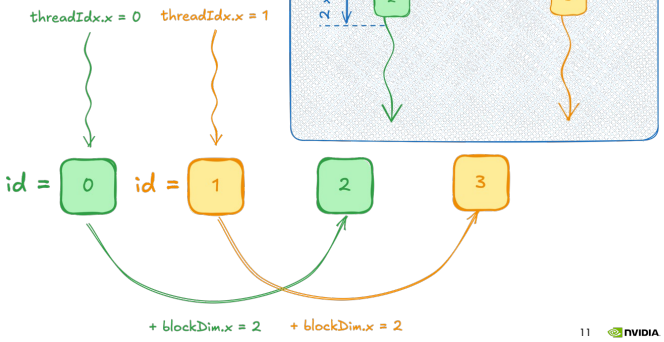
Fortunately, kernels are inherently parallel, so we can increase the number of threads. For instance, to launch two threads, I pass 2 as the second parameter in the triple-chevron syntax. You can see on the right that doing so nearly doubles the kernel's speed. But simply launching more threads isn't enough - if they all process the same cells, there's no actual speedup. To fix this, we need to modify the kernel so each thread can distinguish itself and handle a different subset of cells.

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}
```

CUDA extends device code with a few variables:

- `threadIdx.x` stores index of the current thread



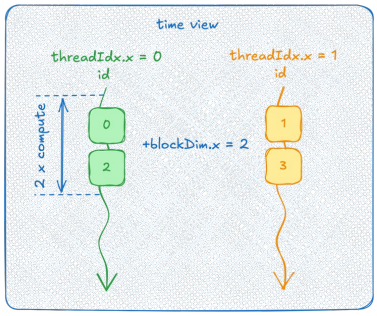
11 NVIDIA

That's where threadIdx.x comes in. threadIdx.x is a built-in variable available in CUDA kernels. It holds the index of the current thread. So, if we launch two threads, threadIdx.x will be equal to 0 in one thread and 1 in another.

NVIDIA

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}
```



Same kernel is executed by multiple GPU threads

```
threadIdx.x = 0
int thread_index = 0;
out[0] = ach::compute(0, in);
out[2] = ach::compute(2, in);
```

```
threadIdx.x = 1
int thread_index = 1;
out[1] = ach::compute(1, in);
out[3] = ach::compute(3, in);
```

12 NVIDIA

With this approach, each thread processes a different subset of cells. In the kernel, we first read threadIdx.x to identify the current thread's index. Then, each thread starts by processing the cell whose index matches its own thread index. That means that the first thread works on cell 0, and the second thread works on cell 1. After finishing one cell, we move ahead by adding the total number of threads to the cell index. This way, the first thread takes cell 2 next, while the second thread moves on to cell 3, and so on.

NVIDIA

CUDA Kernels

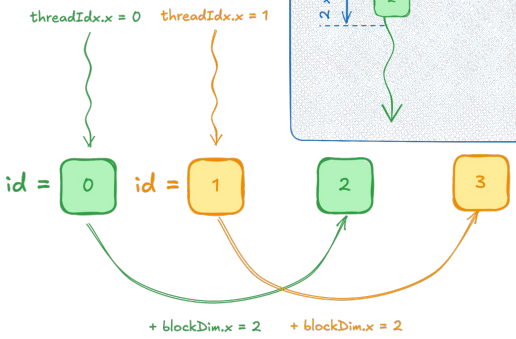
```
const int number_of_threads = 2;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}
```

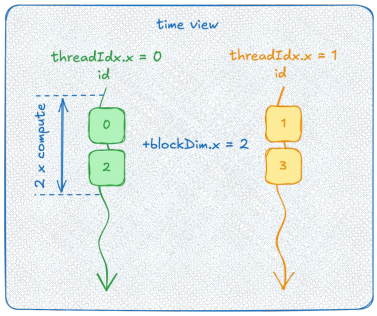
CUDA extends device code with a few variables:

- `threadIdx.x` stores index of the current thread

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15
- 2 threads: 2.6





13 NVIDIA

Notice how this scheme allows us to to compute all cells in parallel such that one cell is never processed by multiple threads. Another observation is that there's nothing specific in our code that would limit it to only two threads. So, let's try increasing the number of threads working on this CUDA kernel.

NVIDIA

CUDA Kernels

```
const int number_of_threads = 256;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}

void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  block_kernel<<<1, number_of_threads, 0, stream>>>(in, out);
}
```

- Scheme works!
- More threads process input faster
- Let's just keep adding them

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15
- 2 threads: 2.6
- 256 threads: 0.037

14 NVIDIA

Here, I've increased the number of threads to 256, and this scheme still works great. Our performance has improved significantly, but we're still nowhere near CUB's speed. Let's see if adding even more threads can bridge this gap.

CUDA Kernels

```
const int number_of_threads = 2048;

__global__
void block_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = threadIdx.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}

void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  block_kernel<<<1, number_of_threads, 0, stream>>>(in, out);
}
```

The error is expected:

- You cannot launch more than 1024 threads
- … in a thread block, which is a HW limit

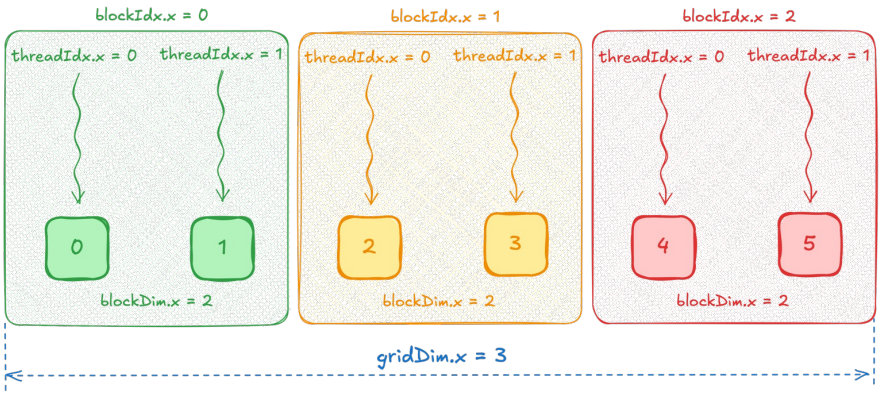
```
terminate called after throwing an instance of
'std::runtime_error'
  what():  invalid configuration argument
```

15 NVIDIA

Now I've tried launching 2048 threads, and suddenly everything fails with a cryptic 'invalid configuration argument' error. As usual, this means there's a gap in our understanding. It turns out that there's a limit on the number of threads.. in a single block. So, what exactly is a thread block, and why does it matter for our kernel launch?

Missing Context

- threads are grouped in blocks
- all blocks are of the same size (max 1024 threads)
- thread indexing is local within a thread block
- collection of blocks is called grid
- `blockIdx.x` stores index of current block within
- `blockDim.x` stores the number of blocks in the grid
- `gridDim.x` stores the number of blocks in the grid



16 NVIDIA

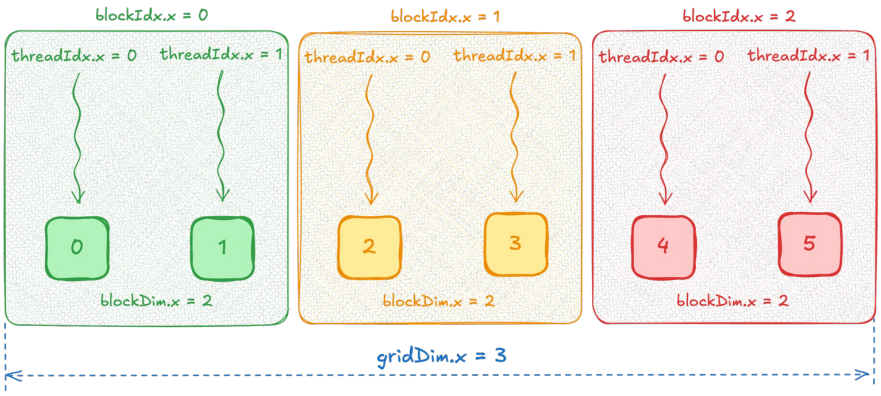
In CUDA, threads are grouped into blocks, and the entire collection of these blocks is called a grid. As we've seen already, each block can contain up to 1024 threads, and every block in a single kernel launch has the same size. The threadIdx.x variable we've seen so far stores the thread's local index within its block. For example, if we have three blocks, each block will have a thread with threadIdx.x set to 0, and another thread with threadIdx.x set to 1. That means we end up with three threads labeled 0, three labeled 1, and so forth. And just as we used threadIdx.x to distinguish threads within a block, we also need a way to identify which block we're in. To do that, CUDA provides additional built-in variables.

NVIDIA

CUDA Kernels

```
__global__
void grid_kernel(...)
{
  int number_of_threads = blockDim.x * gridDim.x;
  ...
```

- collection of blocks is called grid
- `blockIdx.x` stores index of current block within
- `blockDim.x` stores the number of blocks in the grid
- `gridDim.x` stores the number of blocks in the grid



17 NVIDIA

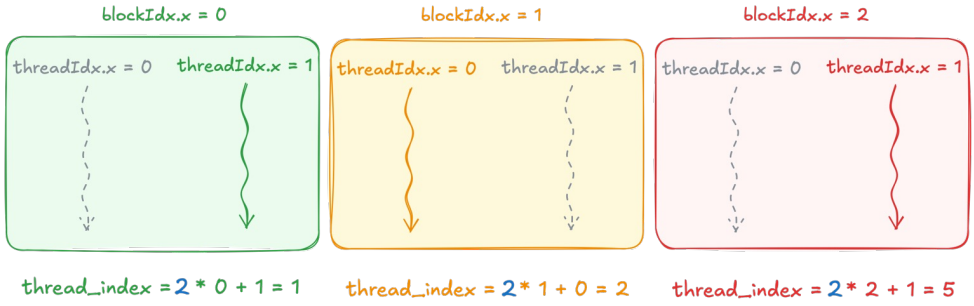
blockDim.x tells us how many threads are there in each block. In the example on the slide, every thread sees 2 for blockDim.x. Meanwhile, gridDim.x gives the number of blocks in the grid. So, if gridDim.x is 3, that means there are 3 blocks total. By multiplying blockDim.x by gridDim.x we can get the total number of threads in the grid. For the example on the slide, 2 * 3 gives us 6 threads in total.

NVIDIA

CUDA Kernels

```
__global__
void grid_kernel(...)
{
  int thread_index = blockDim.x * blockIdx.x
                   + threadIdx.x;
  ...
```

- collection of blocks is called grid
- `blockIdx.x` stores index of current block within
- `blockDim.x` stores the number of blocks in the grid
- `gridDim.x` stores the number of blocks in the grid



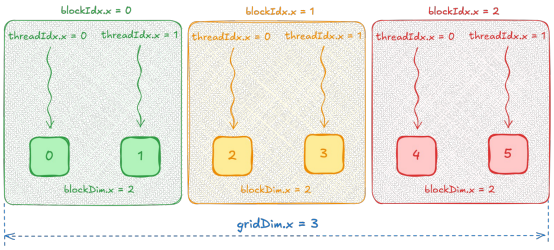
18 NVIDIA

Besides knowing the total number of threads, we also need each thread's unique index across the entire grid. For that, we use another built-in variable, called blockIdx.x, which holds the block's index within the grid. All threads in block 0 see blockIdx.x = 0 while all threads in block 1 see 1, and so on. To calculate a thread's global index, we multiply blockIdx.x by blockDim.x to find out how many threads are there before the current block. Then we add threadIdx.x, which is the thread's local index in its block. For example, the second thread of the first block multiplies 2 by 0, then adds 1 for its local index, giving a global index of 1. Same computation results in index 5 for the last thread of the last block.

NVIDIA

Which Block Size to Use?

- Thread block size doesn't depend on problem size
- There's no block size that fits all the kernels
- As a rule of thumb:
  - Use block sizes that are multiple of **32**
  - Use **256** as a good default
  - Profile for further tuning




```
void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  int block_size = 256;
  ...

  grid_kernel<<<grid_size, block_size, 0, stream>>>(in, out);
}
```

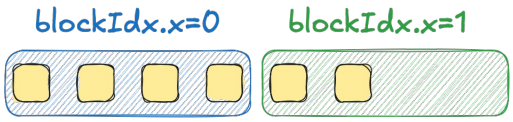
19 NVIDIA

So now we know how to find each thread's unique index within the grid, and we also know the total number of threads. But a few questions remain. For example, what block size should we use? Unfortunately, there's no one-size-fits-all answer. This often comes down to optimization. One helpful guideline is to pick block sizes in multiples of 32, because of how the GPU hardware groups threads. A common default is 256 threads per block, which we'll use throughout this course.

NVIDIA

Which Grid Size to Use?

- We need the number number of thread blocks (grid size)
- Grid size frequently depend on problem size
- As a rule of thumb, use `cuda::ceil_div` to compute grid size




```
void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  int block_size = 256;
  int grid_size = cuda::ceil_div(in.size(), block_size);

  grid_kernel<<<grid_size, block_size, 0, stream>>>(in, out);
}
```

```
int ceil_div(int a, int b)
{
  return (a + b - 1) / b;
}
```


The second question is how many blocks we should launch. Because we typically want to maximize parallelism, the grid size often depends on the problem size. For example, if we want each thread to handle exactly one element, a simple integer division won't work. Imagine that we have 6 cells to process and choose a block size of 4 threads. If we divide 6 by 4, we get 1, which means only one block is launched and some threads end up with extra cells. Instead, we can use `cuda::ceil_div` to perform an integer division that rounds up. Applying cuda::ceil_div gives us 2, ensuring enough blocks so each thread processes exactly one cell. To specify that we want two thread blocks, it's sufficient to pass two as the first parameter of the triple chevron.

CUDA Kernels

```
__global__
void grid_kernel(ach::temperature_grid_f in, float *out)
{
  int thread_index = blockDim.x * blockIdx.x + threadIdx.x;
  int number_of_threads = blockDim.x * gridDim.x;

  for (int id = thread_index; id < in.size(); id += number_of_threads)
  {
    out[id] = ach::compute(id, in);
  }
}

void simulate(ach::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  int block_size = 256;
  int grid_size = cuda::ceil_div(in.size(), block_size);

  grid_kernel<<<grid_size, block_size, 0, stream>>>(in, out);
}
```

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15
- 2 threads: 2.6
- 256 threads: 0.037
- 5'120'000 threads: 0.0003

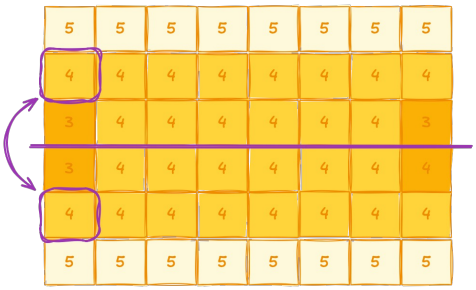
21 NVIDIA

With these changes, we're now launching around five million threads and finally getting close to CUB's performance. Of course, CUB still has more advanced optimizations under the hood, so fully matching its speed is unlikely. But for such a simple kernel, this is already a significant improvement.

Exercise: Detect Asymmetry

- Boundary conditions are symmetric
- Remaining cells should be symmetric as well
- Rewrite the function below as a kernel to verify that

```
void symmetry_check(ach::temperature_grid_f temp, int row)
{
  int column = 0;

  float top = temp(row, column);
  float bottom = temp(temp.extent(0) - 1 - row, column);
  float diff = abs(top - bottom);
  if (diff > 0.1) {
    printf("Error: asymmetry in %d\n", column);
  }
 }
```



03.02-Kernels/03.02.02-Exercise-Symmetry.ipynb

22 NVIDIA

We've modified our simulator but haven't yet confirmed that it still works correctly. Because our setup is symmetric - temperatures at the top and bottom of the grid are identical. So, we can do a quick 'smoke test' by checking that symmetry. The idea is simple: for a given cell, we find its mirrored row by subtracting row's index from the grid's height. If everything is correct, temperatures of these symmetric cells should be nearly the same. On the screen, you'll see a C++ function that performs this check. As part of this exercise, you'll have to turn this function into a CUDA kernel and launch it with the triple-chevron syntax.

NVIDIA

Exercise: Detect Asymmetry

Solution

```
__global__
void symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = 0;

  float top = temp(row, column);
  float bottom = temp(temp.extent(0) - 1 - row, column);
  float diff = abs(top - bottom);
  if (diff > 0.1) {
    printf("Error: asymmetry in %d\n", column);
  }
}

void symmetry_check(ach::temperature_grid_f temp, cudaStream_t stream)
{
  int target_row = 0;
  symmetry_check_kernel<<<1, 1, 0, stream>>>(temp, target_row);
}
```

Annotate function with `__global__` specifier

03.02-Kernels/03.02.02-Exercise-Symmetry.ipynb

23 NVIDIA

To turn the symmetry-check function into a CUDA kernel, we have to annotate it with __global__ specifier.

Exercise: Detect Asymmetry

Solution

```
__global__
void symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = 0;

  float top = temp(row, column);
  float bottom = temp(temp.extent(0) - 1 - row, column);
  float diff = abs(top - bottom);
  if (diff > 0.1) {
    printf("Error: asymmetry in %d\n", column);
  }
}

void symmetry_check(ach::temperature_grid_f temp, cudaStream_t stream)
{
  int target_row = 0;
  symmetry_check_kernel<<<1, 1, 0, stream>>>(temp, target_row);
}
```

Use triple chevron to launch the kernel

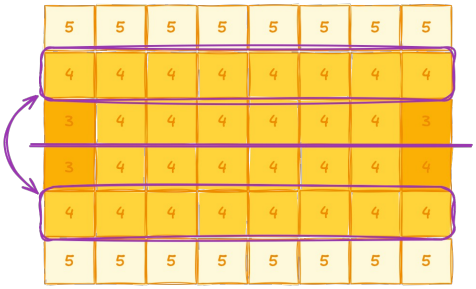
03.02-Kernels/03.02.02-Exercise-Symmetry.ipynb

After that, we need to launch it using the triple-chevron syntax. Because there's no real parallel work here, we'll launch just a single block with a single thread. To make sure this check runs on the correct stream, we specify the stream as the last parameter in the triple-chevron.

Exercise: Detect Row Asymmetry

- Entire row should be symmetric, not just one cell
- Rewrite the kernel below to verify that it's the case
- Assign each thread to verify exactly one column

```
__global__
void symmetry_check(ach::temperature_grid_f temp, int row)
{
  int column = 0;

  float top = temp(row, column);
  float bottom = temp(temp.extent(0) - 1 - row, column);
  float diff = abs(top - bottom);
  if (diff > 0.1) {
    printf("Error: asymmetry in %d\n", column);
  }
 }
```



03.02-Kernels/03.02.03-Exercise-Row-Symmetry.ipynb

Now, when you think about it, the entire row should be symmetrical, not just a single cell! In this exercise, you'll modify the kernel so that each thread handles exactly one column.

Exercise: Detect Row Asymmetry

Solution

```
__global__ void row_symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = blockIdx.x * blockDim.x + threadIdx.x;

  if (abs(temp(row, column) - temp(temp.extent(0) - 1 - row, column)) > 0.1)
  {
    printf("Error: asymmetry in %d / %d\n", column, temp.extent(1));
  }
}

void symmetry_check(ach::temperature_grid_f temp, cudaStream_t stream)
{
  int width      = temp.extent(1);
  int block_size = 256;
  int grid_size  = cuda::ceil_div(width, block_size);

  int target_row = 0;
  row_symmetry_check_kernel<<<grid_size, block_size, 0, stream>>>(temp, target_row);
}
```

03.02-Kernels/03.02.03-Exercise-Row-Symmetry.ipynb

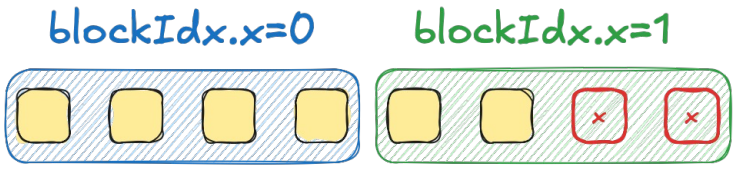
26 NVIDIA

As before, we use the block and thread indices to compute each thread's global index within the grid. Then, we calculate the necessary grid size with cuda::ceil_div and plug that into our triple-chevron launch. This way, each thread handles exactly one column, and the entire row is checked in parallel.

Exercise: Detect Row Asymmetry

Bug in the solution

```
__global__ void symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = blockIdx.x * blockDim.x + threadIdx.x;
```




```
  int width      = temp.extent(1);
  int block_size = 1024;
  int grid_size  = cuda::ceil_div(width, block_size);

  int target_row = 0;
  symmetry_check_kernel<<<grid_size, block_size, 0, stream>>>(temp, target_row);
}
```

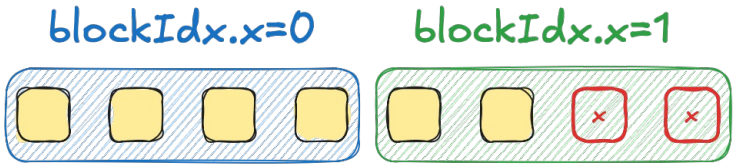
03.02-Kernels/03.02.03-Exercise-Row-Symmetry.ipynb

27 NVIDIA

And… we've just caused our first out-of-bounds access! Because we round up to make sure that each thread has at least one element, we might launch more threads than the actual problem size. In the case of six elements and two blocks of four threads, two last threads will access indices 6 and 7, which are out of range.

NVIDIA

Fixed Row Symmetry Check




```
__global__ void symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = blockIdx.x * blockDim.x + threadIdx.x;

  if (column < temp.extent(1))
  {
    if (abs(temp(row, column) - temp(temp.extent(0) - 1 - row, column)) > 0.1)
    {
        printf("Error: asymmetry in %d / %d\n", column, temp.extent(1));
    }
  }
}
```

Check if a given thread is within bounds of problem size

28 NVIDIA

We can fix this by adding a boundary check in the kernel. Specifically, after computing the global thread index, we compare it against the total number of columns and skip any work if it's outside of the valid range. But how would you discover the source of this out-of-bounds access on your own?

NVIDIA

Compute Sanitizer

```
nvcc --extended-lambda -g -G -o /tmp/a.out ./src/row_symmetry_check.cu

compute-sanitizer /tmp/a.out
```

```
========= Invalid __global__ read of size 4 bytes
=========     at symmetry_check_kernel(cuda::std::mdspan<>, int) in row_symmetry_check.cu:7
=========     by thread (905,0,0) in block (4,0,0)
=========     Address 0x7c5287588004 is out of bounds
=========     and is 5 bytes after the nearest allocation at 0x7c5286200000 of size 20,480,000 bytes
=========     Saved host backtrace up to driver entry point at kernel launch time
```

```
__global__ void symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = blockIdx.x * blockDim.x + threadIdx.x;

  if (abs(temp(row, column) - temp(temp.extent(0) - 1 - row, column)) > 0.1)
  {
    printf("Error: asymmetry in %d / %d\n", column, temp.extent(1));
  }
}
```

- One way to detect the bug would be compute sanitizer

29 NVIDIA

NVIDIA provides a special tool called compute-sanitizer that helps detect out-of-bounds accesses and other memory issues in your CUDA kernels. If you compile the simulator with the flags shown on the slide and then run it through compute-sanitizer, it will pinpoint the exact line where the illegal access occurs. You'll see messages like "invalid read of size 4 bytes in line 7," which make debugging kernels much simpler. However, there's also another way to spot this issue…

Assertions

```
#define CCCL_ENABLE_ASSERTIONS
#include <cuda/std/mdspan>


__global__ void row_symmetry_check_kernel(ach::temperature_grid_f temp, int row)
{
  int column = blockIdx.x * blockDim.x + threadIdx.x;

  if (abs(temp(row, column) - temp(temp.extent(0) - 1 - row, column)) > 0.1)
  {
    printf("Error: asymmetry in %d / %d\n", column, temp.extent(1));
  }
}
```

- Another way to detect the bug would be enabling assertions
- Runtime checks affect performance, so OOB access checks are disable by default
- You could enable them by defining `CCCL_ENABLE_ASSERTIONS` before including any header in debug

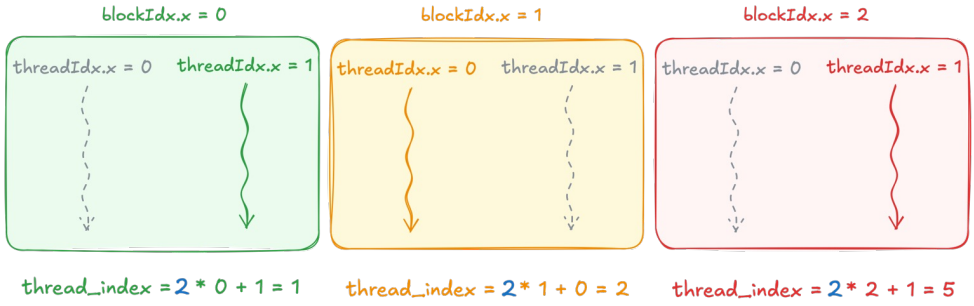
```
mdspan.h:134: __index: block: [0,0,0], thread: [3,0,0]
Assertion `cuda::std::mdspan subscript out of range!` failed.
```

30 NVIDIA

mdspan can actually detect out-of-bounds access on its own, though this feature is off by default to preserve performance. If you define the CCCL_ENABLE_ASSERTIONS macro before including mdspan, it will begin checking all indices. Then, if any out-of-bounds access occurs, mdspan will print an error message like the one you see on the screen.

Why Thread Hierarchy?

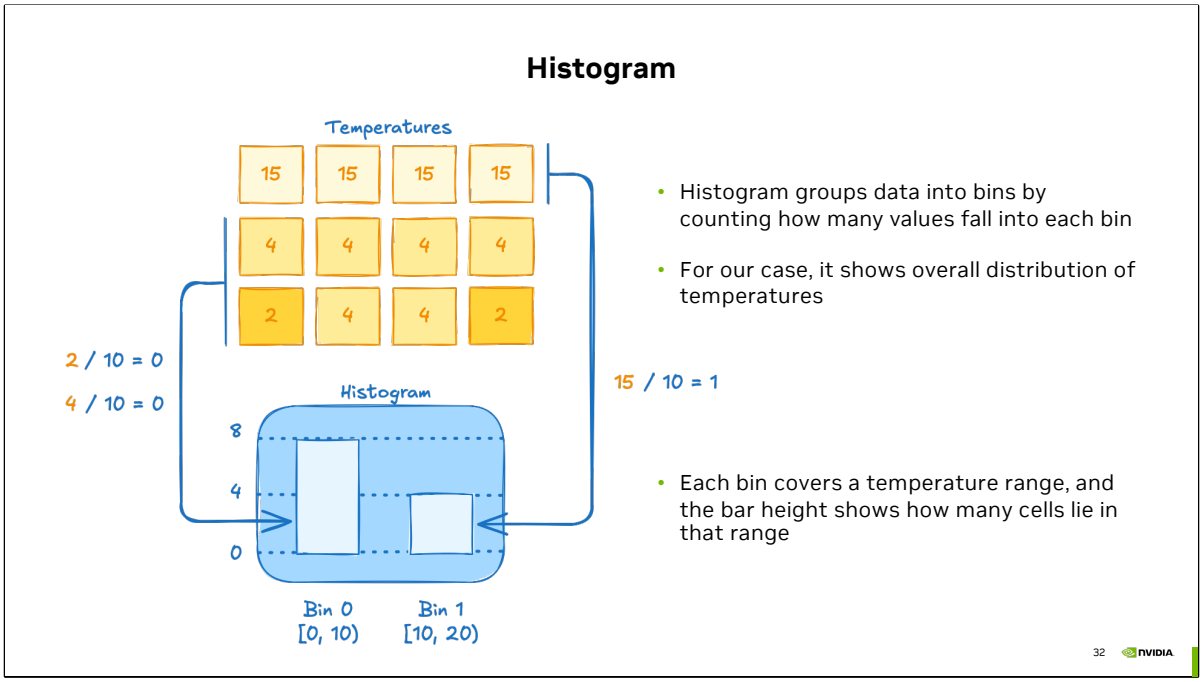
- We just fixed a bug caused by splitting grid into blocks
- It's reasonable to ask a question: why adding all this complexity with thread hierarchy?



31 NVIDIA

We've just fixed a bug caused by splitting threads into blocks. It's only natural to ask: why having all this complexity with thread hierarchy in the first place? To find out, let's modify our problem a bit to see where thread blocks truly make a difference.

NVIDIA



Let's now create a histogram for the temperatures that our simulator produces. A histogram helps us see how frequently certain temperature ranges occur, so let's quickly review how it works. First, we divide the entire temperature range into bins, each covering the same span - in our case, 10 degrees. For each cell in the simulation, we determine which bin its temperature falls into. A simple way to do this is to divide the temperature by the bin width and round down. For example, if a cell's temperature is 4 degrees, we divide by 10 and get 0, which corresponds to the first bin. If a cell's temperature is 15 degrees, dividing by 10 gives us 1, meaning it goes into the second bin. Once we've assigned each cell to the correct bin, we count how many cells landed in each bin. That count becomes the height of the bar in the histogram. In our example, eight cells fall into the first bin and four cells into the second bin. This final visual makes it easy to spot patterns and see which temperature ranges are most common in the simulation.

span

- `cuda::std::span` is like one-dimensional `cuda::std::mdspan`

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   histogram)
{
  ...
```

- Includes size information
- Safer alternative to raw pointers
- Reducing common pointer errors
- Lightweight view over a sequence
- Does not own the data, just references it

33 NVIDIA

Let’s begin by writing our kernel signature. Since our histogram only needs one dimension, mdspan would be an overkill. Instead, we can use span, which is essentially a one-dimensional mdspan. In CUDA, just like in C++, span is preferred over raw pointers because it’s safer, helps prevent common pointer errors, while remaining lightweight.

Interface

```
cuda::std::array<int, 4> a{1, 2, 3, 4};
cuda::std::span<int> span(a.data(), 3);

std::printf("span[0] = %d\n", span[0]); // 1
std::printf("span[1] = %d\n", span[1]); // 2

std::printf("span.size() = %zu\n", span.size()); // 3
```

constructor takes pointer and size

To construct a span, it's sufficient to pass it a raw pointer and a size.

Interface Recap

```
cuda::std::array<int, 4> a{1, 2, 3, 4};
cuda::std::span<int> span(a.data(), 3);

std::printf("span[0] = %d\n", span[0]); // 1
std::printf("span[1] = %d\n", span[1]); // 2

std::printf("span.size() = %zu\n", span.size()); // 3
```

provides subscript operator to access elements

Then, we can use square brackets to access underlying elements.

Interface

```
cuda::std::array<int, 4> a{1, 2, 3, 4};
cuda::std::span<int> span(a.data(), 3);

std::printf("span[0] = %d\n", span[0]); // 1
std::printf("span[1] = %d\n", span[1]); // 2

std::printf("span.size() = %zu\n", span.size()); // 3
```

returns the number of elements in the span

Span also provides a size() member function, which returns the total number of elements in the span.

First Attempt at Implementing Histogram Kernel

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
```

each thread bins exactly one cell

With both the temperatures and histogram passed into our kernel, we'll use already familiar thread-index computation scheme to assign exactly one cell per each thread

First Attempt at Implementing Histogram Kernel

```
constexpr float bin_width = 10;

__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);
```

divide temperature by bin width (10 degrees) to compute the bin index

- In production code, you should check if cell index is within bounds
- We'll omit this check for simplicity and make sure input is multiple of block size

Next, each thread loads its corresponding temperature value and divides it by the bin width to find the correct bin index. In a production environment, you'd also want to ensure that this memory access within bounds. For simplicity in the slides, we'll skip the boundary check and rely on problem sizes that are multiples of the block size instead.

First Attempt at Implementing Histogram Kernel

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  int old_count = histogram[bin];
  int new_count = old_count + 1;
  histogram[bin] = new_count;

}
```

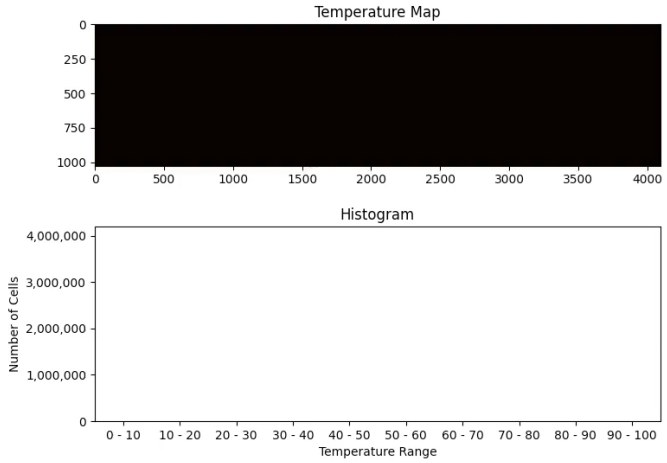
- Read old value of selected bin
- Modify the value we just read (increment)
- Write modified value back into memory

39 NVIDIA

Next, each thread reads the current count from its assigned bin, increments that value, and writes it back into the histogram.

NVIDIA

Something Went Wrong



- We have about 4M cells
- We should see bars adding up to 4M
- But resulting histogram looks empty
- What happened?

40 NVIDIA

But it looks like something went wrong. We have around four million cells in the grid, so we'd expect the histogram bars to sum to four million. But the histogram look empty. What happened?

NVIDIA

Source of the Data Race

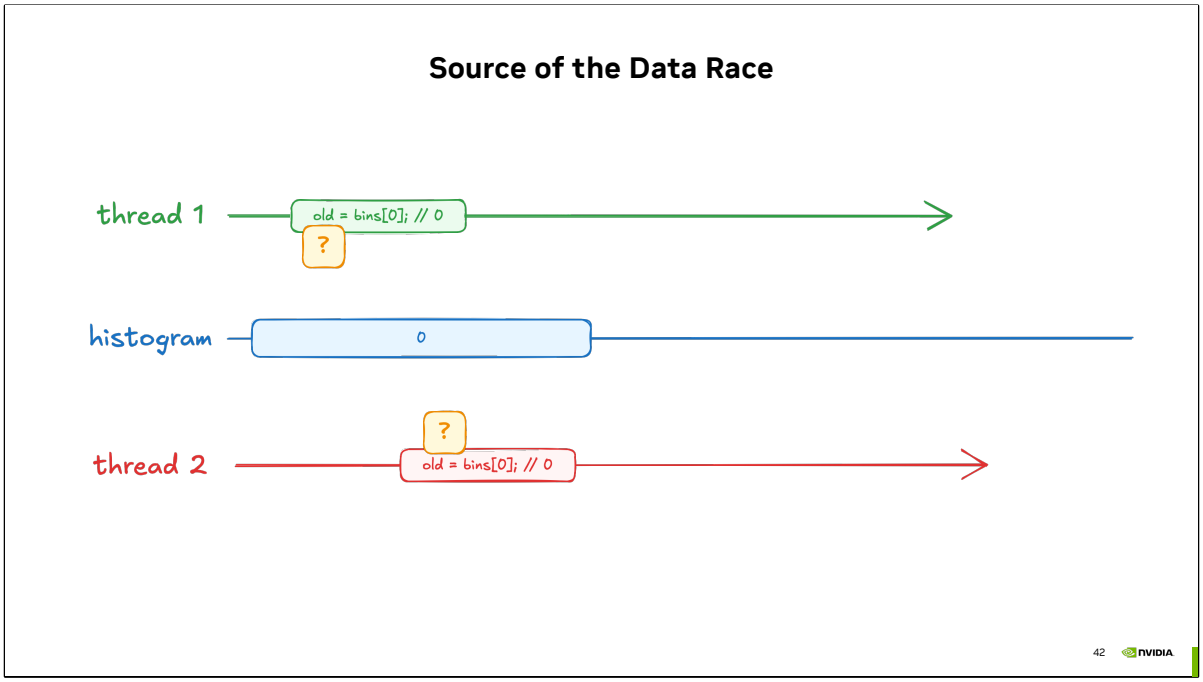
```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);
  int old_count = histogram[bin];
  int new_count = old_count + 1;
  histogram[bin] = new_count;
}
```

- Our kernel has a data race in it
- Millions of threads read and write the same memory location!

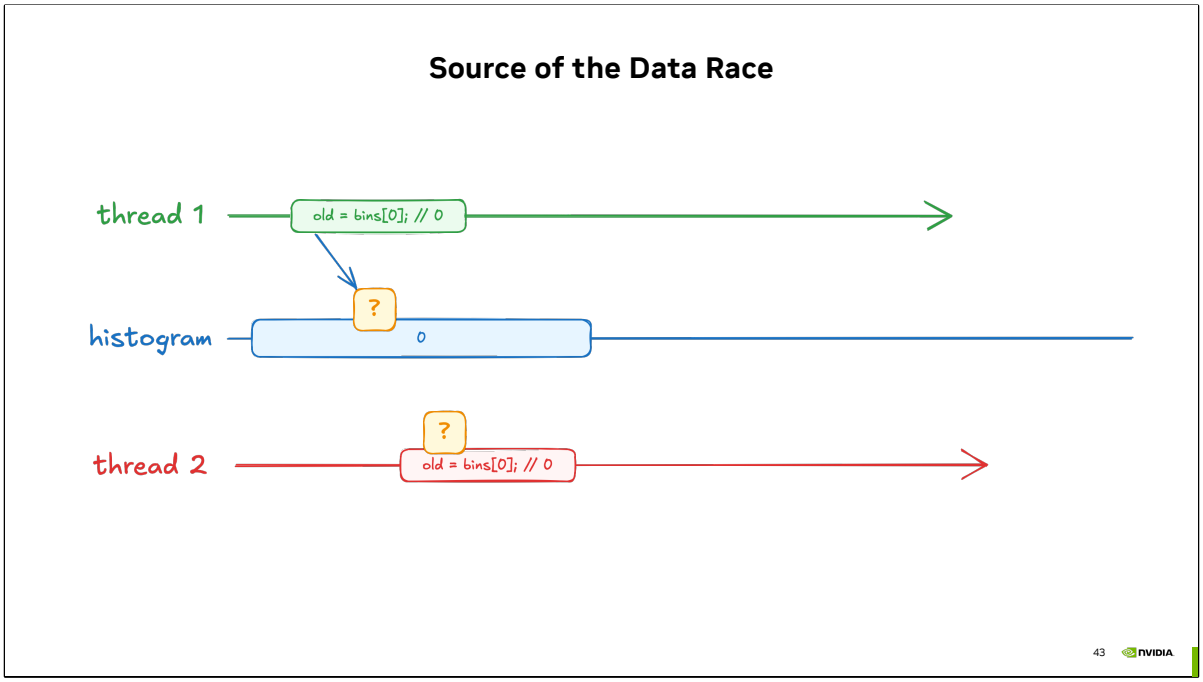
41 NVIDIA

The issue is that our kernel has a data race in the highlighted lines of code. All those millions of threads are reading and writing the same memory location. Let's break this down

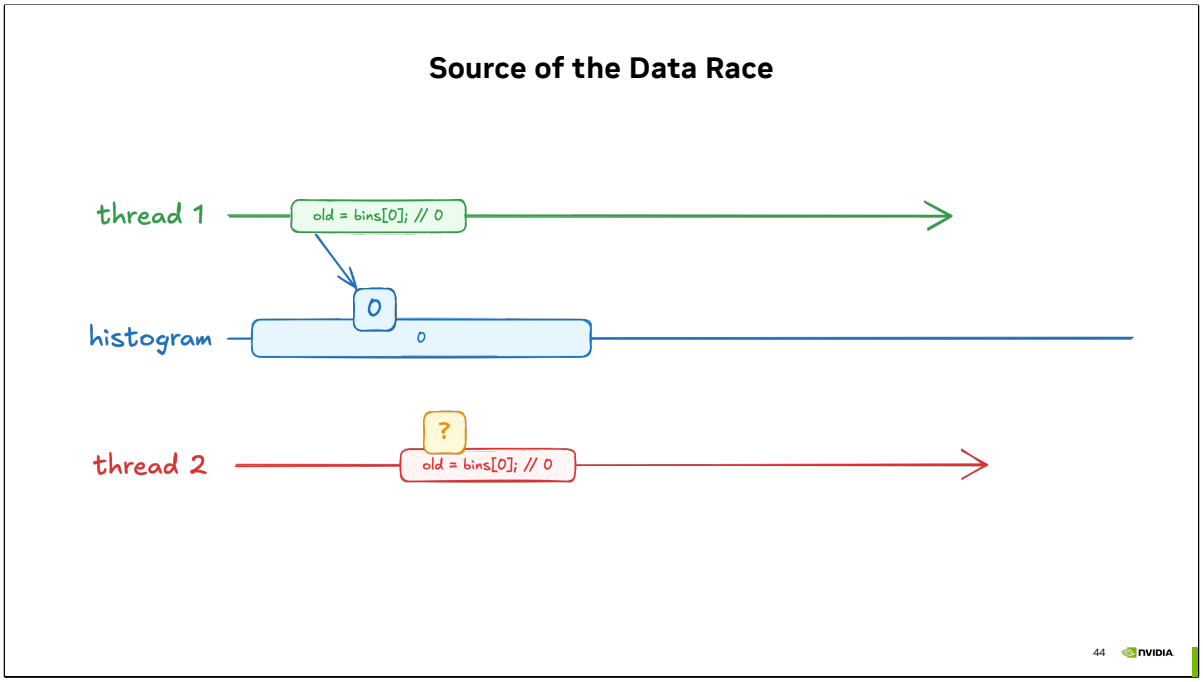
NVIDIA



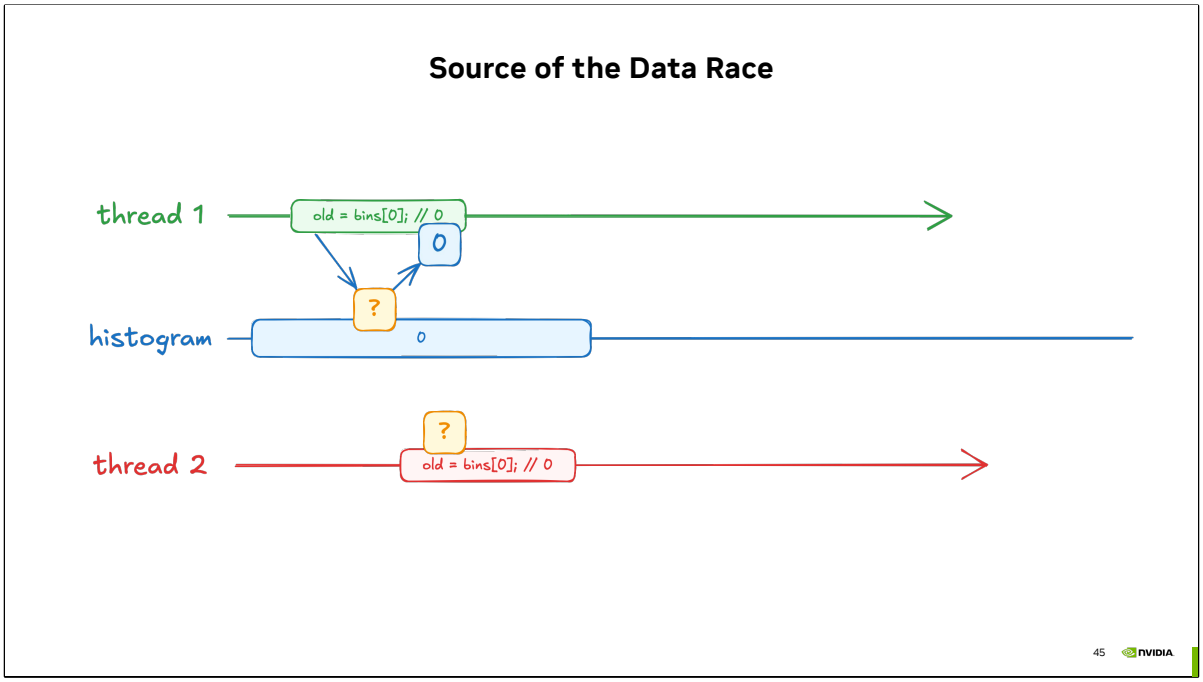
Say we have only two threads. They start by issuing a memory read request.



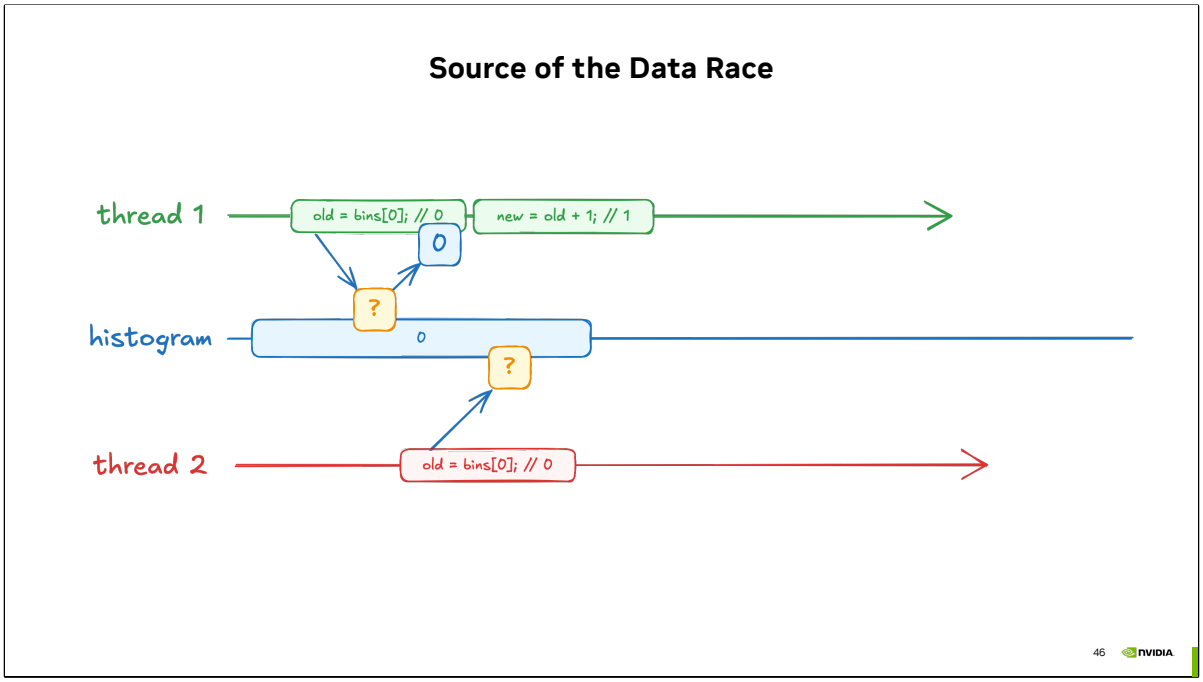
Let’s say memory read from the first thread arrives first.



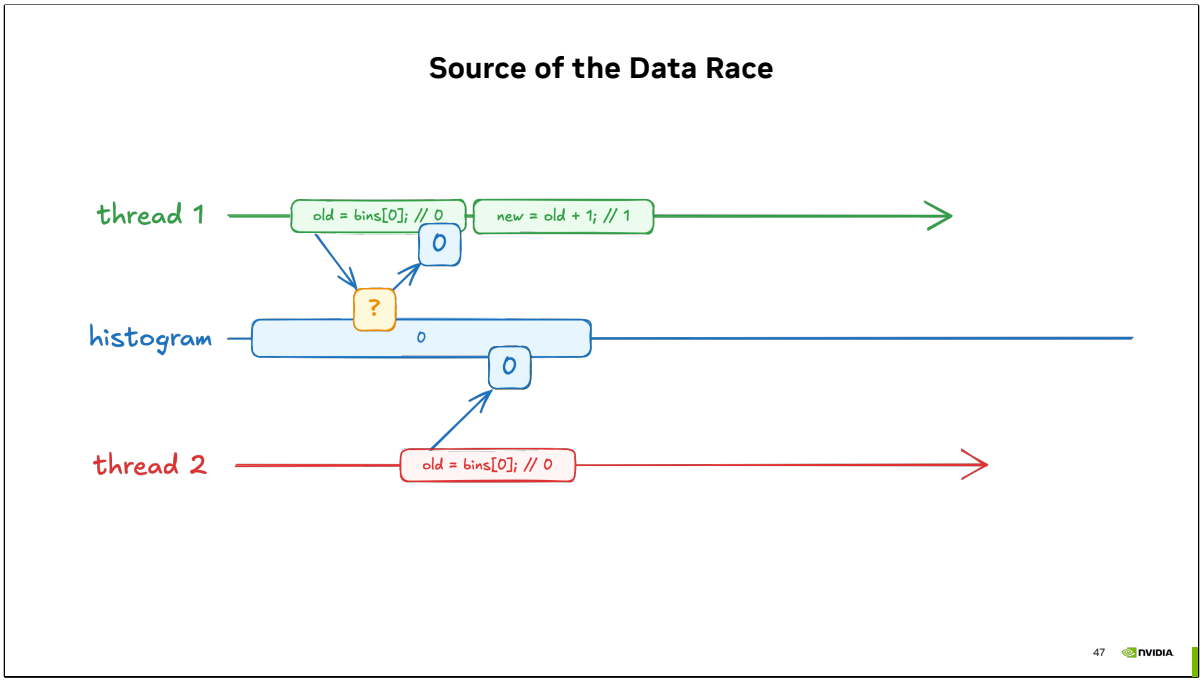
Then memory responds with the current value stored in the histogram, which is zero.



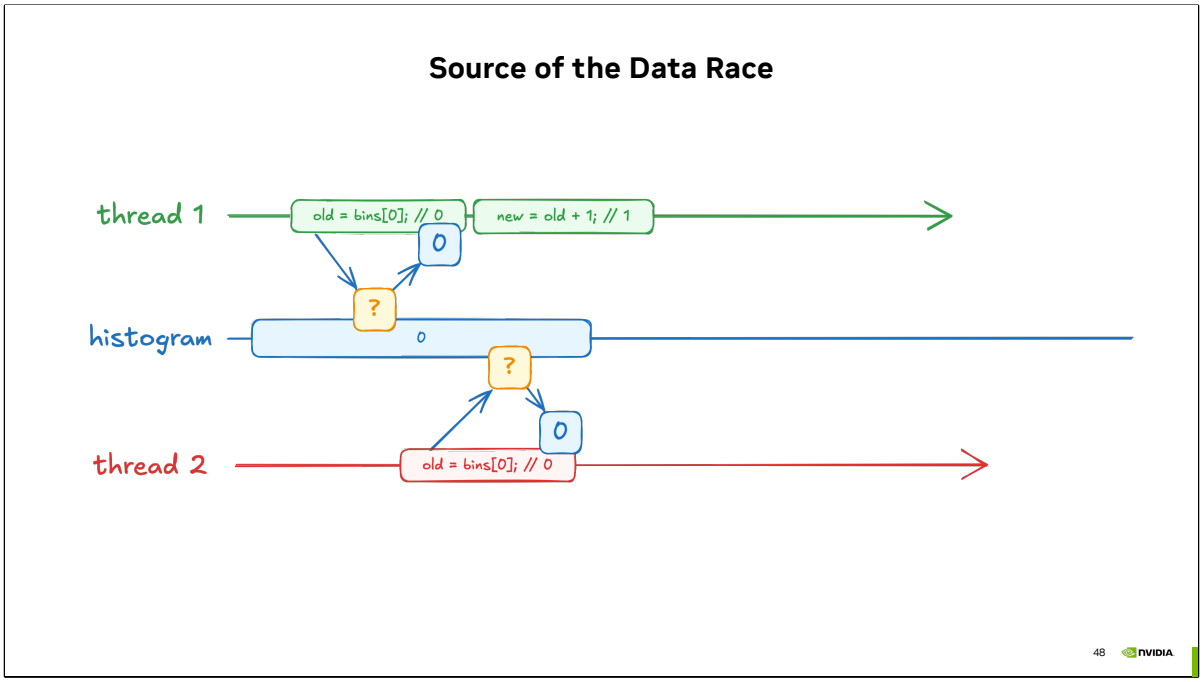
Now old counter stores zero.



Then, first thread increments zero.
New value stores 1.

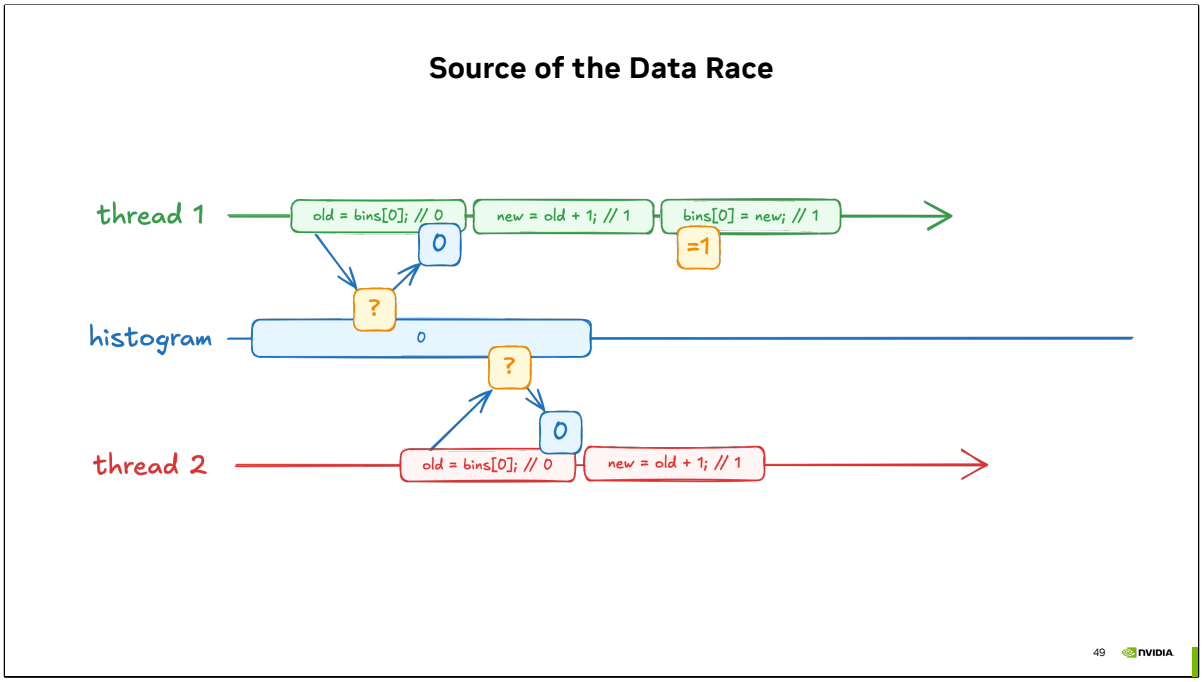


All while the second thread starts reading zero from memory



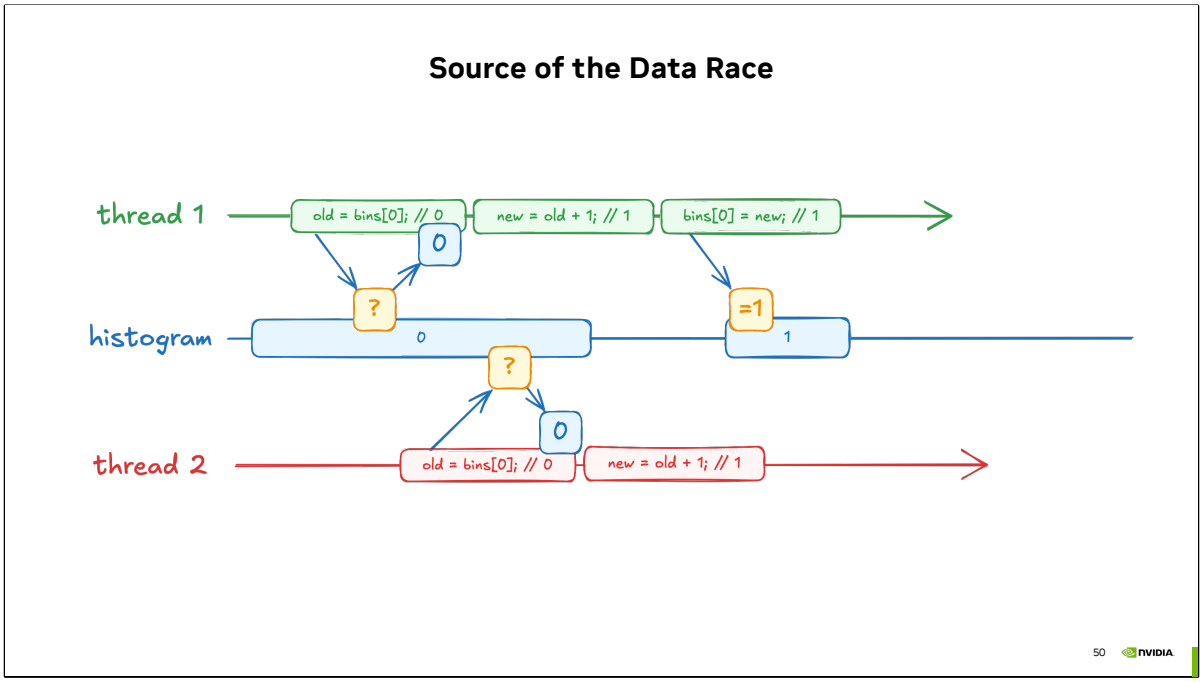
Now second thread's local variable stores zero as well

NVIDIA

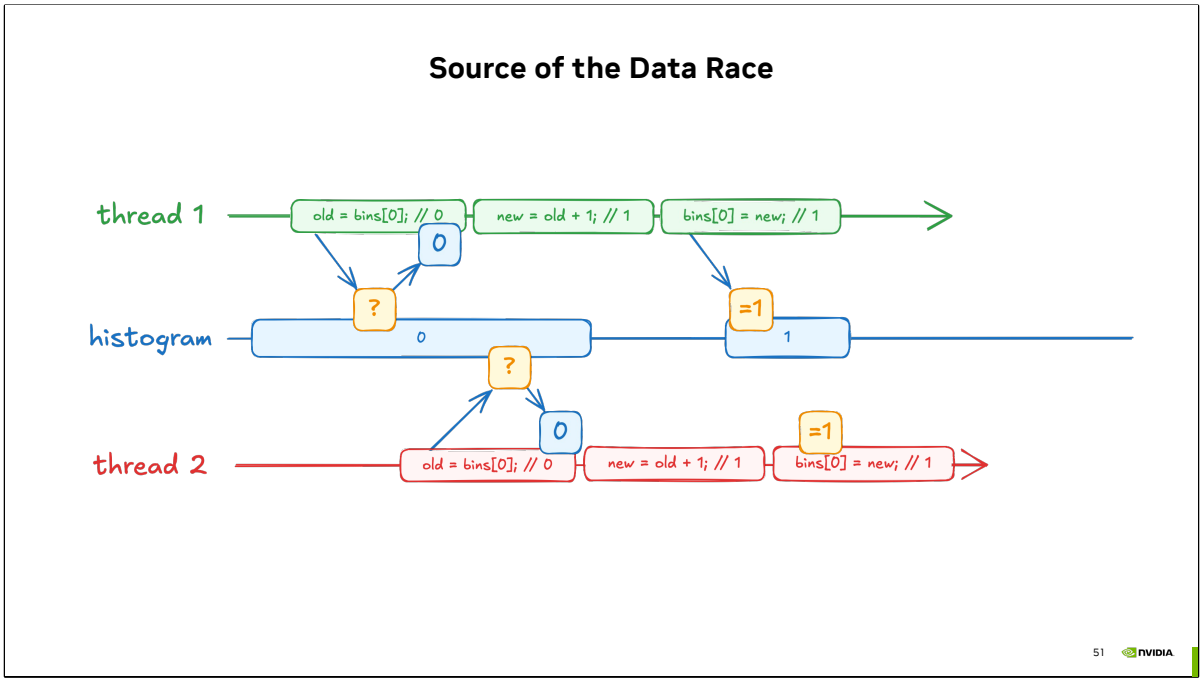


Second thread then increments its local counter, while first thread starts writing new value in memory

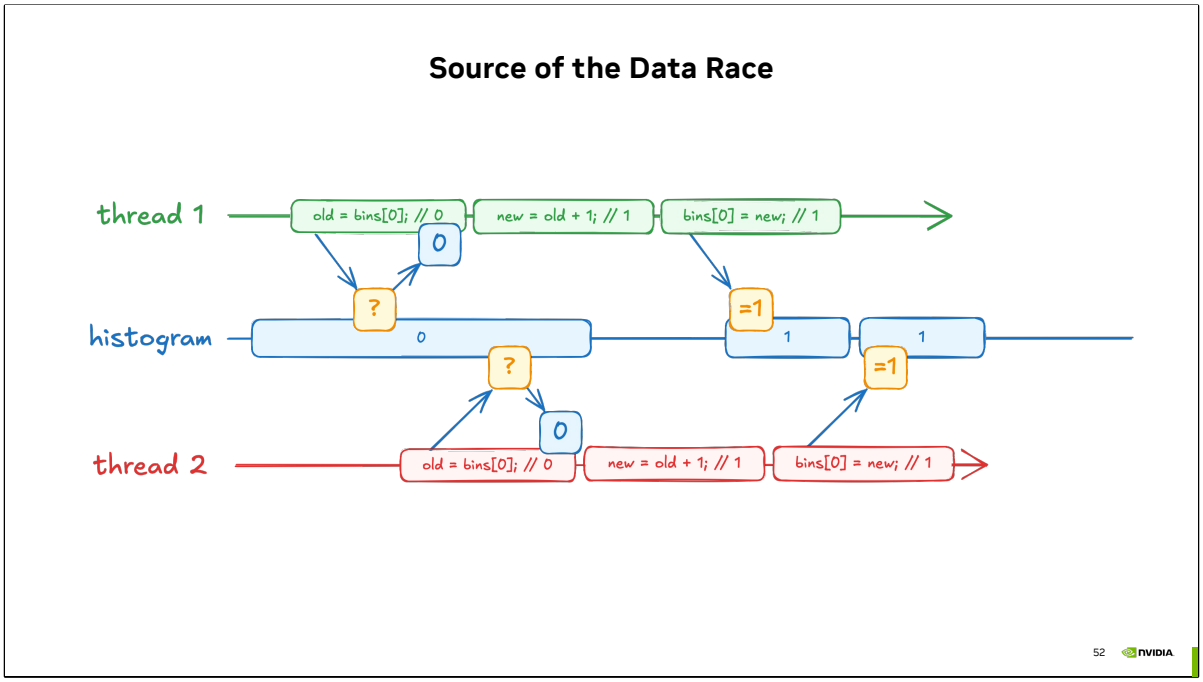
NVIDIA



This write overwrites previous value in the histogram, which now stores one



But the second thread is doing the same



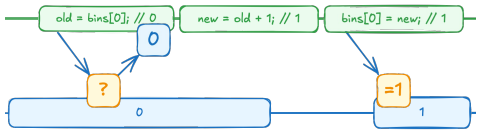
So, in our kernel, we lose most of the increments. Now imagine this happening with millions of threads and it should make it clear why our final histogram appeared empty. We simply lost most of the counters because of this data race.

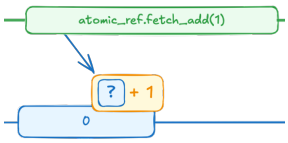
Atomic Memory Operations

To fix data race, we have to make 1. read 2. modify 3. write steps appear as a single **indivisible** read-modify-write operation

```
int old_count = histogram[bin];

int new_count = old_count + 1;

histogram[bin] = new_count;
```

```
histogram[bin].fetch_add(1);
```





53 NVIDIA

To fix this race, we need to ensure the read–modify–write sequence is treated as a single, indivisible memory operation. In C++, these operations are known as atomics. You can think of an atomic as ‘storing’ the instruction itself rather than just bytes. In the example on the right, we store “+1” into memory, and when that operation arrives on memory, it reads the current value, increments it, and writes back the result in one step - preventing any other thread from interfering in the meantime.

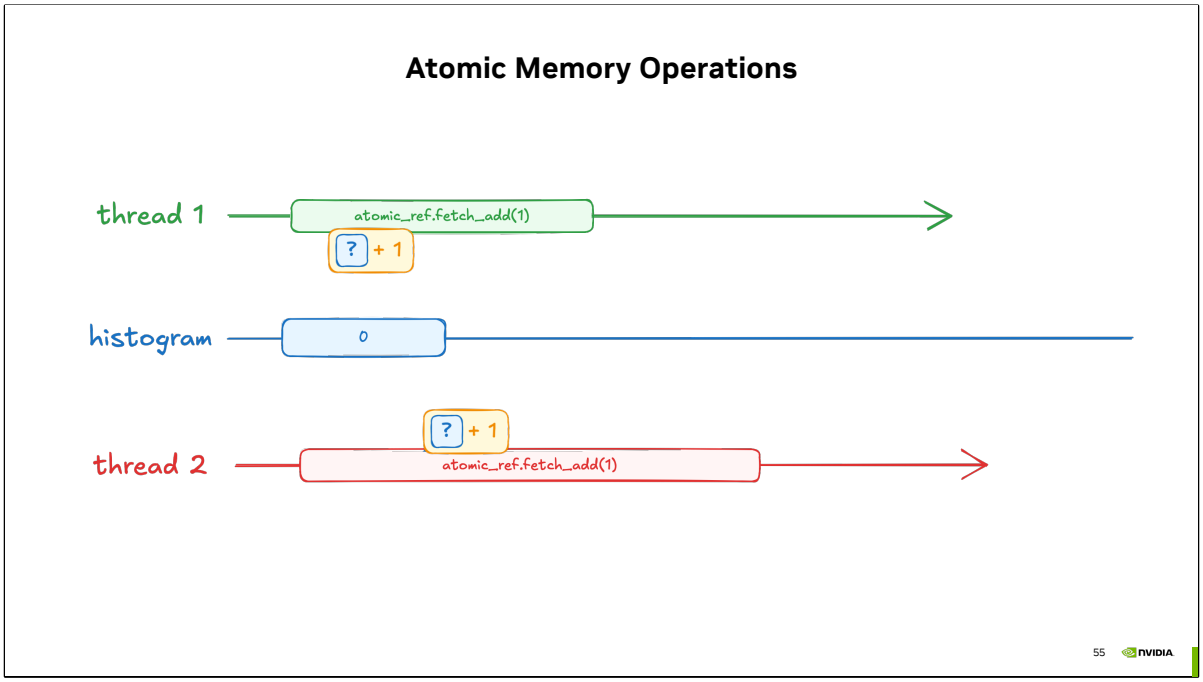
NVIDIA

Atomic Memory Operations

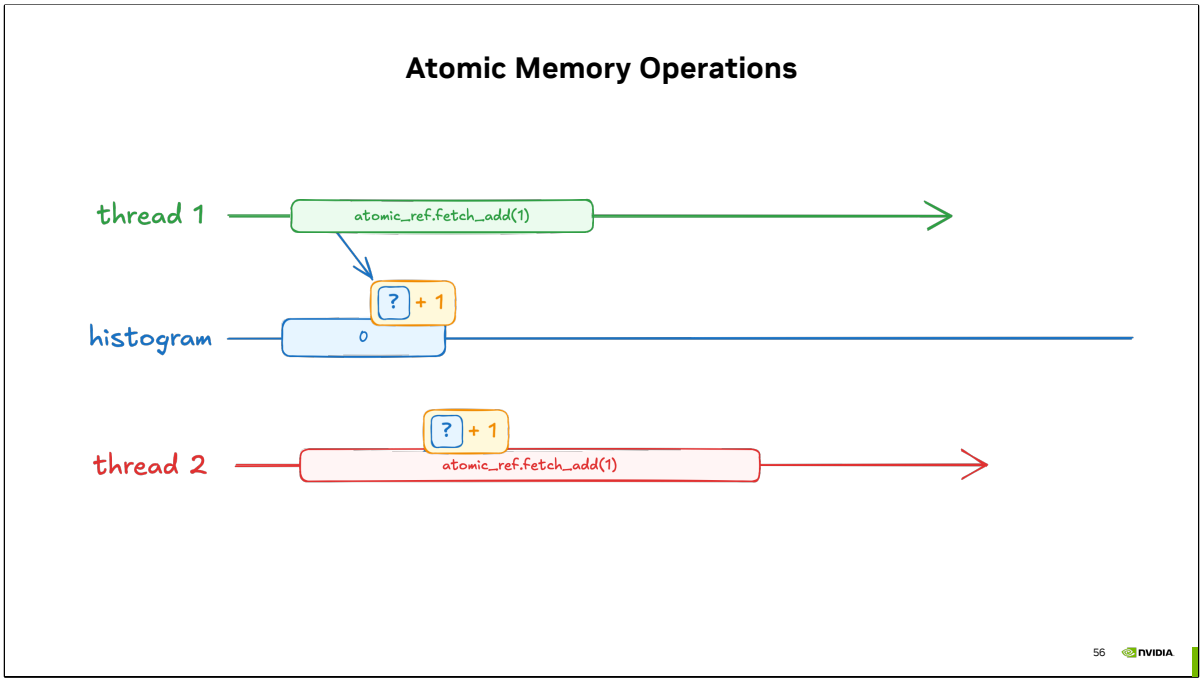
```
cuda::std::atomic_ref<int> ref(count[0]);

ref.fetch_add(3);

ref.fetch_sub(2);

ref.fetch_and(1);

...
```

- Atomics memory operations are indivisible
- `atomic_ref` applies atomic operations to the object it references
- For our example, atomic increment appears as indivisible read-modify-write operation

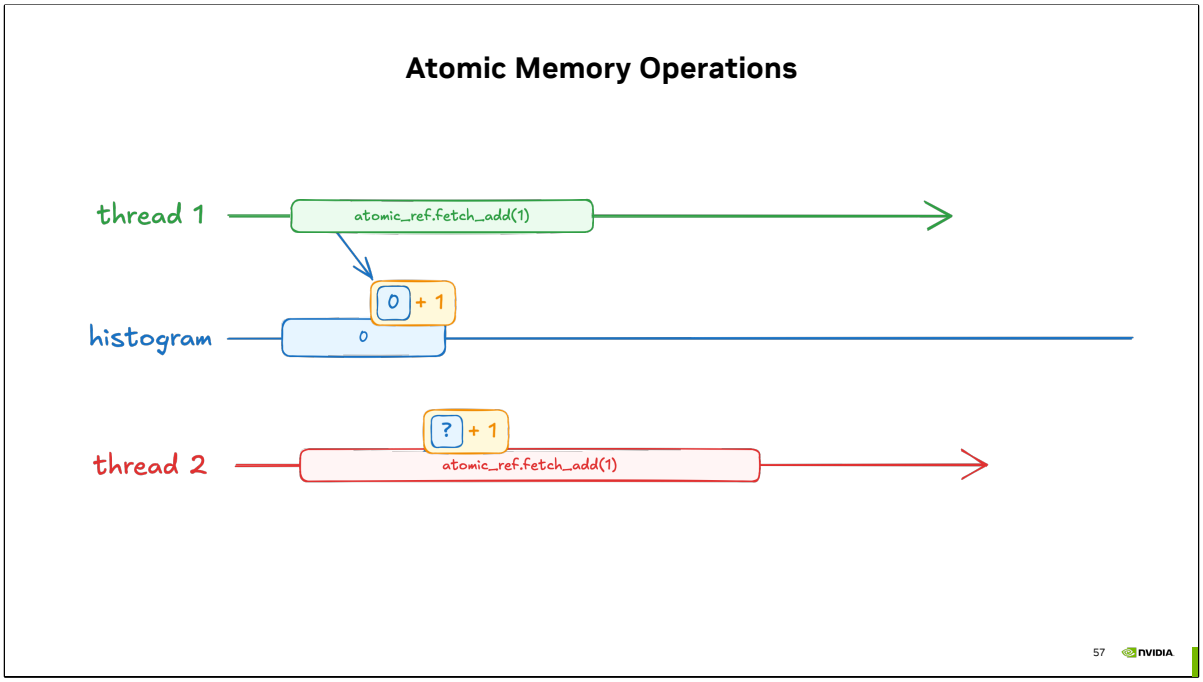
CUDA provides a type called cuda::std::atomic_ref, which lets you treat any existing memory location as an atomic variable. Here, we create an atomic reference to count[0] by wrapping it into atomic reference. Which then provides us with various atomic function available on this reference. For instance, we can atomically add or subtract a value from this memory location. And each of these operations appear as a single, indivisible step to all threads. Let's see how this type can help fix our kernel.



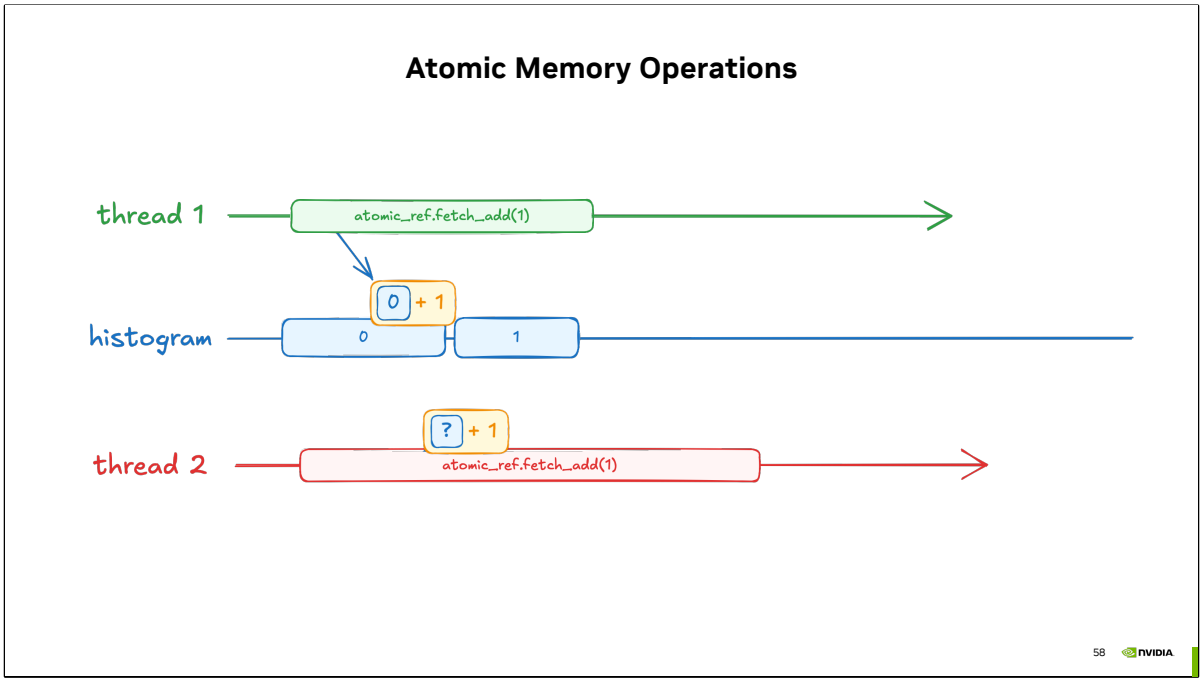
Now both threads issue atomic increment.



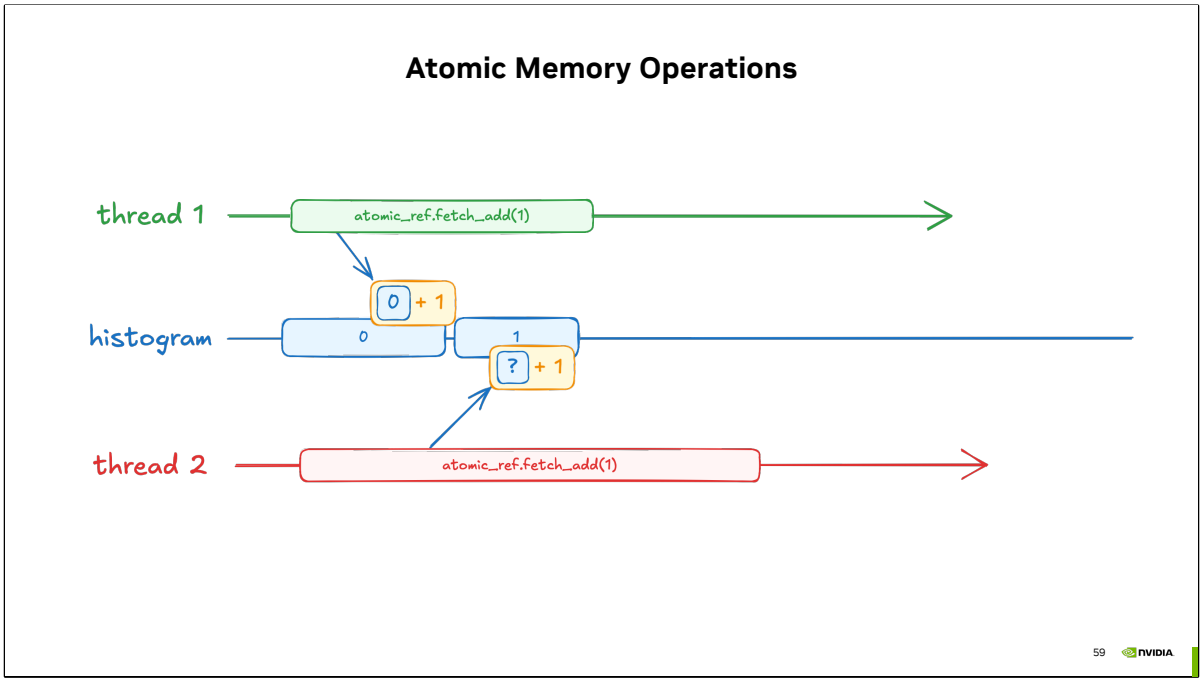
First thread's fetch add arrives first.



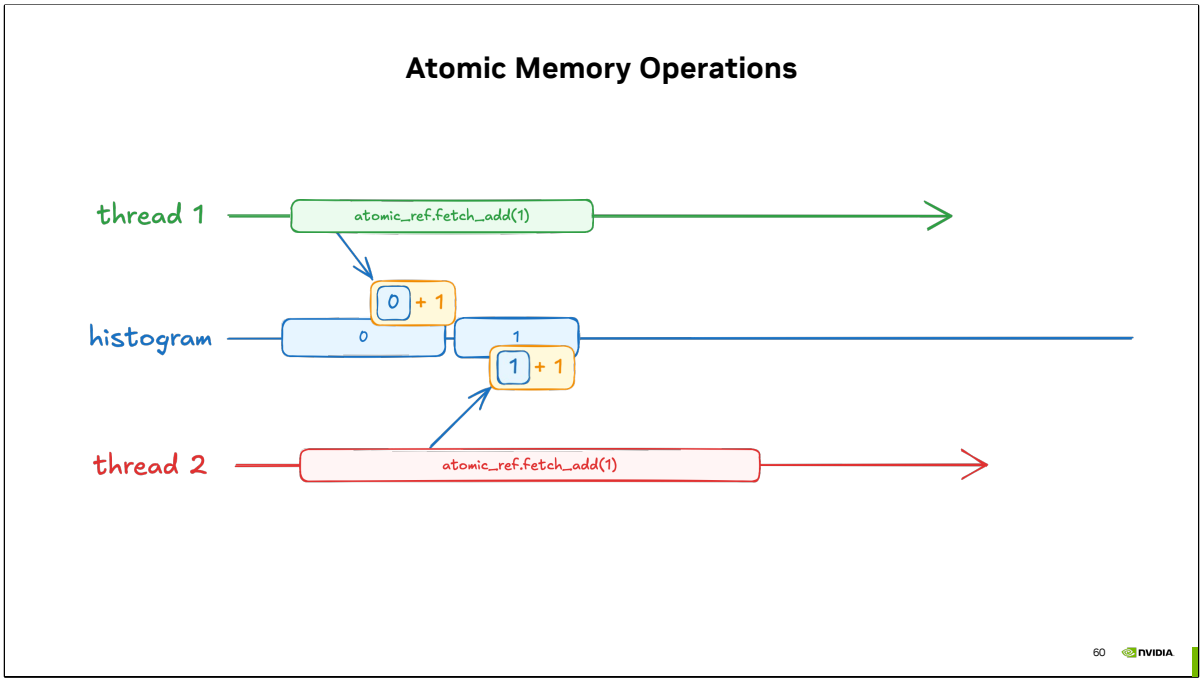
The question mark in the atomic operation is replaced with current value of the bin



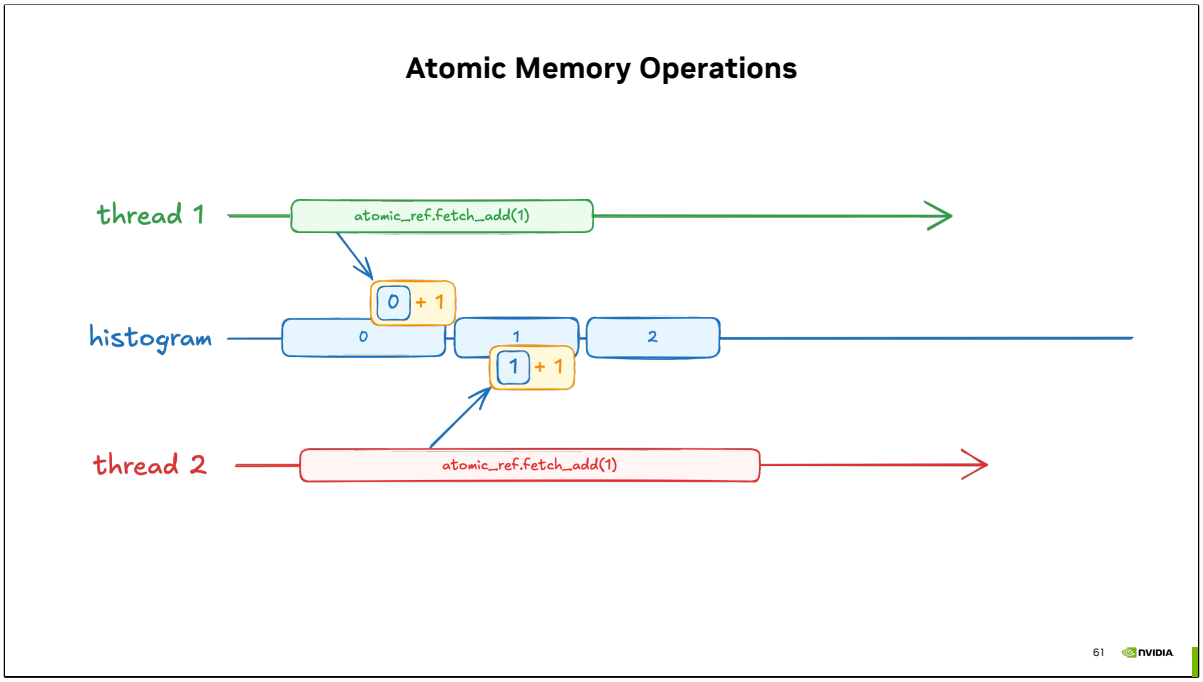
Then it adds one, and stores result back into memory. Now our histogram stores 1.



Second thread's atomic increment arrives second.



And now it's question mark is replaced with current value of the histogram, which ends up being 1.

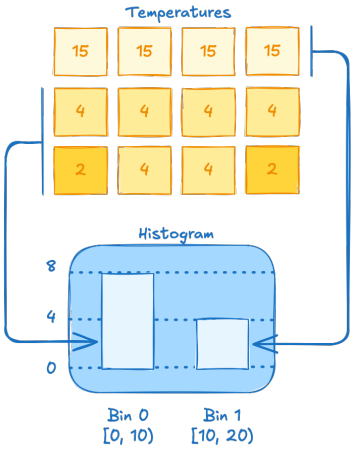


Now we add one to one, and store two back into memory. And as you can see, the memory now stores current result, no increment is lost.

Exercise: Fix Histogram

- Use atomics to fix data race in histogram

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);
  int old_count = histogram[bin];
  int new_count = old_count + 1;
  histogram[bin] = new_count;
}
```



03.03-Atomics/03.03.02-Exercise-Fix-Histogram.ipynb

In this exercise you'll have to fix the data race in the histogram kernel by using atomic references.

Exercise: Fix Histogram

Solution

- Use atomics to fix data race in histogram

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> histogram)
{
  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> ref(histogram[bin]);
  ref.fetch_add(1);
}
```

- wrap reference to bin into atomic
- fetch add to bin using atomic ref

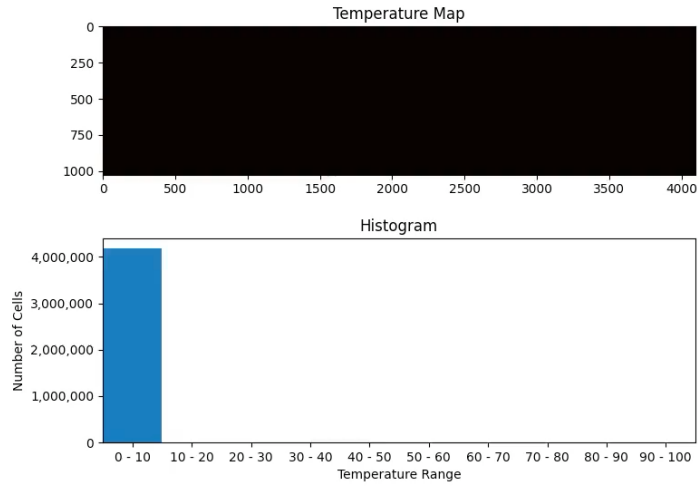
03.03-Atomics/03.03.02-Exercise-Fix-Histogram.ipynb

63 NVIDIA

Let’s go over the solution. Here, we wrap selected bin into an atomic reference, and then, we use fetch add member function to atomically increment it.

Exercise: Fix Histogram

Solution



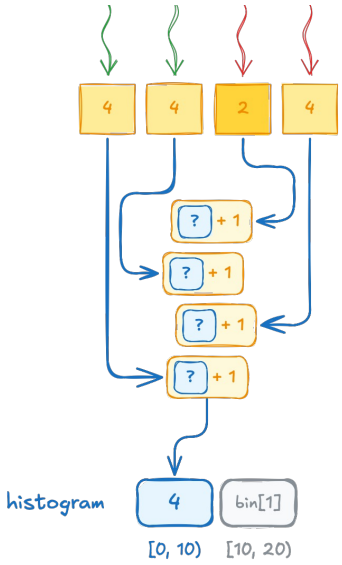
- We are getting correct results now!
- But the performance can't be worse
- Currently getting ~6 GB/s

64 NVIDIA

With this fix in place, our results are finally correct. Notice how the initial temperatures are near zero, so the first bin has the most cells in it. As heat spreads through the grid, we see more bins start to fill up. But we have another issue now, the performance can't be any worse. We are only getting about 6 GB/s, which is nowhere never GPUs bandwidth.

NVIDIA

New Source of Serialization



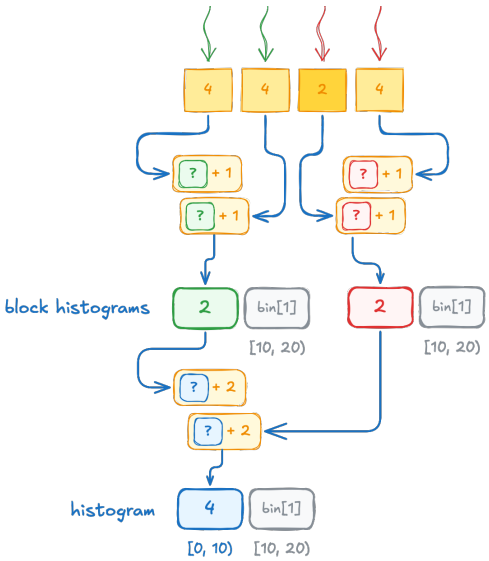
- Atomics let us regain functional correctness
- But this comes at a cost of serialization of memory accesses
- As you know, serialization should be avoided at all costs:
  - Consider thread block size = **256** and grid size = **16384**
  - If all cells mapping into the same bin,
  - we'll get a queue of **4194304** atomics
- So, what are our options?

65 NVIDIA

Last time we saw such poor performance it was because of serialization. But this time, we don't have any loops in our code, so what happened? This time the serialization is coming from the atomic operations themselves. While they solve the data-race problem, they can't run in parallel when all threads target the same memory location. Instead, each atomic gets queued up, one after another. We launch about 256 threads, and about 16 thousand blocks. In total, this gives us a queue of 4 million atomic operations. Which is obviously too much, but what are our options?

NVIDIA

Privatized Histograms



- We could add privatized histograms, one per thread block:
  - Consider thread block size = **256** and grid size = **16384**
  - If all cells mapping into the same bin,
  - we'll get a queue of **256** atomics on private histograms followed by **16384** atomics on resulting histogram
- In limit, this reduces contention by ~ block size

66 NVIDIA

One common solution to this contention is called 'privatization.' Instead of having all threads update the same global histogram, we allocate a small 'private' histogram for each thread block. The threads within a block can then safely use atomics on their private histogram. When they're done, the block aggregates its counts and updates the main histogram with just one atomic operation. In our simple two-block example, each block increments its own private histogram to a total of 2 and then adds that 2 into the global histogram once. Although that might not help much with just two blocks, imagine having 16 thousand of them. In this scenario, we'd have 16 thousand private histograms updated in parallel, followed by just 16 thousand atomic operations. This is far fewer than the four million we'd otherwise have if every thread incremented final histogram directly.

NVIDIA

Implementing Privatized Histograms

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   block_histograms,
  cuda::std::span<int>   histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);

  if (threadIdx.x < histogram.size())
  {
    cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
    ref.fetch_add(block_histogram[threadIdx.x]);
  }
}
```

- Histogram kernel now takes another span with per-thread-block histograms

67 NVIDIA

To implement this privatization, we add new parameter to our kernel, called block histograms.

Implementing Privatized Histograms

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   block_histograms,
  cuda::std::span<int>   histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);

  if (threadIdx.x < histogram.size())
  {
    cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
    ref.fetch_add(block_histogram[threadIdx.x]);
  }
}
```

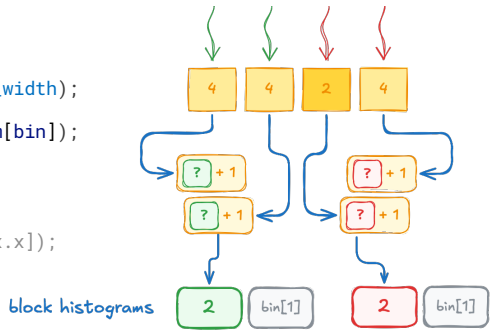
- obtain a view over thread-block private histogram
- subspan offset to private histogram is block index multiplied by histogram size

Next, we create a subspan for the current thread block's portion of the privatized histogram. The subspan function takes an offset and a size. We calculate the offset by multiplying block index by the histogram size. So, the first block has an offset of 0, the second block has an offset of 10, and so on. This ensures each block writes to its own slice of memory.

Implementing Privatized Histograms

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   block_histograms,
  cuda::std::span<int>   histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);

  if (threadIdx.x < histogram.size())
  {
    cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
    ref.fetch_add(block_histogram[threadIdx.x]);
  }
}
```

- Atomically increment thread-block histograms



69 NVIDIA

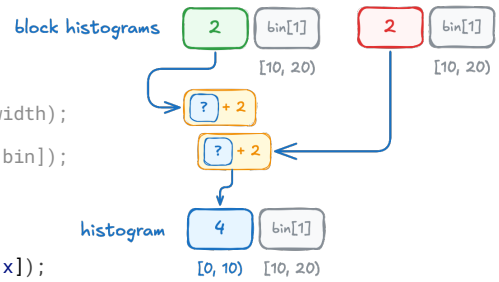
Then, just as before, each thread computes the bin index. However, we now increment the privatized block histogram instead of the global one.

NVIDIA

Implementing Privatized Histograms

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   block_histograms,
  cuda::std::span<int>   histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);

  if (threadIdx.x < histogram.size())
  {
    cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
    ref.fetch_add(block_histogram[threadIdx.x]);
  }
}
```

- Contribute private histograms to resulting one



70 NVIDIA

After that, each block adds its partial counts back into the global histogram in parallel. In the code snippet, the first 10 threads in the block each handle one bin from the block's private histogram. They use an atomic reference to safely add their local counts to the final histogram

NVIDIA

Implementing Privatized Histograms

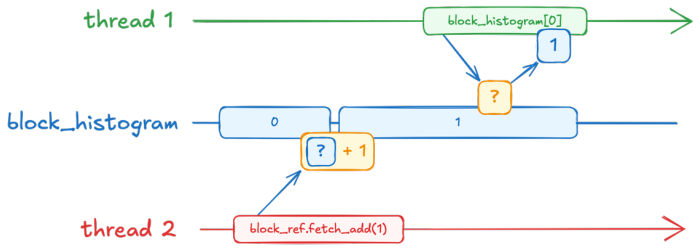
```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int>   block_histograms,
  cuda::std::span<int>   histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);

  if (threadIdx.x < histogram.size())
  {
    cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
    ref.fetch_add(block_histogram[threadIdx.x]);
  }
}
```

- Unfortunately, highlighted code has a bug in it
- We mix non-atomic read with atomic increment, which is an undefined behavior

71 NVIDIA

Unfortunately, highlighted code has a bug in it. Let's take a closer look.

What We Expected




```
cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
block_ref.fetch_add(1);

if (threadIdx.x < histogram.size())
{
  cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
}
```

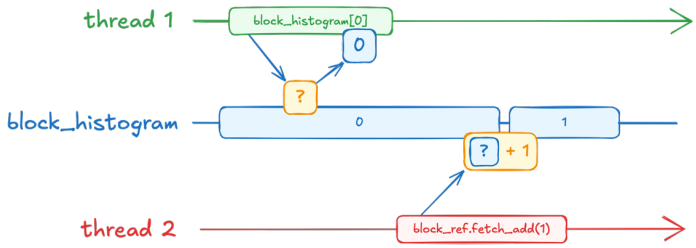
- We expected all block histogram increments to happen first
- And all block histogram reads to happen next

72 NVIDIA

In this code, we've assumed that all threads finish updating the block histogram before the next section of code begins. However, CUDA threads in a block run concurrently, without guaranteed ordering. This means that some threads might've not even started running yet while others already exited the kernel!

NVIDIA

What Can Happen




```
cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
block_ref.fetch_add(1);

if (threadIdx.x < histogram.size())
{
  cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
}
```

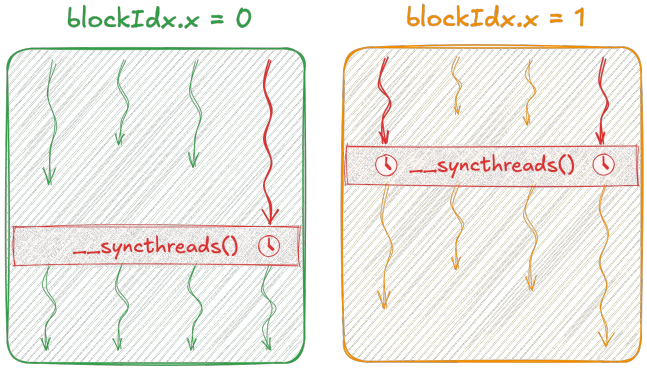
- In practice, we can't assume any order of operations performed by concurrent threads
- We need a mechanism to make sure all threads finished updating block histogram before reading it

73 NVIDIA

One of the possible scenarios is that some thread might've read the block histogram before it was updated. To fix this issue, we have to somehow make sure that all threads finished updating the block histogram, before we read it.

NVIDIA

Synchronizing in a Thread Block



`__syncthreads()`

- invoked cooperatively by all threads in a thread block
- causes threads to wait until all other threads in the thread block arrive
- similar to std::barrier, but not allowed inside conditionals

74 NVIDIA

CUDA provides a special function called __syncthreads(). It acts as a barrier for all threads in a block. When a thread reaches this barrier, it waits until every other thread in the same block has arrived. In that sense, it's similar to std::barrier, but there's one important difference. Every thread in the block must call __syncthreads() from the same logical path.

NVIDIA

Vocabulary Types in libcu++

Compound Types

- `cuda::std::pair`
- `cuda::std::tuple`
- `...`

Optional and Alternatives

- `cuda::std::optional`
- `cuda::std::variant`
- `...`

Math

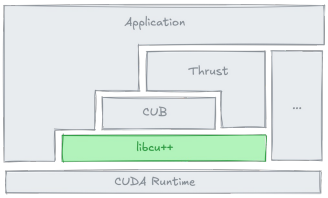
- `cuda::std::complex`
- `...`

Synchronization

- `cuda::std::atomic`
- `cuda::std::atomic_ref`
- `cuda::std::atomic_flag`
- `...`

CUDA Extensions

- `cuda::atomic`
- `cuda::atomic_ref`
- `cuda::atomic_flag`
- `...`



https://nvidia.github.io/cccl/libcudacxx

75 NVIDIA

Anyway, __syncthreads() will solve our functional problem, but there's still a performance concern. Looking back at the libcu++ overview, you'll notice two different sets of atomic references - one in the cuda::std namespace and another in the cuda namespace. Which naturally raises the question: what's the difference between the two?

NVIDIA

Thread Scopes

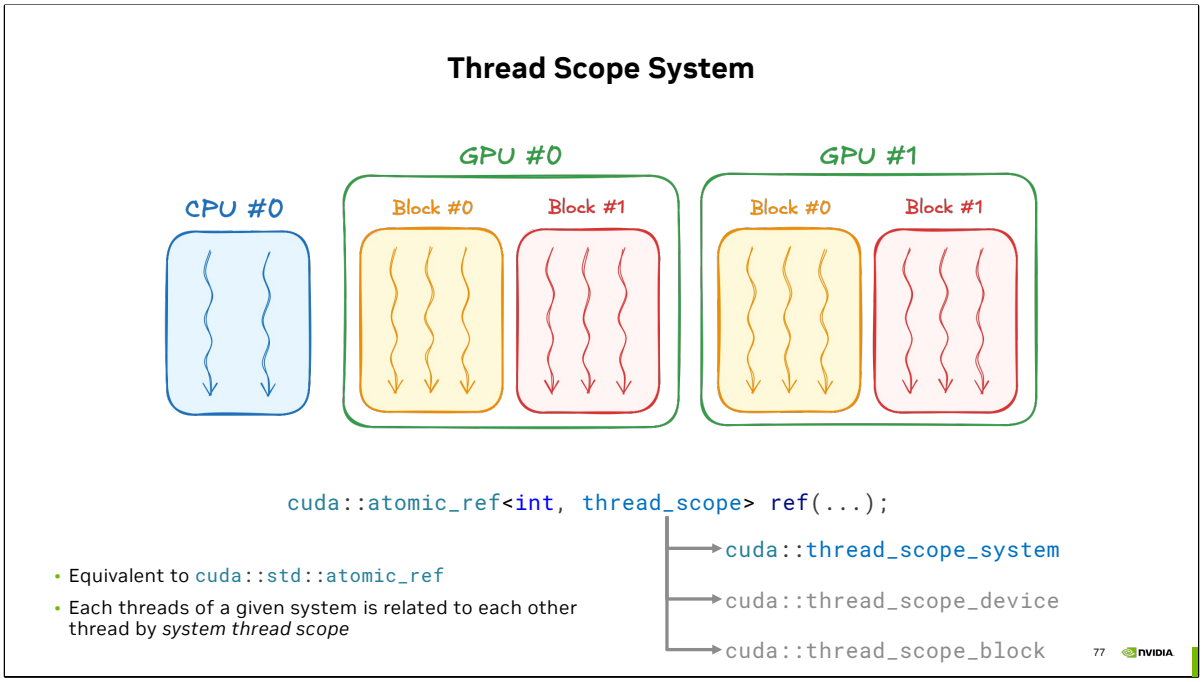
```
cuda::std::atomic_ref<int> ref(...);
```

- Both `cuda::std::atomic_ref` and `cuda::atomic_ref` have equivalent interface
- The only difference is that `cuda::atomic_ref` extends `cuda::std::atomic_ref` with additional `cuda::thread_scope` parameter
- Using appropriate thread scope can significantly affect performance

```
cuda::atomic_ref<int, thread_scope> ref(...);
                                    → cuda::thread_scope_system
                                    → cuda::thread_scope_device
                                    → cuda::thread_scope_block
```

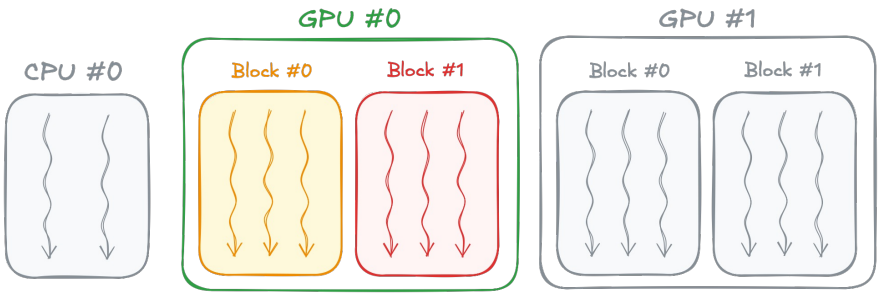
76 NVIDIA

cuda::std::atomic_ref and cuda::atomic_ref provide essentially the same interface. However, cuda::atomic_ref extends cuda::std::atomic_ref by adding a thread_scope template parameter. Choosing the right thread scope can significantly impact performance. So, let's take a closer look at what this thread scope is.



Thread scope represents the set of threads that can synchronize using a given atomic. The scope can be system, device, or block. For instance, all threads are related to each other by `cuda::thread_scope_system`. This means that a thread from any GPU (say, in a multi-GPU system) can synchronize with any other GPU thread, or even a CPU thread. The `cuda::std::atomic_ref` is actually the same thing as cuda::atomic_ref with system scope.

Thread Scope Device




```
cuda::atomic_ref<int, thread_scope> ref(...);
```

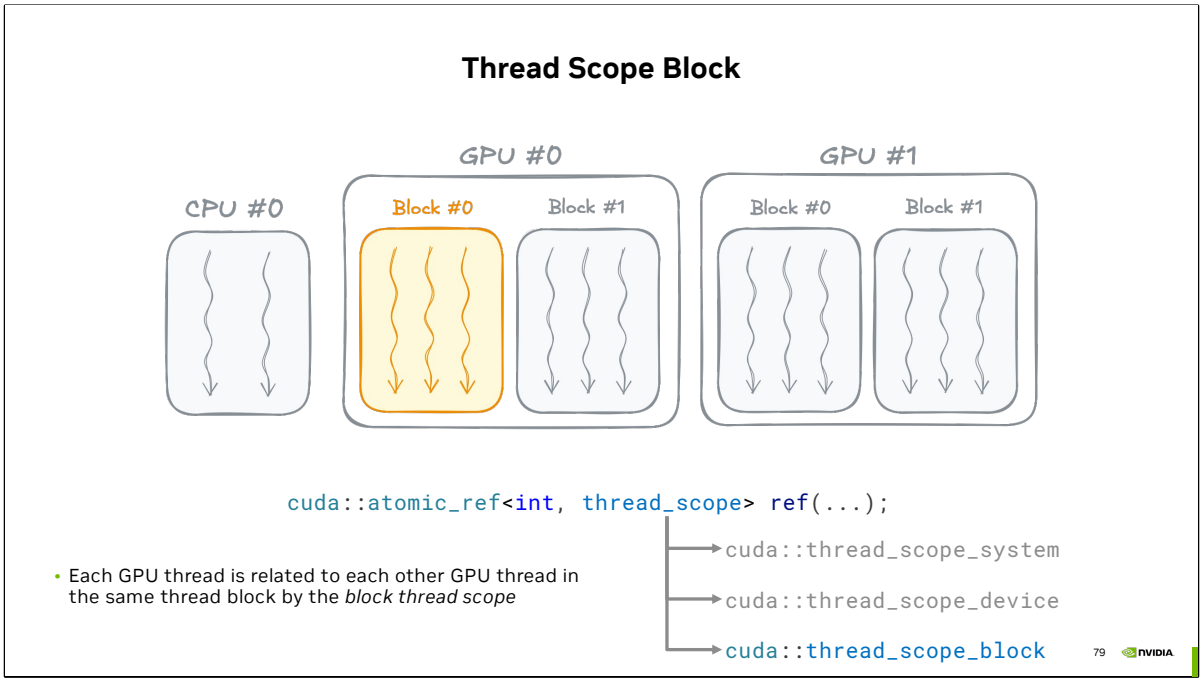
- cuda::thread_scope_system
- cuda::thread_scope_device
- cuda::thread_scope_block

- Each GPU thread is related to each other GPU thread on the same GPU by *device thread scope*

78 NVIDIA

The next is device scope that relates any thread of a given GPU. With this scope, multiple GPUs cannot issue atomics into the same memory location. But since our kernel is running on a single-GPU, device scope is more then enough.

NVIDIA

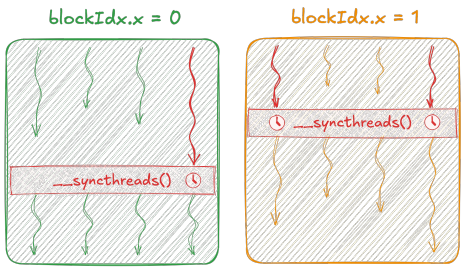


And finally, there's cuda::thread_scope_block, which restricts atomic operations to threads within a single block. In our kernel, this scope is perfect when updating a block's private histogram, because only threads within that block need to perform atomics on the same memory location."

Exercise: Fix Histogram

- Use `__syncthreads()` to make sure all threads updated block histogram before reading it
- Reduce thread scope of atomic operations for better performance (optional)

```
int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bin = static_cast<int>(temperatures[cell] / bin_width);

cuda::std::atomic_ref<int> block_ref(block_histogram[bin]);
block_ref.fetch_add(1);

if (threadIdx.x < histogram.size())
{
  cuda::std::atomic_ref<int> ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
```



03.04-Synchronization/03.04.02-Exercise-Fix-Histogram.ipynb

Which brings us to the next exercise. This time, you'll have to use thread block synchronization to make sure all threads updated block histogram before reading it. Besides that, you'll have to reduce thread scope of atomic operations to improve performance.

Exercise: Fix Histogram

Solution

```
int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bin = static_cast<int>(temperatures[cell] / bin_width);

cuda::atomic_ref<int, cuda::thread_scope_block> block_ref(block_histogram[bin]);
block_ref.fetch_add(1);
__syncthreads();

if (threadIdx.x < histogram.size())
{
  cuda::atomic_ref<int, cuda::thread_scope_device> ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
```

- This change brings us from 6 GB/s to 100 GB/s
- This is much better, but we still have room for improvement

81 NVIDIA

Let's review the solution. We started by swapping cuda::std::atomic_ref for cuda::atomic_ref. Then, we limited the scope of atomic operations on the block histogram to cuda::thread_scope_block and added a thread block synchronization step after those updates. Finally, we used cuda::thread_scope_device for the global increment, since multiple blocks need to update the same memory location. This change brought us from 6 GB/s to 100 GB/s, which is an impressive improvement. That said, we can do even better

Recap

```
cuda::std::span<int> block_histogram =
  block_histograms.subspan(
    blockIdx.x * histogram.size(),
    histogram.size());

int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bin = static_cast<int>(temperatures[cell] / bin_width);

cuda::atomic_ref<int, cuda::thread_scope_block>
  block_ref(block_histogram[bin]);
block_ref.fetch_add(1);
__syncthreads();

if (threadIdx.x < num_bins)
{
  cuda::atomic_ref<int, cuda::thread_scope_device>
    ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
```

We've learned so many things!

- Atomic operations
- Thread scopes
- Sub spans

But what about the initial question?

Why was thread hierarchy needed in the first place?

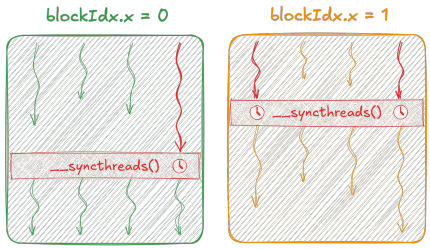
82 NVIDIA

But before optimizing our histogram kernel even further, let's revisit what we've learned so far. We've just covered atomic operations, thread scopes, sub spans, but what about our initial question? Why was thread hierarchy needed in the first place?

NVIDIA

Recap

```
cuda::std::span<int> block_histogram =
  block_histograms.subspan(
    blockIdx.x * histogram.size(),
    histogram.size());

int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bin = static_cast<int>(temperatures[cell] / bin_width);

cuda::atomic_ref<int, cuda::thread_scope_block>
  block_ref(block_histogram[bin]);
block_ref.fetch_add(1);
__syncthreads();

if (threadIdx.x < num_bins)
{
  cuda::atomic_ref<int, cuda::thread_scope_device>
    ref(histogram[threadIdx.x]);
  ref.fetch_add(block_histogram[threadIdx.x]);
}
```

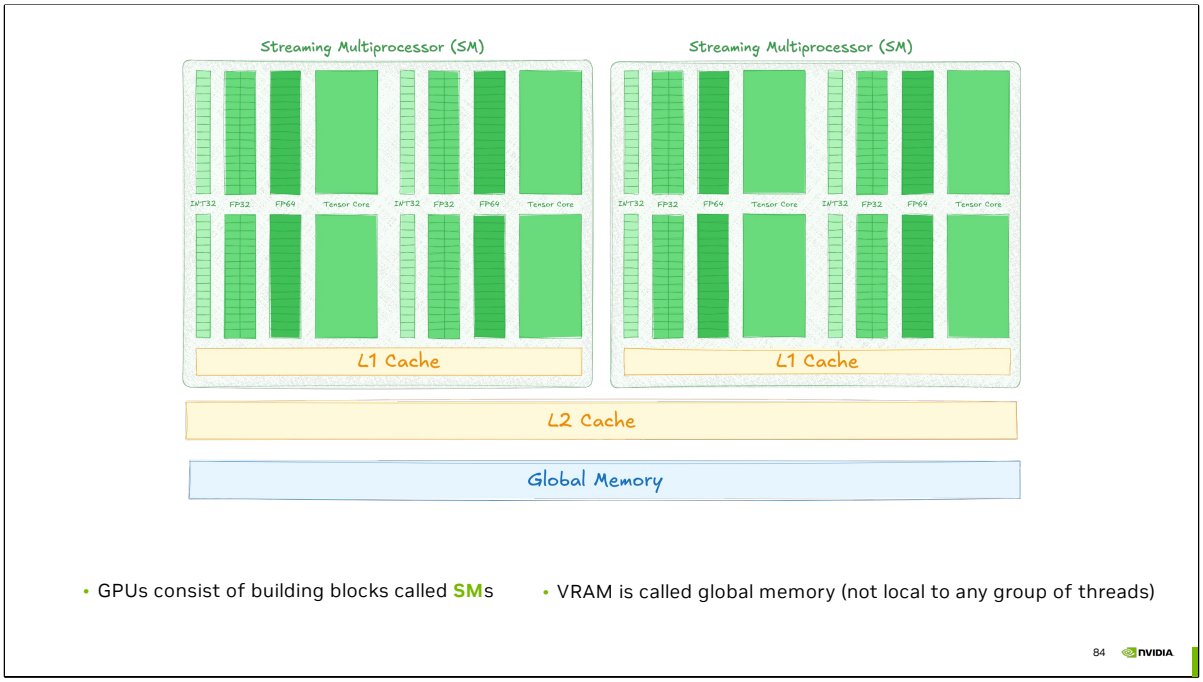


- Let's reconsider `__syncthreads`
- Synchronization is only available between threads of a given thread block
- This makes different groups of threads not equally capable, hence introducing non-uniformity in thread grouping
- Are there any other resources that are only accessible within certain groups of threads?

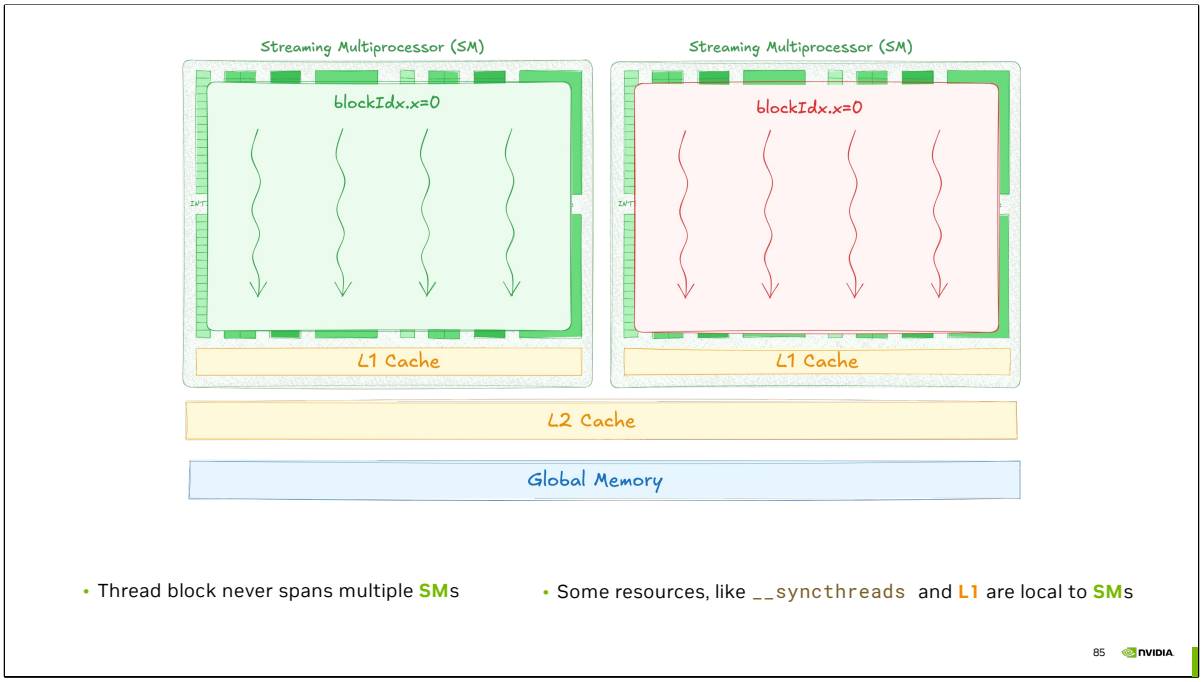
83 NVIDIA

Let's take another look at thread block synchronization, which is only available between threads of a given thread block. This makes different groups of threads not equally capable, hence introducing non-uniformity in thread grouping and thread hierarchy as a result. But are there any other resources that are only accessible within certain groups of threads?

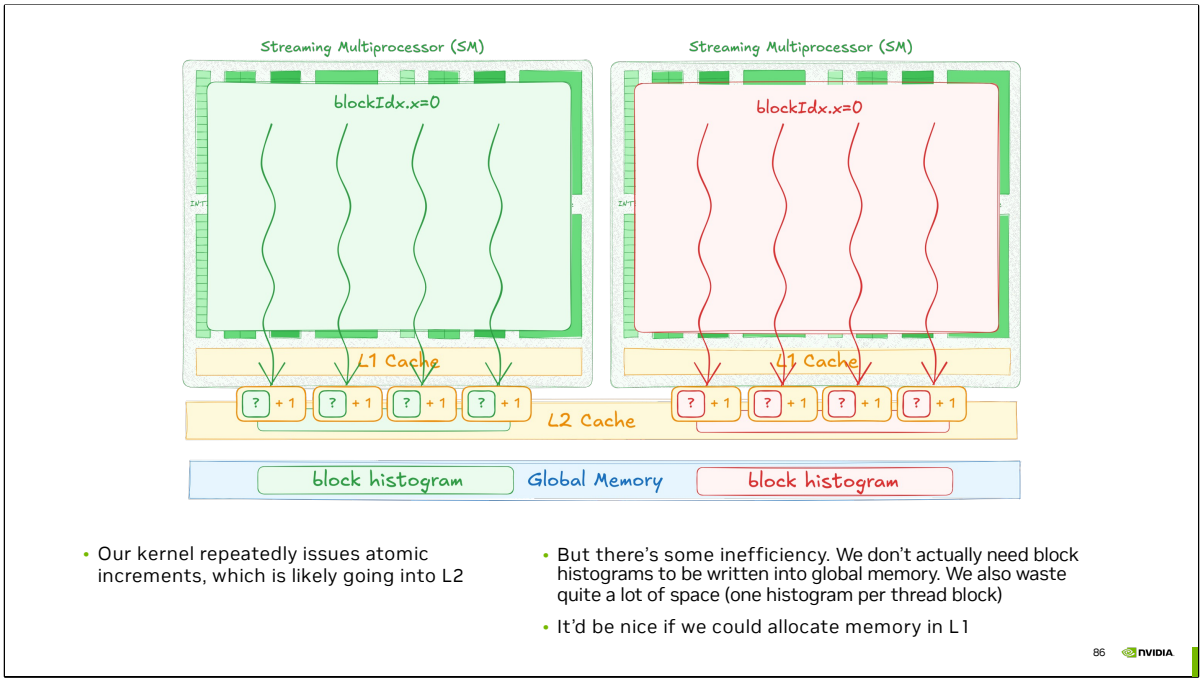
NVIDIA



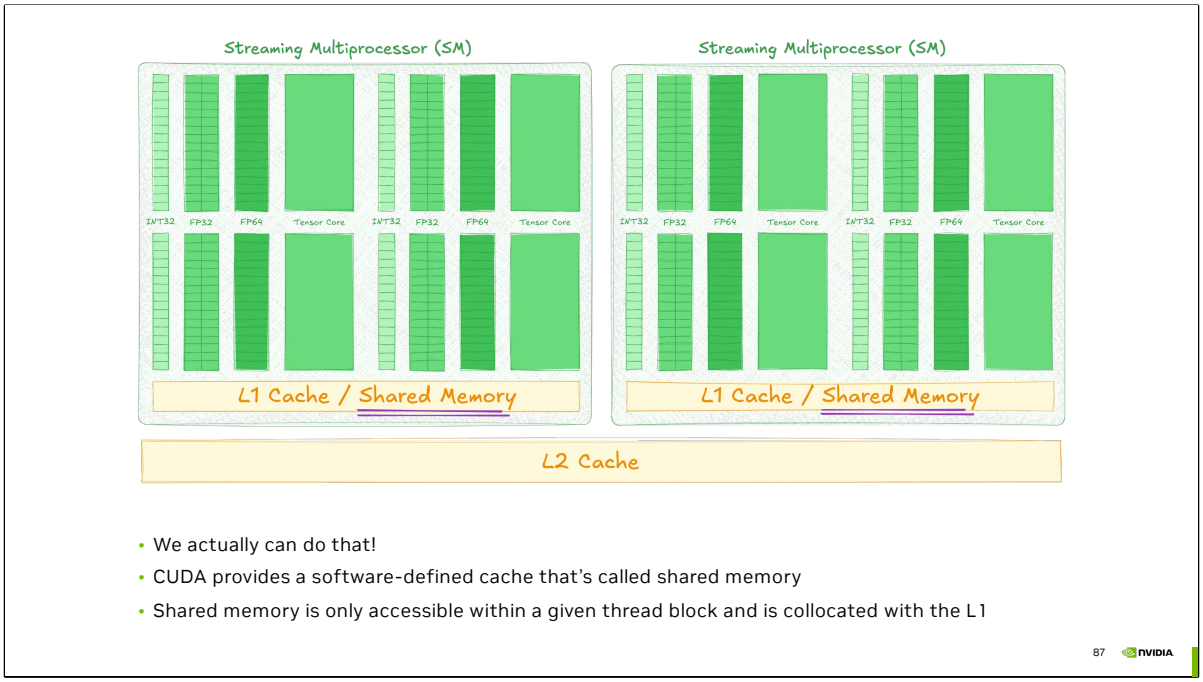
To find out, let's examine the GPU's architecture. GPUs are composed of uniform building blocks called Streaming Multiprocessors, or SMs. A single GPU can contain hundreds them. Each SM has a lot of cores, but also local L1 cache. The main GPU memory lives outside these SMs, which is why we call it 'global' memory.



Some features like thread block synchronization are built into the SM hardware. To use these SM-specific features, thread blocks always run entirely on a single SM.



Let's think about how our histogram kernel maps to this hardware. Each thread block runs on one SM. And block histograms are stored in global memory. Then each thread issues atomics into its block private histogram, which likely go into L2. That's not very efficient because we don't actually need to keep these block histograms around once the kernel finishes.They're just temporary storage. Meanwhile, the L1 cache sits right there on the SM. If we could allocate the block histogram in L1 directly, we'd avoid wasting space in global memory. Besides that, L1 cache seems much closer than global memory, so we'd likely improve performance.



Fortunately, we can do exactly that! CUDA provides a software defended cache that's called shared memory. Shared memory is only accessible within a given thread block and is collocated with L1.

NVIDIA

Shared Memory

```
__global__ void kernel()
{
  __shared__ int shared[4];

  shared[threadIdx.x] = threadIdx.x;

  __syncthreads();

  if (threadIdx.x == 0) {
    for (int i = 0; i < 4; i++) {
      std::printf("shared[%d] = %d\n", i, shared[i]);
    }
  }
}
```

- To allocate array in shared memory space, use `__shared__` specifier

88 NVIDIA

To allocate shared memory, we simply add the __shared__ specifier to a variable. In the example on the slide, I'm declaring an array of four integers as __shared__.

Shared Memory

```
__global__ void kernel()
{
  __shared__ int shared[4];

  shared[threadIdx.x] = threadIdx.x;

  __syncthreads();

  if (threadIdx.x == 0) {
    for (int i = 0; i < 4; i++) {
      std::printf("shared[%d] = %d\n", i, shared[i]);
    }
  }
}
```

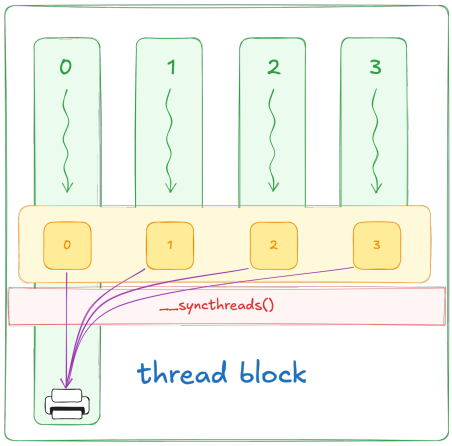
- You can use it as if it was an ordinary array
- Just be mindful of other threads and avoid data races with `__syncthreads`

89 NVIDIA

From within a thread block, you can read and write to this array as you normally would. In this example, each thread writes its own index into the corresponding element of the array. Just keep potential data races in mind and use __syncthreads() so that every thread finishes writing before any thread reads the data.

Shared Memory

```
__global__ void kernel()
{

  __shared__ int shared[4];

  shared[threadIdx.x] = threadIdx.x;

  __syncthreads();

  if (threadIdx.x == 0) {
    for (int i = 0; i < 4; i++) {
      std::printf("shared[%d] = %d\n", i, shared[i]);
    }
  }
}
```



90 NVIDIA

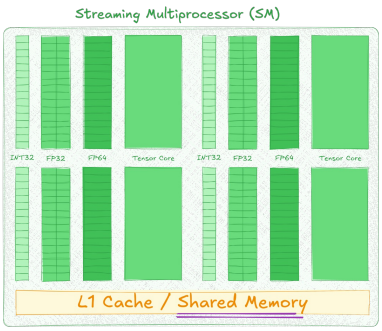
To emphasize that this shared array is indeed shared across all threads in a block, I have the first thread print out every value stored in it. First thread sees the updates from all other threads, and in this case, it prints 0, 1, 2, 3.

NVIDIA

Exercise: Optimize Histogram

- Use shared instead of global memory to store block histogram

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> block_histograms,
  cuda::std::span<int> histogram)
{
  cuda::std::span<int> block_histogram =
    block_histograms.subspan(
      blockIdx.x * histogram.size(),
      histogram.size());

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::atomic_ref<int, cuda::thread_scope_block>
    block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);
  __syncthreads();

  ...
```



03.05-Shared-Memory/03.05.02-Exercise-Optimize-Histogram.ipynb

Which brings us the the next exercise. This time, you'll have to allocate block histogram in shared memory instead of global one.

Exercise: Optimize Histogram

Solution

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> histogram)
{
  __shared__ int block_histogram[num_bins];

  if (threadIdx.x < num_bins) {
    block_histogram[threadIdx.x] = 0;
  }
  __syncthreads();

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::atomic_ref<int, cuda::thread_scope_block>
    block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);
  __syncthreads();
  ...
```

- Kernel doesn't need block histograms allocation now

92 NVIDIA

Let's take a look at the solution. First, we no longer need block histograms in our kernel signature, so we remove it.

Exercise: Optimize Histogram

Solution

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> histogram)
{
  __shared__ int block_histogram[num_bins];

  if (threadIdx.x < num_bins) {
    block_histogram[threadIdx.x] = 0;
  }
  __syncthreads();

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::atomic_ref<int, cuda::thread_scope_block>
    block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);
  __syncthreads();
  ...
```

- We start by allocating block histogram in shared memory
- Then we initialize block histogram to zero by first threads
- And synchronize block to make sure other threads wait for block histogram to be properly initialized

93 NVIDIA

Next, we allocate block histogram in shared memory and then initialize it to zero before the first use. After that, we synchronize thread block, to avoid possible data races.

NVIDIA

Exercise: Optimize Histogram

Solution

```
__global__ void histogram_kernel(
  cuda::std::span<float> temperatures,
  cuda::std::span<int> histogram)
{
  __shared__ int block_histogram[num_bins];

  if (threadIdx.x < num_bins) {
    block_histogram[threadIdx.x] = 0;
  }
  __syncthreads();

  int cell = blockIdx.x * blockDim.x + threadIdx.x;
  int bin = static_cast<int>(temperatures[cell] / bin_width);

  cuda::atomic_ref<int, cuda::thread_scope_block>
    block_ref(block_histogram[bin]);
  block_ref.fetch_add(1);
  __syncthreads();
  ...
```

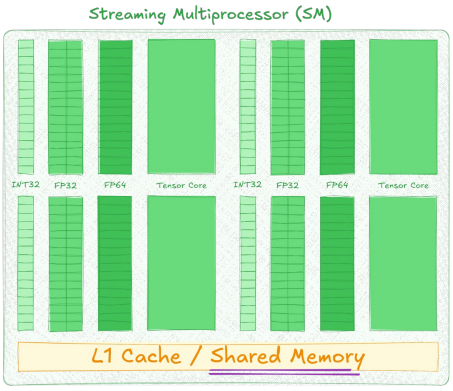
- We then wrap shared memory reference to an atomic reference as usual

94 NVIDIA

And finally, we wrap this shared block histogram in atomic reference, just like before. The rest of the code remains unchanged.

Exercise: Optimize Histogram

Solution

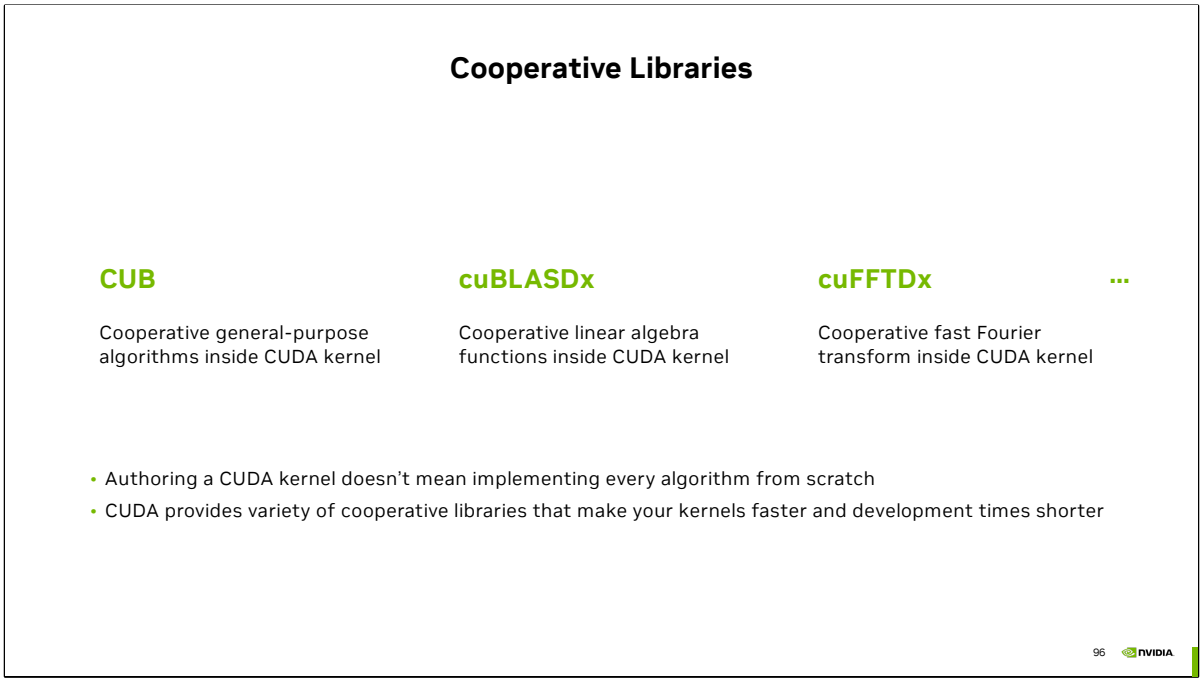


- Overall, this brings us from 100 GB/s to 400 GB/s

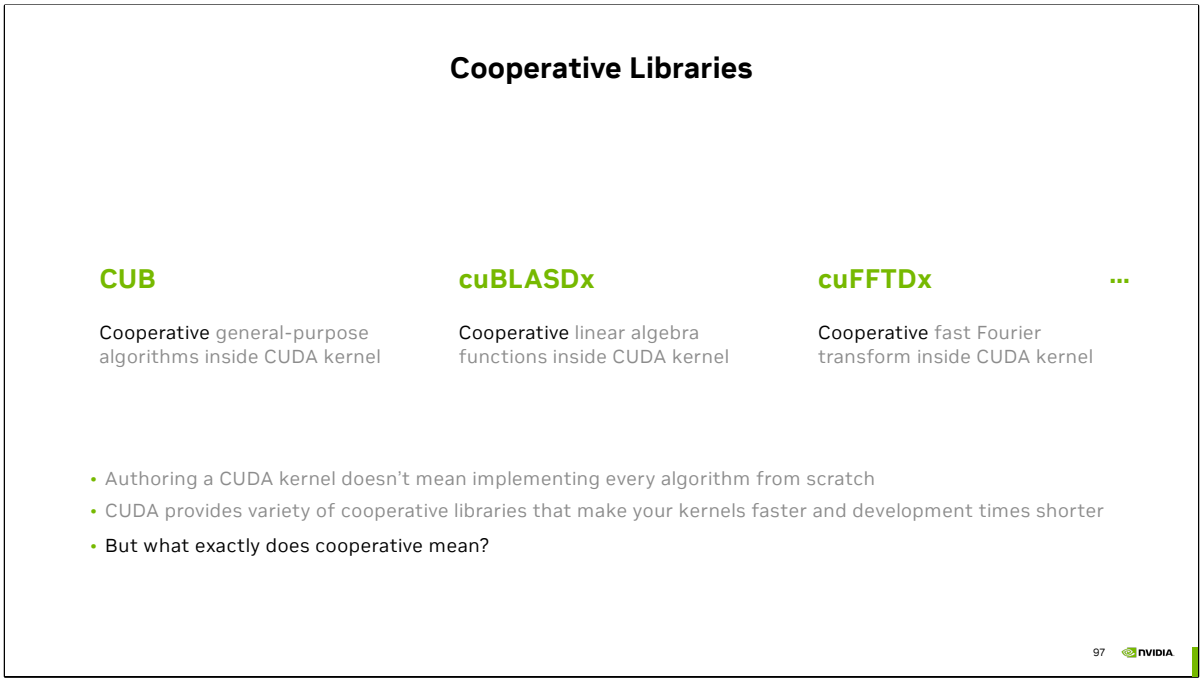
95 NVIDIA

And this change brings us to 400 GB/s, leading to about 4x speedup. Finally, our kernel is functional and performant. But we've made a higher-level mistake. When we first started using GPUs, we relied on accelerated libraries. But as soon as we started writing our own CUDA kernels, we've been implementing everything from scratch.

NVIDIA



Authoring a CUDA kernel doesn't mean you have to reinvent every algorithm from scratch. CUDA offers a range of cooperative libraries that speed up your kernels and shorten development time. For example, CUB provides general-purpose cooperative algorithms for reductions, prefix sums, and sorting. Meanwhile, cuBLASDx handles cooperative linear algebra functions, and cuFFTDx offers cooperative Fourier Transforms.



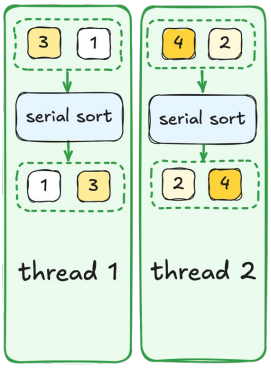
But what exactly does 'cooperative' mean in this context?

NVIDIA

Cooperative Algorithms

Serial

- **One thread** invokes algorithm
- **One thread** executes algorithm



98 NVIDIA

To explain what ‘cooperative’ means, let’s review the types of algorithms we’ve encountered so far. We began with std::sort, which is a serial algorithm. A serial algorithm is both called and executed by a single thread. So, if two different threads each call a serial algorithm, the inputs from one thread can not affect the outputs of the other.

NVIDIA

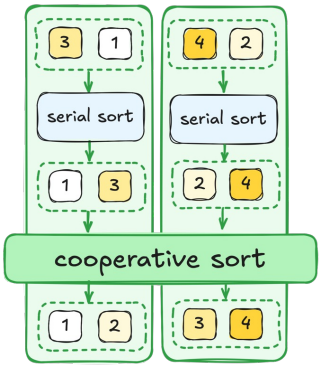
Cooperative Algorithms

Serial

- **One thread** invokes algorithm
- **One thread** executes algorithm

Cooperative

- **Many threads** invoke algorithm
- **Many threads** execute algorithm



99 NVIDIA

Cooperative algorithms, on the other hand, are invoked by multiple threads and are executed cooperatively among them. You can imagine the input to these cooperative algorithms as if one large ‘virtual’ array, that just happened to be split across different threads. The same goes for the output. In the example on the slide, a cooperative sort takes elements [1, 3] from the first thread and [2, 4] from the second. After sorting cooperatively, the first thread’s output becomes [1, 2], even though 2 originally came from the second thread.

NVIDIA

Cooperative Algorithms

Serial

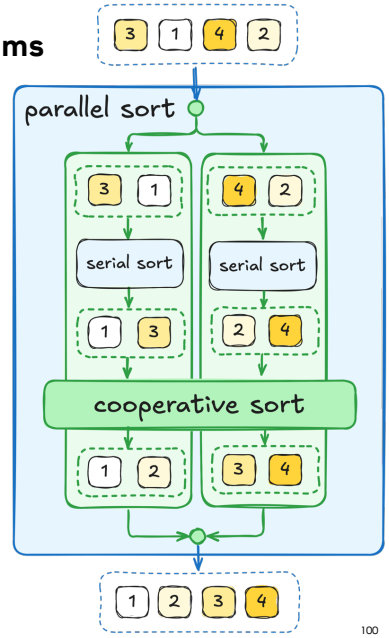
- **One thread** invokes algorithm
- **One thread** executes algorithm

Cooperative

- **Many threads** invoke algorithm
- **Many threads** execute algorithm

Parallel

- **One thread** invokes algorithm
- **Many threads** execute algorithm



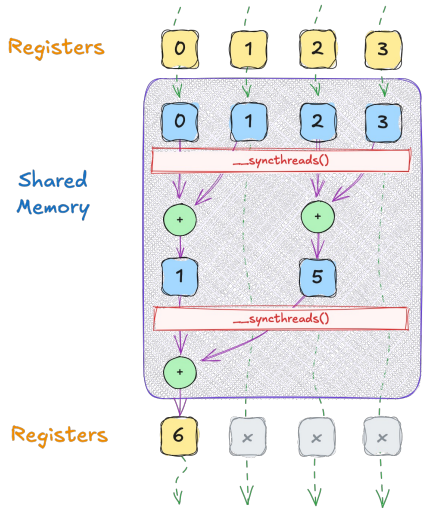
100 NVIDIA

We've also used parallel algorithms, which are called by a single thread but executed by many threads behind the scenes. From a user's perspective, these algorithms appear 'serial,' since you just call them like any other function. However, under the hood, they distribute the work among multiple threads to complete the task faster. Some examples of parallel algorithms we've used so far include thrust::transform and cub::DeviceTransform.

NVIDIA

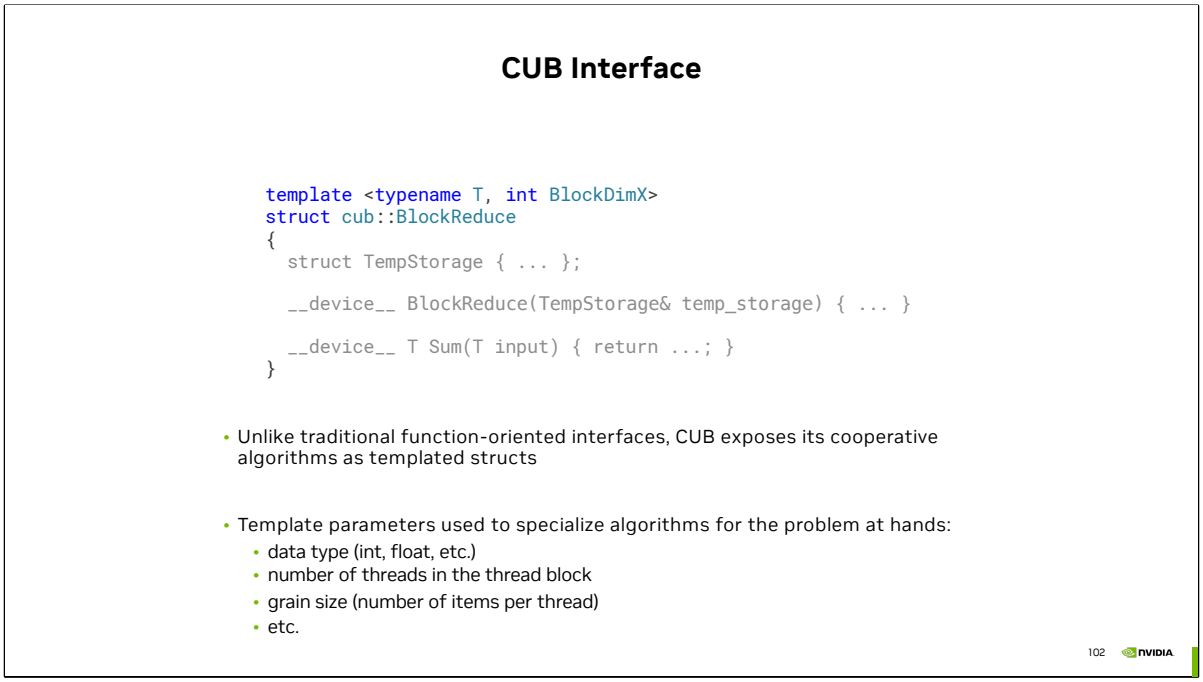
Mental Model

- Cooperative algorithms frequently need shared memory for inter-thread communication
- All threads of a block are expected to participate in block-level cooperative algorithm
- Sometimes output is valid only in main thread



101 NVIDIA

Now, let's look more closely at cooperative algorithms. On the slide you can see how cooperative reduction might work. Here, all threads provide a single value into cooperative reduction. First thread contributes 0, second one contributes 1 at so forth. Inside the cooperative algorithm, these values are written into shared memory. After thread block synchronization, a smaller set of threads computes partial sums and writes them back. After one more synchronization, the total sum (6 in our simplified example) is returned to the first thread. Although this is a very basic model, it illustrates two key points about cooperative algorithms: 1. they rely on shared memory for data exchange between threads 2. they may include synchronization steps. This means if any thread in the block fails to call the cooperative algorithm, the entire kernel could deadlock.



Let's see how this intuition applies to real code. On the screen you can see a cooperative reduction provided by CUB. Unlike traditional function-oriented interfaces, CUB exposes its cooperative algorithms as templated structures. Template parameters are used to specialize algorithms for the

problem at hands. Say, we can specify that we want to reduce integers, within a block of 256 threads.

CUB Interface

```
template <typename T, int BlockDimX>
struct cub::BlockReduce
{
  struct TempStorage { ... };

  __device__ BlockReduce(TempStorage& temp_storage) { ... }

  __device__ T Sum(T input) { return ...; }
}
```

- Nested `TempStorage` type provides type of temporary storage needed by cooperative algorithm for thread communication
- Instance of this type has to be allocated in shared memory
- `TempStorage` allocation must be provided to construct an instance of the algorithm

103 NVIDIA

Next, there's a nested TempStorage type, which specifies the kind of temporary storage the cooperative algorithm needs for its internal communication. We allocate an instance of this type in shared memory, then pass a reference to that shared-memory instance when we construct the cooperative algorithm.

CUB Interface

```
template <typename T, int BlockDimX>
struct cub::BlockReduce
{
  struct TempStorage { ... };

  __device__ BlockReduce(TempStorage& temp_storage) { ... }

  __device__ T Sum(T input) { return ...; }
}
```

- Member functions represent different flavors of a given cooperative algorithms

```
__shared__ cub::BlockReduce<int, 4>::TempStorage storage;

int block_sum = cub::BlockReduce<int, 4>(storage).Sum(threadIdx.x);
```

104 NVIDIA

Finally, member functions provide different flavors of a given cooperative algorithm.

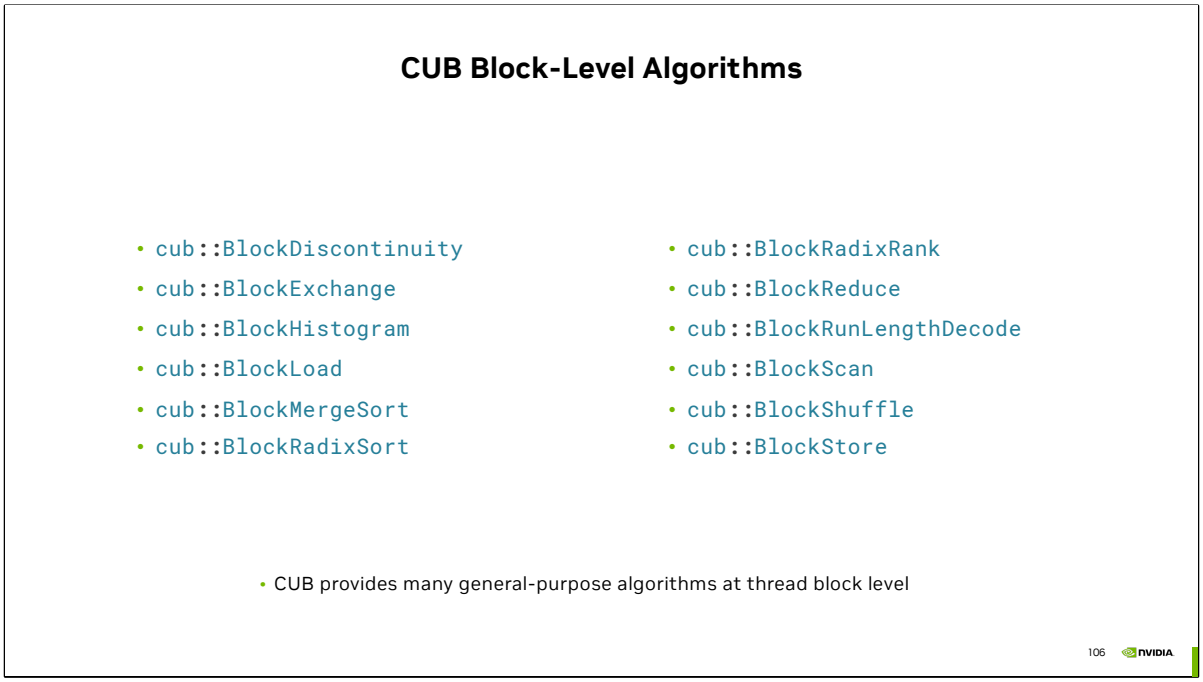
CUB Interface

```
__shared__ cub::BlockReduce<int, 4>::TempStorage storage;

cub::BlockReduce<int, 4> reducer(storage);

int block_sum = reducer.Sum(threadIdx.x);
```

1. Allocate temporary storage in shared memory
2. Construct an instance of the algorithm
3. Invoke method of the algorithm instance

105 NVIDIA

So, putting it all together, usage of this cooperative algorithm looks as follows. We instantiate block reduction for reducing integers within a thread block of size 4. Then allocate nested temp storage type in shared memory, construct an instance of this cooperative algorithm structure and finally call a cooperative algorithm.



CUB provides many general-purpose algorithms at a thread block level

CUB Block-Level Algorithms

- cub::BlockDiscontinuity
- cub::BlockExchange
- cub::BlockHistogram
- cub::BlockLoad
- cub::BlockMergeSort
- cub::BlockRadixSort
- cub::BlockRadixRank
- cub::BlockReduce
- cub::BlockRunLengthDecode
- cub::BlockScan
- cub::BlockShuffle
- cub::BlockStore

- CUB provides many general-purpose algorithms at thread block level

And one of these cooperative algorithms is actually a block histogram

CUB Block-Level Algorithms

```
template <
  typename T,                     -  type of histogram bins (int)
  int BlockDimX,                  -  number of threads in a thread block
  int ItemsPerThread,             -  number of bins in each thread
  int Bins,                       -  number of bins in the histogram
  cub::BlockHistogramAlgorithm    -  algorithm for histogram implementation
    Algorithm = BLOCK_HISTO_SORT>
class cub::BlockHistogram
{
  struct TempStorage
  { ... };

  __device__ BlockHistogram(TempStorage& temp_storage)
  { ... }

  template <typename CounterT>
  __device__ void Histogram(T (&items)[ITEMS_PER_THREAD],  -  thread bins
                            CounterT histogram[BINS])      -  block histogram (in shared memory)
  { ... }
};
```

- `cub::BlockHistogram::Histogram` initializes shared histogram and incorporates bin indices provides by each thread

108 NVIDIA

Block histogram structure has a few template parameters as well. First, we have to specify the type of the histogram elements, which is int in our case. Then, we have to specify the thread block size, like 256. Then, we also specify how many bins each thread is going to contribute. And lastly, we have to specify how many bins are there in our histogram. Then, the block histogram usage is very similar to what we've seen before.

Exercise: Use Cooperative Histogram

- Use `cub::BlockHistogram` to compute block histogram

```
__shared__ int block_histogram[10];

int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bins[1] = {static_cast<int>(temperatures[cell] / 10)};

if (threadIdx.x < 10)
{
  block_histogram[threadIdx.x] = 0;
}
__syncthreads();

cuda::atomic_ref<int, cuda::thread_scope_block>
  block_ref(block_histogram[bin]);
block_ref.fetch_add(1);
__syncthreads();
...
```

03.06-Cooperative-Algorithms/03.06.02-Exercise-Cooperative-Histogram.ipynb

Which brings us to the final exercise in this section.This time, you'll have to use cub::BlockHistogram in our histogram kernel instead of implementing it from scratch.

Exercise: Use Cooperative Histogram

Solution

```
__shared__ int block_histogram[10];

int cell = blockIdx.x * blockDim.x + threadIdx.x;
int bins[1] = {static_cast<int>(temperatures[cell] / 10)};

using histogram_t = cub::BlockHistogram<int, block_size, 1, 10>;
__shared__ typename histogram_t::TempStorage temp_storage;

histogram_t(temp_storage).Histogram(bins, block_histogram);




__syncthreads();
...
```

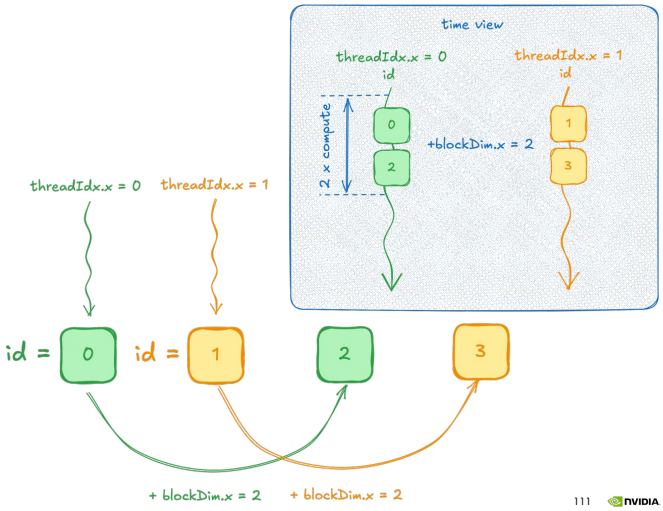
Takeaways

Write custom CUDA kernels only when:

- Your use case is not covered by existing accelerated libraries

10 compute steps take:

- cub: 0.000043
- 1 thread: 4.15
- 2 threads: 2.6
- 256 threads: 0.037
- 5'120'000 threads: 0.0003



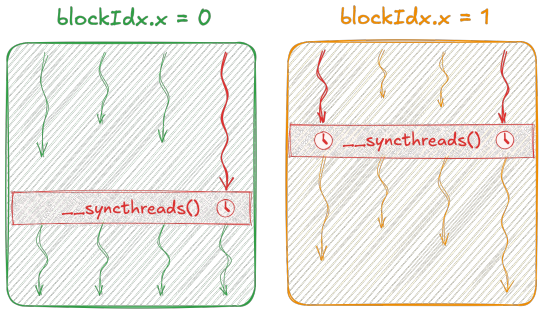
Takeaways

Write custom CUDA kernels only when:

- Your use case is not covered by existing accelerated libraries

Use `__syncthreads`:

- To avoid data races within a thread block



NVIDIA

Takeaways

Write custom CUDA kernels only when:

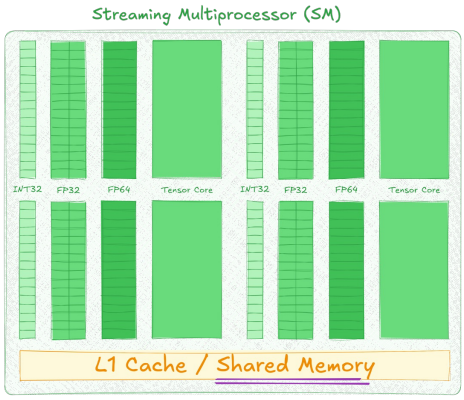
- Your use case is not covered by existing accelerated libraries

Use `__syncthreads`:

- To avoid data races within a thread block

Use `__shared__`:

- When you need a temporary storage inside a thread block
- When you need to exchange data between threads of a thread block
- When your data is not efficiently cached by L1



NVIDIA

Takeaways

Write custom CUDA kernels only when:

- Your use case is not covered by existing accelerated libraries

Use `__syncthreads`:

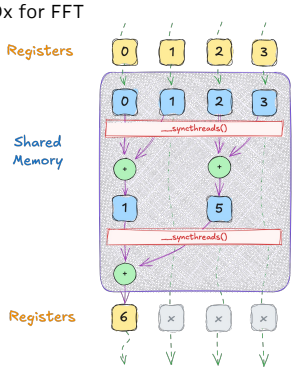
- To avoid data races within a thread block

Use `__shared__`:

- When you need a temporary storage inside a thread block
- When you need to exchange data between threads of a thread block
- When your data is not efficiently cached by L1

Use device-side libraries:

- CUB for block, and warp-scope cooperative general-purpose algorithms
- libcudacxx for vocabulary types
- cuBLASDx for linear algebra
- cuFFTDx for FFT
- etc.



Takeaways

Write custom CUDA kernels only when:

- Your use case is not covered by existing accelerated libraries

Use `__syncthreads`:

- To avoid data races within a thread block

Use `__shared__`:

- When you need a temporary storage inside a thread block
- When you need to exchange data between threads of a thread block
- When your data is not efficiently cached by L1

Use device-side libraries:

- CUB for block, and warp-scope cooperative general-purpose algorithms
- libcudacxx for vocabulary types
- cuBLASDx for linear algebra
- cuFFTDx for FFT
- etc.

Use developer tools:

- compute-sanitizer for CUDA kernels correctness checking
- cuda-gdb for CUDA kernels debugging
- Use NVIDIA Nsight Systems to identify bottlenecks in your application
- NVIDIA Nsight Compute for in-depth kernel profiling


NVIDIA