Implementing New Algorithms with CUDA Kernels

Section at a Glance

Learn the notion of 3.2 **CUDA Kernels** as we implement our first CUDA algorithm

Learn about 3.3 **Atomics** Memory operations

Learn the notion of thread block 3.4 **Synchronization**

Learn the notion of 3.5 **Shared Memory** as accelerate out kernel and achieve 4x speedup

Learn the notion of 3.6 **Cooperative Algorithms** as we coarse the grid and accelerate IO

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

CUDA Kernels

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

CUDA Kernels

- Device functions are:
  - compiled into GPU instructions
  - invoked from device
  - executed on device

CUDA Kernels

- Kernels are:
  - launched with triple chevron syntax <<<?, ?, ?, stream>>> from **CPU**
  - executed in **device execution space**
  - **asynchronous, and**
  - **parallel**

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 dli::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!

CUDA Kernels

```
__global__
void single_thread_kernel(dli::temperature_grid_f in, float *out)
{
  for (int id = 0; id < in.size(); id++)
  {
    out[id] = dli::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(dli::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  single_thread_kernel<<<1, 1, 0, stream>>>(in, out);
}
```

CUDA Kernels

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

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(dli::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] = dli::compute(id, in);
  }
}

void simulate(dli::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

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(dli::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] = dli::compute(id, in);
  }
}
```

CUDA extends device code with a few variables:

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

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(dli::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] = dli::compute(id, in);
  }
}
```

Same kernel is executed by multiple GPU threads

```
threadIdx.x = 0

int thread_index = 0;

out[0] = dli::compute(0, in);
out[2] = dli::compute(2, in);
```

```
threadIdx.x = 1

int thread_index = 1;

out[1] = dli::compute(1, in);
out[3] = dli::compute(3, in);
```

CUDA Kernels

```
const int number_of_threads = 2;

__global__
void block_kernel(dli::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] = dli::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

CUDA Kernels

```
const int number_of_threads = 256;

__global__
void block_kernel(dli::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] = dli::compute(id, in);
  }
}

void simulate(dli::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

CUDA Kernels

```
const int number_of_threads = 2048;

__global__
void block_kernel(dli::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] = dli::compute(id, in);
  }
}

void simulate(dli::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
```

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 grid
- `blockDim.x` stores the number of threads in the block
- `gridDim.x` stores the number of blocks in the grid

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 grid
- `blockDim.x` stores the number of threads in the block
- `gridDim.x` stores the number of blocks in the grid

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 grid
- `blockDim.x` stores the number of threads in the block
- `gridDim.x` stores the number of blocks in the grid

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(dli::temperature_grid_f in,
              float *out,
              cudaStream_t stream)
{
  int block_size = 256;
  ...

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

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(dli::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;
}
```

CUDA Kernels

```
__global__
void grid_kernel(dli::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] = dli::compute(id, in);
  }
}

void simulate(dli::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

Exercise: Detect Asymmetry

5 minutes

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

```
void symmetry_check(dli::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

Exercise: Detect Asymmetry

Solution

```
__global__
void symmetry_check_kernel(dli::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(dli::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

Exercise: Detect Asymmetry

Solution

```
__global__
void symmetry_check_kernel(dli::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(dli::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

Exercise: Detect Row Asymmetry

8 minutes

- 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(dli::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

Exercise: Detect Row Asymmetry

Solution

```
__global__ void row_symmetry_check_kernel(dli::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(dli::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

Exercise: Detect Row Asymmetry

Bug in the solution

```
__global__ void symmetry_check_kernel(dli::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

Fixed Row Symmetry Check

```
__global__ void symmetry_check_kernel(dli::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

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(dli::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

Assertions

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


__global__ void row_symmetry_check_kernel(dli::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.
```

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?

Histogram

- Histogram groups data into bins by counting how many values fall into each bin
- For our case, it shows overall distribution of temperatures
- Each bin covers a temperature range, and the bar height shows how many cells lie in that range

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

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

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

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

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

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

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

Something Went Wrong

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

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!

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the Data Race

Source of the 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);
```

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

Atomic Memory Operations

Atomic Memory Operations

Atomic Memory Operations

Atomic Memory Operations

Atomic Memory Operations

Atomic Memory Operations

Atomic Memory Operations

Exercise: Fix Histogram

5 minutes

- 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

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

Exercise: Fix Histogram

Solution

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

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?

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

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

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

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

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

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

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

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

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

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

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`

Thread Scope System

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

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

- Equivalent to `cuda::std::atomic_ref`
- Each threads of a given system is related to each other thread by *system thread 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*

Thread Scope Block

```
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 in the same thread block by the *block thread scope*

Exercise: Fix Histogram

8 minutes

- 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

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

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?

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?

- GPUs consist of building blocks called **SM**s
- VRAM is called global memory (not local to any group of threads)

- Thread block never spans multiple **SM**s
- Some resources, like `__syncthreads` and **L1** are local to **SM**s

- Our kernel repeatedly issues atomic increments, which is likely going into L2
- But there's some inefficiency. We don't actually need block histograms to be written into global memory. We also waste quite a lot of space (one histogram per thread block)
- It'd be nice if we could allocate memory in L1

- We actually can do that!
- CUDA provides a software-defined cache that's called shared memory
- Shared memory is only accessible within a given thread block and is collocated with the L1

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

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`

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]);
    }
  }
}
```

Exercise: Optimize Histogram

5 minutes

- 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

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

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

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

Exercise: Optimize Histogram

Solution

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

Cooperative Libraries

CUB

Cooperative general-purpose algorithms inside CUDA kernel

cuBLASDx

Cooperative linear algebra functions inside CUDA kernel

cuFFTDx

Cooperative fast Fourier transform inside CUDA kernel

…

- Authoring a CUDA kernel doesn’t mean implementing every algorithm from scratch
- CUDA provides variety of cooperative libraries that make your kernels faster and development times shorter

Cooperative Libraries

CUB

Cooperative general-purpose algorithms inside CUDA kernel

cuBLASDx

Cooperative linear algebra functions inside CUDA kernel

cuFFTDx

Cooperative fast Fourier transform inside CUDA kernel

...

- Authoring a CUDA kernel doesn’t mean implementing every algorithm from scratch
- CUDA provides variety of cooperative libraries that make your kernels faster and development times shorter
- But what exactly does cooperative mean?

Cooperative Algorithms

Serial

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

Cooperative Algorithms

Serial

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

Cooperative

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

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

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

CUB Interface

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

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

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

- Unlike traditional function-oriented interfaces, CUB exposes its cooperative algorithms as templated structs

- Template parameters used to specialize algorithms for the problem at hands:
  - data type (int, float, etc.)
  - number of threads in the thread block
  - grain size (number of items per thread)
  - etc.

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

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);
```

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

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

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

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

Exercise: Use Cooperative Histogram

8 minutes

- 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

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

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

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

Final Assessment

Final Review

Learn 1.2 **Execution Spaces** as we accelerate naïve cooling simulator and achieve16x speedup

Learn 1.3 **Extending Algorithms** with fancy iterators to achieve 2x speedup

Learn 1.4 **Vocabulary Types** as we transition from naive cooling to stencil-based simulation

Learn concept of 1.5 **Parallelism** as we accelerate one of the algorithms by 300x

Learn concept of 1.6 **Memory Spaces** as we accelerate one of the algorithms by 100x

Final Review

Learn the notion of 2.2 **Asynchrony** as we overlap computation and IO to achieve 2x speedup

Learn the notion of 2.3 **CUDA Streams** as we overlap computation with memory transfers

Learn the notion of 2.4 **Pinned Memory** as we make memory transfers asynchronous

Final Review

Learn the notion of 3.2 **CUDA Kernels** as we implement our first CUDA algorithm

Learn about 3.3 **Atomics** Memory operations

Learn the notion of thread block 3.4 **Synchronization**

Learn the notion of 3.5 **Shared Memory** as accelerate out kernel and achieve 4x speedup

Learn the notion of 3.6 **Cooperative Algorithms** as we coarse the grid and accelerate IO

Final Assignment

Step 1

04.01-Assessment/assessment.ipynb

- You are given a CPU simulator of Maxwell's equations
- As part of step 1, you are expected to:
  - Use explicit memory spaces instead of `std::vector`
  - Use explicit execution space specifiers
  - Use GPU-accelerated parallel algorithms instead of `std::transform` etc.
- You are not expected to:
  - Understand the computations
  - Modify any of the lambda functions
  - Use fancy iterators (yet)

```
void update_hx(
  int n, float dx, float dy, float dt, std::vector<float> &hx,
  std::vector<float> &ez, std::vector<float> &buffer)
{
  std::transform(ez.begin() + n, ez.end(), ez.begin(), buffer.begin(),
                 [](float x, float y) { return x - y; });

  std::transform(hx.begin(), hx.end() - n, buffer.begin(), hx.begin(),
                 [dt, dx, dy](float h, float cex) {
                   return h - dli::C0 * dt / 1.3f * cex / dy;
                 });
}
```

...

Final Assignment

Step 2

04.01-Assessment/assessment.ipynb

- You start with code from step 1
- As part of step 2, you'll have to:
  - Use fancy iterators to replace unnecessary materialization
  - Remove any temporary memory allocation remaining in the code from step 1

```
// Materialize cell indices (`i`) in memory
std::vector<int> cell_ids(cells);

for (int i = 0; i < cells; i++)
{
  cell_ids[i] = i;
}
```

Final Assignment

Step 3

04.01-Assessment/assessment.ipynb

- You start with code from step 2
- As part of step 3, you'll have to:
  - Use `cub::BlockReduce` to compute average temperature in a thread block to coarsen grid tile and save IO traffic

```
__global__ void kernel(
  dli::temperature_grid_f fine,
  dli::temperature_grid_f coarse)
{
  ...

  float thread_value = fine(fine_row, fine_col);
  float block_sum = ...;

  coarse(coarse_row, coarse_col) = block_average;
}
```

NVIDIA®