CUDA, Part Two

CS 3220 / CS 5220 Lecture 4-D
Jason Hibbeler
University of Vermont
Spring 2024

Topics

- Dot product
- Shared memory
- Synchronization and potential deadlock
- 2-D computations

Shared Memory

Up until now, we haven't taken advantage of the fact that threads are organized into thread blocks

- we've just been treating each thread individually, without regard to the other threads in the same thread block

As it turns out, there is a way for the threads in each thread block to cooperate

- and have access to the fast, block-local memory on each SM
- the memory we've been using up until now is off-chip memory, cached as needed
- but this hasn't caused a significant problem, since we've just been hitting each element one time

Shared Memory

This block-local memory is called *shared memory*

We declare this block-local storage with `__shared__`

For example:

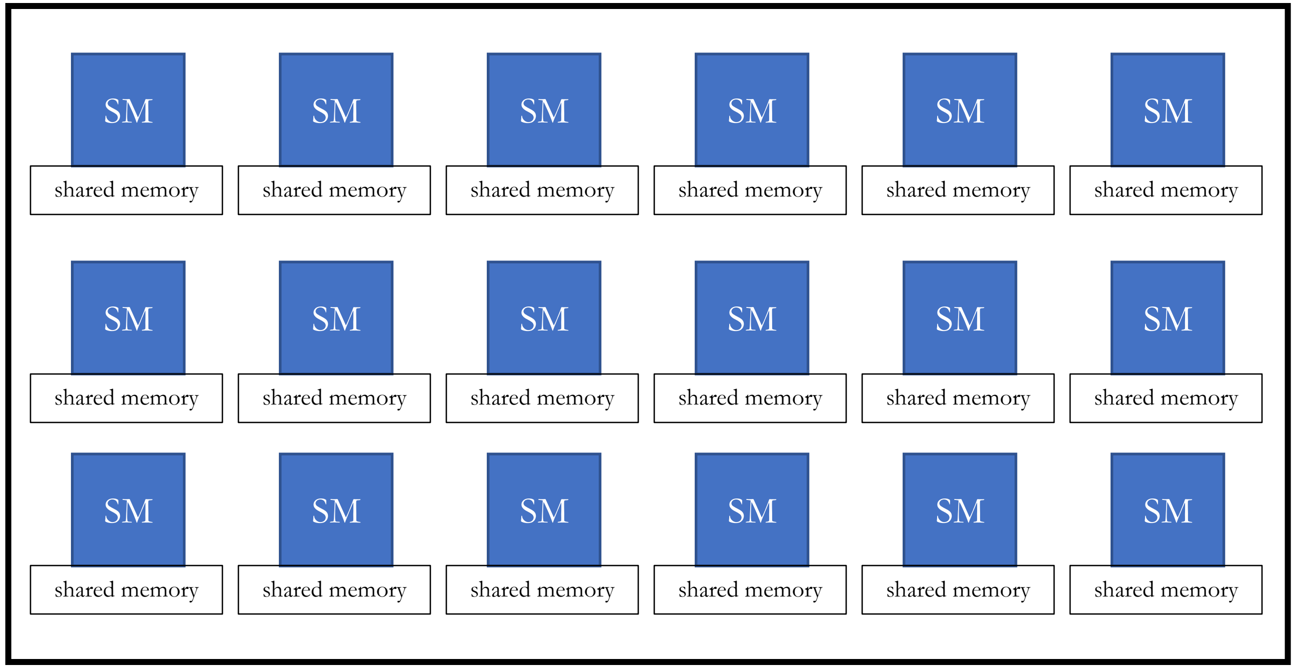
```
__shared__ float localCache[BLOCK_SIZE];
```

Every thread in the thread block can see this memory

- but threads in other blocks cannot—they have their own local shared memory

Shared Memory

This block-local memory is called *shared memory*



this is the GPU

Dot Product

Recall the dot product of two vectors of length n:

$$U \cdot V = u_0 v_0 + u_1 v_1 + \cdots + u_{n-1} v_{n-1}$$

This is an easily parallellized computation

- except for the summation!
- the summation effectively serializes the computation
- more accurately: the summation requires a non-parallel computation
- in general, this is called a reduction operation: reduce many values to a single values

Reduction

A *reduction* (also known as a reduce operation)

- takes some number of values
- and performs a calculation to produce fewer values

This is fundamentally different from the vector addition

- or from the first step ($u_i v_i$) of the dot product

Dot Product

Solution: divide this into subproblems

$$\underbrace{u_0 v_0 + u_1 v_1 + u_2 v_2 + u_3 v_3}_{\text{here's a region}} + \underbrace{u_4 v_4 + u_5 v_5 + u_6 v_6 + u_7 v_7}_{\text{here's a region}} + \cdots + u_{n-1} v_{n-1}$$

etc.

Then we can assign one thread block to each region

- and each thread block can accumulate its pairwise sum in the block-local memory

Dot Product

Details:

- suppose we have n blocks, and each block has m threads
- each thread does a product $u_i v_i$ and writes its result to `localCache[i]` for all the $u_i v_i$ in its region
- in each block, we then sum up the m values in `localCache[]` and put the result in an array indexed by the block number
- after the kernel is complete, we then have an array of length n of partial sums
- we do the final sum on the CPU (since n will be relatively small)

First, without a grid-stride loop

Dot Product

Solution: divide this into subproblems

$$\underbrace{u_0v_0 + u_1v_1 + u_2v_2 + u_3v_3}_{\text{thread block \#0}} + \underbrace{u_4v_4 + u_5v_5 + u_6v_6 + u_7v_7}_{\text{thread block \#1}} + \cdots + u_{n-1}v_{n-1}$$

etc.

partial dot product
from thread block #0

partial dot product
from thread block #1

partial dot product from
thread block #n-1

then add these guys up
on the CPU

Dot Product First Phase: Pairwise Product

Solution: divide this into subproblems

$$u_0 v_0 + u_1 v_1 + u_2 v_2 + u_3 v_3 + u_4 v_4 + u_5 v_5 + u_6 v_6 + u_7 v_7 + \cdots + u_{n-1} v_{n-1}$$

`localcache[0]` `localcache[1]` `localcache[2]` `localcache[3]` `localcache[0]` `localcache[1]` `localcache[2]` `localcache[3]`

thread block #0 thread block #1 etc.

This shows what the array indices would be if the block size were four; in reality, the block size will be 256

- the key thing is that each thread block writes to its own local cache

Dot Product: Pairwise Multiplication

Like this:

```
__global__
void dotp( float *U, float *V, float *partialSum, int n ) {
  __shared__ float localCache[BLOCK_SIZE];
  int tidx = threadIdx.x + blockIdx.x * blockDim.x; // my position in the grid
  localCache[threadIdx.x] = U[tidx] * V[tidx];

  // now, we need to add up the values in localCache[]
  // ...
}
```

here we are assuming that #threads is equal to n; that assumption is OK for this simple example

The Partial Sums

After the kernel completes, we'll have `NUM_BLOCKS` partial sums

- i.e., a small number of partial sums that need to be added together (on the order of 100)
- this is too few things to give to the GPU
- the overhead in copying the memory and getting the GPU cores running is too high
- it would be faster to add the partial sums on the CPU (don't forget: the CPU itself is pretty fast!)

The Partial Sums

But before we get to that point:

- each thread block needs to compute its partial sum—from the grid-stride loop
- in other words, each thread block needs to form the sum `localCache[0] + localCache[1] + ... + localCache[m-1]`

There's a slight complexity though:

- each thread in a thread block is running independently of the others, doing its work ($u_i * v_i$)
- I can only add up the values in `localCache[]` after I know that all of the threads in the thread block have completed — this is actually a read-after-write data hazard
- the only way to do this is through the synchronization of the threads in a thread block
- i.e., don't let the addition happen until each thread in the thread block has completed its product

Synchronization

The CUDA function `__syncthreads()` has the following behavior:

- no thread in a thread block can go past `__syncthreads()` until <u>every</u> thread in the thread block has called it
- it's called a *barrier*
- it's a powerful operation, but it can also lead to subtle problems (deadlock)

Synchronization

In the kernel for the dot product, we'll do this:

(1) each thread computes `localCache[i] = U[i] * V[i]`

(2) each thread waits until all of the other threads have finished their computation

(3) then we sum up the values in `localCache[]`

And the second step above is the `__syncthreads()` call

Dot Product

Like this:

```
__global__
void dotp( float *U, float *V, float *partialSum, int n ) {
  __shared__ float localCache[BLOCK_SIZE];
  int tidx = threadIdx.x + blockIdx.x * blockDim.x; // my position in the grid
  localCache[threadIdx.x] = U[tidx] * V[tidx];
  __syncthreads();

  // now, we need to add up the values in localCache[]
}
```

Computing Each Partial Sum

There's a naïve way and a clever way to sum up the `localCache[]` values in each thread block

Naïve: pick one thread in each block to do this sum, say thread #0

Clever: use a parallel algorithm to perform the reduction

Naïve

One thread adds up all of the values in `localCache[]`:

```
__global__ void dotp( float *U, float *V, float *partialSum, int n ) {
  __shared__ float localCache[BLOCK_SIZE];
  int tidx = threadIdx.x + blockIdx.x * blockDim.x;
  localCache[threadIdx.x] = U[tidx] * V[tidx];
  __syncthreads();

  // now, we need to add up the values in localCache[]
  if (threadIdx.x == 0) {
    float temp = 0.0;
    for (int i=0; i<blockDim.x; ++i)
      temp = temp + localCache[i];
      localCache[0] = temp;
  }
  // now put the result (this thread block's partial sum) in the partialSum array
}
```

Clever: Parallel Reduction

Suppose each thread block has m threads

Then let's use $m/2$ threads this way:

- have the first thread do `localCache[0] = localCache[0] + localCache[m/2]`
- have the second thread do `localCache[1] = localCache[1] + localCache[1+m/2]`
- have thread $m/2 - 1$ do `localCache[m/2-1] = localCache[m/2-1] + localCache[m-1]`

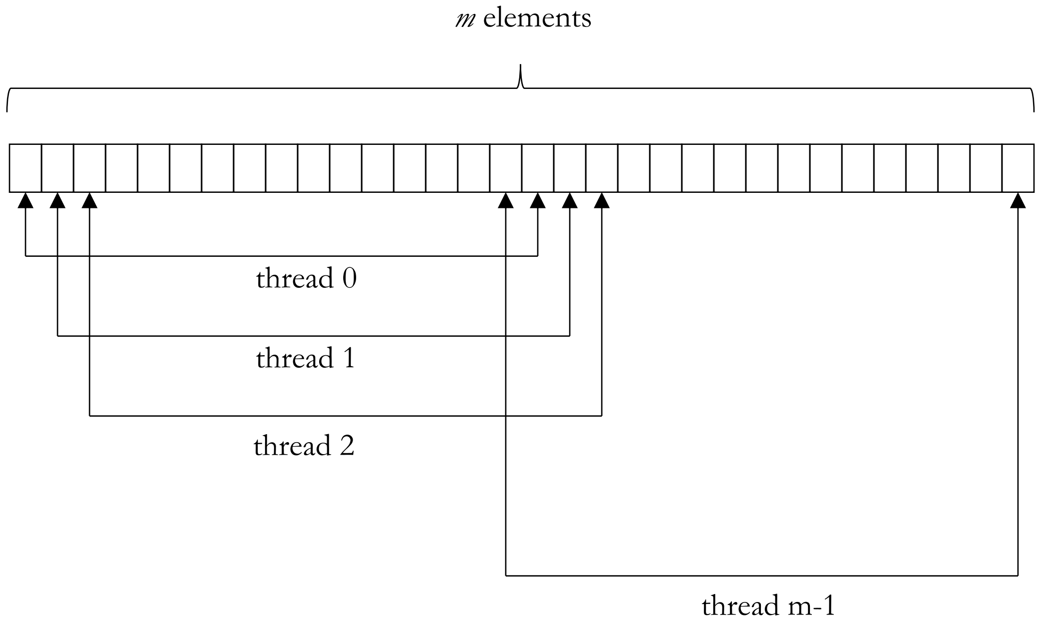
And now we have $m/2$ values to sum!

Then, repeat, with $m/4$ threads

- and successively divide, until we have the final answer sitting in `localCache[0]`
- this assumes that m is an even power of 2, which is a reasonable assumption

Parallel Reduction

Illustration:



Then we'll have $m - 1$ partial sums in array entries 0 to $m - 1$

- and we repeat, using $m/4$ threads
- and repeat again, using $m/8$ threads, etc.

Parallel Reduction: Example with $m = 32$

First pass:

```
localCache[0] = localCache[0] + localCache[16]
localCache[1] = localCache[1] + localCache[17]
...
localCache[14] = localCache[14] + localCache[30]
localCache[15] = localCache[15] + localCache[31]
```

Parallel Reduction: Example with $m = 32$

Second pass:

```
localCache[0] = localCache[0] + localCache[8]
localCache[1] = localCache[1] + localCache[9]
...
localCache[6] = localCache[6] + localCache[14]
localCache[7] = localCache[7] + localCache[15]
```

Parallel Reduction: Example with $m = 32$

Third pass:

```
localCache[0] = localCache[0] + localCache[4]
localCache[1] = localCache[1] + localCache[5]
localCache[2] = localCache[2] + localCache[6]
localCache[3] = localCache[3] + localCache[7]
```

And two more passes gives us the final sum, which will be sitting in `localCache[0]`

Parallel Reduction

After each pass through `localCache[]`, we have to synchronize

- to prevent a read-after-write hazard

Here's the code:

```
cacheIndex = threadIdx.x;
int i = blockDim.x / 2;
while (i > 0) {
  if (cacheIndex < i)
    localCache[cacheIndex] = localCache[cacheIndex] + localCache[cacheIndex + i];
  __syncthreads();
  i = i / 2;
}
```

Parallel Reduction

Last step:

- one thread puts the partial sum in the `partialSum` array
- at an index that is unique to this thread block; specifically, at index `blockIdx.x`

Here's the code:

```
cacheIndex = threadIdx.x;
int i = blockDim.x / 2;
while (i > 0) {
  if (cacheIndex < i)
    localCache[cacheIndex] = localCache[cacheIndex] + localCache[cacheIndex + i];
  __syncthreads();
  i = i / 2;
}

if (cacheIndex == 0)
  partialSum[blockIdx.x] = localCache[cacheIdx];
```

On the CPU Side

Here's the call from the host:

```
U = (float *) malloc(N * sizeof(float));
V = (float *) malloc(N * sizeof(float));
partialSum = (float *) malloc(numBlocks * sizeof(float));
for (int i=0; i<N; ++i) {
  U[i] = (float) (i+1);
  V[i] = 1.0 / U[i];
}

cudaMemcpy( dev_U, U, N*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dev_V, V, N*sizeof(float), cudaMemcpyHostToDevice );
dotp<<<numBlocks, threadsPerBlock>>>( dev_U, dev_V, dev_partialSum, N );
cudaDeviceSynchronize(); // wait for GPU threads to complete; again, not necessary but good pratice
cudaMemcpy( partialSum, dev_partialSum, numBlocks*sizeof(float), cudaMemcpyDeviceToHost );

// finish up on the CPU side
float gpuResult = 0.0;
for (int i=0; i<numBlocks; ++i)
  gpuResult = gpuResult + partialSum[i];
```

Potential Problem

Suppose we put the synchronization inside an if clause:

```
int i = blockDim.x / 2;
while (i > 0) {
  if (cacheIndex < i) {
    localCache[cacheIndex] = localCache[cacheIndex] + localCache[cacheIndex + i];
    __syncthreads(); // only some threads will hit this, causing a deadlock
  }
  i = i / 2;
}
```

`__syncthreads()`

Remember the rule:

- "no thread in a thread block may go past `__syncthreads()` until all threads in the thread block have called it" ("all for one and one for all")
- so if only some of the threads call it (inside the if clause), then those threads are going to wait indefinitely: this is a deadlock

Life lesson from this example:

- use `__syncthreads()` carefully

Dynamic Shared Memory

Before, we declared the size of the shared block in the kernel with a static value

- In other words, the size of the block was known at compile time

```
__shared__ float localCache[BLOCK_SIZE];
```

We can also specify the size dynamically

- and this is certainly more flexible

Dynamic Shared Memory

Specify the size dynamically like this:

```
extern __shared__ float localCache[];
```

And to do this, we have to tell the kernel how big the shared block should be

- and this value becomes a third parameter in the statement that launches the kernel
- the value of the parameter should be the size of the shared region, in bytes

```
dotp<<<numBlocks, threadsPerBlock, blockSize * sizeof(float)>>>( dev_U, dev_V, dev_partialSum, N );
```

Dot Product, with Grid-Stride Loop

Same code, but with a grid-stride loop

- to handle vectors with length larger than total #threads

Dot Product

Like this:

```
__global__ void dotp( float *U, float *V, float *partialSum, int n ) {
  __shared__ float localCache[BLOCK_SIZE];
  int cacheIndex = threadIdx.x; // my position in my threadblock
  int stride = blockDim.x * gridDim.x; // the total number of threads
  float temp = 0.0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i = i + stride)
    temp = temp + U[i] * V[i];

  localCache[cacheIndex] = temp;
  // now, we need to add up the values in localCache[], as shown before
  // ...
}
```

On the CPU Side

Here's the call from the host:

```
U = (float *) malloc(N * sizeof(float));
V = (float *) malloc(N * sizeof(float));
partialSum = (float *) malloc(numBlocks * sizeof(float));
for (int i=0; i<N; ++i) {
  U[i] = (float) (i+1);
  V[i] = 1.0 / U[i];
}

cudaMemcpy( dev_U, U, N*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dev_V, V, N*sizeof(float), cudaMemcpyHostToDevice );
dotp<<<numBlocks, threadsPerBlock, threadsPerBlock * sizeof(float)>>>( dev_U, dev_V, dev_partialSum, N );
cudaDeviceSynchronize(); // wait for GPU threads to complete; again, not necessary but good pratice
cudaMemcpy( partialSum, dev_partialSum, numBlocks*sizeof(float), cudaMemcpyDeviceToHost );

// finish up on the CPU side
float gpuResult = 0.0;
for (int i=0; i<numBlocks; ++i)
  gpuResult = gpuResult + partialSum[i];
```

Two Dimensions

Now suppose we have a 2-D computation

- such as array addition

Now, we will want a 2-D structure for the threads and the thread blocks

CUDA has a built-in datatype `dim3`

- it's a 3-D vector of integers
- the fields are `.x` and `.y` and `.z`
- and each field is initialized to 1 — this is why we were able to use just `blockIdx.x` and `blockDim.x` for the 1-D problems

Two Dimensions

Suppose we have two $N \times N$ matrices

As before, give each thread block a subset of the computation

- but now it's a little trickier, since we want to maintain locality for efficient cache use

We'll store the matrices explicitly as 1-D objects

- with row-major ordering
- so that $A_{i,j}$ will be in `A[(i-1)*N+(j-1)]`
- here we assume that the matrix indices are in $1, \ldots, n$

$$\begin{bmatrix} \texttt{A[0]} & \texttt{A[1]} & \texttt{A[2]} & \ldots & \texttt{A[N-1]} \\ \texttt{A[N]} & \texttt{A[N+1]} & \texttt{A[N+2]} & \cdots & \texttt{A[2*N-1]} \\ \cdots & & & & \\ \texttt{A[N*(N-1)]} & & \cdots & & \texttt{A[N*N-1]} \end{bmatrix}$$

Two Dimensions

Setting up a 2-D array:

```
float *d_x, *d_y, *d_z;
size_t pitch;
cudaMalloc((void**) &d_x, N*N*sizeof(float));
cudaMalloc((void**) &d_y, N*N*sizeof(float));
cudaMalloc((void**) &d_z, N*N*sizeof(float));

cudaMemcpy(d_x, h_x, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, N*N*sizeof(float), cudaMemcpyHostToDevice);
```

Same technique as before

- but now with `N*N` elements

2-D Data: Accessing an Array Element

Now, the position of element $X_{i,j}$ has to take into the number of columns

- we'll assume we have a square matrix, so #columns is the same as #rows

And so, as a matrix (with $1 \leq i,\ j\ \leq N$)

- $X_{i,j}$ will be in `d_x[(i-1)*N + (j-1)]`

With zero-based indexing ($0 \leq i, j\ \leq N - 1$):

- $X_{i,j}$ will be in `d_x[i*N + j]`

Two Dimensions

Launching the kernel:

```
const int BLOCK_SIZE = 16;

dim3 blocks(1, 1, 1);
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE, 1);

blocks.x = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
blocks.y = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;

add2D<<<blocks, threadsPerBlock>>>( d_x, d_y, d_z, N );
cudaDeviceSynchronize(); // this blocks until the device has completed all requested tasks
                         // technically this isn't necessary for these simple kernels
                         // but it's good practice
```

The full block size is 256 (256 threads per block)

- but the blocks are organized as 2-D structures

Two Dimensions: the Kernel

Here's the kernel for matrix addition:

```
__global__
void add2D( float *x, float *y, float *z, int n ) {
  const int tidx = blockDim.x * blockIdx.x + threadIdx.x;
  const int tidy = blockDim.y * blockIdx.y + threadIdx.y;

  if (tidx < n && tidy < n) {
    z[tidx*n + tidy] = x[tidx*n + tidy] + y[tidx*n + tidy];
  }
}
```

The 2-D structure makes this a little more complex

- but we're still indexing a 1-D array
- and here, we're treating the 2-D array as a 1-D array (with zero-based indexing) instead of as a matrix