Warps, Blocks, and Synchronization

K. Cooper[1]

[1] Department of Mathematics
Washington State University

2014

The Trapezoidal Rule

Recall that, given a uniform partition $a = x_0 < x_1 < \cdots < x_N = b$ on an interval $[a, b]$, the composite trapezoidal rule approximates an integral as

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{N-1} f(x_i)\right] + O(h),$$

where $h = (x_{i+1} - x_i)$.

Scalar Trapezoid

We can program this easily enough.

```
integral = 0.;
for(i=1;i<N;i++) integral += f(a+i*h);
integral = h*(integral+0.5*(f(a)+f(b)));
```

This takes about 5.5 seconds to run on our GPU machine.

CUDA Trapezoid

Can we do this using CUDA?
D'oh! Of course!
There is a `for` loop there. We think immediately that it should be made parallel.

But... There is a big problem. The commands in each iteration of the loop depend on the previous iteration. Those familiar with parallel computing recognize this as a *reduce* operation. These are always problematic.

Our Example

We'll integrate the function defined by

$$f(x) = \sin(g(x)) + 2\cos(g(x)),$$

on the interval $[-1, 1]$, where

$$g(x) = -\frac{1}{5}x^3 + \frac{1}{2}x^2 - x + 2.$$

Naturally, being mathematicians, we know we must express g in Horner form:

$$g(x) = 2 + x(-1 + x(0.5 - 0.2x)).$$

The value of the integral is about -0.34702211863386324383.
All examples use 65536 points in the partition.

CUDA Attack

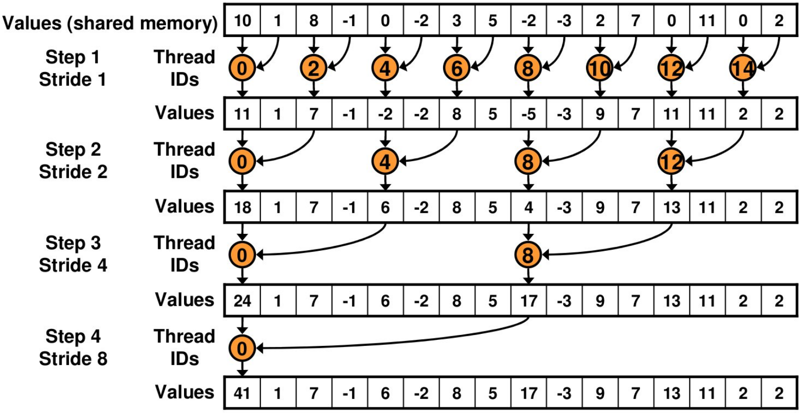
Use a kernel to create an array of function values in parallel.

```
__global__ void
evalfct(double * f_val, double a, double b,
int n){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  double h = (b-a)/(double)(n-1);
  double x = a+i*h;

  f_val[i] = 2.+x*(-1.+x*(0.5-x*0.2));
  f_val[i] = sin(f_val[i])+2*cos(f_val[i]);
  f_val[i] *= h;
  if(i==0) f_val[i] /= 2.;
  if(i==n-1) f_val[i] /= 2.;
}
```

CUDA Reduce

Now all we must do is add up the entries in that array.



Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		2		4		6		8		10		12		14	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				4				8				12			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								8							
	Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8	Thread IDs	0															
	Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

CUDA Reduce

The code looks like this.

```
for(int jump=1;jump<n;jump*=2){
  if(i%(2*jump) != 0) return;
  f_val[i] += f_val[i+jump];
}
```

This runs in about 0.09 seconds on our GPU machine.

It also says the integral value is about -0.326483, though that value changes almost every run.

What’s wrong?!

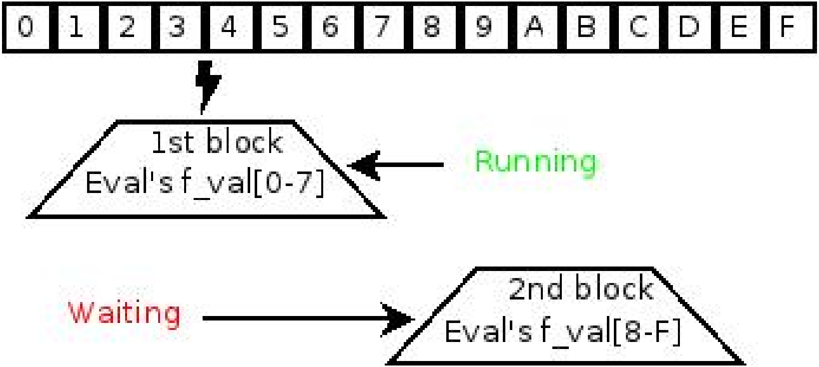
There is nothing we can do about those bad results. Why?

We have 65536 threads. We can work on 2688 of them at a time.
The threads *will* break into blocks, and execute at different times. We need to take control of the blocks, and handle them differently.

This is the fundamental problem facing every shared memory machine: we always wonder whether the value we get from memory has been altered by some other thread.

Execution

The problem is that one set of blocks loads and executes completely before the next set of blocks gets into the GPU.



First block of threads operates only on first 8 entries. `f_val[0]` evaluated *before* `f_val[8]` contains sum of second 8 entries.

Interleaved Reduce

Suppose we have one SM of size 8. . .

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

Stride 1

⋮

11		7		-2		8		-2	-3	2	7	0	11	0	2

Stride 2

⋮

18				6				-2	-3	2	7	0	11	0	2

Stride 4

⋮

24								-2	-3	2	7	0	11	0	2

Stride 8

⋮

22								-2	-3	2	7	0	11	0	2

Blocks

Our code must work on only one block at a time, and it must make sure that all memory has been modified before going to the next stride.

```
for(int jump=1;jump<blockDim.x;jump*=2){
  if(i%(2*jump) != 0) return;
  f_val[i] += f_val[i+jump];
  __syncthreads();
}
```

`__syncthreads()` waits until all threads in block have finished to this point.

Synchthreads()

Suppose we use a block size of 256. SMs on this card have size 192. Blocks are confined to one SM. 64 threads must wait until other threads in block have executed before they start. The `__syncthreads()` function makes sure they keep up.

If you must use `__syncthreads()`, you should probably choose a block size smaller than the SM. Otherwise. . . page faults.

Warps

Threads are loaded into SMs by *warp*. Right now, a warp is 32 threads on all NVidia cards. Any block whose thread count is not a multiple of 32 will result in one warp that is not full.Wasted cycles on some SPs.
On the other hand, whenever a block has more threads than are available on the assigned SM, some warps will not execute at the same time as the rest of the warps from that block.

Blocked Reduce

To summarize, we have two levels where shared memory can lead to problems.

- Between blocks. Blocks do not execute at the same time.
- Within blocks. Warps do not necessarily execute at the same time.

We must adhere to these principles.

- Any time we need *all* blocks to run before we proceed with a calculation, we must quit the current kernel and start another.
- Any time a thread refers to any memory that is not directly associated with that thread, it should call `__syncthreads()` first.

Blocked Reduce

The klutzy way to do this is to use the interleaved reduction on blocks, then just do a sequential sum for those subtotals.

This runs in around 0.27 seconds.

Global Memory

We used global memory to hold the functions values.
Now that we are working block by block, we should use shared memory.
Recall that shared memory is local to each SM. It is faster than global memory. It is limited.

When we can, we should use *registers*. We cannot declare these directly, but small static allocations go into registers. These comprise the fastest memory of all.

Shared Memory

Static shared memory...

```
__global__ void evalfct(double * blocksum,
double a, double b, int n){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = threadIdx.x;
  __shared__ double f[BLOCK_DIM];
  __        __      f[BLOCK_DIM];
  double px;
  px = 2.+x*(-1
  px = 2.+x*(-1
  f[j] = sin(px)+2*cos(px);
  f[j] *= h;
  if(i==0) f[j] /= 2.;
  if(i==n-1) f[j] /= 2.;
  __syncthreads();
```

px goes in a register

static shared memory alloc for whole block

Shared Memory

The reduction…

```
  // Now add them up
  for(int jump=1;jump<blockDim.x;jump*=2){
    if(j%(2*jump) != 0) return;
    f[j] += f[j+jump];
    __synchthreads();
  }
  blocksum[blockIdx.x] = f[0];
  blocksum[blockIdx.x] = f[0];
}
```

Put result in global memory when finished

Shared Memory

This simple change to shared memory cuts the overall execution time to about 0.24 seconds.

We should be able to get more! The problem is probably that stupid stupid sloppy sloppy sequential sum at the end. Now we have a global array of block sums. We can feed that to a new reduce function to repeat the procedure on the smaller array.
Converting to that second interleaved call cuts the time to about 0.10 seconds.

Synchronize

Code for multiple kernel calls.

```
jst = blocksPerGrid/BLOCK_DIM;
evalfct«<blocksPerGrid,
BLOCK_DIM»>(d_blocksum,-1.,1.,n);
cudaDeviceSynchronize();
cudaDeviceSynchronize();
for(jblocks=jst;jblocks>0;jblocks/=BLOCK_DIM){
  reduce«<jblocks
    BLOCK_DIM»>                  ;
  cudaDeviceSynchronize();
  cudaDeviceSynchronize();
  last_block = jblocks;
}
reduce«<1,last_block»>(d_blocksum,last_block);
```

Wait until all blocks have run!

Divergent Branches

Branch refers to an `if` statement that results in different paths being taken through the instruction set.
Any time threads in any given warp take different branches, they are said to be *divergent*. This might mean that time is being wasted – e.g. half of threads are not working.

We have divergent branches. *sigh*

Divergent Branch

Here it is…

```
for(int jump=1;jump<blockDim.x;jump*=2){
  if(j%(2*jump) != 0) return;
  f[j] += f[j+jump];
  __syncthreads();
}
```

Every time we go through the loop we drop half the threads, but in an interleaved fashion. Every warp has at least one thread that hangs on to the bitter end.

Divergent Branch

Count differently.

```
for(int jump=blockDim.x/2;jump>0;jump/=2){
  if(j>=jump) return;
  f[j] += f[j+jump];
  __syncthreads();
}
```

This time we get to drop the entire second half of the block after the first pass through the loop. If e.g. 128 threads per block, then two warps can get out of the SM.

Note that it is bad from one standpoint: on the first couple of iterations, the stride is very large - could be bad for cache.

This version executes in about 0.08 seconds.

So What?

We have taken a scalar program that took five seconds to execute, and sped it up so much that it now takes .09 seconds. The scalar code was about 20 lines; the GPU code is a couple hundred. *sigh*

Of course, if this little block of code executed, say 3600 times in our larger program, then we would have changed a 5.5 hour program into a 5.5 minute program.