CISC 372: Parallel Computing
CUDA, part 3

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Synchronization

We will use synchronization at various levels in CUDA:

1. CPU and kernel calls may synchronize
2. different kernel calls may synchronize
3. the threads in a block may synchronize

CPU and kernel calls may synchronize

- kernel calls execute concurrently with host code
- certain calls block on host until all previous kernels calls complete

```
kernel_1<<<X,Y>>>(...);
   // kernel_1 starts execution, CPU continues to next statement
kernel_2<<<X,Y>>>(...);
   // kernel_2 is placed in queue and will start after kernel_1
   // finishes, CPU continues to next statement
cudaMemcpy(...);
   // CPU blocks until memory is copied, memory copy starts
   // only after kernel_2 finishes
```

When explicit synchronization is needed:

- `cudaDeviceSynchronize()`
  - blocks CPU until all kernel calls complete

Synchronization of kernel calls

- in CUDA, it is possible for different kernel calls to execute concurrently
  - a more advanced topic; not needed for most tasks
- by default: all kernel calls execute in sequence (in a "stream")
  - the second kernel call will not begin until the first completes
  - CUDA runtime maintains a queue of kernel calls

Synchronization of threads within a block

The threads within a block execute concurrently: a parallel program.

- `__syncthreads()`
  - a barrier on all threads in the block
  - a memory fence: all reads and writes made by threads to shared variables complete
- this allows threads in the same block to communicate through shared variables
  1. thread 1 writes to shared variable `x`
  2. `__syncthreads()`
  3. thread 2 reads from `x`
- without the `__syncthreads()`, there would be a data race
  - undefined behavior

Warps and synchronization

- in the GPU, threads within a block are organized into `warps`
  - typically, 32 threads in a warp
- the threads in a warp execute in lockstep
  - each executes the same instruction on the same line of code simultaneously
  - the data on which they execute may differ
- what about a branch? `if (x>0) S;`
  - for some threads in the warp, might have `x>0`, for others, not
  - the threads for which the condition is false will `block` waiting for the other threads in the warp
- what about a branch? `if (x>0) S1; else S2;`
  - first, all the true threads in the warp execute `S1`; other threads block
  - then, all the false threads in the warp execute `S2`; other threads block
- usually, you don't have to know about this, except...
  - performance! with more branch divergence, performance goes down
  - `__syncthreads`...

__syncthreads restriction

This code will not work:

```
if (x>0) {
  S1;
  __syncthreads();
} else {
  S2;
  __syncthreads();
}
```

- the threads taking the false branch block, waiting for the true threads
- the threads taking the true branch are stuck inside `__syncthreads` ...deadlock!

Therefore, in CUDA, for a call to `__syncthreads` to succeed...

- all threads must execute the same textual occurrence of `__syncthreads`

This will work:

```
if (x>0) {
  S1;
} else {
  S2;
}
__syncthreads();
```

Shared variables

- shared variables are declared within the kernel scope with `__shared__`
  - arrays can be shared, but length must be a constant expression
  - preprocessor macros are good for this
- access to shared variables is very fast
- there is not a lot of shared memory
  - 48K bytes on all three types of GPUs available to us
  - 48K bytes = 6144 doubles
  - at 256 threads per block: 24 doubles per thread
  - at 512 threads per block: 12 doubles per thread
  - at 1024 threads per block: 6 doubles per thread
- typical use of shared memory:
  1. all threads load some value from global memory into shared memory
  2. `__syncthreads()`
  3. all threads read the shared values repeatedly
- another typical use: reductions across all threads in a block

Example: dot product

- based on example from **CUDA by Example**, Chapter 5
- two vectors `a` and `b` of floats of length N
- compute the dot product of `a` and `b`:
  - $a_0 b_0 + a_1 b_1 + \cdots + a_{n-1} b_{n-1}$
- strategy
  - fix the number of threads per block (256)
  - fix the number of blocks: 120
    - unless N is small — then the number of blocks is $\lceil N/256 \rceil$
  - cyclic distribution of array of length N
  - each thread computes its partial sum
  - threads within a block perform a reduction to get sum for that block
  - each block writes back its block sum to global memory
  - when kernel returns, CPU finishes up the work by adding up the 120 block sums

dot.cu: kernel, part 1: local sums

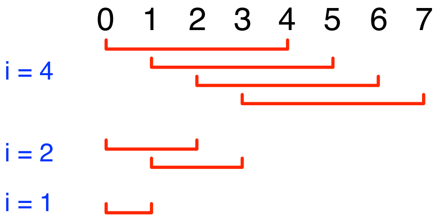
```
/* Does most of the work in computing the dot product of a and b.
   a and b are arrays of length N.
   c is an array of length nblocks.
   Upon return c[i] will hold the portion of the dot product corresponding
   to the indexes for which block i is responsible.
   The final dot product is the sum over all blocks i of c[i].
 */
__global__ void dot(float * a, float * b, float * c) {
  __shared__ float sums[threadsPerBlock];
  const int ltid = threadIdx.x; // local thread ID (within this block)
  const int gtid = ltid + blockIdx.x * blockDim.x; // global thread ID
  const int nthreads = gridDim.x * blockDim.x;
  float thread_sum = 0;

  for (int i = gtid; i < N; i += nthreads) thread_sum += a[i] * b[i];
  sums[ltid] = thread_sum;
  __syncthreads();  // barrier for the threads in this block
```

dot.cu: kernel, part 2: reduction

```
  // reduction over the block. threadsPerBlock must be a power of 2...
  for (int i = blockDim.x/2; i > 0; i /= 2) {
    if (ltid < i) sums[ltid] += sums[ltid + i];
    __syncthreads();
  }
  // at this point, sums[0] holds the sum over all threads.
  if (ltid == 0) c[blockIdx.x] = sums[0];
}
```

- butterfly reduction!
  - similar to butterfly barrier
- threadPerBlock=256, so $\log_2(256) = 8$ iterations
- at end, one thread (0) writes to global memory



dot.cu: host code, part 1

```
#define MIN(a,b) ((a)<(b)?(a):(b))
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)

const int N = 1u<<30;
const int threadsPerBlock = 256;
// use at most 120 blocks.  k40c has 15 SMPs, so that's 8 blocks per
// SMP.  For small values of N, we will use less than 120
// blocks...just enough to have one index per thread...
const int nblocks = MIN(120, (N + threadsPerBlock - 1) / threadsPerBlock);
int main() {
  float * a, * b, * partial_sums, * dev_a, * dev_b, * dev_partial_sums;
  int err;
  double start_time = mytime();

  printf("dot: N = %d, threadsPerBlock = %d, nblocks = %d, nthreads = %d\n",
         N, threadsPerBlock, nblocks, threadsPerBlock*nblocks);
  a = (float*)malloc(N*sizeof(float));  assert(a);
  b = (float*)malloc(N*sizeof(float));  assert(b);
```

dot.cu: host code, part 2

```
partial_sums = (float*)malloc(nblocks*sizeof(float));
err = cudaMalloc((void**)&dev_a, N*sizeof(float));  assert(err == cudaSuccess);
err = cudaMalloc((void**)&dev_b, N*sizeof(float));  assert(err == cudaSuccess);
err = cudaMalloc((void**)&dev_partial_sums, nblocks*sizeof(float));
for (int i = 0; i < N; i++) {
  a[i] = i;
  b[i] = i*2;
}
err = cudaMemcpy(dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice);
assert(err == cudaSuccess);
err = cudaMemcpy(dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice);
assert(err == cudaSuccess);
dot<<<nblocks, threadsPerBlock>>>(dev_a, dev_b, dev_partial_sums);
err = cudaMemcpy(partial_sums, dev_partial_sums, nblocks*sizeof(float),
                 cudaMemcpyDeviceToHost);
assert(err == cudaSuccess);
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_partial_sums);
```

dot.cu: host code, part 3

```
  float result = 0.0f;
  float expected = 2 * sum_squares((float)(N - 1));

  for (int i = 0; i < nblocks; i++) result += partial_sums[i];
  printf("Result = %.12g.  Expected = %.12g.  Time = %lf\n",
 result, expected, mytime() - start_time);
  fflush(stdout);
  assert(result/expected <= 1.0001);
  assert(expected/result <= 1.0001);
  free(a);
  free(b);
  free(partial_sums);
}
```