Table of Contents

1. CUDA Thread Parallelism (S&K, Ch5)
2. Thread Parallelism
3. CUDA SHMEM & Syncrhr
Reading List for GPU/CUDA

- **Reading:** CUDA API: Sanders & Kandrot
  - Intro to CUDA, Ch3
  - Block Parallelism, Ch 4
  - Thread Parallelism, Ch 5

- **Tutorials**
  - CUDA Tutorial:
  - CUDA API:
  - CUDA SDK:
  - CUDA example files on tuckoo in /COMP605/cuda

- **GPU Architectures:**
  - References: NVIDIA online documents
    and lecture notes by S.Weiss

- **More relevant reading**
  - Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model. Robert Hochberg, August 11, 2012
CUDA Thread Parallelism (S&K, Ch5)
Outline:

55555
NVIDIA GPU Threads

- CUDA threads are extremely lightweight: context change between two threads is not a costly operation.
- Threads are grouped into blocks that can contain 64 to 1024 threads (depending on arch).
- Blocks are grouped into Grids.
- Each thread has a unique local index in its block, and each block has a unique index in the grid.
- Warp: a group of 32 threads: minimum size of data processed in SIMD fashion by CUDA multiprocessor.
  - weaving term – designates threads arranged lengthwise on a loom and crossed by the woof.
We split the blocks into threads

Threads can communicate with each other

You can share information between blocks (using global memory and atomics, for example), but not global synchronization.

Threads can be synchronized using `syncthreads()`.

Block parallelism: call kernel with N blocks, 1 thread per block
add<<<N,1>>>( dev_a, dev_b, dev_c);
N blocks x 1 Thread/block = N parallel threads

Block parallelism: split the blocks.

call kernel with 1 block, N threads per block
add<<<1,N>>>( dev_a, dev_b, dev_c);
1 block x N Thread/block = N parallel threads
#include "../common/book.h"

#define N 10

__global__ void add( int *a, int *b, int *c )
{
    int tid = threadIdx.x; <--- cuda variable
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void )
{
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc((void**)&dev_a,N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_b,N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_c,N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ));
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ));

<<<---- call kernel with 1 block, N threads per block
    add<<<1,N>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a  ) );
    HANDLE_ERROR( cudaFree( dev_b  ) );
    HANDLE_ERROR( cudaFree( dev_c  ) );

    return 0;
}
Thread Parallelism: add_loop_blocks.cu (output)

```
[/var/spool/torque/mom_priv/jobs/36453.tuckoo.sdsu.edu.SC: line 10: -r: command not found
------------------------------------------------------
Job is running on node node10
------------------------------------------------------
PBS: qsub is running on tuckoo.sdsu.edu
PBS: originating queue is batch
PBS: executing queue is batch
PBS: working directory is /home/mthomas/pardev/cuda/cuda_by_example/chapter05
PBS: execution mode is PBS_BATCH
PBS: job identifier is 36453.tuckoo.sdsu.edu
PBS: job name is add_loop_blocks
PBS: node file is /var/spool/torque/aux//36453.tuckoo.sdsu.edu
PBS: current home directory is /home/mthomas
PBS: PATH = /opt/pgi/linux86-64/2011/mpi/mpich/include/:/usr/lib64/qt-3.3/bin:/usr/local/bin:
------------------------------------------------------
0 + 0 = 0
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
9 + 81 = 90
```
Thread Parallelism – longer vectors

What happens \#threads is larger than \#blocks*thds requested?

- cannot exceed $maxThreadsPerBlock$, typically 512
- tuckoo.sdsu.edu:
  - Max threads per block: 512
  - Max thread dimensions: (512, 512, 64)
  - Max grid dimensions: (65535, 65535, 1)
  - What is Max Array dimension?
- we will need to distribute the threads
- for large N, we will need a combination of threads and blocks
- we need to convert from a 2D $[\text{block}, \text{thread}]$ space to a 1D indexing scheme
  \[ tid = \text{threadIdx}.x + \text{blockIdx}.x \times \text{blockDim}.x; \]
- \text{blockDim} is constant, stores number of threads along each dimension
- similar to \text{gridDim} which stores number of blocks along each dimension
### Thread Parallelism

Note: for this example, threads represent columns, blocks represent rows.

<table>
<thead>
<tr>
<th>blockDim.x</th>
<th>tid=threadIdx.x + blockIdx.x*blockDim.x</th>
<th>Th0</th>
<th>Th1</th>
<th>Th2</th>
<th>Th3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0+0*4=0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0+1*4=4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0+2*4=8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0+3*4=12</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Vector $V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$

$\text{Elem}(B, TID) = [0, 0, 0, 1, 0, 2, 0, 3, 1, 0, 1, 1, 1, 2, 1, 3, 2, 0, 2, 1, 2, 2, 3, 2, 3, 3, 0, 3, 1]$
Thread Parallelism

Distribute threads using grid dimension as a stride, until all are gone (while loop)

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

This looks remarkably like our original version of vector addition! In fact, compare it to the following CPU implementation from the previous chapter:

```c
void add( int *a, int *b, int *c ) {
    int tid = 0;       // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;        // we have one CPU, so we increment by one
    }
}
```
add_loop_long_blocks.cu (K&S, Ch5)

```
#include "../common/book.h"

#define N (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // allocate the memory on the GPU
    HANDLE_ERROR(cudaMalloc((void**)&dev_a, N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_b, N*sizeof(int)));
    HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
    add<<<128,128>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error: %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
    }
    if (success) printf( "We did it!\n" );

    // free the memory we allocated on the CPU
    free( a ); free( b ); free( c );

    return 0;
}
```
Ripple Example: Extend to 2D matrix:

```c
__global__ void kernel( unsigned char *ptr, int ticks )
{
// map from threadIdx/BlockIdx to pixel position
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;

...}
```

*Figure 5.2* A 2D hierarchy of blocks and threads that could be used to process a 48 x 32 pixel image using one thread per pixel.
The CUDA Memory Model

- The kernel is executed by a batch of threads.
- Threads are organized into a grid of thread blocks.
- Each thread has its own registers, no other thread can access it.
- The kernel uses registers to store private thread data.
- Shared memory: allocated to thread blocks - promotes thread cooperation.
- Global memory: host/threads can read/write.
- Constant and texture memory: host/threads read only.

- Threads in same block can share memory.
- Requires synchronization – essentially communication.
- Example: Dot product:
  \[(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4\]

Source: NVIDIA
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) ) );
    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice ) );
    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );
    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost ) );
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }
    #define sum_squares(x) (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", c,
            2 * sum_squares( (float)(N - 1) ) );
    // free memory on the gpu side
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_partial_c ) );
    // free memory on the cpu side
    free( a );
    free( b );
    free( partial_c );
Shared Memory Model: dot.cu (S&K Ch5)

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock]; // buffer of shared memory - store sum
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0; // each thread computes running sum of product
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp; // set the cache values in the shared buffer
    __syncthreads(); // synchronize threads in this BLOCK

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
Reduction Operation

// for reductions, threadsPerBlock must be a power of 2
//
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex]+=cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
if (cacheIndex == 0) // only need one thread to write to global memory
    c[blockIdx.x] = cache[0];
Calculating smallest multiple of threads per block greater than N

const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

- more efficient (reduce wasted threads)
- should be either 32 or the number calc above
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {
  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
  }
  cache[cacheIndex] = temp; // set the cache values
  __syncthreads(); // synchronize threads in this block

  // for reductions, threadsPerBlock must be a power of 2 because of the following code
  int i = blockDim.x/2;
  while (i != 0) {
    if (cacheIndex < i)
      cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
  }
  if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}
int main( void ) {
    float    *a, *b, c, *partial_c;
    float    *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,blocksPerGrid*sizeof(float) ) );

    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),cudaMemcpyHostToDevice ) );

    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost));
// finish up on the CPU side

    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
    2 * sum_squares( (float)(N - 1) ) );

// free memory on the gpu side
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );

// free memory on the cpu side
free( a );
free( b );