Chapter 5: Thread Coorperation

Splitting parallel blocks

Launch function with **N blocks and M threads**:

```
func<<<N, M>>>(dev_data);
```

Inside add you can obtain the block number with:

```
int tid = blockIdx.x;
```

Inside add you can obtain the thread number with:

```
int tid = threadIdx.x;
```

Limits of blocks

- The hardware limits the number of blocks in a single launch to 65.535

Limits of Threads

- No more then “maxThreadsPerBlock” of threads per block allowed.

- For many gpu’s maxThreadsPerBlock=512 → use combination of threads and blocks.

Standard method for converting for a two-dimensional index to a linear space:

```
int tid = threadId.x + blockIdx.x*blockDim.x
```

- BlockDim: the number of threads along each dimension of the block. (3D: 3-d array of threads per block)
- GridDim: number of blocks along each dimension of the entire grid (2D: 2d grid of blocks)
- So BlockDim+GridDim == 5 dimensions
- use (N+127)/128 instead of std::ceil(N/128)

```
func<<<(N+127)/128, 128 >>>(dev_mem);
```

- Don’t forget to add if to the operation:

```
if(tid < size_vector>{
  out[tid] = input[tid]*2;
}
```

Automatic indexing on any size of gpu

- Combine block and threads and serial execution.
- move by the number of threads (blockDim.x) and the number of blocks(gridDim.x)

```
int tid = threadId.x + blockIdx.x * blockDim.x;
while(tid < size_vector>{
  out[tid] = input[tid]*2;
  tid = tid + blockDim.x * gridDim.x;
}
```

Multi dimension indexing

- You can have up to 2 Dimensions of the grid
- And up to 3 Dimensions on the threads

```
dim3 blocks(2, 2);
dim3 threads(2, 2); // or (2, 2, 2); for 3 dimensions
```

- In the logic itself you can access this with x/y/z

```
__global__ void func(float* input, float* out){
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int tidt = x + y* (blockDim.x*gridDim.x);
  while(tid < size_vector>{
    out[tid] = input[tid]*2;
    tid = tid + blockDim.x * gridDim.x;
  }
}
```

Shared Memory and Synchronization

- “__shared__” C-extension allows threads in the same block to access the same memory.
  - Threads cannot access other blocks memory.
  - The memory is physically on the chip, and not off-chip on the DRAM.

Dot product/Inner product example

- cache is shared among threads, but unique per block → cache index == thread index

```
__global__ void dot(float* a, float*b, float* c){
  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx + blockidx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while (tid < N){
```

```
    temp = temp + a[tid] * b[tid];
    tid = id + blockDIm.x * girdDim.x;
  }
  cache[cachIndex] = temp;

  // synchronize threads in this block.
  __syncthreads();

}
```

- Now reduce the values
  - naive way is to let 1 thread do this after the __syncthreads() call.

```
int i = blockDim.x/2;
while (i != 0) {
  if(cacheIndex < i)
  {
    cache[cacheIndex] = cache[cacheIndex] + cache[cachIndex + 1]
  }
  __syncThreads();
  i = i / 2;
}

if(cacheIndex==0){
  c[blockId.x] = cache[0];
}
```

- The final reduction is typically not done on the gpu, but on the cpu, as it's only as small number of computations the cpu tends to be fast.
- The optimal number of threads/blocks is the highest number of threads that's possible and is still smaller then the vector.
- We limit the size to 32, as your better off using the cpu at that point.

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

The problem with syncthreads

- Don't move syncThreads into the if statement:

```
while (i != 0) {
  if(cacheIndex < i)
  {
    cache[cacheIndex] = cache[cacheIndex] + cache[cachIndex + 1]
    __syncThreads();
  }

  i = i / 2;
}
```

- SynThreads() waits till all threads have executed syncThreads() which will never happen in this case.