Thread Communication and Shared Memory

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
• Communication requires synchronization:
  
  ```c
  __syncthreads()
  ```
  
  • synchronize all threads in a block at this point
  • execution continues only when all threads in the block have executed \textit{up to}
  
  ```c
  __syncthreads()
  ```

```c
__global__ void sumNeighbors(int* globalPtr){
    int i = threadIdx.x;        //put the thread index in a register
    ...
    __syncthreads();            //synchronize all in the block
    ...
}
```
Synchronization Costs

• All threads in a warp will hit the `__syncthreads()` barrier at the same time

• `__syncthreads()` is assumed to be lightweight – very little overhead

• What efficiency costs would you expect?
  • no stalls – execution of the warp terminates and all threads in the block are executed until they reach the same point
  • this can reduce occupancy if only one block is assigned to an SM
Synchronization – Finite Differences

• Calculate the derivative of a function $f(x)$ given a set of $N$ equally spaced samples $f_i = f(x_i)$ where $i \in [0, N - 1]$ and $h = x_i - x_{i-1}$ is the sample spacing

• Central differences – $O(h^2)$ accuracy

$$\frac{d}{dx} f(x) \approx \frac{f(x + h) - f(x - h)}{2h}$$

$$\frac{d}{dx} f[x_i] \approx \frac{f[x_{i+1}] - f[x_{i-1}]}{2h}$$

• Given an array of $N$ discrete samples in host memory
  • estimate the derivative using finite differences
  • minimize the amount of global memory (replace the array with its derivative)
  • what degenerate cases do we have to worry about?
Finite Differences – Handling Edges

• The left and right edges of a function cannot be approximated using central differences because there is missing information.

• Lower-order approximations can be used instead:
  
  • Forward differences – $O(h)$ accuracy
    \[
    \frac{d}{dx} f(x) \approx \frac{f(x + h) - f(x)}{h}
    \]
  
  • Backward differences – $O(h)$ accuracy
    \[
    \frac{d}{dx} f(x) \approx \frac{f(x) - f(x - h)}{h}
    \]
Finite Differences – Configuration

```c
int N;                    //stores the array size
float h;                 //stores the sample spacing
...                      //initialize program
float* f = (float*) malloc(N * sizeof(float));   //allocate host memory
...                      //load the samples into host memory (file, user, etc.)
float* gpu_f;             //create a device pointer

//allocate memory on the device
HANDLE_ERROR( cudaMalloc(&gpu_f, N * sizeof(float)) );
//copy f(x) from the host to the device
HANDLE_ERROR( cudaMemcpy(gpu_f, f, N * sizeof(float), cudaMemcpyHostToDevice) );
int threads = procs.maxThreadsPerBlock;        //use the maximum number of threads
int blocks = N / threads + 1;
kernelderv<<<blocks, threads>>>( gpu_f, h, N );  //call the kernel
```
__global__ void kernelDeriv(float* f, float h, size_t N) {
    size_t xi = blockIdx.x * blockDim.x + threadIdx.x;
    if(xi >= N) return; // exit if outside the function

    size_t bxi = threadIdx.x; // x coordinate within the block
    float dfx; // register stores the derivative

    if(bxi == 0) // beginning of the block
        dfx = (f[xi + 1] - f[xi]) / h; // use forward difference
    else if(bxi == blockDim.x - 1) // end of the block
        dfx = (f[xi] - f[xi - 1]) / h; // use backward difference
    else // if we're in the middle of a block
        dfx = (f[xi + 1] - f[xi - 1]) / (2 * h); // use central difference

    __syncthreads(); // make sure all derivatives are calculated
    f[xi] = dfx; // write the derivative to the input array
}
Finite Differences – Reduce Divergence

```c
__global__ void kernelDeriv( float* f, float h, size_t N ){
    size_t xi = blockIdx.x * blockDim.x + threadIdx.x;
    if(xi >= N) return; //exit if outside the function

    size_t bxi = threadIdx.x; //x coordinate within the block
    float dfx; //register stores the derivative

    size_t xi0 = xi - 1; //register stores lower index
    size_t xi1 = xi + 1; //register stores upper index
    if(bxi == 0) xi0 = xi; //beginning of the block, update lower
    else if(bxi == blockDim.x - 1) xi1 = xi; //end of the block, update upper
    else h *= 2; //middle of block, update step size

    dfx = (f[xi1] - f[xi0]) / h; //use central difference

    __syncthreads(); //make sure all derivatives are calculated
    f[xi] = dfx; //write the derivative to the input array
}
```
Finite Differences – Visualization

- Assume 4 threads per block

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[i]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
<th>t10</th>
<th>t11</th>
<th>t12</th>
<th>t13</th>
<th>t14</th>
<th>t15</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfx</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

__syncthreads()
Shared Memory

- Shared memory is accessible per block
- Shared memory is allocated in the grid configuration
  ```c
  kernelFunc<<< gridDim, blockDim, sharedBytes >>>(...)
  ```
  - the lifetime is the duration of the grid
- A pointer to shared memory is specified in the kernel
  ```c
  extern __shared__ float sharedPtr[];
  ```
  - all pointers specified this way point to the same location in shared memory
  - behave just like normal pointers:
  ```c
definitions...
  ```
  ```c
  __global__ void kernelFunc( float* A, float* B, int N ){
    extern __shared__ unsigned char sharedPtr[]; //pointer to shared memory
    int* intArray = sharedPtr;       //array of ints
    //specify an array of floating point values by dereferencing sharedPtr
    float* floatArray = &sharedPtr[N * sizeof(int)]; //array of floats
    //OR specify an array of floating point values using pointer arithmetic
    float* floatArray = sharedPtr + N * sizeof(int);  //or this
    ...
    //do stuff
  }
  ```
  - all threads in a block have the same value stored in `sharedPtr`
Shared Memory Access

• (code from the previous slide)

```c
__global__ void kernelFunc( float* A, float* B, int N ){
    extern __shared__ unsigned char sharedPtr[]; //pointer to shared memory
    int* intArray = sharedPtr; //array of ints
    float* floatArray = &sharedPtr[N * sizeof(int)]; //array of floats
    ...
}
```

• Assume the following configuration parameters are used

```c
kernelFunc<<< gridDim, blockDim, sharedBytes >>>(...)
```

• How many elements are in `intArray` (in terms of `sharedBytes`)?

  
  \[ N \text{ elements of size } \text{sizeof(int)} \text{ are available in } \text{intArray} \]

• How many elements are available in `floatArray`?

  
  \[ ( \text{sharedBytes} - N \times \text{sizeof(int)} ) / \text{sizeof(float)} \]

• What happens if we access `intArray[N]`?

  
  we overwrite the first element of `floatArray`
Convolution

- Calculate $g(x) = f(x) \ast k$ where $k$ is a box filter:
  - $k(x) = \frac{1}{K}$
  - $K = 4$ is the window size
  - don’t include edges (only evaluate elements where the data is known)

$$\frac{1}{K} \sum_{u=0}^{K-1} f[i + u]$$
Convolution with Shared Memory

- Assume 4 threads per block
- How much data is needed to evaluate all threads in a block?

\[
\begin{align*}
\text{blockIdx.x} &= 0 \\
\text{blockIdx.x} &= 1 \\
\text{blockIdx.x} &= 2
\end{align*}
\]
Copying Data to Shared Memory

```c
int tid = threadIdx.x;
int bd = blockDim.x;
int bid = blockIdx.x;
size_t sN = bd + K - 1;
for(int i = tid; i < sN; i += bd){
    s_ptr[i] = g_ptr[bid * sN + i];
}
__syncthreads();
```

//store the thread ID
//store the # of threads
//store the block ID
//calculate the # of elements to copy
//while there are still elements to copy
//copy from global to shared memory

//synchronize all threads in the block

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[i]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

---

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>block 0 (shared)</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>block 2 (shared)</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Convolution with Shared Memory – Kernel

```c
__global__ void kernelConv(float* g, float* f, size_t K, size_t N) {
    extern __shared__ unsigned char sharedPtr[]; // pointer to shared memory
    size_t bi = blockIdx.x; // index starting the block
    size_t ti = threadIdx.x; // index for the current thread
    size_t bd = blockDim.x; // store the block dimension
    int nS = blockDim.x + K - 1; // # of values needed for the block
    for(int si = tid; si < nS; si += bd) // for each value to copy
        sharedPtr[si] = f[bi * nS + i]; // copy from global to shared
    __syncthreads(); // wait until all values are copied
    // perform the convolution
    float gsum = 0; // create a register for output
    for(int ki = 0; ki < K; k++) // for each value in the kernel
        gsum += sharedPtr[ti + ki]; // add the value to the running sum
    size_t gi = bi + ti; // output index
    g[gi] = gsum / K; // copy the result to global memory
}
```
__device__ void tMemcpy1D(char* dst, char* src, size_t N, int tid, int bdim) {
    for(int i = tid; i < N; i += bdim) dst[i] = src[i];
}
__global__ void kernelConv(float* g, float* f, size_t K, size_t N)
{
    extern __shared__ unsigned char S[];
    // pointer to shared memory

    size_t bi = blockIdx.x;
    // index starting the block

    size_t ti = threadIdx.x;
    // index for the current thread

    size_t bd = blockDim.x;
    // store the block dimension

    int nS = bd + K - 1;
    // # of values needed for the block

    tMemcpy1D(S, f + bi*nS, nS, ti, bd);
    // threaded memory copy function

    __syncthreads();
    // wait until all values are copied

    __syncthreads();
    // perform the convolution

    float gsum = 0;
    // create a register for output

    for(int ki = 0; ki < K; k++)
    {
        gsum += S[ti + ki];
        // add the value to the running sum

        size_t gi = bi + ti;
        // output index

        g[gi] = gsum / K;
        // copy the result to global memory
    }
}