Objective

- To learn about tiled convolution algorithms
  - Some intricate aspects of tiling algorithms
  - Output tiles versus input tiles
Reading

- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 8.4

Tiled 1D Convolution Basic Idea

Four threads – one per output element

Thread block

Output Tile

With a 1x5 mask
Using Shared Memory

- Each thread block loads the elements it needs into its shared memory

Tile 1

- Thread block 1 will load the above elements – note halo elements loaded by two thread blocks

Loading the left halo

```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
    (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```
Loading the internal elements

\[
N = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{bmatrix}
\]

\[
N_{ds} = \begin{bmatrix}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{bmatrix}
\]

\[
N_{ds}[n + \text{threadIdx.x}] = N[\text{blockIdx.x}\times\text{blockDim.x} + \text{threadIdx.x}];
\]

Loading the right halo

\[
N = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{bmatrix}
\]

\[
N_{ds} = \begin{bmatrix}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{bmatrix}
\]

\[
\text{int halo_index_right} = (\text{blockIdx.x} + 1)\times\text{blockDim.x} + \text{threadIdx.x};
\]

\[
\text{if} \ (\text{threadIdx.x} < n) \{
    N_{ds}[n + \text{blockDim.x} + \text{threadIdx.x}] = \{\text{halo_index_right} >= \text{Width}) \ ? 0 : N[\text{halo_index_right}]\};
\]
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
    int n = Mask_Width/2;
    int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
    if (threadIdx.x >= blockDim.x - n) {
        N_ds[threadIdx.x - (blockDim.x - n)] =
            (halo_index_left < 0) ? 0 : N[halo_index_left];
    }
    N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
    int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
    if (threadIdx.x < n) {
        N_ds[n + blockDim.x + threadIdx.x] =
            (halo_index_right >= Width) ? 0 : N[halo_index_right];
    }
    __syncthreads();
    float Pvalue = 0;
    for(int j = 0; j < Mask_Width; j++) {
        Pvalue += N_ds[threadIdx.x + j]*M[j];
    }
    P[i] = Pvalue;
}

Shared Memory Data Reuse

- Element 2 is used by thread 4 (1X)
- Element 3 is used by threads 4, 5 (2X)
- Element 4 is used by threads 4, 5, 6 (3X)
- Element 5 is used by threads 4, 5, 6, 7 (4X)
- Element 6 is used by threads 4, 5, 6, 7 (4X)
- Element 7 is used by threads 5, 6, 7 (3X)
- Element 8 is used by threads 6, 7 (2X)
- Element 9 is used by thread 7 (1X)
Ghost Cells

Reduction in Global Memory Accesses

- \#accesses-global/\#accesses-shared
- Internal tiles
  - (blockDim.x * (2n+1))/(blockDim.x + (2n))
  - Subtract edge effects for boundary tiles
  - For TB size = 128 threads and 1*5 mask ratio = 10.2
2D Convolution

- Use a thread block to calculate a tile of P
  - Thread Block size determined by the TILE_SIZE

High-Level Tiling Strategy

- Load a tile of N into shared memory (SM)
  - All threads participate in loading
  - A subset of threads then use each N element in SM

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016
Output Tiling and Thread Index (P)

- Use a thread block to calculate a tile of P
  - Each output tile is of TILE_SIZE for both x and y

\[
\text{col}_o = \text{blockIdx}.x \times \text{TILE}_\text{SIZE} + \text{tx};
\]

\[
\text{row}_o = \text{blockIdx}.y \times \text{TILE}_\text{SIZE} + \text{ty};
\]

Tiling N

- Each N element is used in calculating up to KERNEL_SIZE * KERNEL_SIZE P elements (all elements in the tile)
Input tiles need to be larger than output tiles.

We will use a strategy where the input tile will be loaded into the shared memory.

Dealing with Mismatch

• Use a thread block that matches input tile
  – Each thread loads one element of the input tile
  – Some threads do not participate in calculating output
    • There will be if statements and control divergence
Shifting from output coordinates to input coordinates

```c
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y * TILE_SIZE + ty;
int col_o = blockIdx.x * TILE_SIZE + tx;
int row_i = row_o - 2;
int col_i = col_o - 2;
```
Threads that loads halos outside N should return 0.0

Setting Block Size

```c
#define BLOCK_SIZE (TILE_SIZE + 4)

dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);

In general, block size should be
tile size + (kernel size -1)

Dim3 dimGrid(N.width/TILE_SIZE, N.height/TILE_SIZE, 1)
```
In General

• BLOCK_SIZE is limited by the maximal number of threads in a thread block

• Input tile sizes could be N* TILE_SIZE + (KERNEL_SIZE - 1)
  – By having each thread to calculate N input points (thread coarsening)
  – N is limited by the shared memory size

• KERNEL_SIZE is decided by application needs

QUESTIONS?