Objective

- To learn about tiled convolution algorithms
  - Some intricate aspects of tiling algorithms
  - Output tiles versus input tiles
  - Three different styles of input tile loading
  - To prepare for MP-4

Tiled 1D Convolution Basic Idea

Three Tiling Strategies
Strategy 1: Loading the left halo

\[
\text{n} = \frac{\text{Mask Width}}{2}; \\
\text{ halo_index_left} = (\text{blockIdx.x} - 1) \times \text{blockDim.x} + \text{threadIdx.x}; \\
\text{ if (threadIdx.x} >= \text{blockDim.x} - \text{radius}) \{ \\
\quad \text{N_ds[threadIdx.x} - (\text{blockDim.x} - \text{radius})] = \\
\quad \quad (\text{halo_index_left} < 0) \quad ? \quad 0 \quad : \quad \text{N[halo_index_left]}; \\
\} \\
\]

If (blockIdx.x \times blockDim.x + threadIdx.x < Width) {
\quad N_ds[\text{radius} + threadIdx.x] = N[\text{blockIdx.x} \times \text{blockDim.x} + threadIdx.x]; 
\}
\else {
\quad N_ds[\text{radius} + threadIdx.x] = 0.0f;
\}
\]

__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width, int Width) {
\quad \text{i} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}; 
\quad \text{__shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];} 
\quad \text{int radius} = \frac{\text{Mask Width}}{2}; 
\quad \text{int halo_index_left} = (\text{blockIdx.x} - 1) \times \text{blockDim.x} + \text{threadIdx.x}; 
\quad \text{if (threadIdx.x} >= \text{blockDim.x} - \text{radius}) \{ 
\quad \quad \text{N_ds[threadIdx.x} - (\text{blockDim.x} - \text{radius})] = 
\quad \quad \quad (\text{halo_index_left} < 0) \quad ? \quad 0 \quad : \quad \text{N[halo_index_left]}; 
\quad \}
\quad \text{N_ds[\text{radius} + threadIdx.x]} = \text{N[\text{blockIdx.x} \times \text{blockDim.x} + threadIdx.x]}; 
\quad \}
\quad \text{\_syncthreads();}
\quad \text{float Pvalue} = 0; 
\quad \text{for(int } j = 0; j < \text{Mask_Width}; j++) \{ 
\quad \quad \text{Pvalue} += \text{N_ds[threadIdx.x + j]} \times \text{M[j]}; 
\quad \}
\quad \text{P[i]} = \text{Pvalue}; 
}

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498/University of Illinois, 2007-2018
Shared Memory Data Reuse

<table>
<thead>
<tr>
<th>N_ds</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>
</tr>
</thead>
</table>

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

```
__global__ void convolution_1D_tiled_cache_kernel(float *N, float *P,
int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float N_ds[TILE_WIDTH];
    N_ds[threadIdx.x] = N[i];
    __syncthreads();
    int radius = Mask_Width/2;
    int This_tile_start_point = blockIdx.x * blockDim.x + i;
    int N_start_point = i - radius;
    float Pvalue = 0;
    for (int j = 0; j < Mask_Width; j++) {
        int N_index = N_start_point + j;
        if (N_index >= 0 && N_index < Width) {
            if ((N_index >= This_tile_start_point) && (N_index < This_tile_start_point)) {
                Pvalue += N_ds[threadIdx.x-radius+j]*M[j];
            } else {
                Pvalue += N[N_index]*M[j];
            }
        }
    }
    P[i] = Pvalue;
}
```

Strategy 2
Output Tiling and Thread Index (P)

```
col_o = blockIdx.x * TILE_WIDTH + threadIdx.x;
row_o = blockIdx.y * TILE_SIZE + threadIdx.y;
```

Strategy 2 Tile Loading in 2D

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

```
Strategy 3

int i = blockIdx.x*blockDim.x + threadIdx.x;
__shared__ float N_ds[TILE_WIDTH];
N_ds[threadIdx.x] = N[i];
__syncthreads();
int radius = Mask_Width/2;
int This_tile_start_point = blockIdx.x * blockDim.x + i;
int N_start_point = i - radius;
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j++) {
    int N_index = N_start_point + j;
    if (N_index >= 0 && N_index < Width) {
        if ((N_index >= This_tile_start_point) && (N_index < This_tile_start_point)) {
            Pvalue += N_ds[threadIdx.x-radius+j]*M[j];
        } else {
            Pvalue += N[N_index] * M[j];
        }
    }
}
P[i] = Pvalue;
```
Input tiles need to be larger than output tiles.

- We will show strategy 2 where the input tile will be loaded into the shared memory.

Setting Block Width

```cpp
dim3 dimBlock(TILE_WIDTH+4, TILE_WIDTH+4, 1);
```

In general, block width should be

```
TILE_WIDTH + (Mask_Width - 1)
```

```cpp
Dim3 dimGrid(ceil(P.width/(1.0*TILE_WIDTH)), ceil(P.height/(1.0*TILE_WIDTH)), 1)
```

There need to be enough thread blocks to generate all P elements.

Dealing with Mismatch

- Use a thread block that matches input tile size
  - 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
Shifting from output coordinates to input coordinate

```c
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y * TILE_WIDTH + ty;
int col_o = blockIdx.x * TILE_WIDTH + tx;

int row_i = row_o - 2;
int col_i = col_o - 2;
```

Threads that loads halos outside N should return 0.0

Some threads do not participate in calculating output.

```c
float output = 0.0f;

if((row_i >= 0) && (row_i < N.height) &&
   (col_i >= 0) && (col_i < N.width)) {
   Ns[ty][tx] = N.elements[row_i*N.width + col_i];
} else{
   Ns[ty][tx] = 0.0f;
}
```
Some threads do not write output

```plaintext
if(row_o < P.height && col_o < P.width)
P.elements[row_o * P.width + col_o] = output;
}
```

Alternatively

- You can extend the 1D strategy 3 tiled convolution into a 2D strategy 3 tiled convolution.
  - Each input tile matches its corresponding output tile
  - All halo elements will be loaded from global memory
  - If condition and divergence during inner product computation

ANY MORE QUESTIONS?
READ CHAPTER 7