CS/EE 217: GPU Architecture and Parallel Programming

Tiled Convolution
Objective

• To learn about tiled convolution algorithms
  – Some intricate aspects of tiling algorithms
  – Output tiles versus input tiles
# Tiled 1D Convolution Basic Idea

<table>
<thead>
<tr>
<th>P</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>
</table>

<table>
<thead>
<tr>
<th>N</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>
</table>

- **Tile 0**
  - ghost
  - 0 1 2 3 4 5

- **Tile 1**
  - 2 3 4 5 6 7 8 9

- **Tile 2**
  - 6 7 8 9 10 11 12 13

- **Tile 3**
  - 10 11 12 13 14 15 ghost

**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

\[ N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x] \]
Loading the right halo

```c
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];
}
```
__global__ void convolution_1D_basic_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)

Mask_Width is 5
Ghost Cells

N


__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float N_ds[TILE_SIZE];
    N_ds[threadIdx.x] = N[i];
    __syncthreads();

    int This_tile_start_point = blockIdx.x * blockDim.x;
    int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
    int N_start_point = i - (Mask_Width/2);
    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 < Next_tile_start_point)) {
                Value += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
            } else {
                Pvalue += N[N_index] * M[j];
            }
        }
    }
    P[i] = Pvalue;
}
2D convolution with Tiling P

• Use a thread block to calculate a tile of P
  – Thread Block size determined by the TILE_SIZE
Tiling N

• Each element in the tile is used in calculating up to MASK_SIZE * MASK_SIZE elements (all elements in the tile)
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
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

$$col_o = blockIdx.x \times TILE\_SIZE + tx;$$
$$row_o = blockIdx.y \times TILE\_SIZE + ty;$$
Input tiles need to be larger than output tiles.
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
Setting up blocks

#define O_TILE_WIDTH  12
#define BLOCK_WIDTH  (O_TILE_WIDTH + 4)

dim3 dimBlock (BLOCK_WIDTH, BLOCK_WIDTH);
dim3 dimGrid ((imageWidth - 1)/O_TILE_WIDTH + 1, (imageHeight-1)/O_TILE_WIDTH+1, 1);

• In general, block width = Tile width + mask width – 1;
Using constant memory for mask

• Since mask is used by all threads and not modified:
  – All threads in a warp access the same locations at every time
  – Take advantage of the cachable constant memory!
  – Magnify memory bandwidth without consuming shared memory
• Syntax:
  __global__ void convolution_2D_kernel (float *P, *float N, height, width, channels, 
  const float __restrict__ *M) {

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_SIZE + ty;
int col_o = blockIdx.x * TILE_SIZE + tx;

int row_i = row_o - 2;  //MASK_SIZE/2
int col_i = col_o - 2;  //MASK_SIZE/2
```
Threads that loads halos outside N should return 0.0
Taking Care of Boundaries

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 participate in calculating output.

```c
if(ty < TILE_SIZE && tx < TILE_SIZE){
    for(i = 0; i < MASK_SIZE; i++) {
        for(j = 0; j < MASK_SIZE; j++) {
            output += Mc[i][j] * Ns[i+ty][j+tx];
        }
    }
}
```
Some threads do not write output

if(row_o < P.height && col_o < P.width)
    P.elements[row_o * P.width + col_o] = output;
In General

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

• Input tile sizes could be $k \times \text{TILE\_SIZE} + (\text{MASK\_SIZE}-1)$
  – For 1D convolution – what is it for 2D convolution?
  – By having each thread to calculate $k$ input points (thread coarsening)
  – $k$ is limited by the shared memory size

• MASK_SIZE is decided by application needs
ANY MORE QUESTIONS?
READ CHAPTER 8