CSE 599 I
Accelerated Computing - Programming GPUS

Parallel Patterns: Stencil (Convolution)
Module 8.1 – Parallel Computation Patterns (Stencil)
Convolution
Objective

– To learn convolution, an important method
  – Widely used in audio, image and video processing
  – Foundational to stencil computation used in many science and engineering applications
  – Basic 1D and 2D convolution kernels
Convolution as a Filter

– Often performed as a filter that transforms signal or pixel values into more desirable values.
  – Some filters smooth out the signal values so that one can see the big-picture trend
  – Others like Gaussian filters can be used to sharpen boundaries and edges of objects in images.
Convolution – a computational definition

- An array operation where each output data element is a weighted sum of a collection of neighboring input elements
- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
  - We will refer to these mask arrays as convolution masks to avoid confusion.
  - The value pattern of the mask array elements defines the type of filtering done
  - Our image blur example in Module 3 is a special case where all mask elements are of the same value and hard coded into the source code.
1D Convolution Example

- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- The figure shows calculation of P[2]

Calculation of $P[3]$
Convolution Boundary Condition

Calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements

- Different policies (0, replicates of boundary values, etc.)
A 1D Convolution Kernel with Boundary Condition Handling

This kernel forces all elements outside the valid input range to 0

```c
__global__ void convolution_1D_basic_kernel(float *N, float *M,
float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i – (Mask_Width/2);

    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 && N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```
A 1D Convolution Kernel with Boundary Condition Handling

- This kernel forces all elements outside the valid input range to 0

```c
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    float Pvalue = 0;
    int N_start_point = i – (Mask_Width/2);

    if (i < Width) {
        for (int j = 0; j < Mask_Width; j++) {
            if (N_start_point + j >= 0 && N_start_point + j < Width) {
                Pvalue += N[N_start_point + j]*M[j];
            }
        }
        P[i] = Pvalue;
    }
}
```
2D Convolution

N

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 3 & 4 & 5 & 6 & 7 & 8 \\
3 & 4 & 5 & 6 & 7 & 8 & 9 \\
4 & 5 & 6 & 7 & 8 & 5 & 6 \\
5 & 6 & 7 & 8 & 5 & 6 & 7 \\
6 & 7 & 8 & 9 & 0 & 1 & 2 \\
7 & 8 & 9 & 0 & 1 & 2 & 3 \\
\end{array}
\]

P

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 4 & 5 & 6 & 7 & 8 \\
4 & 5 & 6 & 7 & 8 & 5 \\
5 & 6 & 7 & 8 & 5 & 6 \\
6 & 7 & 8 & 9 & 0 & 1 \\
7 & 8 & 9 & 0 & 1 & 2 \\
\end{array}
\]

M

\[
\begin{array}{cccc}
1 & 2 & 3 & 2 \\
2 & 3 & 4 & 3 \\
3 & 4 & 5 & 4 \\
2 & 3 & 4 & 3 \\
1 & 2 & 3 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 4 & 9 & 8 \\
4 & 9 & 16 & 5 \\
4 & 16 & 25 & 21 \\
8 & 15 & 24 & 16 \\
5 & 12 & 21 & 16 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 2 \\
2 & 3 & 4 & 3 \\
3 & 4 & 5 & 4 \\
2 & 3 & 4 & 3 \\
1 & 2 & 3 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 4 & 9 & 8 \\
4 & 9 & 16 & 15 \\
4 & 16 & 24 & 21 \\
8 & 15 & 24 & 16 \\
5 & 12 & 21 & 16 \\
\end{array}
\]
2D Convolution – Ghost Cells

N

0 0 0 0 0
0 3 4 5 6
0 2 3 4 5
0 3 5 6 7
0 1 1 3 1

P

GHOST CELLS
(apron cells, halo cells)

M

1 2 3 2 1
2 3 4 3 2
3 4 5 4 3
2 3 4 3 2
1 2 3 2 1

0 0 0 0 0
0 9 16 15 12
0 8 15 16 15
0 9 20 18 14
0 2 3 6 1
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskwidth, int w, int h) {

    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;

        int N_start_col = Col - (maskwidth/2);
        int N_start_row = Row - (maskwidth/2);

        // Get the offset of the surrounding box
        for (int j = 0; j < maskwidth; ++j) {
            for (int k = 0; k < maskwidth; ++k) {
                int curRow = N_start_row + j;
                int curCol = N_start_col + k;

                // Verify we have a valid image pixel
                if (curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol] * mask[j*maskwidth + k];
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal);
    }
}
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
                                int maskwidth, int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;

        N_start_col = Col - (maskwidth/2);
        N_start_row = Row - (maskwidth/2);

        // Get the of the surrounding box
        for(int j = 0; j < maskwidth; ++j) {
            for(int k = 0; k < maskwidth; ++k) {

                int curRow = N_start_row + j;
                int curCol = N_start_col + k;

                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal);
    }
}
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
  int maskwidth, int w, int h) {
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col = Col – (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskwidth; ++j) {
      for(int k = 0; k < maskwidth; ++k) {

        int curRow = N_Start_row + j;
        int curCol = N_start_col + k;

        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
        }
      }
    }

    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
Module 8.2 – Parallel Computation Patterns (Stencil)
Tiled Convolution
Objective

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

- Calculation of adjacent output elements involve shared input elements
- We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses
Input Data Needs

- Assume that we want to have each block to calculate $T$ output elements
  - $T + \text{Mask\_Width} - 1$ input elements are needed to calculate $T$ output elements
  - $T + \text{Mask\_Width} - 1$ is usually not a multiple of $T$, except for small $T$ values
  - $T$ is usually significantly larger than $\text{Mask\_Width}$
Definition – output tile

Each thread block calculates an output tile

Each output tile width is O TILE WIDTH

For each thread,

O TILE WIDTH is 4 in this example
Definition - Input Tiles

Each input tile has all values needed to calculate the corresponding output tile.
Two Design Options

– **Design 1**: The size of each thread block matches the size of an output tile
  – All threads participate in calculating output elements
  – blockDim.x would be 4 in our example
  – Some threads need to load more than one input element into the shared memory

– **Design 2**: The size of each thread block matches the size of an input tile
  – Some threads will not participate in calculating output elements
  – blockDim.x would be 8 in our example
  – Each thread loads one input element into the shared memory

– We will present Design 2 and leave Design 1 as an exercise.
For each thread,
Index_i = index_o – n

where n is Mask_Width / 2
n is 2 in this example
All Threads Participate in Loading Input Tiles

```c
float output = 0.0f;

if((index_i >= 0) && (index_i < Width)) {
    Ns[tx] = N[index_i];
}
else{
    Ns[tx] = 0.0f;
}
```
Some threads do not participate in calculating output

```c
if (threadIdx.x < O_TILE_WIDTH) {
    output = 0.0f;
    for(j = 0; j < Mask_Width; j++) {
        output += M[j] * Ns[j+threadIdx.x];
    }
    P[index_o] = output;
}
```

- `index_o = blockIdx.x*O_TILE_WIDTH + threadIdx.x`

- Only Threads 0 through O TILE_WIDTH-1 participate in calculation of output.
Setting Block Size

#define O_TILE_WIDTH 1020
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)

dim3 dimBlock(BLOCK_WIDTH, 1, 1);

dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)

The Mask_Width is 5 in this example
In general, block width should be
output tile width + (mask width-1)
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

N

O

O


O

O

P[0]

P[1]

P[2]

P[3]

P[4]

P[5]

P[6]
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
Module 8.3 – Parallel Computation Patterns (Stencil)
Tile Boundary Conditions
Objective

- To learn to write a 2D convolution kernel
  - 2D Image data types and API functions
  - Using constant caching
  - Input tiles vs. output tiles in 2D
  - Thread to data index mapping
  - Handling boundary conditions
### 2D Image Matrix with Automated Padding

- It is sometimes desirable to pad each row of a 2D matrix to multiples of DRAM bursts
  - So each row starts at the DRAM burst boundary
  - Effectively adding columns
  - This is usually done automatically by matrix allocation function
  - Pitch can be different for different hardware

- Example: a 3X3 matrix padded into a 3X4 matrix

<table>
<thead>
<tr>
<th>Height is 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width is 3</td>
</tr>
<tr>
<td>Channels is 1 (See MP Description)</td>
</tr>
<tr>
<td>Pitch is 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Height</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₀,</td>
<td>M₀,</td>
</tr>
<tr>
<td>M₁,</td>
<td>M₁,</td>
</tr>
<tr>
<td>M₂,</td>
<td>M₂,</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Padded elements
Row-Major Layout with Pitch

Row*Pitch + Col = 2*4 + 1 = 9

M

M

Padded elements
Image Matrix Type in this Course

// Image Matrix Structure declaration
//
typedef struct {
    int width;
    int height;
    int pitch;
    int channels;
    float* data;
} * wbImage_t;

This type will only be used in the host code of the MP.
wbImage_t API Function for Your Lab

```c
wbImage_t  wbImage_new(int height, int width, int channels)
wbImage_t  wbImport(char * File);

void wbImage_delete(wbImage_t img)

int wbImage_getWidth(wbImage_t img)
int wbImage_getHeight(wbImage_t img)
int wbImage_getChannels(wbImage_t img)
int wbImage_getPitch(wbImage_t img)

float *wbImage_getData(wbImage_t img)
```

For simplicity, the pitch of all matrices are set to be width * channels (no padding) for our labs.

The use of all API functions has been done in the provided host code.
Setting Block Size

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

dim3 dimBlock(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 dimGrid((wbImage_getWidth(N)-1)/O_TILE_WIDTH+1,
              (wbImage_getHeight(N)-1)/O_TILE_WIDTH+1, 1)

In general, BLOCK_WIDTH should be
            O_TILE_WIDTH + (MASK_WIDTH-1)
```
Using constant memory and caching for Mask

- Mask is used by all threads but not modified in the convolution kernel
  - All threads in a warp access the same locations at each point in time
- CUDA devices provide constant memory whose contents are aggressively cached
  - Cached values are broadcast to all threads in a warp
  - Effectively magnifies memory bandwidth without consuming shared memory
- Use of const __restrict__ qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

```c
__global__ void convolution_2D_kernel(float *P,
    float *N, height, width, channels,
    const float __restrict__ *M) {
```
Shifting from output coordinates to input coordinate

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

int row_i = row_o - 2;
int col_i = col_o - 2;
```
Taking Care of Boundaries (1 channel example)

```c
if((row_i >= 0) && (row_i < height) &&
   (col_i >= 0) && (col_i < width)) {
    Ns[ty][tx] = data[row_i * width + col_i];
} else{
    Ns[ty][tx] = 0.0f;
}
```

Use of width here is OK since pitch is set to width for this MP.
Some threads do not participate in calculating output. (1 channel example)

```c
float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
    for(i = 0; i < MASK_WIDTH; i++) {
        for(j = 0; j < MASK_WIDTH; j++) {
            output += Ms[i][j] * Ns[i+ty][j+tx];
        }
    }
}
```
Some threads do not write output (1 channel example)

```c
if (row_o < height && col_o < width)
    data[row_o*width + col_o] = output;
```
Access Pattern for M

- M is the convolution mask
- Elements of M are the convolution coefficients
- Calculation of all output P elements need M
- M is not changed during kernel

- Bonus: M elements are accessed in the same order when calculating all P elements

- M is a good candidate for constant memory
Programmer view of CUDA Memories (review)

- Each thread can:
  - Read/write per-thread **Registers** (~1 cycle)
  - Read/write per-block **shared memory** (~5 cycles)
  - Read/write per-grid **global memory** (~500 cycles)
  - Read-only per-grid **constant memory** (~5 cycles with caching)
How to Use Constant Memory

- Host code allocates, initializes variables (src) the same way as any other variables that need to be copied to the device

- Declare a constant memory variable (dest) to be used by the device

- Use `cudaMemcpyToSymbol(dest, src, size)` to copy the variable into the device constant memory

- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.
Host Code

// global variable, outside any function
__constant__ float Mc[KERNEL_SIZE][KERNEL_SIZE];
...

// allocate N, P, initialize N elements, copy N to Nd
Matrix M;
M = AllocateMatrix(KERNEL_SIZE,KERNEL_SIZE,1);
// initialize M elements
...

cudaMemcpyToSymbol(Mc, M.elements,
KERNEL_SIZE*KERNEL_SIZE*sizeof(float));
ConvolutionKernel<<<dimGrid,dimBlock>>>(...);
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
Module 8.4 – Parallel Computation Patterns (Stencil)
Analyzing Data Reuse in Tiled Convolution
Objective

- To learn to analyze the cost and benefit of tiled parallel convolution algorithms
  - More complex reuse pattern than matrix multiplication
  - Less uniform access patterns
An 8-element Convolution Tile

For Mask_Width=5, we load 8+5-1=12 elements (12 memory loads)
Each output P element uses 5 N elements

P[8] uses N[6], N[7], N[8], N[9], N[10]
...
P[14] uses N[12], N[13], N[14], N[15], N[16]
A simple way to calculate tiling benefit

- $(8+5-1)=12$ elements loaded
- $8 \times 5$ global memory accesses replaced by shared memory accesses
- This gives a bandwidth reduction of $40/12=3.3$
In General, for 1D TILED CONVOLUTION

- \( O_{\text{TILE \_ WIDTH}} + \text{MASK\_WIDTH} - 1 \) elements loaded for each input tile
- \( O_{\text{TILE \_ WIDTH}} \times \text{MASK\_WIDTH} \) global memory accesses replaced by shared memory accesses
- This gives a reduction factor of

\[
\frac{O_{\text{TILE \_ WIDTH}} \times \text{MASK\_WIDTH}}{O_{\text{TILE \_ WIDTH}} + \text{MASK\_WIDTH} - 1}
\]

This ignores ghost elements in edge tiles.
Another Way to Look at Reuse

N[6] is used by P[8] (1X)
N[7] is used by P[8], P[9] (2X)
N[8] is used by P[8], P[9], P[10] (3X)
N[9] is used by P[8], P[9], P[10], P[11] (4X)
N[10] is used by P[8], P[9], P[10], P[11], P[12] (5X)
... (5X)
N[14] is used by P[12], P[13], P[14], P[15] (4X)
N[15] is used by P[13], P[14], P[15] (3X)
Another Way to Look at Reuse

The total number of global memory accesses (to the \((8+5-1)=12\) N elements) replaced by shared memory accesses is:

\[
1 + 2 + 3 + 4 + 5 \times (8-5+1) + 4 + 3 + 2 + 1
= 10 + 20 + 10
= 40
\]

So the reduction is:

\[
40/12 = 3.3
\]
In General, for 1D

- The total number of global memory accesses to the input tile can be calculated as

\[
1 + 2 + \ldots + \text{MASK_WIDTH}-1 + \\
\text{MASK_WIDTH} \times (\text{O_TILE_WIDTH} - \text{MASK_WIDTH} + 1) + \text{MASK_WIDTH}-1 + \\
\ldots + 2 + 1
\]

\[
= \text{MASK_WIDTH} \times (\text{MASK_WIDTH}-1) + \text{MASK_WIDTH} \times \\
(\text{O_TILE_WIDTH} - \text{MASK_WIDTH} + 1)
\]

\[
= \text{MASK_WIDTH} \times \text{O_TILE_WIDTH}
\]

For a total of \text{O_TILE_WIDTH} + \text{MASK_WIDTH} -1 input tile elements
Examples of Bandwidth Reduction for 1D

The reduction ratio is:

\[
\text{MASK_WIDTH} \times \frac{\text{O_TILE_WIDTH}}{\text{O_TILE_WIDTH} + \text{MASK_WIDTH} - 1}
\]

<table>
<thead>
<tr>
<th>O_TILE_WIDTH</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_WIDTH= 5</td>
<td>4.0</td>
<td>4.4</td>
<td>4.7</td>
<td>4.9</td>
<td>4.9</td>
</tr>
<tr>
<td>MASK_WIDTH = 9</td>
<td>6.0</td>
<td>7.2</td>
<td>8.0</td>
<td>8.5</td>
<td>8.7</td>
</tr>
</tbody>
</table>
For 2D Convolution Tiles

- \((O\_TILE\_WIDTH+MASK\_WIDTH-1)^2\) input elements need to be loaded into shared memory.
- The calculation of each output element needs to access \(MASK\_WIDTH^2\) input elements.
- \(O\_TILE\_WIDTH^2 \times MASK\_WIDTH^2\) global memory accesses are converted into shared memory accesses.
- The reduction ratio is

\[
O\_TILE\_WIDTH^2 \times MASK\_WIDTH^2 / (O\_TILE\_WIDTH+MASK\_WIDTH-1)^2
\]
Bandwidth Reduction for 2D

The reduction ratio is:

\[
\frac{O_{\text{TILE\_WIDTH}}^2 \times \text{MASK\_WIDTH}^2}{(O_{\text{TILE\_WIDTH}}+\text{MASK\_WIDTH}-1)^2}
\]

<table>
<thead>
<tr>
<th>O_{\text{TILE_WIDTH}}</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_WIDTH = 5</td>
<td>11.1</td>
<td>16</td>
<td>19.7</td>
<td>22.1</td>
</tr>
<tr>
<td>MASK_WIDTH = 9</td>
<td>20.3</td>
<td>36</td>
<td>51.8</td>
<td>64</td>
</tr>
</tbody>
</table>

Tile size has significant effect on the memory bandwidth reduction ratio.

This often argues for larger shared memory size.
A simpler alternative approach

```c
__global__ void convolution_1D_tiled_caching_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)) {
            Pvalue += N_ds[threadIdx.x + j - (Mask_Width/2)] * M[j];
        } else {
            Pvalue += N[N_index] * M[j];
        }
    }
}
P[i] = Pvalue;
```
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.