Convolution, Constant Memory and Constant Caching
Case Study: Convolution

• To learn convolution, an important parallel computation pattern
  – Widely used in signal, image and video processing
  – Foundational to stencil computation used in many science and engineering

• To learn about 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)
Memory Hierarchies

• If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything
• On today’s processors, main memory accesses take hundreds of cycles

• One solution: Caches
Cache - Cont’d

• Cache is unit of volatile memory storage

• A cache is an “array” of cache lines

• Cache line can usually hold data from several consecutive memory addresses

• When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests
Caches - Cont’d

Some definitions:

– **Spatial locality**: is when the data elements stored in consecutive memory locations are accessed consecutively.

– **Temporal locality**: is when the same data element is accessed multiple times in a short period of time.

• Both spatial locality and temporal locality improve the performance of caches.
Scratchpad vs. Cache

- Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.
- In terms of distance from the processor, scratchpad is similar to L1 cache.
- Unlike cache, scratchpad does not necessarily hold a copy of data that is also in main memory.
- It requires explicit data transfer instructions, whereas cache doesn’t
Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)
CPU and GPU have different caching philosophy

• CPU L1 caches are usually coherent
  – L1 is also replicated for each core
  – Even data that will be changed can be cached in L1
  – Updates to local cache copy invalidates (or less commonly updates) copies in other caches
  – Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

• GPU L1 caches are usually incoherent
  – Avoid caching data that will be modified
How to Use Constant Memory

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

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

- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.
More on Constant Caching

- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads

- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence

This is not a problem if a variable is NOT modified by a kernel.
Case Study: Convolution

• Each output element is a weighted sum of neighboring input elements

• The weights are defined as the convolution kernel
  – The same convolution mask is typically used for all elements of the array.
Gaussian Blur

Simple Integer Gaussian Kernel

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>4</th>
<th>7</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16</td>
<td>26</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>26</td>
<td>41</td>
<td>26</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>26</td>
<td>16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\[
\frac{1}{273}
\]
1D Convolution Example

• Commonly used for audio processing
  – Mask size is usually an odd number of elements for symmetry (5 in this example)

• Calculation of P[2]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
1D Convolution Boundary Condition

• Calculation of output elements near the boundaries (beginning and end) of the input 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 image 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;
}
```
2D Convolution

N

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

P

M

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>9</th>
<th>8</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>16</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>25</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>24</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>21</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>
### 2D Convolution Boundary Condition

#### Matrix N

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

#### Matrix M

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

#### Matrix P

```
|   | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 4 | 6 | 6 | 0 | 0 |
| 0 | 0 | 10 | 12 | 12 | 0 | 0 | 0 |
| 0 | 0 | 12 | 12 | 10 | 0 | 0 | 0 |
| 0 | 0 | 12 | 10 | 6 | 0 | 0 | 0 |
```
2D Convolution – Ghost Cells

M

N

P

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

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

ghost cells
(apron cells, halo cells)
Access Pattern for M

• M is referred to as mask (a.k.a. kernel, filter, etc.)
  – Elements of M are called mask (kernel, filter) 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
#define KERNEL_SIZE 5

// Matrix Structure declaration
typedef struct {
    unsigned int width;
    unsigned int height;
    float* elements;
} Matrix;
AllocateMatrix()

// Allocate a device matrix of dimensions height*width
//   If init == 0, initialize to all zeroes.
//   If init == 1, perform random initialization.
//   If init == 2, initialize matrix parameters, but do not allocate memory

Matrix AllocateMatrix(int height, int width, int init)
{
    Matrix M;
    M.width = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
}
AllocateMatrix() (Cont.)

// don't allocate memory on option 2
if(init == 2) return M;
M.elements = (float*) malloc(size*sizeof(float));
for(unsigned int i = 0; i < M.height * M.width; i++)
{
    M.elements[i] = (init == 0) ? (0.0f) :
        (rand() / (float)RAND_MAX);
    if(rand() % 2) M.elements[i] = - M.elements[i]
}
return M;
// 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>>>(Nd, Pd);