Objective

- 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 applications
  - Critical component of Neural Networks and Deep Learning
- Important techniques
  - Taking advantage of cached memories

Convolution Applications

- A popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, computer vision, and machine learning.
- Convolution is often performed as a filter that transforms signals and pixels 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 Computation

- 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 or convolution filters to avoid confusion.
  - The same convolution mask is typically used for all elements of the array.
1D Convolution Example

- Commonly used for audio processing
  - Mask_Width is usually an odd number of elements for symmetry (5 in this example)
  - Mask_Radius is the number of elements used in convolution on each side of the pixel being calculated (2 in this example).

- Calculation of P[2]:

<table>
<thead>
<tr>
<th>N0</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
<th>N6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

- Calculation of P[3]:

<table>
<thead>
<tr>
<th>N0</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
<th>N6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</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 valid data index 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;
}
```
2D Convolution

Elements of M are called mask (kernel, filter) coefficients (weights).
- Calculation of all output P elements needs M
- M is not changed during grid execution

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

M is a good candidate for Constant Memory

Access Pattern for M

- Elements of M are called mask (kernel, filter) coefficients (weights)
  - Calculation of all output P elements needs M
  - M is not changed during grid execution

- 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)

Memory Hierarchies

• Review: If we had to go to global memory to access data all the time, the execution speed of GPUs would be limited by the global memory bandwidth
  – We saw the use of shared memory (scratchpad) in tiled matrix multiplication.

• Another important solution: Caches

Caches Store Lines of Memory

Recall: memory bursts
• contain around 1024 bits (128B) from consecutive (linear) addresses.
• Let’s call a single burst a line.

What’s a cache?
• An array of cache lines (and tags).
• Memory read produces a line,
• cache stores a copy of the line, and
• tag records line’s memory address.

Memory Accesses Show Locality

An executing program
• loads and store data from memory.
• Consider sequence of addresses accessed. Sequence usually shows two types of locality:
  – spatial: accessing X implies accessing X+1 (and X+2, and so forth) soon
  – temporal: accessing X implies accessing X again soon
(Caches improve performance for both types.)
Caches Can’t Hold Everything

Caches are smaller than memory.

When cache is full,
• must make room for new line,
• usually by discarding least recently used line.

GPU Has Constant and L1 Caches

To support writes (modification of lines),
• changes must be copied back to memory, and
• cache must track modification status.
• L1 cache in GPU (for global memory accesses) supports writes.

Cache for constant / texture memory
• Special case: lines are read-only
• Enables higher-throughput access than L1 for common GPU kernel access patterns.

Cache vs. Scratchpad (GPU Shared Mem.)

• Caches vs. shared memory
  – Both on chip*, with similar performance
  – (As of Volta generation, both using the same physical resources, allocated dynamically!)

What’s the difference?
• Programmer controls shared memory contents (called a scratchpad)
• Microarchitecture automatically determines contents of cache.
  *Static RAM, not DRAM, by the way—see ECE120/CS233.

How to Use Constant Memory

Host code is similar to previous versions, but…

Allocate device memory for M (the mask)
• outside of all functions
• using __constant__ (tells GPU that caching is safe).

For copying to device memory, use
• cudaMemcpyToSymbol(dst, src, size)
• with destination defined as above.
Example of Host Code

(MASK_WIDTH is the size of the mask.)

// outside of any kernel/function
static __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];

// allocate N, P, initialize N elements, copy N to Nd

// in host code:
float* M; // host memory copy of mask
// initialize M
cudamemcpyToSymbol(Mc, M, 
    MASK_WIDTH * MASK_WIDTH * sizeof(M[0]));
ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
// (note that file-scope Mc is visible to kernel)