ECE408/CS483/CSE408 Spring 2018

Applied Parallel Programming

Lecture 7-8: Convolution, Constant Memory and Constant Caching

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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 cache memories

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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..

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

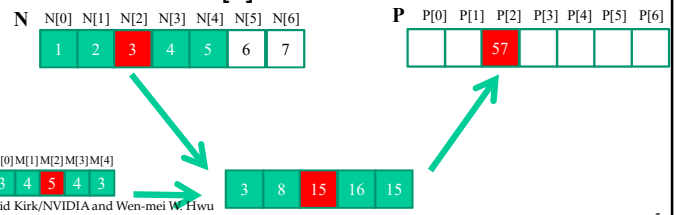
Convolution Computation

- 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.

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

1D Convolution Example

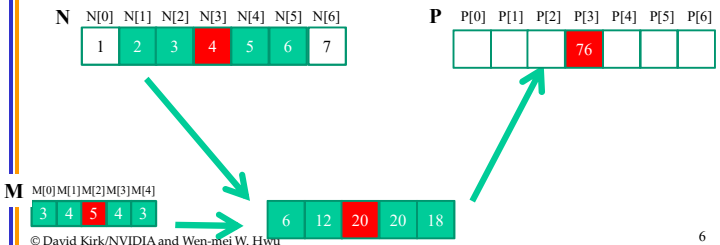
- Commonly used for audio processing
  - Mask_Width is usually an odd number of elements for symmetry (5 in this example)
  - Mask_Radius is the number of elements used in convolution on each side of the pixel being calculated (2 in this example).
- Calculation of P[2]:



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

1D Convolution Example - more on inside elements

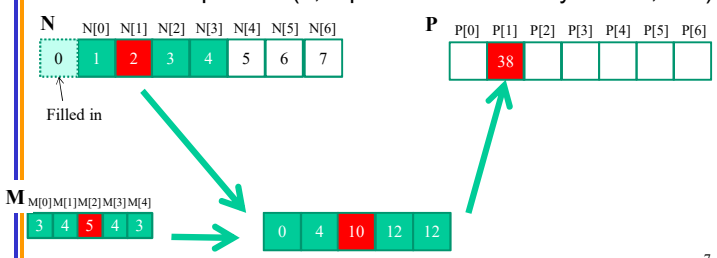
- Calculation of P[3]



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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.)



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

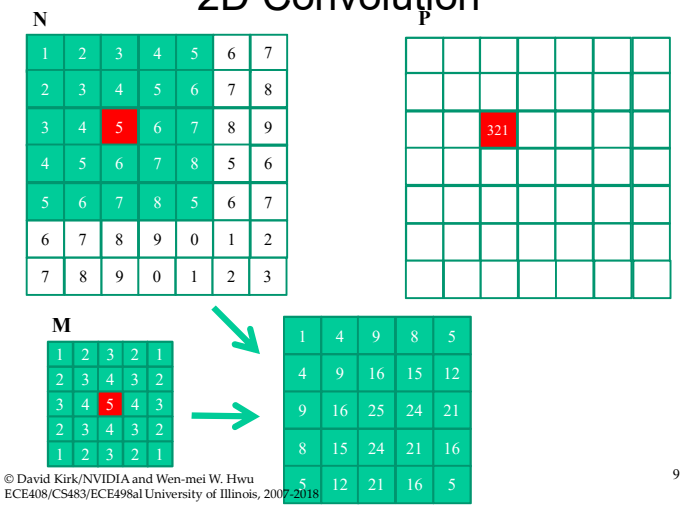
A 1D Convolution Kernel with Boundary Condition Handling

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

```
__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;

}
```

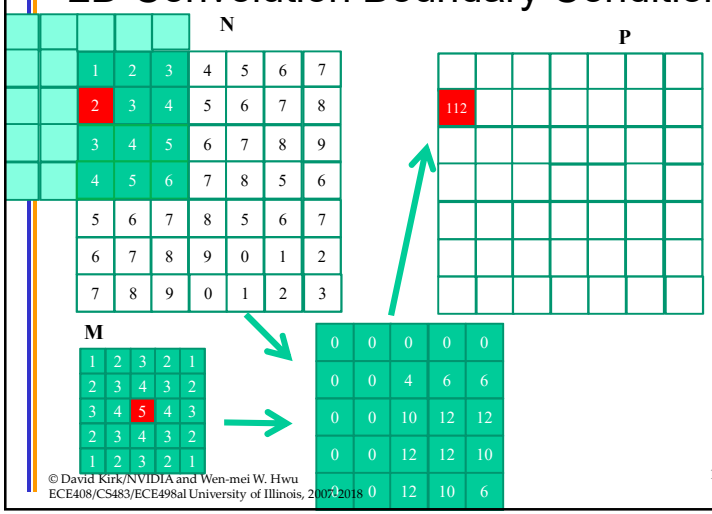
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

2D Convolution



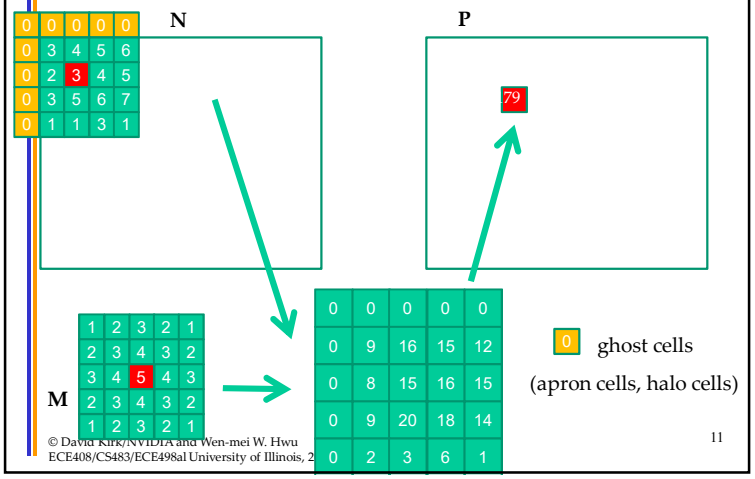
© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

2D Convolution Boundary Condition



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

2D Convolution – Ghost Cells



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2

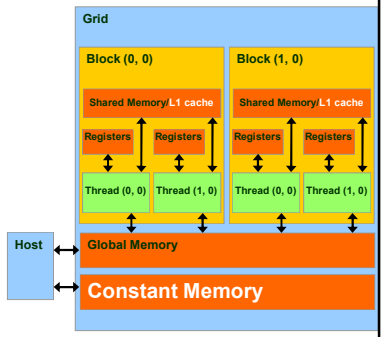
Access Pattern for M

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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)**



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

Cache - Cont'd

- A cache is an "array" of cache lines
  - A cache line can usually hold data from several consecutive memory addresses
- When data is requested from the global memory, an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests
  - The data in the cache is a "copy" of the original data in global memory
  - Additional hardware is used to remember the addresses of the data in the cache line

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

Caches - Cont'd

Some definitions:

- Spatial locality: when the data elements stored in consecutive memory locations are accessed consecutively
- Temporal locality: when the same data element is accessed multiple times in short period of time

- Both spatial locality and temporal locality improve the performance of caches

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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 SMs, scratchpad is similar to L1 cache.
- Unlike cache, scratchpad does not necessarily hold a copy of data that is also in main memory
  - Scratchpad requires explicit data transfer instructions into locations in the scratchpad , whereas cache doesn't

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

Constant Cache in GPUs

- Modification to cached data needs to be (eventually) reflected back to the original data in global memory
  - Requires logic to track the modified status, etc.
- Constant cache is a special cache for constant data that will not be modified during kernel execution by a grid
  - Data declared in the constant memory will not be modified during kernel execution.
  - Constant cache can be accessed with higher throughput than L1 cache for some common patterns

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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.

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

Some Header File Stuff for M

```
#define MASK_WIDTH 5

// Matrix Structure declaration
typedef struct {
  unsigned int width;
  unsigned int height;
  unsigned int pitch;
  float* elements;
} Matrix;
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

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 = M.pitch = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

AllocateMatrix() (Cont.)

```
// don't allocate memory on option 2
  if(init == 2) return M;
  int size = height * width;
  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;
```


Host Code

```
  // global variable, outside any kernel/function
  __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
  // allocate N, P, initialize N elements, copy N to Nd
  Matrix  M;
  M  = AllocateMatrix(MASK_WIDTH, MASK_WIDTH, 1);
  // initialize M elements
….
  cudaMemcpyToSymbol(Mc, M.elements,
        MASK_WIDTH*MASK_WIDTH*sizeof(float));
  ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018

ANY MORE QUESTIONS?
READ CHAPTER 7

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2018