Week 8

Convolution

1

Road map

- Intro to Convolution
- Sequential program
- OMP or MPI version
- CUDA Intro and Convolution in CUDA

2

Convolution

Convolution applies a filter or mask or kernel* on each element of the input array to obtain a new value, which is a weighted sum of a set of neighboring input elements

* The term “kernel” may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

3

Convolution and its Applications

- Convolution is a widely-used operation in signal processing (1D), image processing(2D), video processing(3D), and computer vision
  - Smoothing, sharpening, or blurring an image
  - Finding edges in an image
  - Removing noise, etc.
- Applications in machine learning and artificial intelligence
  - Convolutional Neural Networks (CNN or ConvNets)

* The term “kernel” may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

4

Mathematically formulation (Discrete Structures…)

$$y_i = \sum_{j=-r}^{r} f_{i+j} \times x_i$$

- $x_i \in [x_0, x_1, \ldots, x_{n-1}]$ //input array
- $y_i \in [y_0, y_1, \ldots, y_{n-1}]$ //output array
- $f_i \in [f_0,\ f_1, \ldots, f_{2r}]$ //filter array

5

Some observations

- The size of the filter is an odd number ($2r + 1$),
  - the weighted sum calculation is symmetric around the element that is being calculated.
- That is, the weighted sum involves ***r*** input elements on each side of the position that is being calculated,
- Which is the reason ***r*** is referred to as the radius of the filter.

6

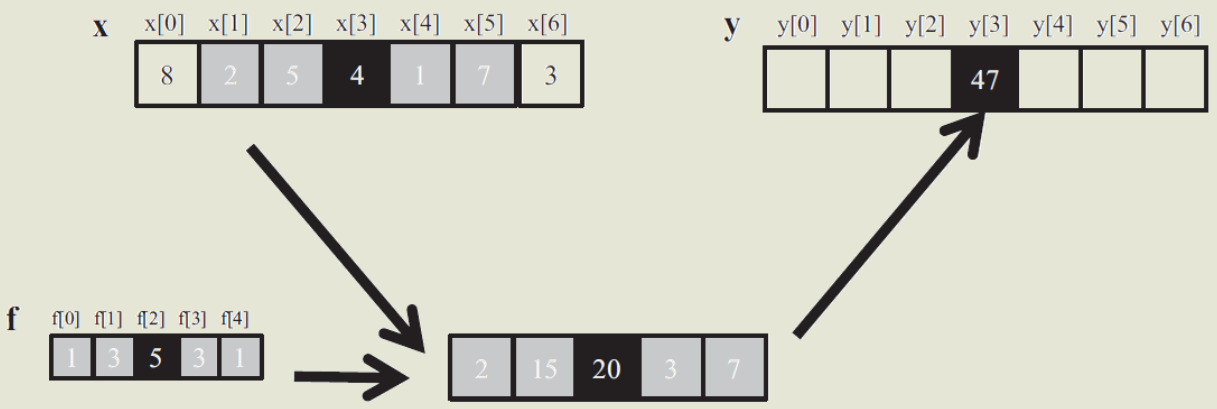


FIGURE 7.2

1D convolution, calculation of y[3].

7

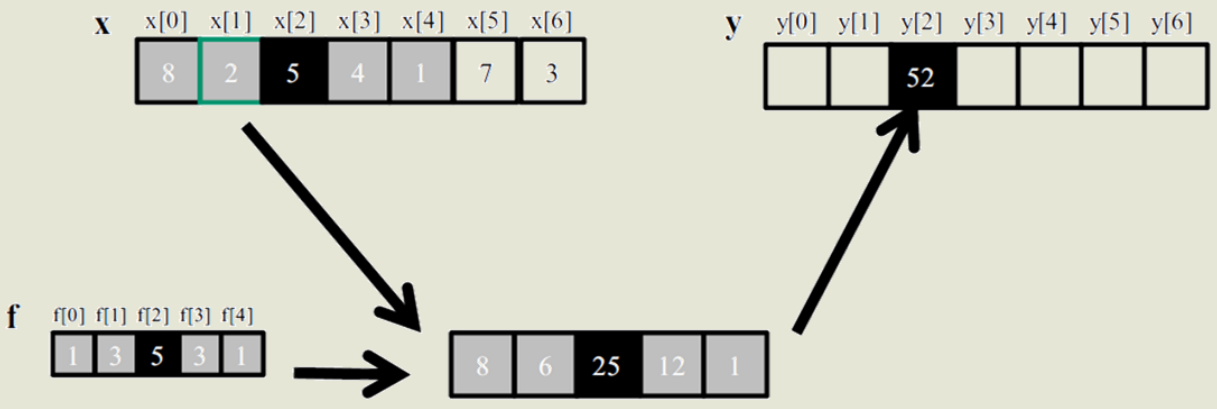


FIGURE 7.1

A 1D convolution example, inside elements.

8

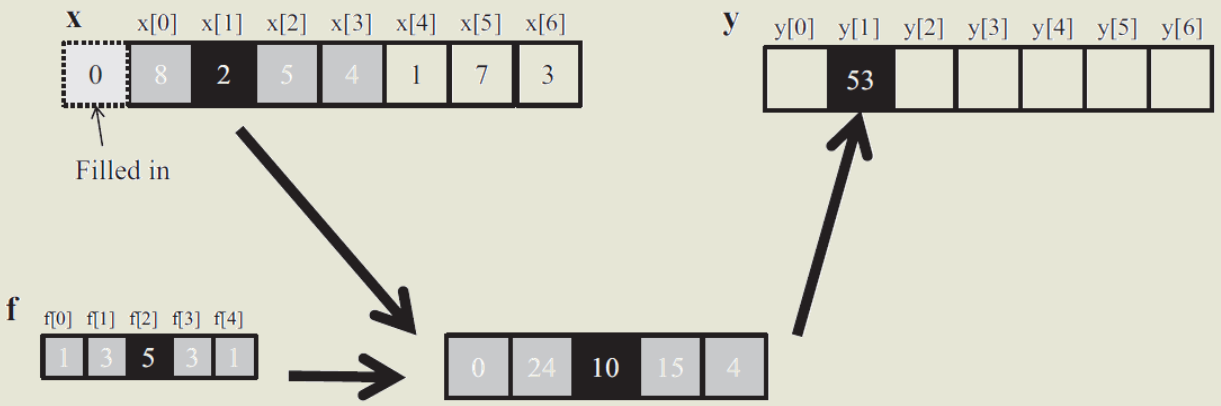


FIGURE 7.3

A 1D convolution boundary condition.

9

Ghost Elements

- 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.)

10

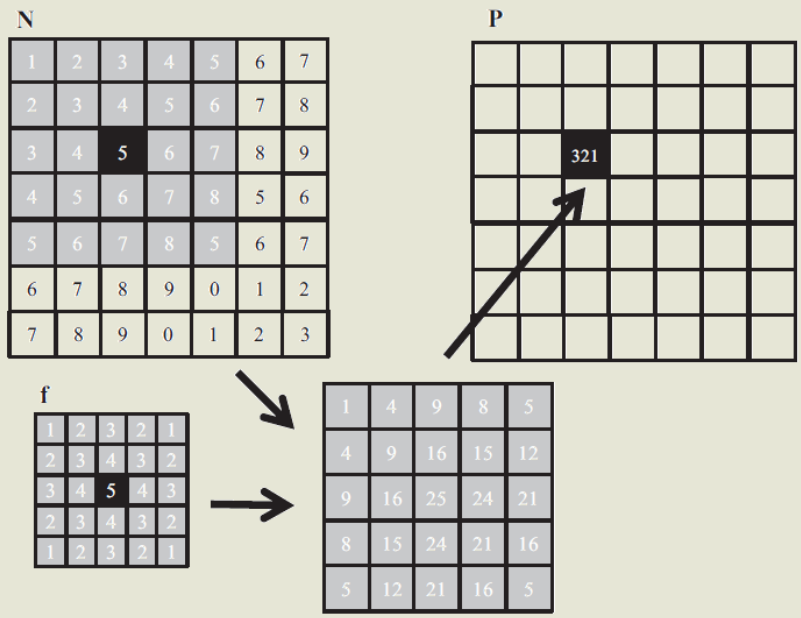


FIGURE 7.4
A 2D convolution example.

11

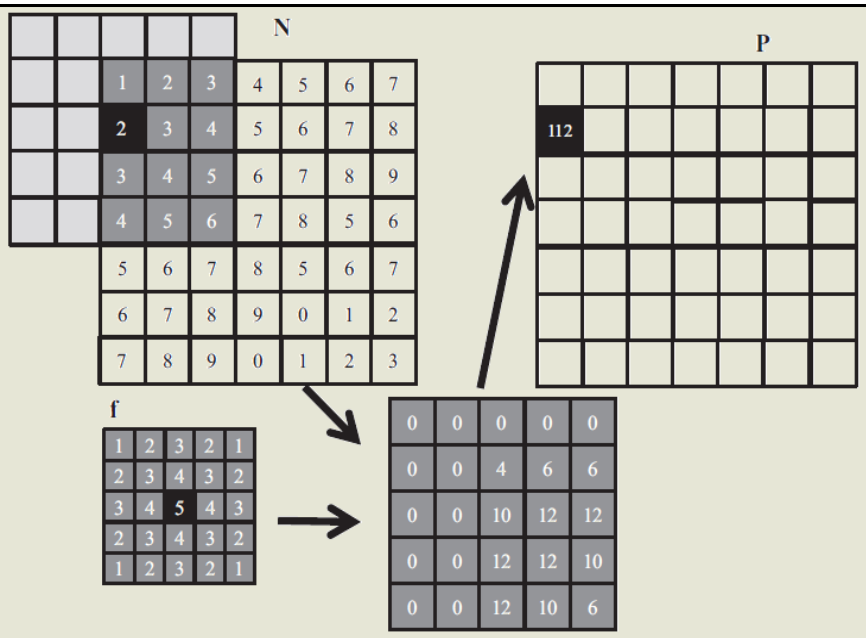


FIGURE 7.5
A 2D convolution boundary condition.

12

Storing the Mask in Constant Memory

- We can store the mask in constant memory
  - The mask is small
  - It is constant
  - It is accessed by all threads
- Constant memory is cached inside each CPU, and it is particularly fast.

13

Storing the Mask in Constant Memory

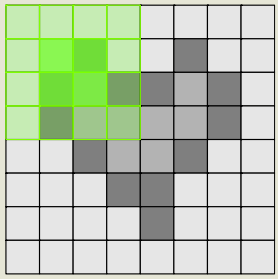
- Declare the mask as a global variable

```
#define MASK_WIDTH 5
const float M[MASK_WIDTH];
```

14

Recall: Data Reuse: Tiling

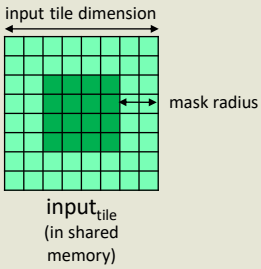
- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory

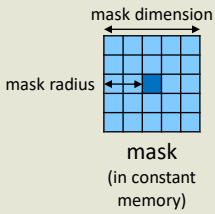


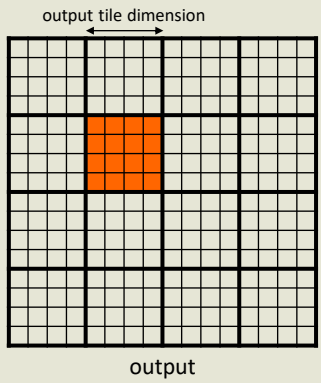
15

15

Loading Tiles into Shared Memory







Challenge: Input and output tiles have different dimensions

(input tile dimension = output tile dimension + 2 × mask radius)

Slide credit: Izzat El Hajj

16

Sequential Code: Initializing….

```
1.    void convolve(RGBQUAD** image, int width, int height) {
2.        // Allocate memory for the 2D array
3.        RGBQUAD* t_image = (RGBQUAD*)malloc(height * width * sizeof(RGBQUAD*));
4.        RGBQUAD** resultImage = (RGBQUAD**)malloc(height * sizeof(RGBQUAD**));
5.        if (t_image == NULL || resultImage == NULL) {
6.            printf("Unable to allocate memory.\n");   return;}
7.        for (int i = 0; i < height; i++) {
8.            resultImage[i] = &t_image[i*width];
9.        }

10.       // Set kernel Radius
11.       const int kernelRadius = kernelSize / 2;
```

17

Sequential Code: actual calculations

```
1.    // Apply convolution filter
2.    for (int y = 0; y < height; y++) {
3.      for (int x = 0; x < width; x++) {
4.        int sumR = 0, sumG = 0, sumB = 0;
5.        for (int ky = -kernelRadius; ky <= kernelRadius; ky++) {
6.           for (int kx = -kernelRadius; kx <= kernelRadius; kx++) {
7.             int pixelX = x + kx;
8.             int pixelY = y + ky;
9.             // Handle boundary conditions (ghost values assumed to be zero)
10.            if (pixelX >= 0 && pixelX < width && pixelY >= 0 && pixelY < height) {
11.   sumR += image[pixelY][pixelX].rgbRed * kernel5x5[ky + kernelRadius][kx + kernelRadius];
12.   sumG += image[pixelY][pixelX].rgbGreen * kernel5x5[ky + kernelRadius][kx + kernelRadius];
13.   sumB += image[pixelY][pixelX].rgbBlue * kernel5x5[ky + kernelRadius][kx + kernelRadius];
14.            } // end if
15.          } // for
16.       } // for
17.   …
18.   …
```

18

Sequential Code: Updating the result image

```
1.  // Apply convolution filter
2.  for (int y = 0; y < height; y++) {
3.    for (int x = 0; x < width; x++) {
4.  …
5.  …

6.  resultImage[y][x].rgbRed = sumR/256;
7.  resultImage[y][x].rgbGreen = sumG/256;
8.  resultImage[y][x].rgbBlue = sumB/256;

9.  // Ensure RGB values are within valid range
10. resultImage[y][x].rgbRed = (resultImage[y][x].rgbRed > 255) ? 255 :
11.      (resultImage[y][x].rgbRed < 0) ? 0 : resultImage[y][x].rgbRed;
12. resultImage[y][x].rgbGreen = (resultImage[y][x].rgbGreen > 255) ? 255 :
13.      (resultImage[y][x].rgbGreen < 0) ? 0 : resultImage[y][x].rgbGreen;
14. resultImage[y][x].rgbBlue = (resultImage[y][x].rgbBlue > 255) ? 255 :
15.      (resultImage[y][x].rgbBlue < 0) ? 0 : resultImage[y][x].rgbBlue;
16.         }
17.     }
```

19

Sequential Code: updataing input image

```
1.    // Copy the result image back to the original image
2.    for (int y = 0; y < height; y++) {
3.        for (int x = 0; x < width; x++) {
4.            image[y][x] = resultImage[y][x];
5.        }
6.    }

7.    free(resultImage);
8.    free(t_image);
9.    printf("Convolution Applied successfully.\n");
10.   }
```

20