

COMS4040A & COMS7045A Assignment 3 – Report

Abdulkadir Dere - 752817 - Computer Science Hons

29 May 2020

1 Introduction

In this report, we will focus on the different design methodologies used to create 2D convolution on provided images. We will look at serial implementation, CUDA implementation using both global memory and constant memory, CUDA implementation using both shared memory and constant memory and CUDA implementation using texture memory. Each design methodology is defined and process has been explained with the code.

2 Methodology

2.1 Serial Computation

We will first implement the serial version of 2D image convolution algorithm using the CPU. Serial version of the algorithm will follow the following design methodology:

- Read the image using built-in functions. 2D image is returned as 1D image and assigned to a variable.
- Convert the read image from 1D to 2D for easier computation of convolution algorithm since the image is in 2D.
- Pad the image so mask we can compute the corners of the image. Image will be padded with zero padding.
- Apply the convolution mask/filter to the padded image. The convolution algorithm will be executed as a recursive algorithm. The process will start by selecting the first element of the array excluding the padded pixels. We will subtract padded size from offset (half size of the mask) to find the starting row and column of the image. This pixel must be same as the original pixel. We will apply the convolution mask to this pixel. (Algorithm 1)

- The convolution mask application process starts by identifying the neighbours of the given pixel location. We will compute neighbours according to the mask size since image will be padded by the factor of half of mask on all four sides. We will not run out-of-bounds when retrieving the pixel values for each neighbour since the image is padded. We will create a convolution results array with the dimensions of the mask size to compute the results. We will do element-wise multiplication between neighbours and the mask. All of these values will be added together to compute the convolution result of the given pixel location. (Algorithm 2)
- The convolution result for the pixel location will be saved in the output image and process will continue with the next pixel location until algorithm iterates through all the pixels. (Algorithm 1)
- The padded area will be removed by un-padding the padded image to retrieve the results in original image dimensions.
- The resulting image will be converted from 2D array to 1D to save the resulting convolution image.

The code for the detailed methodology has been provided with comments. Pre-processing and post-processing functions are not included in the report. These functions can be found in the following file: *serialConvolution.cu* under *src* folder.

```
1 // 2D serial convolution method
2 double **serial_convolution(double **input, double **output){
3     int range = padded_size - offset;
4     // printf("range: %d \n", range);
5
6     for (int i = offset; i<range; i++){
7         for (int j = offset; j<range; j++){
8             output[i][j] = applyMask(input, i, j);
9         }
10    }
11    return output;
12 }
```

Listing 1: Add zero padding to the image

```
1 double applyMask(double **array, int row, int col){
2     int n_size = offset * 2 + 1;
3
4     // neighbours of given location
5     double **neighbours = allocateMatrix(n_size, n_size);
6
7     // dynamically get the neighbours range
8     int n1 = 0;
9     for (int r=row - 1; r <= row + offset; r++){
```

```

10     int n2 = 0;
11     for (int c = col - 1; c <= col + offset; c++){
12         neighbours[n1][n2] = array[r][c];
13         n2++;
14     }
15     n1++;
16 }

18 double **convolution = allocateMatrix(n_size, n_size);
19 double value = 0;
20 for (int r=0; r<3; r++){
21     for(int c=0; c<3; c++){
22         convolution[r][c] = mask[r][c] * neighbours[r][c];
23         value = value + convolution[r][c];
24     }
25 }
26 return value;
27 }

```

Listing 2: Apply convolution mask to the given pixel

2.2 CUDA implementation using both global memory and constant memory

Image convolution using CUDA C has been implemented using both global memory and constant memory. The convolution mask is constant through out the convolution process. It is beneficial to cache the convolution mask in the constant memory as this informs the CUDA runtime that mask values will not change during kernel execution (Kirk and Hwu [2010]) (Algorithm 3). The constant memory will set the mask value as read-only and it will be broadcasted to all elements in the convolution kernel execution. The convolution process will be done in global memory using a CUDA kernel (Algorithm 4). The kernel parameters consist of original image, allocated space for resulting image, width and height of the original image. The row and column indexes will be computed to identify which index will be computed by which thread. The starting index for row and column will be calculated by subtracting the offset value because we want to ignore the padded area. This will let the convolution process to start and end at dimensions of the original image so we don't run out-of-bounds. The kernel will calculate each convolution value by calculating all the elements within the mask filter size (dimension) and add all of them to get the convolution value for a specific pixel. The row and column indexes will be verified so that we are within the dimensions. The resulting value will be saved to the resulting image space.

```

1 // Convolution Mask Dimension
2 #define MASK_DIM 3

```

```
3 #define OFFSET (MASK_DIM/2)

5 // allocate mask in constant memory
6 __constant__ float d_mask_global[MASK_DIM * MASK_DIM];
```

Listing 3: Cache mask in to the constant memory

```
1 // 2D convolution using global and constant memory
2 __global__ void global_convolution(float *d_Data, float *d_result, int
   width, int height) {
3     // calculate the row and column index to compute for each thread
4     int row = blockIdx.y * blockDim.y + threadIdx.y;
5     int col = blockIdx.x * blockDim.x + threadIdx.x;

7     // Starting index for convolution so we can ignore the padded area
8     int i_row = row - OFFSET;
9     int i_col = col - OFFSET;

11    // convolution value to be calculated for each pixel's row and
       column
12    double value = 0;
13    // iterate over all rows and column using the mask dimension.
14    // this will calculate all the neighbours and origin pixel and sum
       these values to give
15    // us the value of the origin pixel
16    for (int i = 0; i < MASK_DIM; i++) {
17        for (int j = 0; j < MASK_DIM; j++) {
18            if ((i_row + i) >= 0 && (i_row + i) < height && (i_col + j) >= 0
                && (i_col + j) < width) {
19                // sum all the values within the range of the mask to get
                origin pixel's value
20                value += d_Data[(i_row + i) * width + (i_col + j)] *
                    d_mask_global[i * MASK_DIM + j];
21            }
22        }
23    }
24    // write back convolution result
25    d_result[row * width + col] = value;
26 }
```

Listing 4: 2D Convolution using the global memory

2.3 CUDA implementation using both shared memory and constant memory

In this section, we will look at image convolution using both shared memory and constant memory.

Listing 5: Cache mask in to the constant memory

Listing 6: 2D Convolution using the shared memory

2.4 CUDA implementation using texture memory

3 Questions and Answers

4 Experiment

4.1 Experiment Setup

Experiments are conducted on MacBook Pro (15-inch, 2019) by Apple laptop. Processing power of the laptop is 2.6 GHz 6-Core Intel Core i7. Memory specifications of the MacBook is 16 GB 2400 MHz DDR4. Operating system of the MacBook is macOS Catalina. Experiments conducted using different combination of distance and sorting algorithms with varying sizes for the m reference, q query and d dimensional points. These varying sizes are recorded and kept constant to not affect the results.

4.2 Experiment Data

Experiments are done using synthetic data. Synthetic data is generated by using a random number generator algorithm for the reference and query matrices. Following sizes have been used for the variables m , n and d in different experiments as shown in results. We have used 100000 for m references and 10 for d dimensions. We have altered q query points between 100 and 200.

4.3 Experiment Results

Experimental results are shown with different distance and sorting algorithms and with varying n query points.

4.4 Summary of Results

5 Conclusion

References

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2nd edition.