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

We will be using averaging, edge detection and sharpening masks for the convolution process.

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.

-1	-1	-1
-1	9	-1
-1	-1	-1

-1	0	1
-2	0	2
-1	0	1

1	1	1
1	1	1
1	1	1

Table 1: Masks used for convolution: sharpening, edge detection and averaging respectively

- 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 founds in the following file: *serialConvolution.cu* under *src* folder.

```
// 2D serial convolution method
double **serial_convolution(double **input, double **output){
    int range = padded_size - offset;
    // printf("range: %d \n", range);

    for (int i = offset; i<range; i++){
        for (int j = offset; j<range; j++){
            output[i][j] = applyMask(input, i, j);
        }
    }
    return output;
}
```

Listing 1: Add zero padding to the image

```c
double applyMask(double **array, int row, int col){
    int n_size = offset * 2 + 1;

    // neighbours of given location
    double **neighbours = allocateMatrix(n_size, n_size);

    // dynamically get the neighbours range
    int n1 = 0;
    for (int r=row - 1; r <= row + offset; r++){
        int n2 = 0;
        for (int c =col - 1; c <= col + offset; c++){
            neighbours[n1][n2] = array[r][c];
            n2++;
        }
        n1++;
    }

    double **convolution = allocateMatrix(n_size, n_size);
    double value = 0;
    for (int r=0; r<3; r++){
        for(int c=0; c<3; c++){
            convolution[r][c] = mask[r][c] * neighbours[r][c];
            value = value + convolution[r][c];
        }
    }
    return value;
}
```

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.

```
// Convolution Mask Dimension
#define MASK_DIM 3
#define OFFSET (MASK_DIM/2)

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

Listing 3: Cache mask in to the constant memory

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

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

  // convolution value to be calculated for each pixel's row and column
  double value = 0;
  // iterate over all rows and column using the mask dimension.
  // this will calculate all the neighbours and origin pixel and sum these values to give
  // us the value of the origin pixel
  for (int i = 0; i < MASK_DIM; i++) {
    for (int j = 0; j < MASK_DIM; j++) {
      if ((i_row + i) >= 0 && (i_row + i) < height && (i_col + j) >= 0 && (i_col + j) < width) {
        // sum all the values within the range of the mask to get origin pixel's value
        value += d_Data[(i_row + i) * width + (i_col + j)] *
```

```
                d_mask_global[i * MASK_DIM + j];
      }
    }
  }
  // write back convolution result
  d_result[row * width + col] = value;
}
```

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.

```
// Convolution Mask Dimension
#define MASK_DIM 3
#define OFFSET (MASK_DIM/2)

#define TILE_WIDTH 16
#define RADIUS 2
#define BLOCK_WIDTH (TILE_WIDTH+(2*RADIUS))

#define DIAMETER (RADIUS*2+1) // mask diameter
#define SIZE (RADIUS*DIAMETER) // mask size

// allocate mask in constant memory
__constant__ float d_mask_shared[MASK_DIM * MASK_DIM];
```

Listing 5: Cache mask in to the constant memory

```
__global__ void shared_convolution(float* dData, float* dResult,
   unsigned int width, unsigned int height){

  // create tile in shared memrory for the convolution
  __shared__ float shared[BLOCK_WIDTH * BLOCK_WIDTH];

    // for simplicity to use threadIdx
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
```

```
  int by = blockIdx.y;

  // get row and column index of pixels in the tile
  int col = bx * TILE_WIDTH + tx - RADIUS;
  int row = by * TILE_WIDTH + ty - RADIUS;

  // Find the last and first pixel locations within the image
  col = max(0, col);
  col = min(col, width-1);
  row = max(row, 0);
  row = min(row, height-1);

  // load the tile pixels from the global memory into shared memory
  // this will help us to reduce global memory access by the factor of 1/TILE_WIDTH
  // ignore any pixels which are out-of-bounds (i.e. padded area)
  unsigned int index = row * width + col;
  unsigned int block_index = ty * blockDim.y + tx;
  shared[block_index] = dData[index];

  // thread barrier to wait for all the threads to finish loading from
  // global memory to shared memory
  __syncthreads();

  // Elementwise multiplication of pixel and mask values and add all of the values within the mask
  // range to get output value of one pixel. Verify that we are not working out-of-bounds of the image
  // We will iterate over rows and columns within the mask dimensions (i.e. all the neighbours)
  float value = 0;
  if (((tx >= RADIUS) && (tx < BLOCK_WIDTH-RADIUS)) && ((ty>=RADIUS) && (ty<=BLOCK_WIDTH-RADIUS))){
    for(int i = 0; i<MASK_DIM; i++){
        for(int j = 0; j<MASK_DIM; j++){
          value += shared[block_index+(i*blockDim.x)+j] * d_mask_shared[i*3+j];
        }
    }
    dResult[index] = value;
}
```

```
}
```

Listing 6: 2D Convolution using the shared memory

3 Questions and Answers

4 Experiment

4.1 Experiment Setup

Experiments are conducted on a cluster. The details for the CUDA device are listed.
CUDA Device 0
Major revision number: 6
Minor revision number: 1
Name: GeForce GTX 1060 6GB
Total global memory: 6371475456
Total shared memory per block: 49152
Total registers per block: 65536
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Clock rate: 1784500
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 10
Kernel execution timeout: Yes

Image	**image21.pgm**	**lena_bw.pgm**
Matrix Size	512x512	512x512
Tile Size	16x16	16x16
Serial Convolution Time (ms)	154.862244	152.380386
Global Memory Time (ms)	0.093184	0.094880
Shared Memory Time (ms)	0.030656	0.032704
Speedup of global memory kernel (ms)	1661.9	1606.03
Speedup of shared memory kernel (ms)	5051.61	4659.38
Throughput of serial implementation (GFLOPS)	0.0152348	0.0154829
Throughput of global memory implementation (GFLOPS)	25.3187	24.8661
Throughput of shared memory implementation (GFLOPS)	76.9603	72.1409
Performance improvement: global over serial	1661.9x	1606.03x
Performance improvement: shared over serial	5051.61x	4659.38x
Performance improvement: shared over global	3.03967x	2.90117x

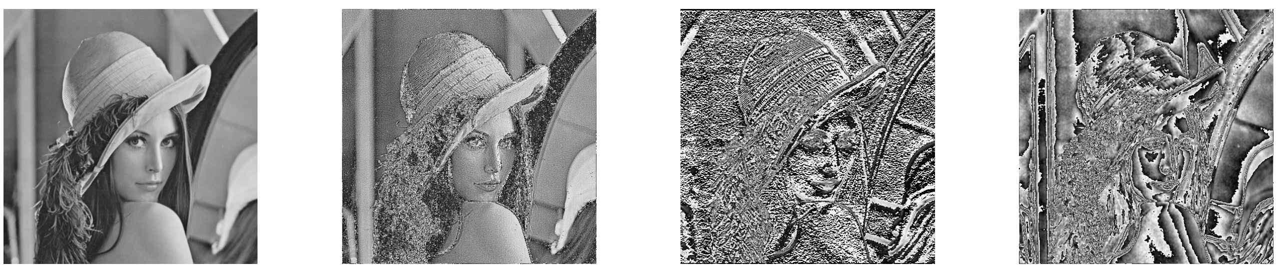
Table 2: Results of the convolutions applied to the given images.

4.2 Experiment Results

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

(a) (b) (c) (d)

Figure 1: Convolution results for the *image*21 image (a) Original Image (b) Sharpening (c) Edge detection (d) Averaging



(a) (b) (c) (d)

Figure 2: Convolution results for the *lena_bw* image (a) Original Image (b) Sharpening (c) Edge detection (d) Averaging

4.3 Summary of Results

As we can see from the Table 2 that global memory version has greatly improved on the serial implementation. The shared memory version has improved on global memory and serial implementations. However, global memory implementation is still not at its top efficiency since every thread calculates its own output. This can be increased by implementing threads to compute a tile of the image similar to shared memory implementation.

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.