Lab 4

George Onwubuya

November 1, 2018

1 Output

1.1 Matrix Output (m x n)

```
Setting up the problem...0.006791 s
    Image: 600 x 1000
    Mask: 5 x 5
Allocating device variables...0.201096 s
Copying data from host to device...0.001548 s
Launching kernel...0.000209 s
Copying data from device to host...0.001337 s
Verifying results...TEST PASSED


Setting up the problem...0.025171 s
    Image: 1200 x 2000
    Mask: 5 x 5
Allocating device variables...0.181609 s
Copying data from host to device...0.004396 s
Launching kernel...0.000497 s
Copying data from device to host...0.006667 s
Verifying results...TEST PASSED


Setting up the problem...0.097721 s
    Image: 2400 x 4000
    Mask: 5 x 5
Allocating device variables...0.184917 s
Copying data from host to device...0.020479 s
Launching kernel...0.001661 s
Copying data from device to host...0.032120 s
Verifying results...TEST PASSED
```

```
Setting up the problem...0.400631 s
    Image: 4800 x 8000
    Mask: 5 x 5
Allocating device variables...0.183295 s
Copying data from host to device...0.077317 s
Launching kernel...0.006211 s
Copying data from device to host...0.121019 s
Verifying results...TEST PASSED


Setting up the problem...1.571396 s
    Image: 9600 x 16000
    Mask: 5 x 5
Allocating device variables...0.192857 s
Copying data from host to device...0.259799 s
Launching kernel...0.024441 s
Copying data from device to host...0.475482 s
Verifying results...TEST PASSED
```

1.2 Matrix Output (m x m)

```
Setting up the problem...0.012148 s
    Image: 1000 x 1000
    Mask: 5 x 5
Allocating device variables...0.160189 s
Copying data from host to device...0.002237 s
Launching kernel...0.000304 s
Copying data from device to host...0.003576 s
Verifying results...TEST PASSED


Setting up the problem...0.042840 s
    Image: 2000 x 2000
    Mask: 5 x 5
Allocating device variables...0.147785 s
Copying data from host to device...0.007397 s
Launching kernel...0.000784 s
Copying data from device to host...0.010515 s
Verifying results...TEST PASSED


Setting up the problem...0.168949 s
    Image: 4000 x 4000
    Mask: 5 x 5
```

```
Allocating device variables...0.146379 s
Copying data from host to device...0.031356 s
Launching kernel...0.002791 s
Copying data from device to host...0.043496 s
Verifying results...TEST PASSED


Setting up the problem...0.667698 s
    Image: 8000 x 8000
    Mask: 5 x 5
Allocating device variables...0.147719 s
Copying data from host to device...0.129577 s
Launching kernel...0.010494 s
Copying data from device to host...0.222300 s
Verifying results...TEST PASSED


Setting up the problem...2.687160 s
    Image: 16000 x 16000
    Mask: 5 x 5
Allocating device variables...0.156885 s
Copying data from host to device...0.535993 s
Launching kernel...0.041381 s
Copying data from device to host...0.637439 s
Verifying results...TEST PASSED
```

2 Performance Analysis

2.1 Rectangle Matrices (m x n)

Execution Time (seconds) for Each Process					
Elements(m*n)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
600 x 1000	0.006791	0.0.201096	0.001548	0.000209	0.001337
1200 x 2000	0.025171	0.181609	0.004396	0.000497	0.006667
2400 x 4000	0.097721	0.184917	0.020479	0.001661	0.032120
4800 x 8000	0.400631	0.183295	0.077317	0.006211	0.121019
9600 x 16000	1.571396	0.192857	0.259799	0.024441	0.475482

2.2 Square Matrices (m x m)

Execution Time (seconds) for Each Process					
Elements(m*m)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
1000	0.012148	0.160189	0.002237	0.000304	0.003576
2000	0.042840	0.147785	0.007397	0.000784	0.010515
4000	0.168949	0.146379	0.031356	0.002791	0.043496
8000	0.667698	0.147719	0.129577	0.010494	0.222300
16000	2.687160	0.156885	0.535993	0.041381	0.637439

2.3 Comments

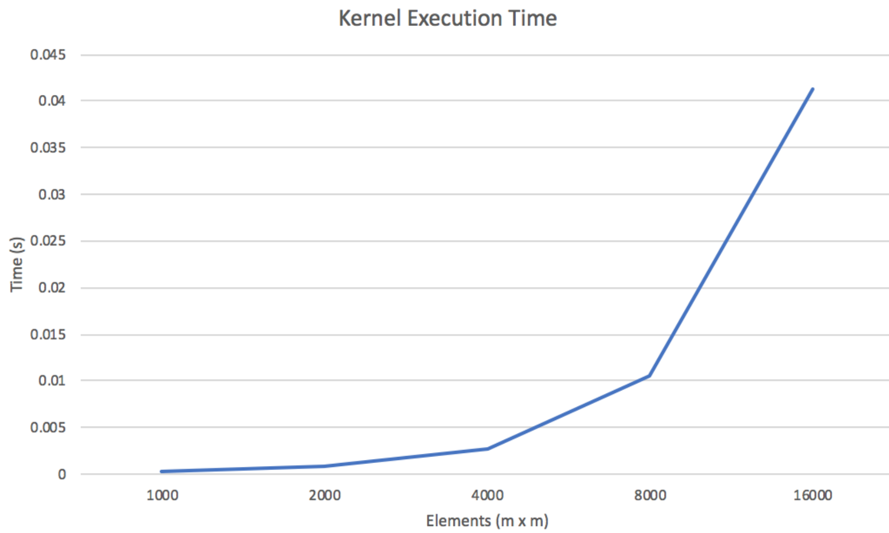
There is an observable increase in time for each execution subsection as the number of elements increases. The time taken to set up the problem, copy data from host to device, launch the kernel and copy from device to host increases. There is a direct proportion between the execution times and the number of elements. This observation is true both rectangle and square matrices

3 Answers

3.1 C(i)

The floating point computation rate varies as the size of the matrices increase. The choice of matrix size was double the size of m for a series of executions in order to observe how the kernel times changed. That means a matrix with m = 2000 would scale by 4 if m is doubled, m = 4000. The kernel execution times for the matrices are shown below in the table. When the number of elements increases from 1000 to 2000, the time is scaled by 2.6, and with subsequent increases (2000, 4000, 8000 16000) the times are scaled by 3.6, 3.8 4.

Kernel Execution Time (seconds)	
Elements (m x m)	Launch Kernel
1000	0.000304
2000	0.000784
4000	0.002791
8000	0.010494
16000	0.041381



3.2 C(ii)

The table below shows the overhead, which is the total time spent on the GPU side as a ratio over the total execution time expressed as a %. The amount of time spent in the GPU increases as the number of elements increase. GPU overhead accounts for more than 95% of the total execution time

Overhead Time as a Percentage			
Elements (m x m)	Total Time	Device Time	Overhead
1000	0.166306	0.166002	99.81720443
2000	0.166481	0.165697	99.52907539
4000	0.224022	0.221231	98.75414022
8000	0.51009	0.499596	97.94271599
16000	1.371698	1.330317	96.98322809

4 Main

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
```

```
*******************************************************************************/

#include <stdio.h>
#include "support.h"
#include "kernel.cu"

int main(int argc, char* argv[])
{
    Timer timer;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

        Matrix M_h, N_h, P_h; // M: filter, N: input image, P: output image
        Matrix N_d, P_d;
        unsigned imageHeight, imageWidth;
        cudaError_t cuda_ret;
        dim3 dim_grid, dim_block;

        /* Read image dimensions */
    if (argc == 1) {
        imageHeight = 600;
        imageWidth = 1000;
    } else if (argc == 2) {
        imageHeight = atoi(argv[1]);
        imageWidth = atoi(argv[1]);
    } else if (argc == 3) {
        imageHeight = atoi(argv[1]);
        imageWidth = atoi(argv[2]);
    } else {
        printf("\n    Invalid input parameters!"
           "\n    Usage: ./convolution          # Image is 600 x 1000"
           "\n    Usage: ./convolution <m>      # Image is m x m"
           "\n    Usage: ./convolution <m> <n>  # Image is m x n"
           "\n");
        exit(0);
    }

        /* Allocate host memory */
        M_h = allocateMatrix(FILTER_SIZE, FILTER_SIZE);
```

```
        N_h = allocateMatrix(imageHeight, imageWidth);
        P_h = allocateMatrix(imageHeight, imageWidth);

        /* Initialize filter and images */
        initMatrix(M_h);
        initMatrix(N_h);

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));
    printf("    Image: %u x %u\n", imageHeight, imageWidth);
    printf("    Mask: %u x %u\n", FILTER_SIZE, FILTER_SIZE);

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

        N_d = allocateDeviceMatrix(imageHeight, imageWidth);
        P_d = allocateDeviceMatrix(imageHeight, imageWidth);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy host variables to device ------------------------------------------

    printf("Copying data from host to device..."); fflush(stdout);
    startTime(&timer);

        /* Copy image to device global memory */
        copyToDeviceMatrix(N_d, N_h);

        /* Copy mask to device constant memory */

        cuda_ret = cudaMemcpyToSymbol(M_c, M_h.elements,
            M_h.height*M_h.width * sizeof(float));
        if(cuda_ret != cudaSuccess) FATAL("Unable to copy to constant memory");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Launch kernel ----------------------------------------------------------
    printf("Launching kernel..."); fflush(stdout);
```

```
    startTime(&timer);

        dim_block.x = BLOCK_SIZE;
        dim_block.y = BLOCK_SIZE;
          dim_block.z = 1;

        dim_grid.x = imageWidth/TILE_SIZE;
        if(imageWidth%TILE_SIZE != 0) dim_grid.x++;
        dim_grid.y = imageHeight/TILE_SIZE;
        if(imageHeight%TILE_SIZE != 0) dim_grid.y++;
        dim_grid.z = 1;

        convolution<<<dim_grid, dim_block>>>(N_d, P_d);

        cuda_ret = cudaDeviceSynchronize();
        if(cuda_ret != cudaSuccess) FATAL("Unable to launch/execute kernel");

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy device variables from host ----------------------------------------

    printf("Copying data from device to host..."); fflush(stdout);
    startTime(&timer);

    copyFromDeviceMatrix(P_h, P_d);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Verify correctness -----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);

    verify(M_h, N_h, P_h);

    // Free memory ------------------------------------------------------------

        freeMatrix(M_h);
        freeMatrix(N_h);
        freeMatrix(P_h);
```

```
        freeDeviceMatrix(N_d);
        freeDeviceMatrix(P_d);

        return 0;
}
```

5 Kernel

```
/********************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ******************************************************************************/

__constant__ float M_c[FILTER_SIZE][FILTER_SIZE];

/*__device__ float getElement(Matrix *N, const int row, const int col)
{
        return N->elements[row*N->width+col];
}
*/
/*__device__ void retElem(Matrix *P, const int row, const int col, float value)
{
        P->elements[row*P->width+col] = value;

        return;
}*/

__global__ void convolution(Matrix N, Matrix P)
{
        /*******************************************************************
        Determine input and output indexes of each thread
        Load a tile of the input image to shared memory
        Apply the filter on the input image tile
        Write the compute values to the output image at the correct indexes
        *******************************************************************/

```

```
        //INSERT KERNEL CODE HERE
        /*int col = blockIdx.x * blockDim.x + threadIdx.x;
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col_zeroIndex = col - FILTER_SIZE/2;
        int row_zeroIndex = row - FILTER_SIZE/2;
        float sum = 0;

        for(int j = 0; j < FILTER_SIZE; ++j){
                for(int k = 0; k < FILTER_SIZE; ++k){
                        if((row_zeroIndex + j >= 0) && (row_zeroIndex + j < N.height) &&
                             (col_zeroIndex + k >= 0) && (col_zeroIndex+ k < N.width)){
                                        //sum = M_c[j][k] * getElement(&N, row_zeroIndex + j, col_zeroIndex + k);

                                        sum += M_c[j][k] * N.elements[(row_zeroIndex + j)*N.width + col_zeroIndex +k];
                        }
                }
         }
        if( row < P.height  && col < P.width)

                //retElem(&P, row, col, sum);

                P.elements[row * P.width + col] = sum;*/


        int row = blockIdx.y * TILE_SIZE + threadIdx.y;
        int col = blockIdx.x * TILE_SIZE + threadIdx.x;
        int rowZeroIndex = row - FILTER_SIZE/2;
        int colZeroIndex = col - FILTER_SIZE/2;


        __shared__ float N_ds[TILE_SIZE + FILTER_SIZE - 1][TILE_SIZE + FILTER_SIZE - 1];

        if((rowZeroIndex >= 0) && (rowZeroIndex < N.height) && (colZeroIndex >= 0) && (colZeroIndex < N.width)){

                N_ds[threadIdx.y][threadIdx.x] = N.elements[rowZeroIndex * N.width + colZeroIndex];
        }
```

```
        else{
                N_ds[threadIdx.y][threadIdx.x] = 0.0f;
        }

        __syncthreads();

        float sum = 0.0f;

        if(threadIdx.y < TILE_SIZE && threadIdx.x < TILE_SIZE){

                for(int dr = 0; dr < FILTER_SIZE; ++dr){

                        for(int dc = 0; dc < FILTER_SIZE; ++dc){

                                sum += M_c[dr][dc] * N_ds[threadIdx.y + dr][threadIdx.x + dc];
                        }
                }

        if(row < P.height && col < P.width){

                P.elements[row * P.width + col] = sum;
        }

        }
}
```