CPE810 – GPU And Multicore Programming

Lab Report #3: Homework 3

Convolution

Instructor: Zhou Feng

TA: Ali Aghdaei

James Fong

I pledge my honor that I have abided by the Stevens Honor System

Table of Contents

Objective

Implement a convolution routine using texture and shared memory in CUDA. Your code should be able handle arbitrary 2D input sizes (1D kernel and 2D image pixel values should be randomly generated between 0~15). You only need to use a 1D kernel (mask) for both row and column convolutions that can be performed separately.

Required Functions and Implementation Details

1. Allocate Device Memory

```
h_Kernel    = (float *)malloc(KERNEL_LENGTH * sizeof(float));
h_Input     = (float *)malloc(imageW * imageH * sizeof(float));
h_Buffer    = (float *)malloc(imageW * imageH * sizeof(float));
h_OutputCPU = (float *)malloc(imageW * imageH * sizeof(float));
h_OutputGPU = (float *)malloc(imageW * imageH * sizeof(float));
```

2. Copy Host Memory to Device

```
checkCudaErrors(cudaMallocArray(&a_Src, &floatTex, imageW, imageH));
checkCudaErrors(cudaMalloc((void **)&d_Output, imageW * imageH * sizeof(float)));

cudaResourceDesc            texRes;
memset(&texRes,0,sizeof(cudaResourceDesc));

texRes.resType              = cudaResourceTypeArray;
texRes.res.array.array      = a_Src;

cudaTextureDesc             texDescr;
memset(&texDescr,0,sizeof(cudaTextureDesc));

texDescr.normalizedCoords = false;
texDescr.filterMode       = cudaFilterModeLinear;
texDescr.addressMode[0] = cudaAddressModeWrap;
texDescr.addressMode[1] = cudaAddressModeWrap;
texDescr.readMode = cudaReadModeElementType;

checkCudaErrors(cudaCreateTextureObject(&texSrc, &texRes, &texDescr, NULL));

srand(2009);

for (unsigned int i = 0; i < KERNEL_LENGTH; i++)
{
    h_Kernel[i] = (float)(rand() % 16);
}

for (unsigned int i = 0; i < imageW * imageH; i++)
{
    h_Input[i] = (float)(rand() % 16);
}

setConvolutionKernel(h_Kernel);
checkCudaErrors(cudaMemcpyToArray(a_Src, 0, 0, h_Input, imageW * imageH *
sizeof(float), cudaMemcpyHostToDevice));
```

3. Initialize Thread Block and Kernel Grid Dimensions

```
dim3 dimsInputMatrix;
int maskLength;
int threadCount = 512;
```

```
if (argc == 4) {
if (atoi(argv[1]) > 0)
        dimsInputMatrix.y = atoi(argv[1]);

    if (atoi(argv[2]) > 0)
        dimsInputMatrix.x = atoi(argv[2]);

    dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y;

    if (atoi(argv[3]) > 0)
        maskLength = atoi(argv[3]);

    cout << "Three Command Line Arguments Accepted." << endl;
    }

else if (argc == 5) {
    if (atoi(argv[1]) > 0)
       dimsInputMatrix.y = atoi(argv[1]);

    if (atoi(argv[2]) > 0)
       dimsInputMatrix.x = atoi(argv[2]);

    dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y;

    if (atoi(argv[3]) > 0)
        maskLength = atoi(argv[3]);

    if (atoi(argv[4]) > 0)
        threadCount = atoi(argv[4]);

    cout << "Four Command Line Arguments Accepted." << endl;
}

else {
    cout << "Enter row dimensions: ";
    cin >> dimsInputMatrix.y;

    cout << "Enter column dimensions: ";
    cin >> dimsInputMatrix.x;

    //Total number of elements in the input matrix
    dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y;

    cout << "Enter the length of the mask: ";
    cin >> maskLength;

    cout << "Enter the threads per block: ";
    cin >> threadCount;

    if (threadCount > 32)
        threadCount = 32;
}

const int imageW = 3072;
const int imageH = 3072 / 2;
const unsigned int iterations = 10;
```

4. Invoke CUDA Kernel

```
const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);
const float  x = (float)ix + 0.5f;
const float  y = (float)iy + 0.5f;

if (ix >= imageW || iy >= imageH)
{
    return;
}

float sum = 0;
```

5. Copy Results from Device to Host

```
checkCudaErrors(cudaDeviceSynchronize());
sdkResetTimer(&hTimer);
sdkStartTimer(&hTimer);

for (unsigned int i = 0; i < iterations; i++)
{
    convolutionRowsGPU(
        d_Output,
        a_Src,
        imageW,
        imageH,
        texSrc
    );
}

checkCudaErrors(cudaDeviceSynchronize());
sdkStopTimer(&hTimer);
gpuTime = sdkGetTimerValue(&hTimer) / (float)iterations;
printf("Average convolutionRowsGPU() time: %f msecs; //%f Mpix/s\n", gpuTime,
imageW * imageH * 1e-6 / (0.001 * gpuTime));

//While CUDA kernels can't write to textures directly, this copy is inevitable
printf("Copying convolutionRowGPU() output back to the texture...\n");
checkCudaErrors(cudaDeviceSynchronize());
sdkResetTimer(&hTimer);
sdkStartTimer(&hTimer);
checkCudaErrors(cudaMemcpyToArray(a_Src, 0, 0, d_Output, imageW * imageH *
sizeof(float), cudaMemcpyDeviceToDevice));
checkCudaErrors(cudaDeviceSynchronize());
sdkStopTimer(&hTimer);
gpuTime = sdkGetTimerValue(&hTimer);
printf("cudaMemcpyToArray() time: %f msecs; //%f Mpix/s\n", gpuTime, imageW *
imageH * 1e-6 / (0.001 * gpuTime));

printf("Running GPU columns convolution (%i iterations)\n", iterations);
checkCudaErrors(cudaDeviceSynchronize());
sdkResetTimer(&hTimer);
sdkStartTimer(&hTimer);
```

```
for (int i = 0; i < iterations; i++)
{
    convolutionColumnsGPU(
        d_Output,
        a_Src,
        imageW,
        imageH,
        texSrc
    );
}

checkCudaErrors(cudaDeviceSynchronize());
sdkStopTimer(&hTimer);
gpuTime = sdkGetTimerValue(&hTimer) / (float)iterations;
printf("Average convolutionColumnsGPU() time: %f msecs; //%f Mpix/s\n", gpuTime,
imageW * imageH * 1e-6 / (0.001 * gpuTime));

printf("Reading back GPU results...\n");

checkCudaErrors(cudaMemcpy(h_OutputGPU, d_Output, imageW * imageH * sizeof(float),
cudaMemcpyDeviceToHost));
```

6. Deallocate Device Memory

```
checkCudaErrors(cudaFree(d_Output));
checkCudaErrors(cudaFreeArray(a_Src));
free(h_OutputGPU);
free(h_Buffer);
free(h_Input);
free(h_Kernel);
```

7. Implement The 2D Image Convolution Kernel Using Texture and Shared Memories

```
__global__ void convolutionRowsKernel(
    float* d_Dst,
    int imageW,
    int imageH,
    float* d_Mask,
    int KERNEL_RADIUS,
    cudaTextureObject_t texSrc
)
{
    //allocate shared memory
    extern __shared__ float c_Kernel[];

    //initialize shared memory
    if (threadIdx.x < (2 * KERNEL_RADIUS)) {
        c_Kernel[threadIdx.x] = d_Mask[threadIdx.x];
    }

    //wait for threads to clear shared memory
    __syncthreads();
```

```
        const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
        const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);
        const float  x = (float)ix + 0.5f;
        const float  y = (float)iy + 0.5f;

        if (ix >= imageW || iy >= imageH)
            {
               return;
            }

        float sum = 0;

        for (int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
            {
                    sum += tex2D<float>(texSrc, x + (float)k, y) *
                    c_Kernel[KERNEL_RADIUS - k];
            }

        d_Dst[IMAD(iy, imageW, ix)] = sum;
    }
    __global__ void convolutionColumnsGPUKernel(
            float* d_Dst,
            int imageW,
            int imageH,
            float* d_Mask,
            int KERNEL_RADIUS,
            cudaTextureObject_t texSrc
    )
    {
            //allocate shared memory
            extern __shared__ float c_Kernel[];

            //initialize shared memory
            if (threadIdx.x < (2 * KERNEL_RADIUS)) {
                    c_Kernel[threadIdx.x] = d_Mask[threadIdx.x];
            }

            //wait for threads to clear shared memory
            __syncthreads();

            const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
            const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);
            const float  x = (float)ix + 0.5f;
            const float  y = (float)iy + 0.5f;

            if (ix >= imageW || iy >= imageH)
            {
                return;
            }

            float sum = 0;

            for (int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
            {
                sum += tex2D<float>(texSrc, x, y + (float)k) * c_Kernel[KERNEL_RADIUS -
k];
            }
```

```
            d_Dst[IMAD(iy, imageW, ix)] = sum;
        }
```

8. Handle Thread Divergence When Dealing With Arbitrary 2D Input and Kernel Sizes

```
    dim3 dimsInputMatrix;    //Dimensions for Input Matrix
    int maskLength;          //Mask Length
    int threadCount;  //Thread Count
    int iterations;

    if (argc == 4) {
        if (atoi(argv[1]) > 0)
            dimsInputMatrix.y = atoi(argv[1]); //Row count for Input Matrix of image

        if (atoi(argv[2]) > 0)
            dimsInputMatrix.x = atoi(argv[2]); //Columns count for Input Matrix of image

        dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y;  //Total number of
elements in the input matrix

        if (atoi(argv[3]) > 0)
            maskLength = atoi(argv[3]); //Argument for Mask Length
        else {
            cout << "Command Line Argument not accepted." << endl;
            return 0;
        }

        cout << "Three Command Line Arguments Accepted." << endl;
    }

    else if (argc == 5) {
        if (atoi(argv[1]) > 0)
            dimsInputMatrix.y = atoi(argv[1]);

        if (atoi(argv[2]) > 0)
            dimsInputMatrix.x = atoi(argv[2]);

        dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y; //Total number of
elements in the input matrix

        if (atoi(argv[3]) > 0)
            maskLength = atoi(argv[3]);

        if (atoi(argv[4]) > 0)
            threadCount = atoi(argv[4]); // Argument for thread length
        else {
            cout << "Command Line Argument not accepted." << endl;
            return 0;
        }

        cout << "Four Command Line Arguments Accepted." << endl;
    }

    else if (argc == 6) {
```

```
        if (atoi(argv[1]) > 0)
            dimsInputMatrix.y = atoi(argv[1]);

        if (atoi(argv[2]) > 0)
            dimsInputMatrix.x = atoi(argv[2]);

        dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y; //Total number of
elements in the input matrix

        if (atoi(argv[3]) > 0)
            maskLength = atoi(argv[3]);

        if (atoi(argv[4]) > 0)
            threadCount = atoi(argv[4]);

        if (atoi(argv[5]) > 0)
            iterations = atoi(argv[5]); //Total number of iterations.
        else {
            cout << "Command Line Argument not accepted." << endl;
            return 0;
        }

        cout << "Five Command Line Arguments Accepted." << endl;
    }

    else {
        cout << "Enter row dimensions: ";
        cin >> dimsInputMatrix.y;

        cout << "Enter column dimensions: ";
        cin >> dimsInputMatrix.x;

        //Total number of elements in the input matrix
        dimsInputMatrix.z = dimsInputMatrix.x * dimsInputMatrix.y;

        cout << "Enter the length of the mask: ";
        cin >> maskLength;

        cout << "Enter the threads per block: ";
        cin >> threadCount;

        if (threadCount > 32)
        {
            cout << "Maximum number of threads is 32, thus setting current number of
threads as 32" << endl;
            threadCount = 32;
        }

        cout << "Enter the iterations per block: ";
        cin >> iterations;


    }
```

Questions

1. Name 3 applications of convolution.

Image Processing, Signal Processing, and computer vision. [1].

2. How many floating-point operations are being performed in your convolution kernel (expressed in dimX, dimY and dimK)? Explain.

Based on the operations being performed by the kernel:

```
const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);
const float  x = (float)ix + 0.5f;
const float  y = (float)iy + 0.5f;

if (ix >= imageW || iy >= imageH)
{
    return;
}

float sum = 0;

for (int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
{

    sum += tex2D<float>(texSrc, x, y + (float)k) * c_Kernel[KERNEL_RADIUS - k];

 }
```

There are a total of three separate floating point operations done by the convolution kernel for rows, and three separate floating point operations done by the convolution kernel for columns. Thus, there are six separate floating operations done per calculation, and two other float operations calculated for x and y, for row and columns before beginning calculations.

3. How many global memory reads are being performed by your kernel (expressed in dimX, dimY and dimK)? Explain.

The global memory reads being performed by the kernel are done after every operation performed by the kernel, after initializing shared memory with inputted data calculation. This can be seen below using the code for utilizing shared memory:

```
//allocate shared memory
extern __shared__ float c_Kernel[];

//initialize shared memory
if (threadIdx.x < (2 * KERNEL_RADIUS)) {
    c_Kernel[threadIdx.x] = d_Mask[threadIdx.x];
}

//wait for threads to clear shared memory

__syncthreads();
```

4. How many global memory writes are being performed by your kernel (expressed in dimX, dimY and dimK)? Explain.

The global memory writes being performed by my kernel, is after every convolution row calculation, is done after every write using dimX to dimK, for Row convolution, with a similar process for column convolution. This can be seen below using the code for utilizing shared memory:

```
//allocate shared memory
extern __shared__ float c_Kernel[];

//initialize shared memory
if (threadIdx.x < (2 * KERNEL_RADIUS)) {
    c_Kernel[threadIdx.x] = d_Mask[threadIdx.x];
}

//wait for threads to clear shared memory

__syncthreads();
```

5. What is the minimum, maximum, and average number of real operations that a thread will perform (expressed in dimX, dimY and dimK)? Real operations are those that directly contribute to the final output value.

The minimum number of real operations a thread will perform are dependent on the thread count and total number of elements (given by dimX and dimY), in which the same amount of elements and the same amount of thread count is equal to a minimum of 1 real operations. By default, the machine is set to accept a maximum of 32 thread count, so the maximum number of elements for real operations is 32 elements.

The maximum number of real operations that a thread will perform are dependent on the thread count and the total number of elements (given by dimX and dimY), in which reaching the maximum amount of elements vs the total thread count is equal to the total number of real operations. The maximum allowable number of elements are technically unlimited, but to avoid idle threads, it should not larger than the given kernel dimK.

The average number of real operations that a thread will perform are the given total number of elements divided by the given thread count. This number is then divided by the maximum total of real operations, determined by the given kernel dimK.

6. What is the measured floating-point computation rate for the CPU and GPU kernels in this application? How do they each scale with the size of the input?

The measured floating point computation rate for the CPU ad GPU kernels in this application are indicated by chart 1 for GPU computation, and chart 2 for CPU computation, which display the computational rate vs the X or Y size of input of the matrices. Charts can be located in the charts section of the report.

7. What will happens if the mask (convolution kernel) size is too large (e.g. 1024)? What do you end up spending most of your time doing? Does that put other constraints on the way you'd write your algorithm (think of the shared/constant memory size)?

If the mask size is too large, it will result in a large amount of idle threads and will waste the total available parallelism [2]. As the radius of the filter increases, the percentage of idle threads increase, and will result in wasting memory by causing too many idle threads, thus requiring the use of separable filters to increase efficiency. This can be done through tiling to reduce the number of unnecessary data loads by dividing the process into separate passes. [3]

Most of the time spent for this assignment was done attempting to include shared memory working in conjunction with textured memory, with not many examples being available online to understand and in practice, required a lot of time to include. This put other constraints on how the algorithm would have been written by changing how mask length is included, and how defining the other needed inputs would be accepted.

Experimentation

Various forms of experimentation came into play with different types of sizes for convolution. The main method of altering the sizes of matrices.

For simplicity, matrices were kept as squared numbers. This was then compared with different types of mask lengths, which were used to test approximations for avoiding Idle Threads.

Threads per block were kept to a maximum allowable limit of 32, and number of iterations can be adjusted but were kept to a default amount of 10.

Calculations were kept to a default, determined by the sample code left by CUDA developers. Part of the shared memory code was in part adapted by convolutionSeparable example, but most remained with convolutionTextured sample code due to requiring textured memory.

Chart 1 compares GPU performance, with Computation Rate [in Mpix/s] compared to the size of the input. GPU computation rate tends to increase logarithmically.

Chart 2 compares CPU performance with Computation Rate [in Mpix/s] compared to the size of the input. CPU performance tends to increase linearly with the amount of computation increasing with the input. However, compared to the GPU performance, it is dramatically slower than the GPU performance.

Conclusion

CUDA offers many different implementations for convolution algorithms with various types of performance techniques; often, these techniques demonstrate various forms of performance enhancement, or are performed for different kinds of image filtering.

Our specific implementation utilizes both textured memory and shared memory, which is good for a dynamically allocated space for memory and utilizing shared memory for high memory throughput. The GPU is limited by the amount of available shared memory, while CPU implementation is limited by cache size.

Tables

Input [dimsX * dimsY]	Rows Computation Rate [Mpix/s]	Computation Rate Time (ms)	CudaMemcpyToArray [Mpix/s]	CudaMemcpyToArray Time (ms)	Average convolutionColumnsGPU [Mpix/s]	Average convolutionColumnsGPU (ms)
10	0.003841	21.062109	71.428574	0.0014	166.666659	0.00085
50	0.11875	21.052608	1785.714349	0.0014	4237.288	0.00074
100	0.474051	21.09478	4999.999763	0.002	16666.66588	0.00057
200	32786.88588	0.00122	338.12342	0.1183	74074.06896	0.00056
300	71428.5687	0.00126	689.127077	0.13066	157894.7277	0.00057
400	129032.2568	0.00124	1176.470524	0.136	280701.7382	0.00057
500	187969.9249	0.00133	3396.739055	0.0736	471698.1478	0.00053
750	396126.7645	0.00142	3695.795043	0.1522	1004464.363	0.00056
1000	689655.1812	0.00145	10131.71207	0.0987	1639344.294	0.00061

Table 1: Computed data using fixed data for mask size, thread count, and iterations.

Graphs

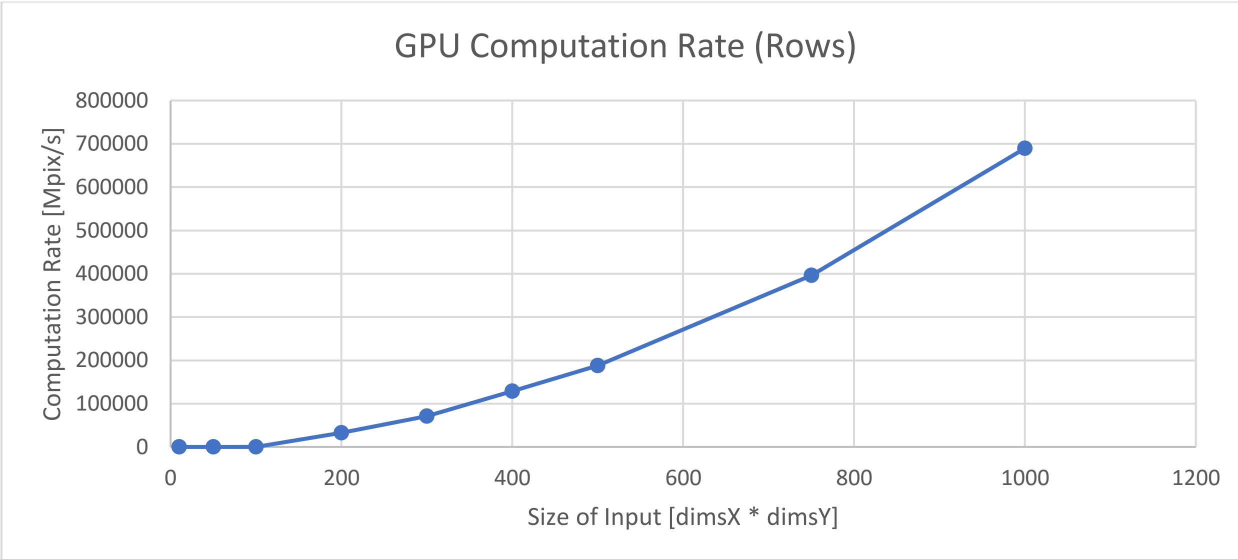

Chart 1. GPU Computation Rate. (Rows)

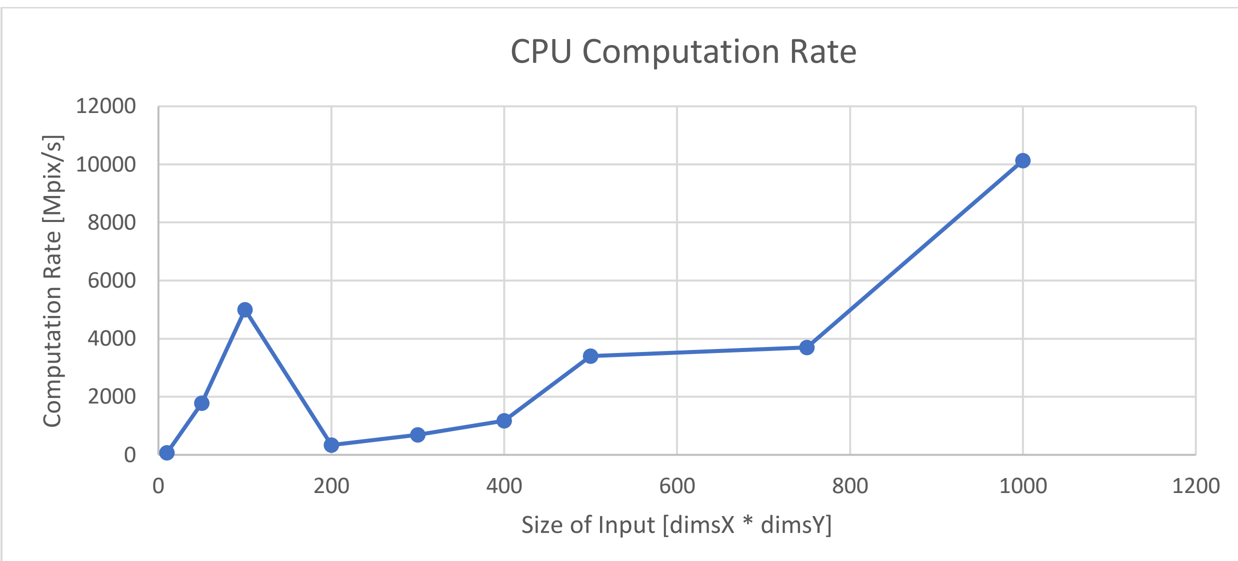

Chart 2. CPU Computation Rate.

References

[1]:
https://www.projectrhea.org/rhea/index.php/Applications_of_Convolution:_Simple_Image_Blurring

[2]: https://developer.download.nvidia.com/assets/cuda/files/convolutionSeparable.pdf

[3]:https://docs.nvidia.com/cuda/samples/3_Imaging/convolutionSeparable/doc/convolutionSeparable.pdf