## 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.

## Instructions

Make a new folder for your project. Copy and modify the code of "convolutionTexture" in CUDA sample code (3_Imaging folder) to include the following key functions:

1. Allocate device memory
2. Copy host memory to device
3. Initialize thread block and kernel grid dimensions
4. Invoke CUDA kernel
5. Copy results from device to host
6. Deallocate device memory
7. Implement the 2D image convolution kernel using texture and shared memories
8. Handle thread divergence when dealing with arbitrary 2D input and kernel sizes

## **Preface**

My implementation is an altered version of the CUDA convolutionTexture.

### Code Execution Guide

My code can be ran via several methods. Valid command line arguments options include:

```
./CudaConvolution.exe <dimX> <dimX > <dimK>
```
Or…
```
./CudaConvolution.exe <dimX> <dimX > <dimK> <threadCount>
```
Or…
```
./CudaConvolution.exe <dimX> <dimX > <dimK> <threadCount> <testIterations>
```

| | |
|---|---|
| <dimX> | X-dimension of the input 2D matrix. |
| <dimY> | Y-dimension of the input 2D matrix. |
| <dimK> | The size of the 1D matrix. Must be less than or equal to threadCount. |
| <threadCount> | The number of threads per block to be used for GPU computation. A maximum of 32. |
| <testIterations> | Number of times to run each GPU kernel (For testing/experimentation resolution). |

Or via console user prompt. To access this, exclude command line arguments when running the executable. The prompts will guide your operation.

I have included an instruction checklist which allows for quick indexing of my code to ensure I followed the instructions outlined on the previous page.

| **Instruction Checklist** | | |
|---|---|---|
| | Line of Occurrence | Function/Kernel/Definition |
| 1 | 184 – 187 | `checkCudaErrors(cudaMalloc(&d_OutputGPU, z_full_bytes)); …` |
| 2 | 219 – 220 | `checkCudaErrors(cudaMemcpyToArray(a_Src, 0, 0, h_Input, z_full_bytes, cudaMemcpyHostToDevice)); …` |
| 3 | 84-85; 92-93 | `dim3 threads(threadCount, threadCount);` `dim3 blocks(iDivUp(matrixCol_x, threads.x), iDivUp(matrixRow_y, threads.y)); …` |
| 4 | 87; 95 | `convolutionRowsGPUKernel <<<blocks, threads, SHARMEM>>> (d_Dst, matrixCol_x, matrixRow_y, d_mask, mask_len_rad, texSrc);` `convolutionColsGPUKernel <<<blocks, threads, SHARMEM >>> (d_Dst, matrixCol_x, matrixRow_y, d_mask, mask_len_rad, texSrc);` |
| 5 | 273 | `checkCudaErrors(cudaMemcpy(h_OutputGPU, d_OutputGPU, inputDims.z * sizeof(float), cudaMemcpyDeviceToHost));` |
| 6 | 297 – 299 | `checkCudaErrors(cudaFree(d_OutputGPU));` `checkCudaErrors(cudaFree(d_Mask));` `checkCudaErrors(cudaFreeArray(a_Src));` |
| 7 | 32; 56 | `__global__ void convolutionRowsGPUKernel(…)` `__global__ void convolutionColsGPUKernel(…)` |
| 8 | 38, 62; | `__syncthreads();` |

## Questions

1. Name 3 applications of convolution.
    a. Signal Processing
    b. Image Processing
    c. Differential Equations

---

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

Since my code is utilizing textured memory with wrapping to resolve the out of bounds memory calls. There are no inactive threads during mask indexing.

The number of floating-point operations is equivalent to:

$$2 * 3 * dimK * dimX * dimY$$

- The 2 is for each convolution kernel (Rows and Columns).
- The 3 is for each convolution step:

"
```
[sum += [tex2D<float>(texSrc, [x + (float)k], y) * shared_mask[mask_len_rad.second - k] ] ];
```
"

I have colored the three different operations in question.

Since this line is ran in a loop for each element of the mask and then via each thread for each element of the input matrix, this is equivalent to multiplying the 3 by the dimension of the mask and the dimensions of the input matrix.

---

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

Initializing the shared memory version of the convolution mask requires global memory reads. This value is equivalent to number of mask elements times the number of blocks deployed. $[dimK * numBlocks]$

Since texture memory is global memory, every read from my texture object is a read from global memory. Here that number is equivalent to the number of elements in the input array times the number of elements in the mask (or $dimX * dimY * dimK$); this assumes that the compiler isn't holding this value in register for future accesses during the aforementioned for loop. And if you assumed the compiler does this clear cut optimization, the number of global memory reads would be equal to $dimX * dimY$ instead.

The variables here must be multiplied by 2 because there are two kernel functions being done here. So in the end this is the equation for the number of global memory reads:

$$2 * (dimK * numBlocks + dimX * dimY)$$

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

   Standard global memory is used for storing the output of the convolutions, and these are the only operations writing to global memory. That number is equivalent to the number of elements in the output matrix, $dimX * dimY$.

   But, we must multiply this function by 2 to account of the two kernel calls.

$$2 * dimX * dimY$$

   This is the final equation for the number of global memory writes.

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.

   Assuming that the only operation that matters here is the summation of the output element.

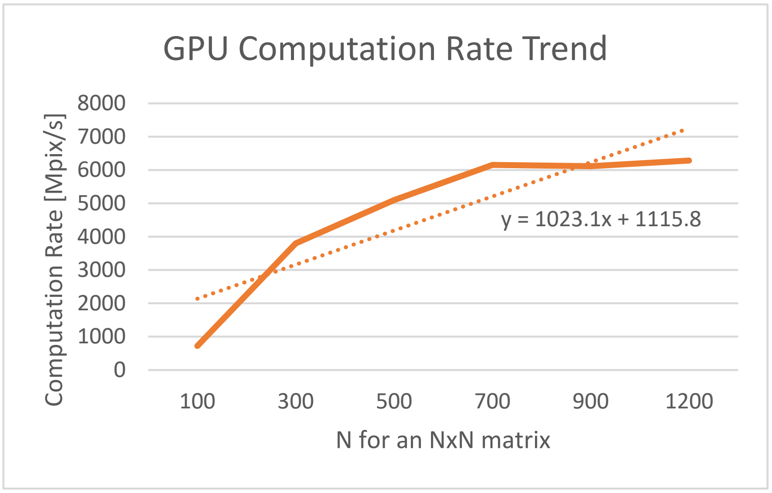
   $dimK$ real operations are performed per thread.

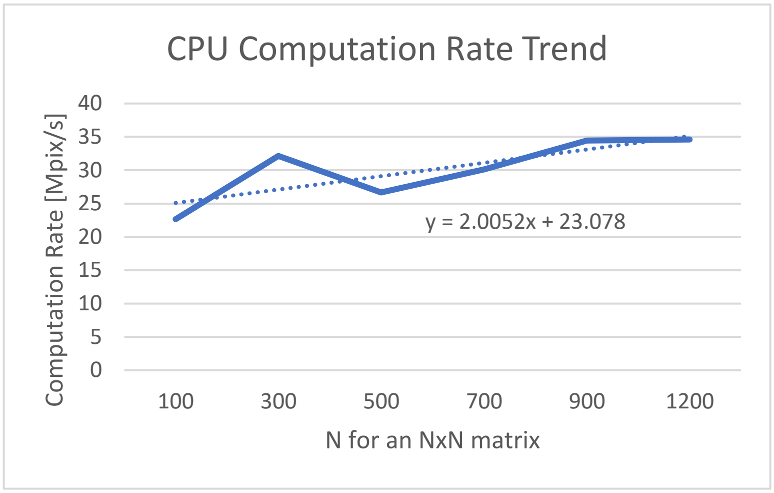
   Again, this is because the NVIDIA sample code and my code both use:

```
texDescr.addressMode[0] = cudaAddressModeWrap;
texDescr.addressMode[1] = cudaAddressModeWrap;
```

Which ensures there are no out-of-bounds accesses, as well as ensuring that there is always a calculation going on since the, would be, out-of-bounds accesses are wrapped around to valid positions and calculated on. Because of this there is no variation in the number of real operations. Max = Min = Avg, for this algorithm setup. I could see this question being more meaningful for an algorithm that intends on ignoring out-of-bound indices rather than wrapping. Since the CUDA sample, which we were instructed to utilize and alter for this assignment, uses wrapping my code does as well.

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?



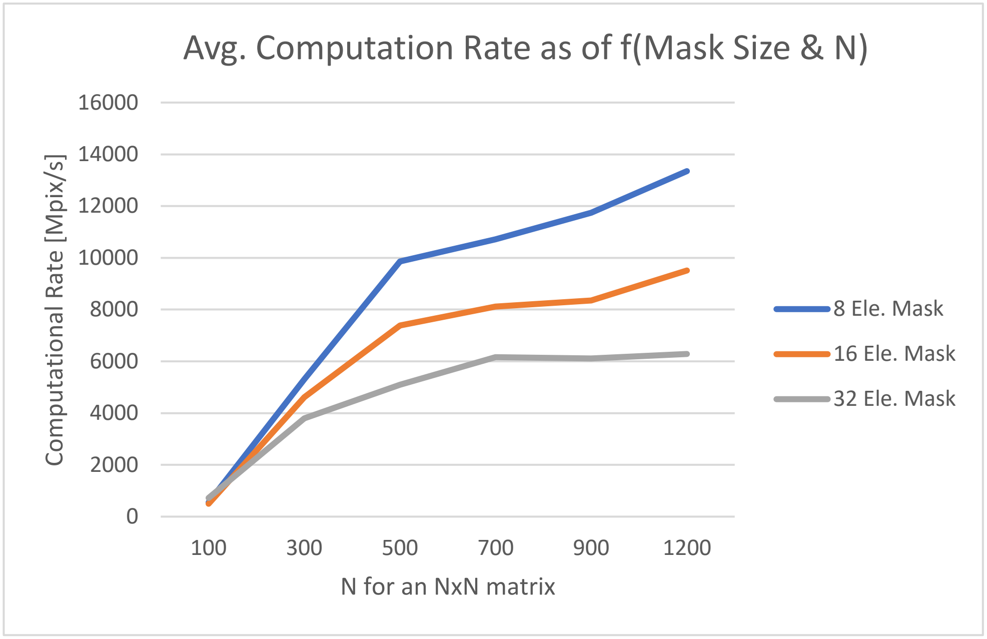


These charts display the computational rate of a computational device vs. input matrix size. As input size increases, the GPU's computational rate increases logarithmically. And CPU's computational rate increases seemingly linearly.

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)?

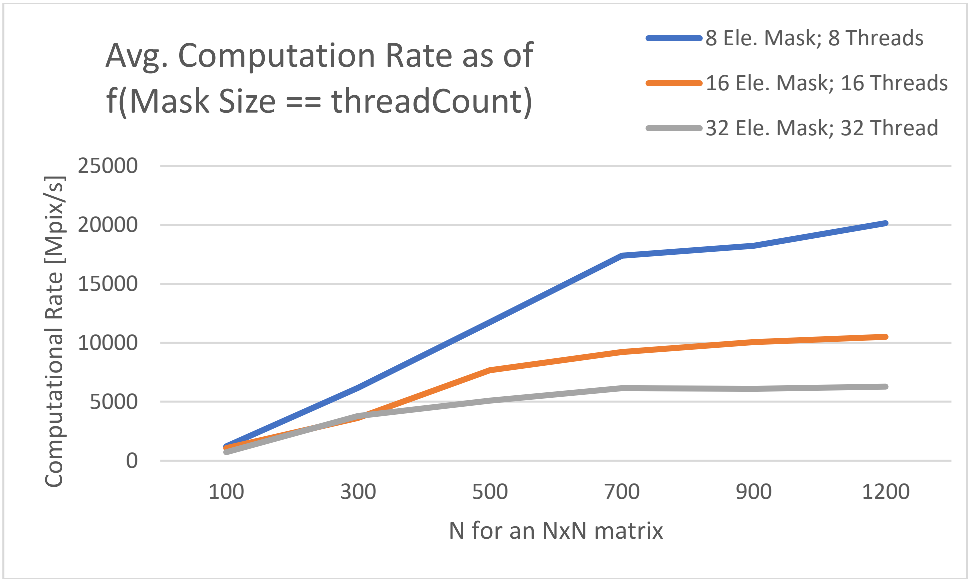
The larger the mask is the more time that will be spent moving memory from global to shared memory. You'd have to analyze the time to read from constant memory vs. reading from global, storing in shared then computing. If operating from constant memory solely would end up faster, for large masks you could call a different CUDA kernel with the mask stored in constant memory rather than in shared memory.

## Experimentation



Ran with 32, the max. threads

Clearly, the smaller the mask the more performant the code is. Less time is spend per kernel call and per thread.



This chart further shows the mask's length dependance. We can also observe that the thread count total does have a positive increase in computational rate, this is due to the inclusion of additional block which can operate at a larger parallelized scale.

I did not analyze the performance of non-square matrices, because that would bias one side of the convolution more than another.

## Conclusion

This was by far the most difficult code to implement. I squarely believe this is due to the fact we did not cover the topic of implementing texture memory in class. As such a was left to teach myself the topic, which seems to be a severe lack of useful documentation. I was in the end able to piece everything together however, and have a good idea of out to utilize texture memory in CUDA as well as on NVIDIA gpus.