5SIA0 ECA - GPU Assignment

Omar Zohir Aly - 1586459

I. Introduction

In this report, we discuss a number of optimizations in implementing 1D convolution and 2D convolution on a Nvidia GPU (using CUDA and C++) as well as the effect of the different sizes of the input arrays, as well as size of the mask used in the convolution on the performance of the kernel implemented on the GPU.

II. 1D Convolution

The 1D convolution algorithm implies convolution of a one-dimensional input array with a one-dimensional mask, i.e., centering the mask on each element of the input array, multiply the elements of the mask and the array, then saving the output in the corresponding output array.

Each step of the convolution process involves loading the chunk of the input array being processed (either from the global memory or from the shared memory), then loading the mask from the memory, and multiplying the input array chunk with the mask, adding it up, and then saving the output in the output array.

Since the process is implemented on a GPU, one shall think spatially about the problem in hand, i.e., one should think of the role of each thread and how memory shall be arranged within each block to maximize the benefit from the spatial locality as well as the parallelization offered by the GPU.

In the naive implementation of the 1D convolution, each thread shall be responsible for calculating one element in the output array. For example, a thread of a global index of 0 shall multiply the mask (centered at the element 0 in the input array) with the corresponding proportion of the input array (which involves padding the input array). In this implementation, both the mask and the input arrays are loaded from the global memory, in which its access may take around hundreds of cycles. The main aim of the following optimizations is to assure that the memory access latency is reduced as much as possible.

It is worthy to mention that, ahead of the start of the kernel, one shall allocate memory for the arrays on the host and the device (i.e., the GPU) memory. Also, the input array and the mask shall be randomly initialized on the host memory, then copied to the memory allocated on the device using cudaMemcpy with the flag cudaMemcpyHostToDevice. After the kernel is run, the result is copied back to the host, again using cudaMemcpy with the flag cudaMemcpyDeviceToHost. Then both the host and the device implementation are checked against one another, to assure that the kernel implementation is correct. Then, all the dynamically-allocated memory on the host or the device shall be freed. Also, to assure that the comparison is fair, both the host and the device implementations are run for a 1000 times, and then the duration of each is compared to one another, to gain an insight on the performance optimization gained through the different implementations. The following style shall be followed all along the optimizations mentioned in this report.

A. Task 1. Implementing tiled 1D convolution

One may leverage the availability of the shared memory, which is a scratchpad memory private for each thread block, to store the elements of the input array that are being processed by the elements of the thread block. In this way, the global memory will need to be accessed only once or twice per thread (to load the memory needed from the global memory to the local shared memory), compared to accessing the global memory for `MASK_SIZE` times (which will be typically larger than 2) and thus reducing the number of times in which the threads have to access the global memory, thus reducing the kernel latency. Since the shared memory is private per block, it might make sense to set the size of each tile to the block size (i.e., the number of threads per block). However, my implementation requires a slightly different tile size, discussed in the next paragraph.

In my implementation, I decided to pad the input array with `int (MASK_SIZE/2)` elements (i.e., the mask radius) *on each side*. This should allow to swipe the mask easily on the padded memory array in the kernel, reducing the number of if statements needed in the kernel, and thus reducing warp divergences. Based on that padding technique, one shall also need a pad each tile with twice the size of the mask radius, so that the first and the last element in the tile can be implemented. Therefore the size of each tile shall be `BLOCK_SIZE + 2 * int(MASK_SIZE/2)`.

For loading elements into the tile, all of the threads in the block shall load the respective element from the input array (i.e., the element with index equivalent to the global thread index). In addition, since the tile itself is padded, the first `2 * int(MASK_SIZE/2)` threads shall load an extra element from the next block with the same thread index (to basically add the padded elements needed for calculation of the last outputs of required by the thread block). Then after all the memory load is done successfully, all the threads need to synced together, and then the calculation of the output elements may start, since all the elements needed from the input array shall be available in the shared memory of the block. The aforementioned implementation of the shared memory assumes that block size is larger than the mask size, which may or may not be the case

B. Task 2. Constant memory with tiled 1D convolution

An extra and an obvious implementation is to reduce the access latency of the mask, through storing the mask in the constant memory. The constant memory is global scratchpad memory, of a much smaller size (typically 64 KB) with a much smaller access time compared to that to the global memory (around 50 - 100 times less). Therefore, one can declare the mask as a constant memory on the GPU, and declare an array for it on the host then initialize it, then use cudaMemcpyToSymbol to copy the initialized memory on the host to the device's constant memory. Although this such a simple optimization, it will clearly show that it was a useful one.

C. Task 3. Performance comparisons

1) Different Mask Sizes

To check the different effect of different mask sizes, the host and the kernel implementations were both run on a Nvidia RTX2080 Ti with an input size `N` = 51200, and a block size of 256. Since the aforementioned implementation assumes that the mask size shall be smaller than the block size, as well as that the mask size is an odd number, the following values of M (Mask size) are used = 1,3,7,33,65,129,155,255. Figure 1 and Figure 2 show the execution times for both the C code and the different kernels used, respectively.

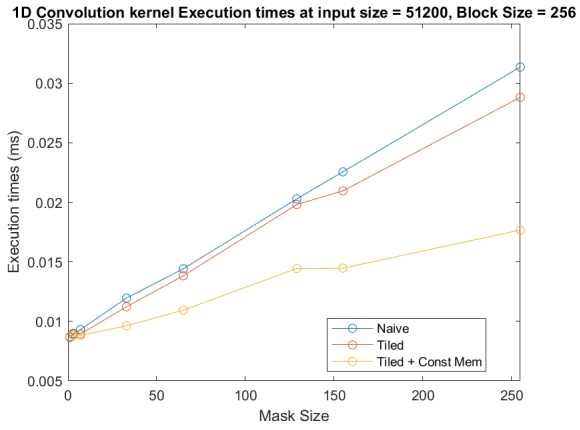

Figure 1: 1D Convolution Kernel Execution times at N = 51200, B = 256

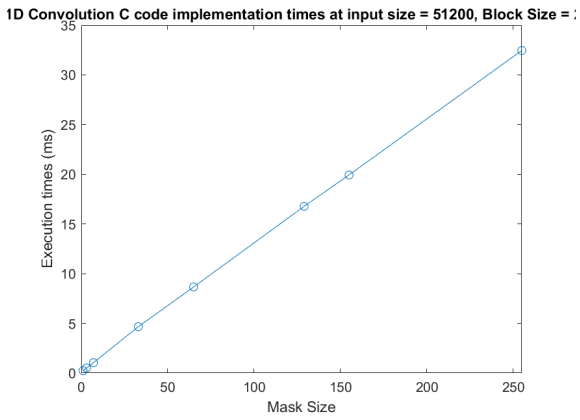

Figure 2: 1D Convolution host code Execution times at N = 51200, B = 256

Figure 1 and Figure 2 suggest that the Tiled implementation performs better than the naive one slightly. However, the constant memory does contribute to a significant optimization, especially for larger mask sizes, since this would imply much more accesses to the global memory in the case of the tiled implementation as well as the naive one. In addition, it is clear that as the mask size increase the execution times, for both the host and the device implementations, are linearly increased, since the number of computations is also linearly increased (in the case of the 1D convolution). One can also notice that the host device implementation is significantly slower than those of the device implementations, which confirms the hypothesis that the GPU parallelization shall reduce the execution times.

2) Different Block Sizes

To test if different block sizes may have an effect on the execution times, the input size will be kept as 51200, while the mask size will be tested twice, once at M = 255, and at M = 63. It is worthy to remind that the block size has to be larger than the mask, and at the same time, there is a limit of the shared memory per block, which is 48 KB in the RTX2080 Ti. Therefore, the block size will not be increased than 1024 threads.

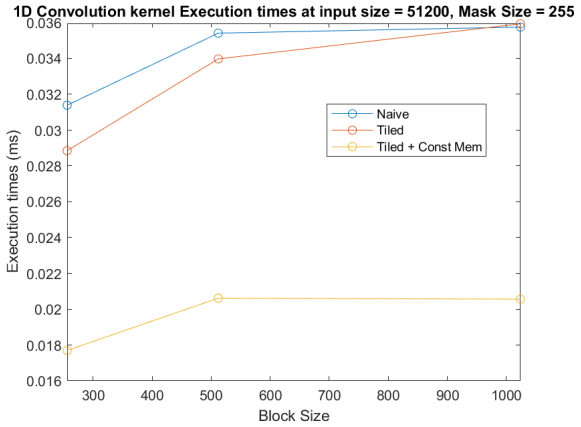

Figure 3: 1D Convolution kernel execution times at N = 51200, M = 255

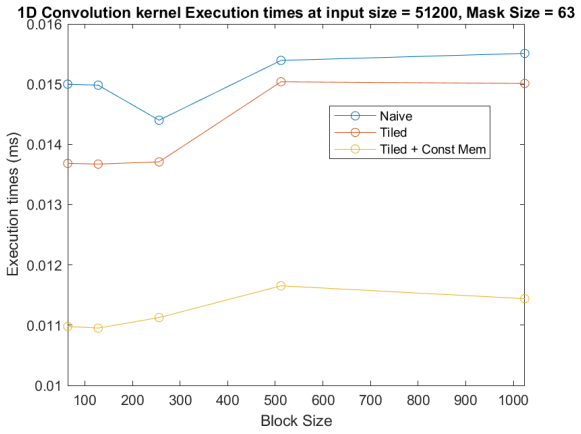

Figure 4: 1D Convolution kernel execution times at N = 51200, M = 63

Figures 3 and 4 show that the execution times slightly increase as the number of threads per blocks increase. However, that increase usually stops as the number of blocks increase from 512 to 1024. Also, there is a significant drop in the execution times in the naive implementation when B = 256, which is also unexpected. My expectation was that as the number of threads per block increase (as long as the number of threads per block is divisible by 32, since threads execute in warps of 32 either way), the running time of the kernel shall decrease due to increased parallelization. However, digging a bit deeper into the theory of it, I realised that usually the optimal number of threads per block depends on a number of factors that include the maximum number of active threads in the GPU (which differs from one GPU to the other), the number of warp schedulers in the GPU, as well as the number of active blocks per the streaming multiprocessors. Therefore, I have not came out with a certain conclusion from this section other than that the difference between the execution times is not that massive at different block sizes for my 1D convolution kernel. However, that may vary quite a lot from one processor to the other, and from one kernel to the other. However, If I have to choose an optimal block size for the 1D convolution kernel, I will typically go for 256 threads per block.

3) *Different Input sizes*

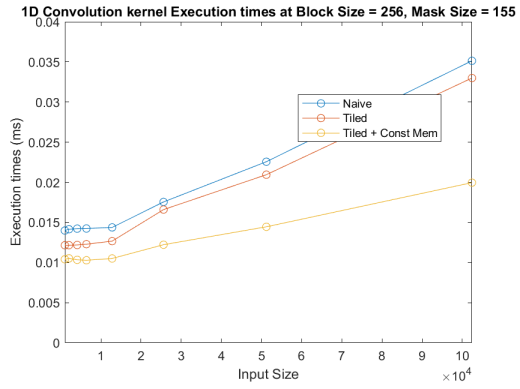

Figure 5: 1D Convolution kernel execution times at B = 256, M = 155

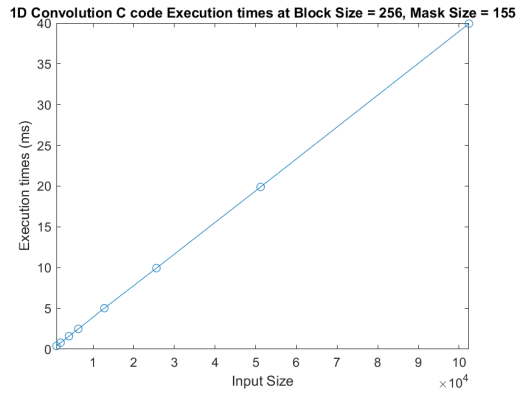

Figure 6: 1D Convolution host code execution times at B = 256, M = 155

The number of data elements shall be a multiple of the block size, to assure that each element is calculated by only one thread. In that section, the block size is kept at 256 and the mask size is kept as 155. Figures 5 and 6 show that as the number of input elements increase, the execution times increase linearly, since the number of computations increases linearly in a 1D convolution. However, it also shows that, as seen earlier, the constant memory optimization is quite significant compared to the tiling optimization. Also, we can see that the C code execution times is much much higher than that of the kernel due to the massive parallelization implemented in the kernel.

4) *Conclusion*

As the mask size, or the input size increase, the computation time shall increase due to the increased number of computations. As a heuristic, one shall try a block size of 256 as an optimal block size, and then try to seek slightly larger and smaller sizes (128,512) in an attempt to find an optimal block size (As long as the amount of shared memory needed per block does not offered by the GPU under test).

III. 2D Convolution

A few assumptions made in the 2D convolution is that, the threadblock is a square block, and the mask is also a square matrix. Again, the block size is assumed to be larger than the mask size in my implementation.

A. *Naive implementation*

The naive implementation of the 2D convolution algorithm involves the very same steps outlined in the 1D convolution. The significant difference is that the mask and the input arrays are 2D arrays (in a sense, not implementation wise), which involves a slightly more tedious calculation to obtain the row and the column of the element that thread is supposed to produce an output for. Again, each thread is responsible for only 1 element in the output array, and that would include centring the mask on the row and the col that the thread is currently processing (i.e, if a thread is supposed to compute the element (3,3) in the output array, then the mask should be centered on the input array at (3,3)). Finally, the mask size (i.e., the size of each dimension of the square mask) is assumed to be an odd number.

B. *Optimizations*

1) *Tiling and constant memory*

Implementing the constant memory is identical to that discussed in section II-B, only with the difference that the mask is now a 2D array instead of a 1D one. Tiling is slightly more different, since padding will be added above, below, on the right, and on the left of the block, all with size `int(Mask size/2)` (i.e., the mask radius).

The rest of the padding will be explained through an example to keep more understandable. Assuming that the input size is 8 (i.e., input array of size 8*8), Block size of 4, and the mask size is 3. The tile should be of size

6*6 (i.e,size of each dimension should be Block size + Mask size - 1).

Therefore in this case, 4*4 threads are launched, and they need to load a 6*6 array from the global memory to the shared array. My implementation was that the first 2 rows of the block need to load the last (2*4) elements in the tile (the last 2 rows of the pad, partially), while the first two columns of the block should load the last (4*2) elements in the tile (i.e., the last 2 columns of the pad partially). However, there is still a portion of the tile that is not loaded, which are the last (ones on the most right hand side) (2*2) elements of the tile. Loading those 2*2 elements is the responsibility of the first 2*2 threads launched in the block (i.e, threads (0,0), (0,1), (1,0), (1,1)).

Other than that is pretty much the same compared to the naive one, except that the calculation of the output needs the shared memory (Tile)and the constant memory (Mask) which take much less time for accessing.

2) Avoiding bank conflicts

I have tried to implement the bank checker as guided in the MatrixMul example. However, I could not find the header file needed for implementing the bank checker, and thus the bank checker definition was never set to 1. It also seems that, per what I have found online, the bank checker was included in an on older SDK and was then deprecated due to its ”best effort” performance.

C. Performance comparison

Using a square Block size of dimension size of 32 (a multiple of 32), a mask of size 7*7, and an input array of size 256*256, the host implementation takes around 14.5 ms, while the naive implementation takes around 0.01647 ms and the optimized one (tiling + constant memory) took around 0.01179 ms, which is 40% better than the naive implementation. It is expected that if the mask size (per dimension), or the block size (per dimension) was reduced to a half, the number of computations will be reduced to a quarter, and thus the computational time shall be reduced by a quarter. To check that (in a quite primitive way), the size of each dimension of the input array was reduced to a half. The C implementation took around 3.61 ms, which is quite close to what was expected. For the Naive and optimized CUDA, it took 0.01604 and 0.01152 ms respectively, which shows a 2.6% and a 2.3% improvement respectively. This can be attributed to the way the grid size (per dimension) was set (i.e., the number of block launched in the program), which was designed as (N + B - 1)/ B , where N is the number of input elements per dimension, and B is the number of blocks per dimension. And therefore, if the number of input elements was increased or decreased, the grid size would also shrink correspondingly.

IV. Conclusion

In this assignment, we have explored extensively how a 1D convolution and a 2D Convolution kernel would be implemented. In addition, we have observed the different effects of changing the mask size, the input array size, or the block size on the performance of the kernel.