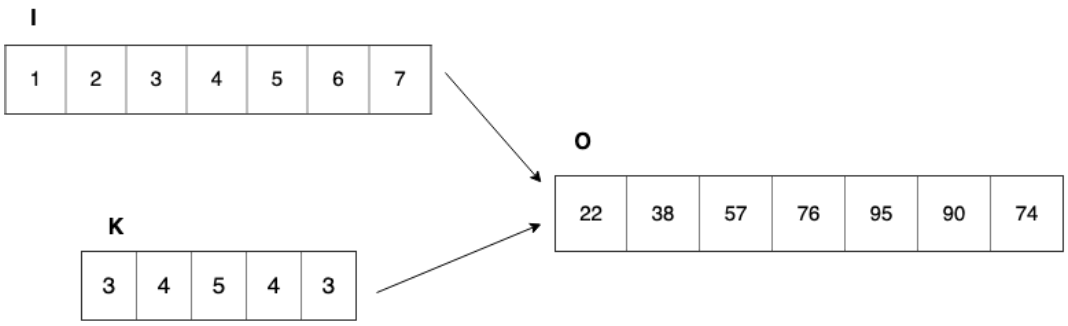
Implementing Convolutions in CUDA

The convolution operation has many applications in both image processing and deep learning (i.e. convolutional neural networks). Since convolutions can be performed on different parts of the input array (or image) independently of each other, it is a great fit for parallelization which is why convolutions are commonly performed on GPU. This blog post will cover some efficient convolution implementations on GPU using CUDA. This blog post will focus on 1D convolutions but can be extended to higher dimensional cases.

What is a Convolution?

A convolution is an operation that takes two parameters - an input array and a convolutional kernel array - and outputs another array. The convolutional kernel array is typically much smaller than the input array and iterates through the input array and at each iteration it computes a weighted sum of the current input element as well as its neighbouring input elements and the result is placed in the output array. This process is much easier to understand with an example which is shown below.



$$
\begin{aligned}
O[0] &= I[0]*K[2] + I[1]*K[3] + I[2]*K[4] \\
&= 1*5 + 2*4 + 3*3 = 22 \\
O[1] &= I[0]*K[1] + I[1]*K[2] + I[2]*K[3] + I[3]*K[4] \\
&= 1*4 + 2*5 + 3*4 + 4*3 = 38 \\
O[2] &= I[0]*K[0] + I[1]*K[1] + I[2]*K[2] + I[3]*K[3] + I[4]*K[4] \\
&= 1*3 + 2*4 + 3*5 + 4*4 + 5*3 = 57 \\
O[3] &= I[1]*K[1] + I[2]*K[2] + I[3]*K[3] + I[4]*K[4] + I[5]*K[5] \\
&= 2*3 + 3*4 + 4*5 + 5*4 + 6*3 = 76 \\
O[4] &= I[2]*K[1] + I[3]*K[2] + I[4]*K[3] + I[5]*K[4] + I[6]*K[5] \\
&= 3*3 + 4*4 + 5*5 + 6*4 + 7*3 = 95 \\
O[5] &= I[3]*K[1] + I[4]*K[2] + I[5]*K[3] + I[6]*K[4] \\
&= 4*3 + 5*4 + 6*5 + 7*4 = 90 \\
O[6] &= I[4]*K[1] + I[5]*K[2] + I[6]*K[3] \\
&= 5*3 + 6*4 + 7*5 = 74
\end{aligned}
$$

Here I is the input array, K is the convolutional kernel and O is the output array. It should be noted that for input cells near the boundaries (i.e. at the beginning and the end) there are not enough elements for a full weighted sum with the convolutional kernel. In these boundary cases we just pretend there are a sufficient number zero elements on the truncated side of the input array - these are called *ghost cells*. For example, when computing O[0] we assume that the first

two elements of the kernel, K[0] and K[1], have each been multiplied by zero-valued *ghost cells* at the beginning of the input array and therefore contribute nothing to the overall weighted sum.

Naive CUDA Implementation

Let's start off with a simple baseline CUDA implementation which we can build on later. As stated earlier, each element in the output array can be computed independently so it would make sense to have one thread per output index which is responsible for its computation. This means that this thread must

1. Read the input elements in it's neighbourhood.
2. Read the convolutional kernel.
3. Perform the convolution computation.
4. Write the result of the computation to it's index in the output array.

```
__global__ void convolution_global_memory(float *N, float *M, float *P, int Width){

        int i = blockIdx.x*blockDim.x+threadIdx.x;

        float Pvalue = 0;

        int n_start_point = i-(MASK_WIDTH/2);

        for(int j =0; j<MASK_WIDTH;j++){
                if(n_start_point+j ⩾0 && n_start_point+j < Width){
                        Pvalue+= N[n_start_point+j]*M[j];
                }
        }

        P[i]=Pvalue;
}
```

This code is relatively straight-forward. Each thread computes it's P[i] value by iterating over it's neighbourhood of input values starting at `i-(MASK_WIDTH/2)` and ending at `i+(MASK_WIDTH/2)`. The problem is that both N and M are in global memory so at each iteration the operation `N[n_start_point+j]*M[j]` involves two calls to global memory which is not as efficient as it could be.

Optimized CUDA Implementation using Constant Memory

A couple things to notice about the convolutional operation are that the convolutional kernel is never modified and that it is almost always fairly small. For these reasons, we can increase efficiency by putting the convolutional kernel in constant memory. The CUDA runtime will initially read the convolutional kernel from global memory as before however now it will cache it since it knows it will never be modified. The disadvantage of constant memory is that it is small (only 64 KB) but since our convolutional kernel is also small this shouldn't be a problem. Very few changes need to be made to put M in constant memory.

First M must be defined as a global variable with `__constant__`.

```
#define MASK_WIDTH 5
__constant__ float M[MASK_WIDTH];
```

Then the kernel is copied into constant memory with `cudaMemcpyToSymbol`.

```
cudaMemcpyToSymbol(M,h_M,MASK_WIDTH*sizeof(float));
```

Now since, M is a global variable we can remove it from the kernel function signature.

```
__global__ void convolution_constant_memory(float *N, float *P, int Width){
```

While this is an improvement over the naive approach, there is still a lot of inefficiency associated with reading the input array from global memory.

Even More Optimized CUDA Implementation using Shared Memory

In order to speed up the input array reads let's put the array in shared memory. Shared memory is memory that is shared within each block and is much faster to read from than global memory. Below is the shared memory implementation.

```
#define TILE_SIZE 4
#define INPUT_SIZE 12
#define MASK_WIDTH 5
__constant__ float M[MASK_WIDTH];

__global__ void convolution_shared_memory(float *N, float *P){

        int i = blockIdx.x*blockDim.x+threadIdx.x;

        __shared__ float N_s[TILE_SIZE];

        N_s[threadIdx.x]=N[i];

        __syncthreads();

        int this_title_start_point = blockIdx.x*blockDim.x;
        int next_tile_start_point = (blockIdx.x+1)*blockDim.x;
        int n_start_point = i-(MASK_WIDTH/2);
        float Pvalue = 0;

        for(int j =0; j < MASK_WIDTH; j++){

                int N_index = n_start_point+j;

                if(N_index >= 0 && N_index < INPUT_SIZE){
                        if((N_index >= this_title_start_point) && (N_index < next_tile_start_point)){
                                Pvalue+=N_s[threadIdx.x+j-(MASK_WIDTH/2)]*M[j];
                        }
                        else{
                                Pvalue+=N[N_index]*M[j];
                        }
                }
        }

        P[i]=Pvalue;
}
```

The first step is to create the shared memory array and populate it with the input elements for that block. We segment the input array across several blocks and call the segments *tiles*.

```
__shared__ float N_s[TILE_SIZE];
N_s[threadIdx.x]=N[i];
```

Then we have two variables, `this_tile_start_point` and `next_tile_start_point` which store the first thread id of the current block and the first thread id of the next

block respectively. These will be used later.
`int this_title_start_point = blockIdx.x*blockDim.x;`
`int next_tile_start_point = (blockIdx.x+1)*blockDim.x;`

The rest of the code is the same except for the conditional `if((N_index ⩾ this_title_start_point) && (N_index < next_tile_start_point))` which checks if the current index refers to an input element that is within the current block. If so then it can be retrieved efficiently from shared memory with
`Pvalue+=N_s[threadIdx.x+j-(MASK_WIDTH/2)]*M[j];`

However the problem with this approach is that elements on the boundaries will have neighbouring elements in other blocks and shared memory is only at the block level so it cannot be exploited for these elements. Therefore in these boundary cases we must read from global memory as before with
`Pvalue+=N[N_index]*M[j];`

However, the good news is that it is probable that these elements would have already been loaded to the L2 cache if they had been read into the shared memory of another block by the time this call is made. In this case it would actually have been read from the L2 cache rather than global memory which is faster.

Thank you for reading.

References