EECS498 Final Project

A. Introduction

In this project, we need to optimize convolutional neural network which can be seen as a 4D matrix multiplication. There are two layers which we need to optimizes. In order to give specific parameters for each of them, we first output their values.

	Layer 1	Layer 2
Batch Size (B)	10000	10000
Channel (C)	1	12
Output Feature (M)	12	24
Kernel Width(K)	7	7
Image Height (H)	72	33
Image Width (W)	72	33
H_out (H+K-1)	66	27
W_out (W+k-1)	66	27

We also need to output the original Op time of two layers.

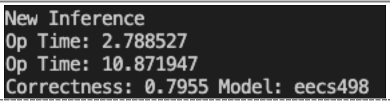
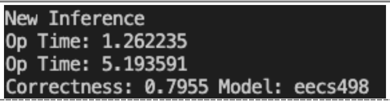
	Layer 1	Layer 2
Operation (Op) Time	2.78	10.87

B. Optimizations

1) Block size value

   <Motivation>

   In the original file, the block size is set to 512. To observe the difference of Op time between the block size I try to set it to 32. The following is op time of two layers.

	Op Time (block size = 512)	Op Time (block size = 32)
Layer 1	2.78	1.26
Layer 2	10.87	5.19
Picture		

```
New Inference
Op Time: 2.788527
Op Time: 10.871947
Correctness: 0.7955 Model: eecs498
```

```
New Inference
Op Time: 1.262235
Op Time: 5.193591
Correctness: 0.7955 Model: eecs498
```

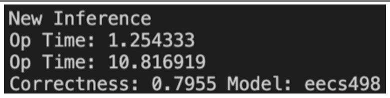
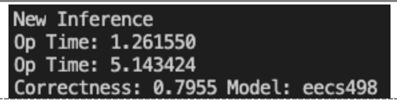
   <Result>

   With the block size 32, the time improve 54%/52% compare with the original one. The reason may be that we set the blockdim to 32 but there will be more grid(batch_size+31/32) work on the paralelism which will make the computation more efficiently.

2) Weight matrix in constant memory

   <Motivation>

   The characteristic of the constant memory is read-only type with less cycles comparing with the global memory. Since the weight matrix are use every time without change, we can save it to the constant memory so that it will not need to read from the global memory every time.

	Op Time (block size = 512) + Constant memory	Op Time (block size = 32) + Constant memory
Layer 1	1.25	1.26
Layer 2	10.81	5.14
Picture	 New Inference Op Time: 1.254333 Op Time: 10.816919 Correctness: 0.7955 Model: eecs498	 New Inference Op Time: 1.261550 Op Time: 5.143424 Correctness: 0.7955 Model: eecs498

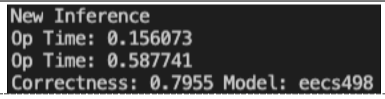
<Result>

With the use of the constant memory the result improves 55%/1% on block size 512 and 0.05%/0.09% on block size 32. It seems to not improve a lot on both layers and block size. The reason may be that in the original file the weight matrix is const int in the function parameter which makes a little different in the result with the constant memory one.

3) Parallelism in input image and Batch and Channel

<Motivation>

Since every element in the image is doing the same operation- multiplication, I come up with idea to do the parallelism in the image which can make the computation more efficiently. The ways I do on the host side are (1) set the blockDim to (tile_width, tile_width, 1) so that tile_width*tile_width numbers of thread can do together. The tile_size set to 16 on both layer. (2) set the gridDim to (B, M, (H_out/tile_width)*(W_out/tile_width)) so that every batch and channel can also do the same computation and thus reduce two for loops in the kernel side. On the kernel side (3) we can reduce three for loops because of parallelism and the output image width is calculated by blockIdx.z and thread.x and image height is calculated by blockIdx.z and thread.y.

	Op Time (last step)	Op Time (with parallelism)
Layer 1	1.26	0.15
Layer 2	5.14	0.58
Picture		 New Inference Op Time: 0.156073 Op Time: 0.587741 Correctness: 0.7955 Model: eecs498

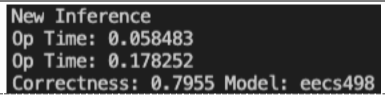
<Result>

The result improves 88%/88% compare with the last one and improve 94%/94% compare with the original one. The result improves a lot because it does parallelism on both image and batch and channel which make use of GPU characteristic of doing the same instruction together.

4) Shared Memory in input value

<Motivation>

As teacher said in the class "A profitable way of performing computation on the device is to tile the input data to take advantage of fast shared memory". Since each element in the input matrix will use many times, we can put it into shared memory from global memory and thus reuse it to reduce memory bandwidth. As the first step, we need to set up the shared memory width, since each output element will be the sum up of (input element + i)*(kernel+i) where i is from 0 to K-1. We set up the shared width with (tile_size+K-1). Secondly, we need to save the input data from global memory to the local One. Though there are 4 dimensions in the input array, there are actual 3 dimension doing the multiplication and the other one is the batch size. The shared memory is set to C*shared_width*shared_width. Because on the kernel side we set the blockDim(shared_width, shared_width), we need to be careful about the boundary condition when doing the multiplication. The boundary condition in the multiplication not only need to consider the image_width < W_out and image_height < H_out but also need to consider the threadIdx.x < tile_width and threadIdx.y < tile_width.

	Op Time (last step)	Op Time (with shared memory)
Layer 1	0.15	0.05
Layer 2	0.58	0.17
Picture		

```
New Inference
Op Time: 0.058483
Op Time: 0.178252
Correctness: 0.7955 Model: eecs498
```

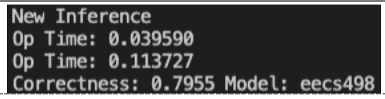
<Result>

The operation time improves 66%/70% compare with the last step. This shows that load the data from the local memory reduce a lot of time compare with load directly from the memory. Now the same bandwidth can support with more Flops.

5) Unroll matrix Multiplication

<Motivation>

In order to reduce the loop to optimize, we can use the unroll method to unroll the last 7 steps in the last for loop. Without unrolling, warps execute every iteration of the for loop. Instead of using the for (p = 0; p < K, ++p), the original multiplication line was separated into 7 lines. For example, for the first line it does the first element multiplication and the second line does the second element multiplication.

	Op Time (last step)	Op Time (with Unroll)
Layer 1	0.05	0.03
Layer 2	0.17	0.11
Picture		

```
New Inference
Op Time: 0.039590
Op Time: 0.113727
Correctness: 0.7955 Model: eecs498
```

<Result>

The result improves 40%/36% compare with the last step. The reason may be that unrolling can reduce the useless warps and thus make efficient use of memory bandwidth.

C. Conclusion

In this project, I did four optimizations including putting the weight matrix into the constant memory, parallelism on image, channel and batch, putting the input element into the shared memory and unroll the matrix. In these four methods, I think parallelism optimizes the process most because it makes efficient use of memory bandwidth and more threads and blocks can do the instruction at the same time.