Jose Moti
CE17B118
29/03/2020

CS6886: Systems Engineering for Deep Learning

REPORT

Assignment 2

Introduction

The common building blocks used in Deep Learning are implemented using AVX in this assignment. The whole alexnet operations are written with for loops without using any inbuilt functions. Also, different dataflows such as OS, WS, IS is tried out on different conv layers and times reported separately. Tiling is done for conv layers using AVX. Different memory declaration layouts were tried on different conv layers in order to find which one is the best for each dataflow and each conv layer.

PART-A

Machine Configurations

Processor	Intel Core i5 - 7200U
Cores	2
Threads	4
L1D Cache size	32K
L2 Cache size	256K
L3 cache size	3072K

AVX Intrinsics

(*The data type is changed from int32_t to float. This is done because most of the avx instructions are applicable only to float data type and not to int data type.)

__m256 _mm256_load_ps (float const * mem_addr) Load 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from memory. It need to be aligned on boundaries.
__m256 _mm256_loadu_ps (float const * mem_addr) Load 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from memory. It need not be aligned on boundaries.
__m256 _mm256_maskload_ps (float const * mem_addr, __m256i mask) Load 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from memory using mask. That is masked elements are zeroed out. For conv_2d.
__m256 _mm256_mul_ps (__m256 a, __m256 b)

Multiply two 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements). Output is of __m256 type. For conv_2d and linear.
__m256 _mm256_add_ps (__m256 a, __m256 b) Add two 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements). Output is of __m256 type. For conv_2d and linear.
__m256 _mm256_setzero_ps (void) Return vector of type __m256 with all elements set to zero. Used for conv_2d.
void _mm256_store_ps (float * mem_addr, __m256 a) Store 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from a into memory. Boundary alignment is required.
void _mm256_storeu_ps (float * mem_addr, __m256 a) Store 256-bits (composed of 8 packed single-precision (32-bit) floating-point elements) from a into memory. Boundary alignment not required.
__m256i _mm256_set_epi32 (int e7, int e6, int e5, int e4, int e3, int e2, int e1, int e0) Set packed 32-bit integers in dst with the supplied values. Used for conv_2d.
__m256 _mm256_max_ps (__m256 a, __m256 b) Returns a vector containing a maximum element from each point of the two input vectors. Used in maxpool layer.
__m256 _mm256_permute_ps (__m256 a, int imm8) Shuffles the vector in a combination as given by imm8. Used in maxpool layer.
__m256 _mm256_set1_ps (float a) Broadcasts the float value ‘a’ to the whole vector. Eight 32-bit float data type.

PART-B

Step1 - Correctness

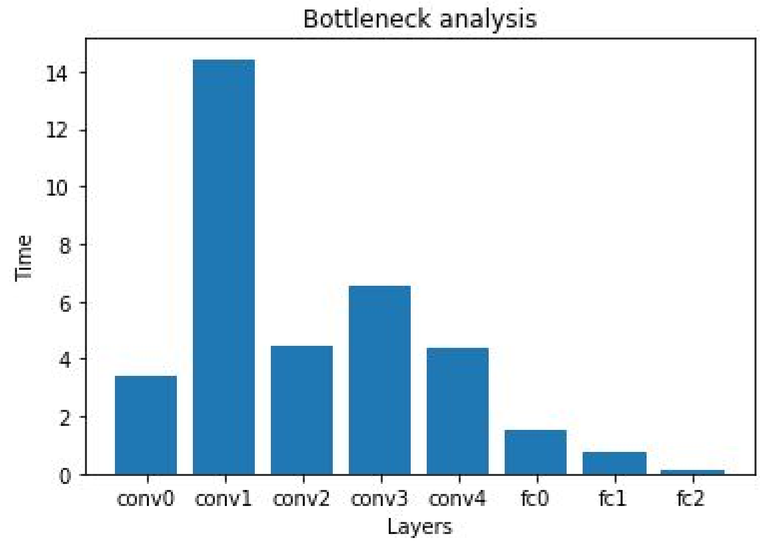
In this part, the giving function codes were completed. The default optimization flag was used for the whole assignment so that no optimization is used in the naive implementation.

1. For **2d convolution**, naive implementation was done because vectorizing/ unrolling the final loop of the code makes it similar to Output Stationary dataflow which we would be implementing in Step 3a. See conv_2d in util.cpp for further details. The steps done are as follows:
   - Create memory space for input, output, weights.
   - Padding is done without vector extensions.
   - Computation with 7 for loops.
   - Input memory is freed.
   - The output memory pointer is returned.
2. For **fully connected layer**, implemented using AVX. The final loop was vectorized in order to perform eight float operations simultaneously. The AVX instructions used are given below:
   - loadu_ps
   - storeu_ps
   - mul_ps
   - add_ps
3. For **2d maxpool operation**, it is implemented using avx instructions. The code snippet showing AVX instructions used is given below. Two consequent vectors are loaded and a max-min vector out of them is returned. The last element of the returned vector will be storing the max value. Using avx instructions the runtime for the maxpool layer was found to decrease by 5 times compared to maxpool naive implementation. See code in util.cpp. AVX instructions used are:
   - max_ps
   - permute_ps
   - loadu_ps
   - storeu_ps

4. For **relu,** no AVX was used. This is because even if AVX is to be used it involves the creation of a mask that itself takes almost the same time as the naive operation. It is of void type as it just changes the values in the memory of input and it is then passed for further operations.

Step2-Bottleneck analysis

The timings for each layer of the net with the blocks as described above are plotted. I am getting better time results while giving further optimizations while compiling the code. Here I am reporting the values that I got with default optimization flags. There was a mistake in the first conv layer.
(*The padding in both directions was set as 2, where it is 0 actually. It was changed before implementation.)



[3.385s ,14.43s ,4.461s ,6.502s ,4.343s ,1.499s ,0.743s ,0.15s]

Total time = 35.531s
The time taken is maximum for the second conv layer. The reason for this is explained below.

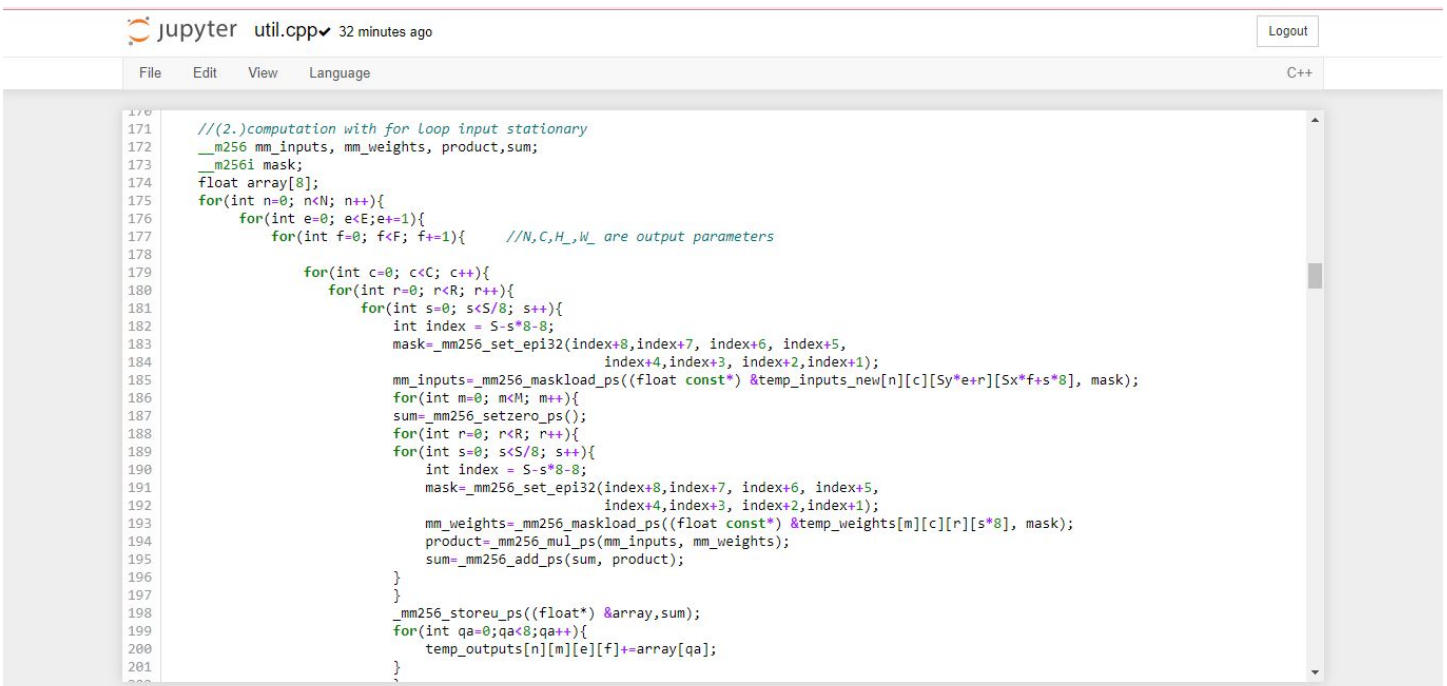
	Operations(MAC) M*C*R*S*N*E*F	**Bits** (MNEF+MCRS+NCHW)*32	**op/bits**
conv0	105M	15M	7
conv1	448M	28M	16
conv2	149M	32M	4.65
conv3	224M	46M	4.86
conv4	149M	32M	4.65
fc0	37M	38M	1.1
fc1	16M	17M	1.1
fc2	4M	4M	1

This table can explain the anomaly in the conv1 layer.

- op/bits value is higher relative to others.
- The bar plot shows proportionality between operations and time taken. This means there is enough bandwidth for memory and more time is taken for MAC operations.
- This shows the need for parallelism in the system so that the operations can be shared in the given space in order to make these less time-consuming.
- Hence computation bottleneck and not memory bottleneck. Thus the performance can be improved using AVX intrinsics.

Step 3a-Implementing different dataflows

Different dataflows naming Output stationery, Weight stationery, Input Stationery are implemented using AVX.
A screenshot of the computation loop for Input stationery dataflow. See conv2d_OS, conv2d_WS, conv2d_IS in util.cpp for full code.



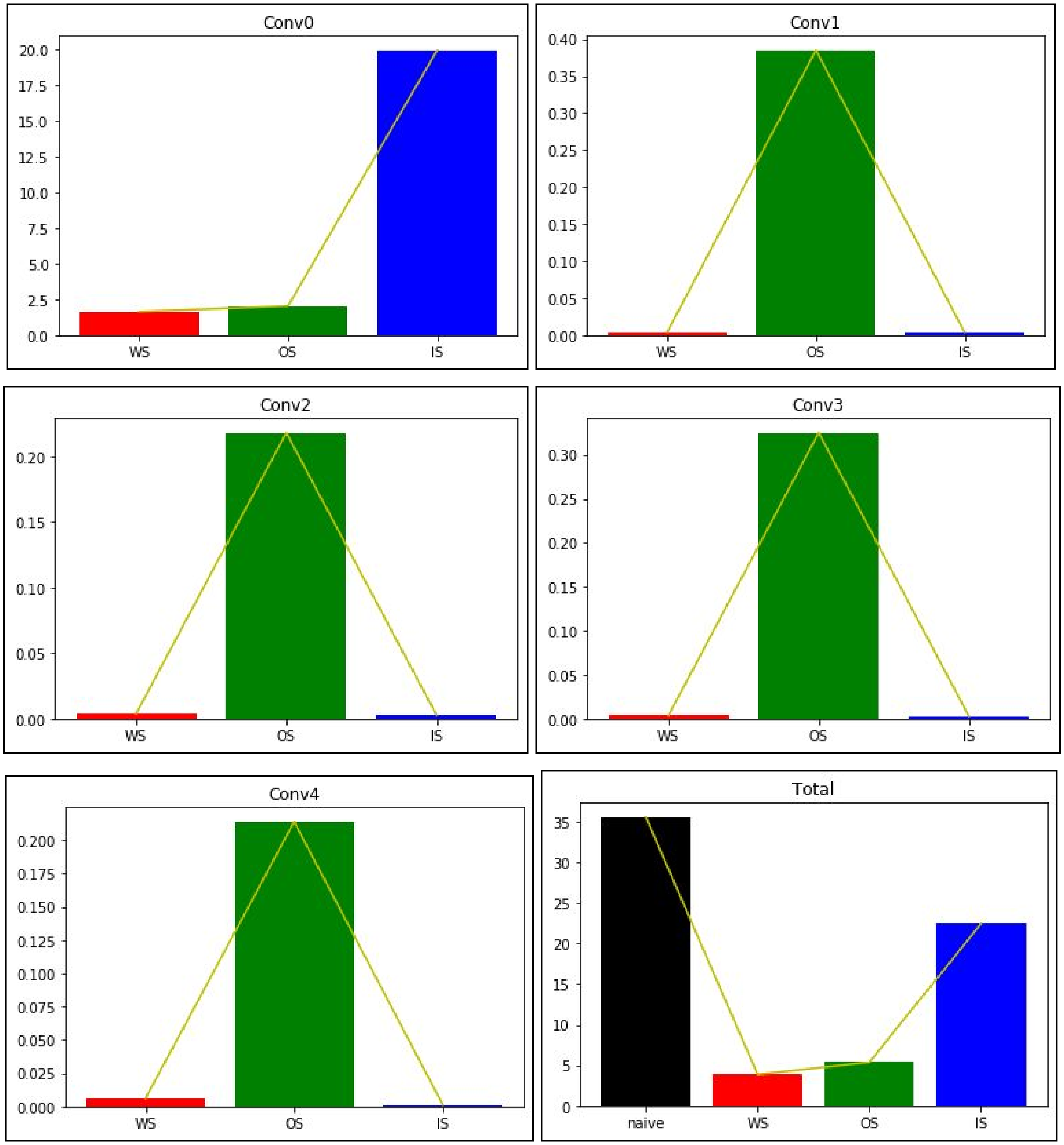

```cpp
    //(2.)computation with for loop input stationary
    __m256 mm_inputs, mm_weights, product,sum;
    __m256i mask;
    float array[8];
    for(int n=0; n<N; n++){
         for(int e=0; e<E;e+=1){
             for(int f=0; f<F; f+=1){     //N,C,H_,W_ are output parameters

                 for(int c=0; c<C; c++){
                    for(int r=0; r<R; r++){
                        for(int s=0; s<S/8; s++){
                            int index = S-s*8-8;
                            mask=_mm256_set_epi32(index+8,index+7, index+6, index+5,
                                                  index+4,index+3, index+2,index+1);
                            mm_inputs=_mm256_maskload_ps((float const*) &temp_inputs_new[n][c][Sy*e+r][Sx*f+s*8], mask);
                            for(int m=0; m<M; m++){
                            sum=_mm256_setzero_ps();
                            for(int r=0; r<R; r++){
                            for(int s=0; s<S/8; s++){
                                int index = S-s*8-8;
                                mask=_mm256_set_epi32(index+8,index+7, index+6, index+5,
                                                  index+4,index+3, index+2,index+1);
                                mm_weights=_mm256_maskload_ps((float const*) &temp_weights[m][c][r][s*8], mask);
                                product=_mm256_mul_ps(mm_inputs, mm_weights);
                                sum=_mm256_add_ps(sum, product);
                            }
                            }
                            _mm256_storeu_ps((float*) &array,sum);
                            for(int qa=0;qa<8;qa++){
                                temp_outputs[n][m][e][f]+=array[qa];
                            }
```

- In OS N, M, E, F are the outer loops that make the output array stationary inside this nested loop.
- In WS M, C, R, S are the outer loops that make the weight array stationary inside this nested loop.
- In IS N, C, E, F are the outer loops that make the input array stationary inside this nested loop.

Step 3b-Implementing different dataflows

The time plots for the 3 dataflow for the 5 conv layers are given below.



naive=[3.385,14.43,4.461,6.502,4.343,1.499,0.743,0.15] total=35.531
WS=[1.638,0.004,0.004,0.004,0.006,1.445,0.635,0.133] total=3.874
OS=[2.027,0.385,0.218,0.325,0.214,1.441,0.618,0.137] total=5.373
IS=[19.902,0.004,0.003,0.003,0.001,1.630,0.717,0.158] total=22.426

The best dataflow pattern for each layer is given below.

conv0	**WS**
conv1	**WS**
conv2	**IS**
conv3	**IS**
conv4	**IS**

So each conv layers are set to the corresponding dataflow in order to obtain the minimum time of 3.8 seconds. It is set as given below:

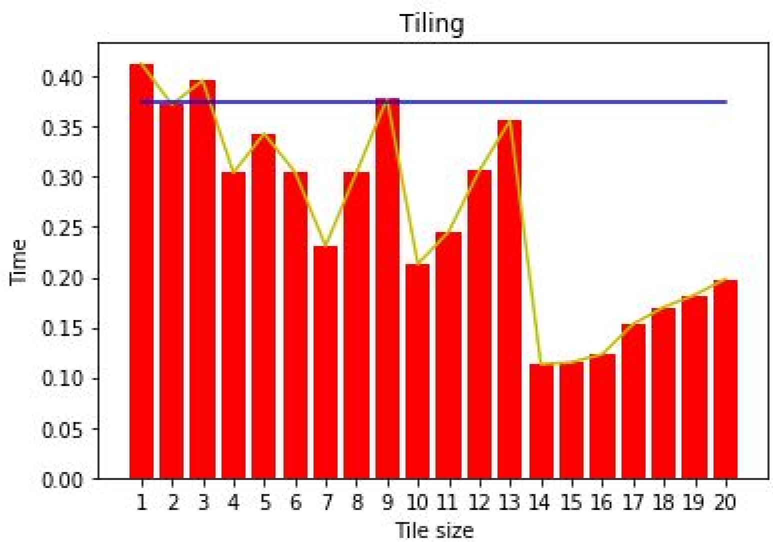
```
temp = conv_layers[0]->conv2d_WS(temp);  //conv_2d conv2d_optimized conv2d_WS
relu(temp);
temp = maxpool_2d(temp, 3, 3, 2, 2);
temp = conv_layers[1]->conv2d_WS(temp);
relu(temp);
temp = maxpool_2d(temp, 3, 3, 2, 2);
temp = conv_layers[2]->conv2d_IS(temp);
relu(temp);
temp = conv_layers[3]->conv2d_IS(temp);
relu(temp);
temp = conv_layers[4]->conv2d_IS(temp);
relu(temp);
temp = maxpool_2d(temp, 3, 3, 2, 2);
```

Inferences:

- We can see a high value for IS in the first conv layer but it decreases subsequently. This is because with conv layers the size of the input matrix decreases and because of that reuse works more efficiently than when there are larger inputs.
- WS gives the best time for almost all layers. This is because WS filters are usually small that once loaded these can compute across a large section of input. But as the filter size increases it leads to multiple vector declaration even inside the row of a filter.
- OS, on the other hand, is giving the worst result for 4 conv layers. This is because a large loading of inputs and weights takes place each time. And for this same small filters should be called from memory again and again and this leads to time loss. This does not happen in the case of WS and IS when their sizes are low.
- From this, we can understand that OS will not always remain as the best dataflow. It depends on the sizes of the weights and inputs. It mainly depends on which part can be reused most in order to decrease the memory loading cost.
- So in convolution operations, I think WS is the best dataflow pattern.

Step 4-Tiling

Tiling for the second conv layer is to be done. Here the padding is 2 and stride is 1 with a filter size of 5. So the output height is
E=(H-5+4+1)=H
Ie, it remains the same. So here both input and output are declared with E and E_tile is a tile to be given to conv2d_OS. For conv2d_optimized both the input feature map and tile size are given as input.
The code for tiling is given in conv2d_optimized. My friend Sooryakiran P helped me with the coding part of tiling.
The implementation time for the conv1 layer without tiling is 0.375. The time taken for one forward pass in this layer with different tile sizes is given in the plot below.



tile_time=[0.413,0.372,0.396,0.304,0.343,0.305,0.231,0.304,0.378,0.213,0.245,0.306,0.356,0.113,0.115,0.123,0.153,0.170,0.182,0.198]

Inferences:

- Using tiles we can decrease the time taken for the second conv layer.
- Low values of tiles require higher computation times compared to that of higher tile values.
- Minimum times are obtained for tile sizes of 14,15,16 which is less than half of un-tiled time.
- This shows there is significantly higher reusability as the tile size approaches 16 and also 8 since AVX vectors take multiples of 8 values at a time.
- A tile size of 15 can be used for the best output and saving a lot of computational time.

PART-C

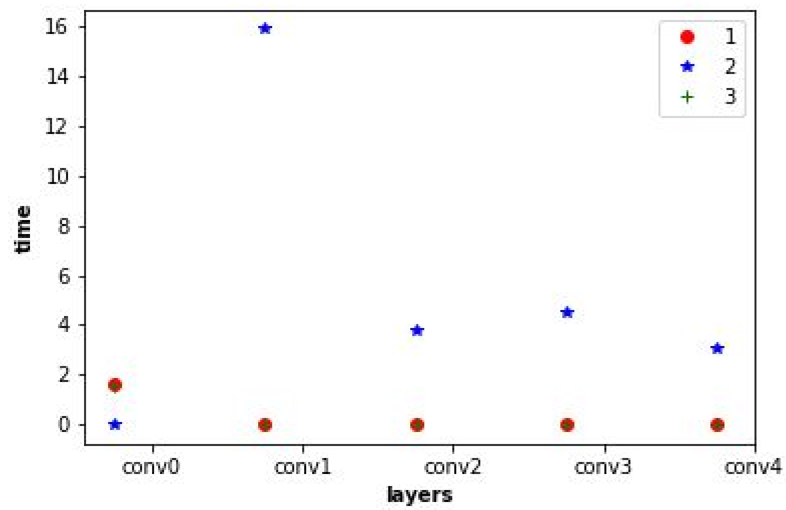
Memory Layout

Weight Stationary

Either of C, H, W loop can be vectorized. N and M cannot come in the end.

1. 1Input-(N,C,H,W), Output-(N,M,E,F), Weights-(M,C,R,S) → WS1
2. Input-(N,H,W,C), Output-(N,M,E,F), Weights-(M,R,S,C) → WS2
3. Input-(C,N,H,W), Output-(N,M,E,F), Weights-(C,M,R,S) → WS3

The plots of time for these 3 memory layout are given as plot below.



y_ws1=[1.638,0.004,0.004,0.004,0.006] #3.874
y_ws2=[0.004,15.905,3.802,4.55,3.039] #29.433
y_ws3=[1.581,0.004,0.004,0.004,0.004] #3.697

Inferences:

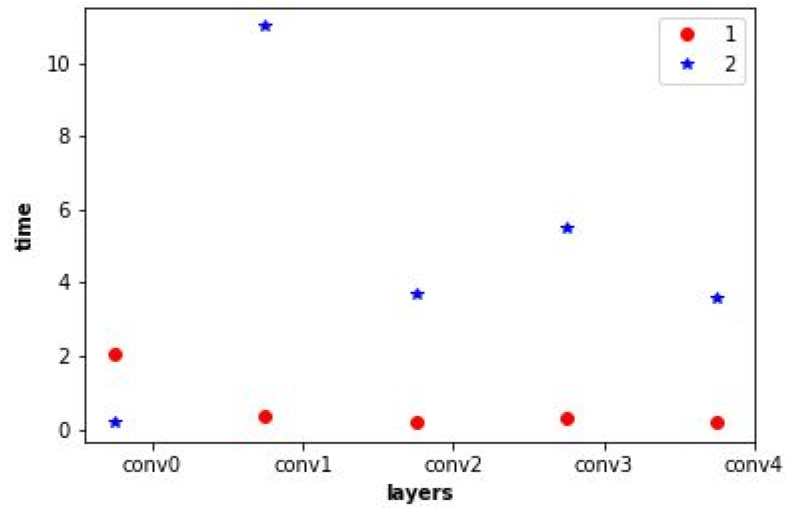
- 1 and 3 are same because both C and M acts as outer loop in this case.
- In 2 C loop is vectorized and in 1 and 3 S loop is vectorized. In conv0 C was 3 while S was 11 and due to this 2 had better results. Only one round of vectorization was required in C while it was two for R and S. Thus 2 worked best on conv0.
- In the remaining layers the depth of the filters was much higher than height and width and because of this 2 had the worst results.

Output Stationary

Either of C, H, W loop can be vectorized. N and M cannot come in the end.

1. Input-(N,C,H,W), Output-(N,M,E,F), Weights-(M,C,R,S) → OS1
2. Input-(N,H,W,C), Output-(N,M,E,F), Weights-(M,R,S,C) → OS2
3. Input-(C,N,H,W), Output-(N,M,E,F), Weights-(C,M,R,S)→ same as OS1
4. Input-(C,N,W,H), Output-(N,M,E,F), Weights-(C,M,S,R)→ same as OS1

Any other changes do not affect the computation times as they are part of outer loops.
The plot is given below:



y_os1=[2.027,0.385,0.218,0.325,0.214] #5.373
y_os2=[0.188,10.991,3.678,5.482,3.599] #26.064

Inferences:

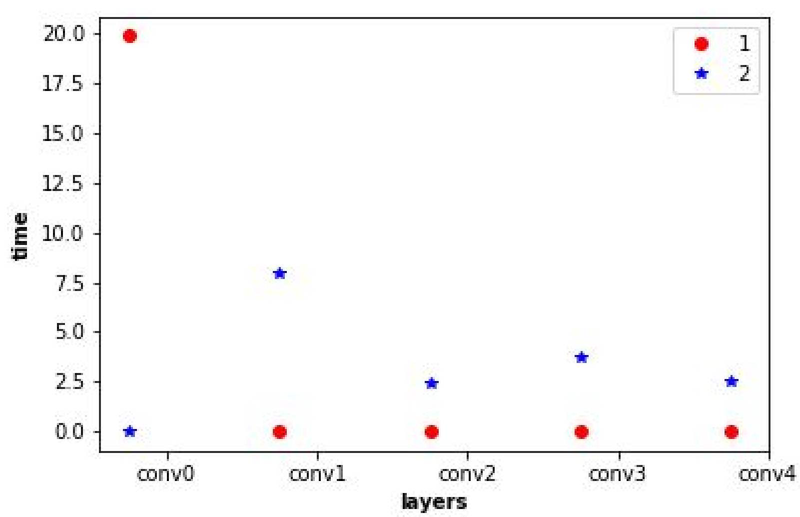
- Same as WS. Memory layout 2 is best for conv0 while 1 is better for the remaining conv layers. Reason- C is smaller tha S in conv0 and not for remaining.

Input Stationary

Either of C, H, W loop can be vectorized. N and M cannot come in the end.

1. Input-(N,C,H,W), Output-(N,M,E,F), Weights-(M,C,R,S) → IS1
2. Input-(N,H,W,C), Output-(N,M,E,F), Weights-(M,R,S,C) → IS2

Any other changes do not affect the computation times as they are part of outer loops.
The plot is given below:



y_is1= [19.902,0.004,0.003,0.003,0.001] #22.426
y_is2= [0.004,7.936,2.459,3.737,2.480] #18.734

Inferences:

- Same results as that of OS and WS. Using 2 it reads the input depthwise rather than width wise. Since depth in conv0 is 3 and width is 227 2 gives lesser computation time than 1.
- For all other conv layers depth comes out to be larger than input size leading to more computation time.

Conclusion

Did different types of dataflow and memory allocation techniques on different layers of Alexnet. Different layers are found to behave differently because the input parameters of the convolution layers differ. With the different parameters, we should be able to predict the right dataflow or memory declarations as the time losses are very significant.

References

- AVX Intrinsics
- Stackoverflow
- AVX Tutorial Link
- Sooryakiran P ME17B174 helped me with the tiling code.

Codes

- Part B in util.cpp
- Part C in mem_layout/util.cpp
- Diagrams in plot drawn with jupyter notebook plot.ipynb
- g++ alexnet.cpp util.cpp -o test -mavx   //no extra optimiser used in this assignment.
- From boilerplate code 1 additional function create_fmap in util.cpp.

THANK YOU