CS6886: Systems Engineering for Deep Learning

Assignment 2

Submitted by: Sooryakiran P (ME17B174)

Part A

Step 1

Machine configuration: Output of *lscpu*

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             996.184
CPU max MHz:         3400.0000
CPU min MHz:         400.0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7
Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d
```

Step 2

The following are the AVX instructions used for the assignment.

Sl. No	Instruction	Description	Use case
1	_mm256_loadu_ps(...)	Loads 8 single precision floats to the 256bit AVX registers.	Used to transfer all data to AVX registers. This was used in almost all functions to transfer the data to the registers for vector operations.
2	_mm256_mul_ps(...)	Multiplies two single precision floating point vectors element wise.	This function was used for the multiply part in all the MAC operations in *linear_optimized* and *conv2d* functions.
3	_mm256_add_ps(...)	Adds two single precision floating point vectors element wise.	This function was used for accumulating the partial sums in MAC operations in *linear_optimized* and *conv2d* functions.
4	_mm256_storeu_ps(...)	Transfers the data from 256 AVX registers back to the specified memory location.	This function was used to read the output of all AVX computation used. This was used in almost all functions implemented.
5	_mm256_maskload_ps(...)	Loads single precision floats to the 256bit AVX registers, according to the specified mask.	This is used only if the input array size is less than 8 floats. This was used at the end when the array sizes was not divisible by 8 in *linear_optimized* function. This was also used for all *conv2d* functions because the convolution kernels sizes are usually smaller than 8 floats along a dimension.
6	_mm256_set_epi32(...)	Sets the given array as integer values in the 256 bit registers.	This was used to set the mask before using *__mm256_maskload_ps* function.
7	_mm256_maskstore_ps(...)	Transfers the data from 256 AVX registers back to the specified memory location, according to the mask specified.	This was used to get back the data after computation wherever the inputs were loaded using the mask.

Part B

The datatype of DATA was changed from int_32t to float.

Step 1, 3a

Implementation details of all the functions are as given below.

Linear Layer

This is a naive implementation for the fully connected layers using nested loops. AVX instructions were not used for this function. The algorithm is as given below.

```
for each input in batch:
   for each output_element in output_vector:
      output_element = 0
      for each input_element in input_vector:
         output_element += input_element * weight(input_element,
                           output_element)
```

Linear Layer Optimized

This is an optimized version of the fully connected layers. The last loop is vectorized using AVX instruction. The algorithm is as given below

```
for each input in batch:
   for each output_element in output_vector:
      partial_sum[8] = {0, . . ., 0} // __mm_256 partial sum

      for each group of 8 input_element in input_vector:
         load all data into 256 bit AVX registers
         partial_sum += input_element * weight(input_element,
                        output_element)

      for the remaining group of n<8 input elements:
         create a mask and load into 256 bit AVX registers.
         partial_sum += input_element * weight(input_element,
                        output_element)

      output_element = sum(partial_sum)
```

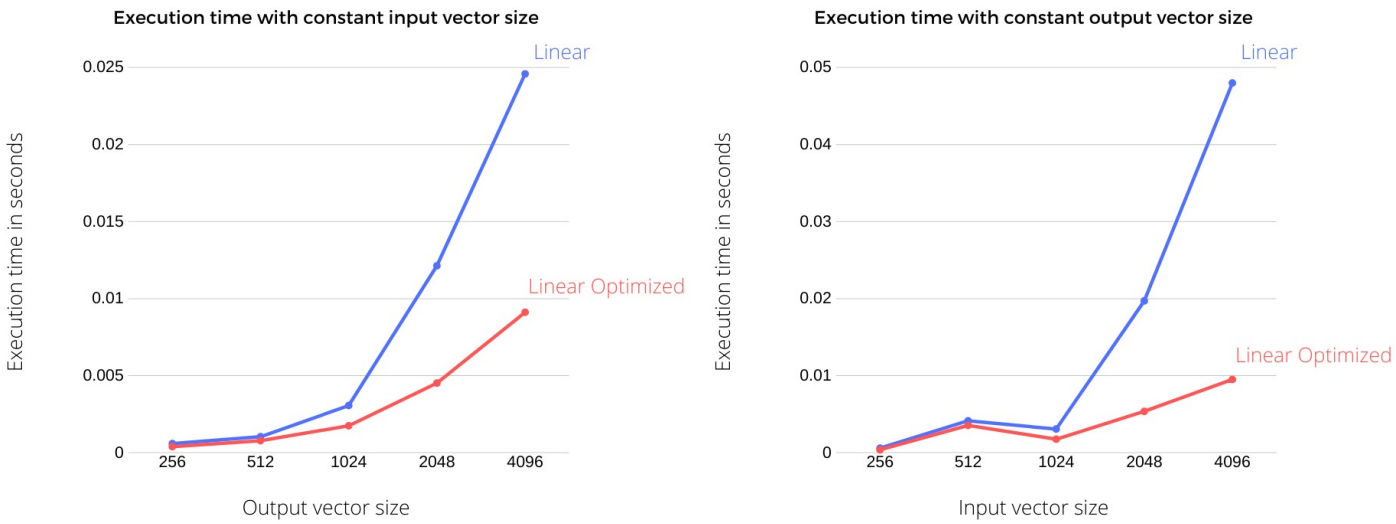


Fig 1. Execution time for various input output combinations for linear and linear optimized with AVX.

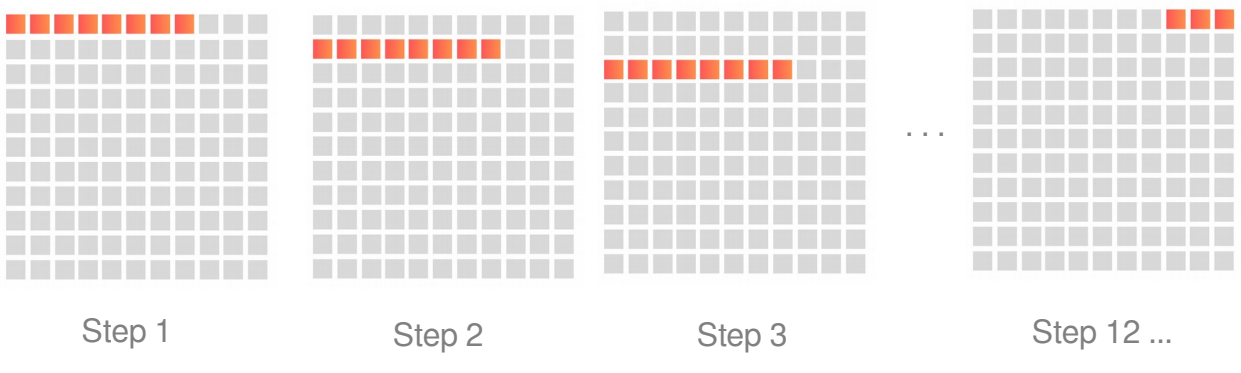
We can see that the optimized version of the function scales better.

Conv2D layer

This is a naive implementation for the 2D convolution operation. This consists of all the 7 loops that are not parallelized using AVX. This will serve as a baseline to compare other implementations.

Conv2D Optimized Output Stationary.

AVX instructions were used to implement an output stationary 2D convolution. Masked load is used for loading the weights and inputs to AVX registers because Conv filter sizes along a dimension is usually less than 8. A graphical view of the algorithm is shown below for a 11x11 convolution.



Partial sums are accumulated for iterations along the Y direction because they share a common mask i.e All steps from 1 to 11 share a common mask. Steps 11n to 11(n+1) share a common mask. So once 11 steps are done, the partial sum vector is flushed and added, and the row shifts to the right.

Conv2D Optimized Weight Stationary.

In this implementation, weights are first loaded to an AVX register. Then all the inputs that the weights can act upon are loaded and the output is updated. The algorithm is as given below.

```
Initialize outputs as zeros.
for each filter in filters:
   for each channel in channels:
      load weights as vectors of float[8] into AVX registers
      for each vector[8] in weight:
         for all input patches:
            compute dot product of input patch & weight vector and add
            them to corresponding output index.
```

Conv2D Optimized Input Stationary.

In this case, parts of inputs are first stored in the AVX registers. Then weights are loaded accordingly and the computed output is stored.

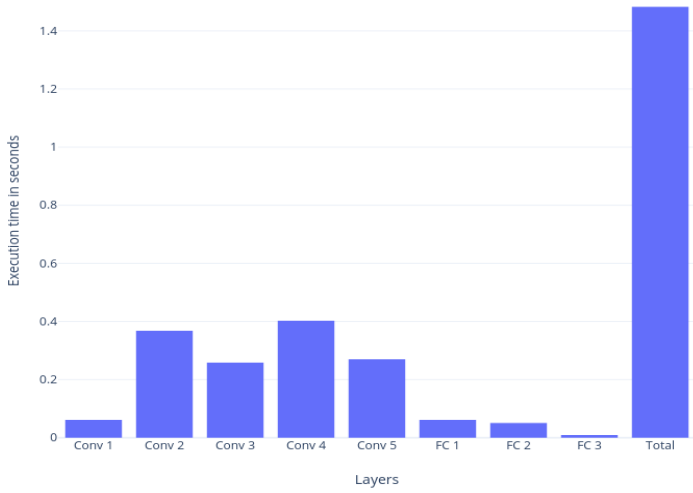
```
Initialize outputs as zeros.
for each input in batches:
   for each channel in channels:
      load each input patches as vectors of float[8] into AVX registers.
      for each vector[8] in input:
         for each filter in filters:
            compute dot product of input patch & weight vector and add
            them to corresponding output index.
```

ReLU, MaxPool and Padding2D are also implemented.

Step 2

Bottleneck analysis.

Each layer of the given Alexnet implementation was timed. The network was run 100 times and the average execution time was calculated. The execution time for each layer is as given below. These timings are for the optimized version of the layers. (All times in s)



Conv 1	Conv 2	Conv 3	Conv 4	Conv 5	FC 1	FC 2	FC 3	Total
0.0611	0.3682	0.2581	0.4028	0.057369	0.2702	0.0504	0.0092	1.4830

Step 3a

Different data flow patterns, i.e. Output stationary, Input Stationary and Weight stationary are implemented using AVX instructions. The details of their implementations are given in section 1. Their performance comparison is given in the succeeding section.

Step 3b

Different convolutional layers are timed for different data-flow patterns independently. The results are as follows. (All times in seconds)

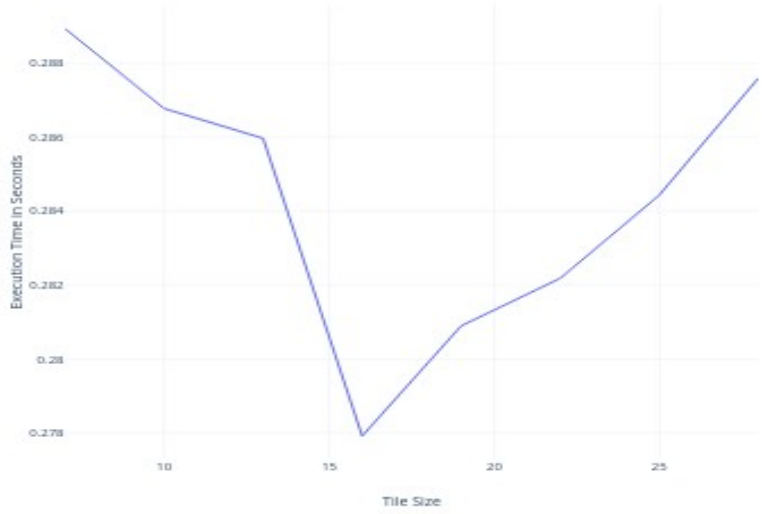
We can see that the implementation of output stationary performs faster than all other implementations.

Step 4

2D Tiling was implemented to speed up convolutions in the second convolutional layer of Alexnet. Different configurations of tile sizes are tried and the results are as given below. There is no large difference between tile sizes as much of the time is lost in allocating the data in tiles are reading output than actual computation. Also actual timings depend on the workloads that are already running on the PC.

The 2nd layer had an input size of 55 x 55 with 96 channels. The filter size along H and W was 5 x 5 with padding 2 on both axes. Tile sizes in input space were lower bound by 5 and upper bound by 55.

The execution times vs tile_size (in output space) are given below:



We can see that there is a perceivable dip in execution time when the output tile size is 16, which correspond to nearly 3 x 3 tile configuration in the input space.

Conclusion

Common functions used in Deep Learning like 2d convolution, fully connected layers, 2d max pooling and activation functions like ReLU was implemented. Further more, convolutional and fully connected layers were optimized using AVX instructions. Alexnet model which was supplied along with the boilerplate code was used to test the inference capabilities of the developed layers.