ELEC 4511/5511
Midterm Exam

Name: **Charles Bollig**

Instructions:
- Build a Python and C model of a Deep Neural Network architecture
- An example of 1-hidden layer 1-output layer has been provided
- You are not allowed to work with any other student for the duration of this assignment

Always show your work.

Problem Category	Max Points	Earned Points
Python Implementation of Multilayer Solution	25	
C Code Implementation of Multilayer Solution	40	
Optimization of C Code Implementation using SSE instruction set	25	
Presentation/Organization of Results	10	
Graduate students: Execution on Raspberry PI	10	
Graduate students: Use of NEON instruction set	10	

Overview

[Task Python]
Multilayer implementation. Provided a single hidden layer code that uses random numbers for biases and weights. Must be generalized to multi-stages.

Must take in a network description example: **example_Input2_Hidden_2x3_Output_4.txt**

[Task C_Implementation]
Multilayer implementation. Provided a single hidden layer code that uses random numbers for biases and weights. Must be generalized to multi-stages.

[Task C_Optimization]
Develop an SSE-bases solution that accelerates part of the weight * activation input for each layer. A good starting point would be to understand the NVIDIA TensorCore. YOU

[Presentation/Organization]
Clearly show your results

****Instructions to run code:**

C file needs to be compiled before running.

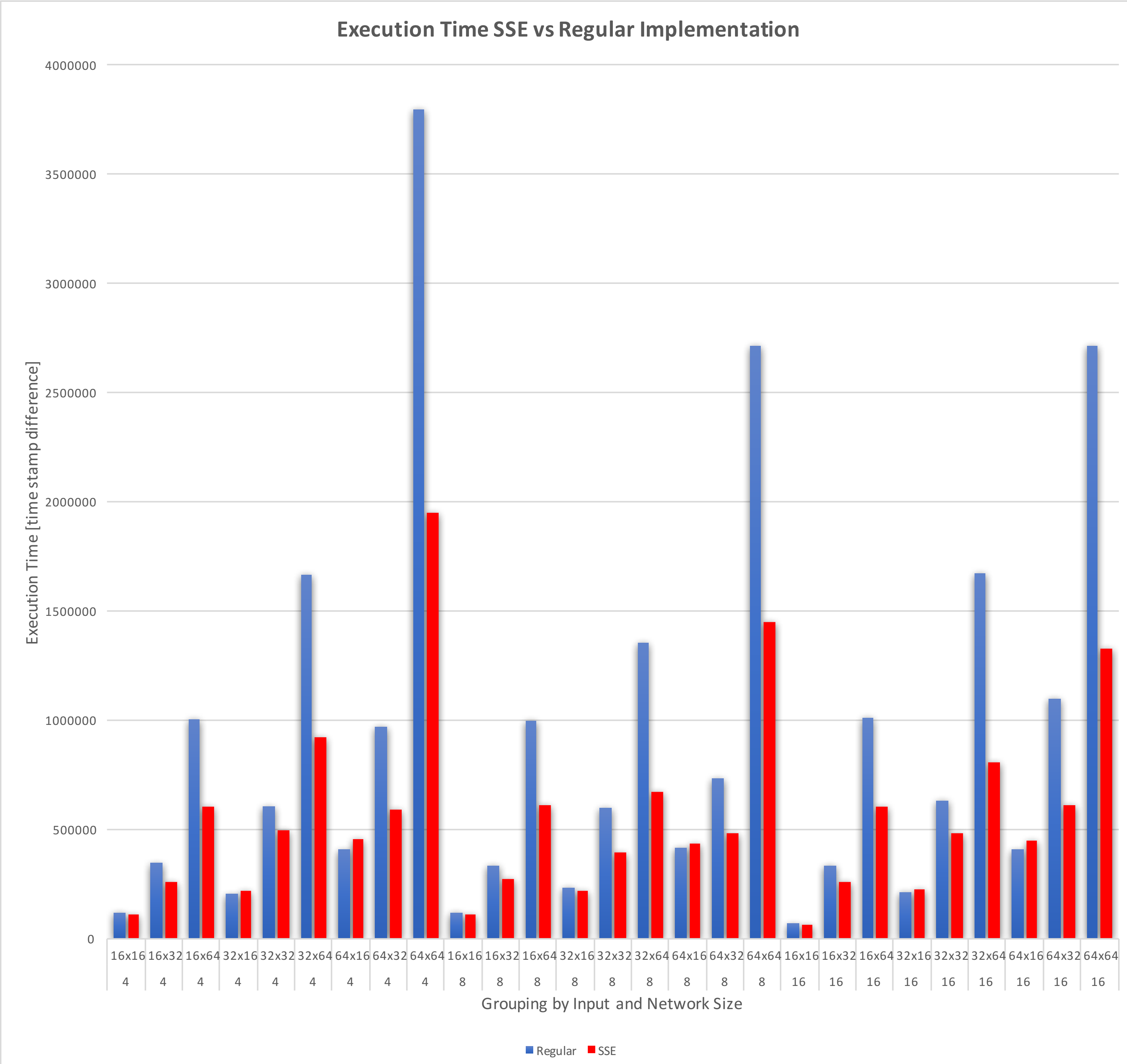
To compile code, enter:

```
gcc DNN.c -lm -o DNN_out -msse3
```

Then, enter:

```
Python runExp.py
```

The results of the C implementation and optimization with SSE are shown below. The x-axis is organized according to Network (hidden layers x hidden nodes) and input nodes (4, 8, or 16).



The results show that there is significant average speed increase when using SSE intrinsics on the x86 architecture, as opposed to using traditional coding methods. The general trend for execution time was for execution time to increase as the number of hidden nodes increased.

I observed that the number of inputs has surprisingly small effect on the overall execution time of the network. After all, the only difference in execution time made by the input is the number of multiplications in the very first layer. After that, execution time is more strongly determined by the number of hidden layers and hidden nodes.

For the most part, SSE implementation greatly lowered the total execution time for the network. The effect was not as pronounced until larger networks were utilized. For example, there is a negligible difference in execution time of the 8 input 16x16 network [regular: 112492] and [SSE: 108718]. However, for a large network: 8 input 64x64 network [regular: 2711488] and [SSE: 1448736], the difference is immense – almost a 2x speed up!

In addition, there was one situation across all three input levels where the SSE time took longer than the regular time: 64x64 network.

In the running of this code, there was one anomaly in the execution time that did occur. If we look at the plot above, we can see that the input size has very little effect on the overall execution time. However, when the input was 4, the execution time of the largest network (64x64 network), was much greater than the execution of the networks with greater inputs. I attribute this to the likelihood that there were other processes running on the server at that time. Many other students are running their simulations.

To speed up this execution, I used the intrinsic data type __m128, which is four aligned floats. With this data type, I separated input and weights into 4 float partitions. Then, I multiplied these together. So, instead of traversing the loops for the number of times that there are numbers of input nodes, I only traversed time/4 of that and used intrinsic multipliers to speed up the process.

Data:

Input	Network Size	Output	Regular Time	SSE Time
4	16x16	4	118427	109890
4	16x32	4	342177	258533
4	16x64	4	1001353	600504
4	32x16	4	205707	217892
4	32x32	4	604200	496782
4	32x64	4	1663964	916868
4	64x16	4	406163	451079
4	64x32	4	965082	588795
4	64x64	4	3794224	1951134
8	16x16	4	112492	108718
8	16x32	4	334235	268476
8	16x64	4	995795	609809
8	32x16	4	230147	217423
8	32x32	4	596514	394440
8	32x64	4	1349973	669606
8	64x16	4	413421	435948
8	64x32	4	734011	480892
8	64x64	4	2711488	1448736
16	16x16	4	70507	63016
16	16x32	4	330467	258274
16	16x64	4	1011304	601963
16	32x16	4	212166	220653
16	32x32	4	630491	482873
16	32x64	4	1673608	805072
16	64x16	4	404586	449369
16	64x32	4	1098074	612403
16	64x64	4	2712808	1327900

[Graduate students]

[Task: Raspberry Pi C Code Implementation]
[Task: Raspberry Pi Optimization]

****Instructions to run code:**

C file needs to be compiled before running.

To compile code, enter:

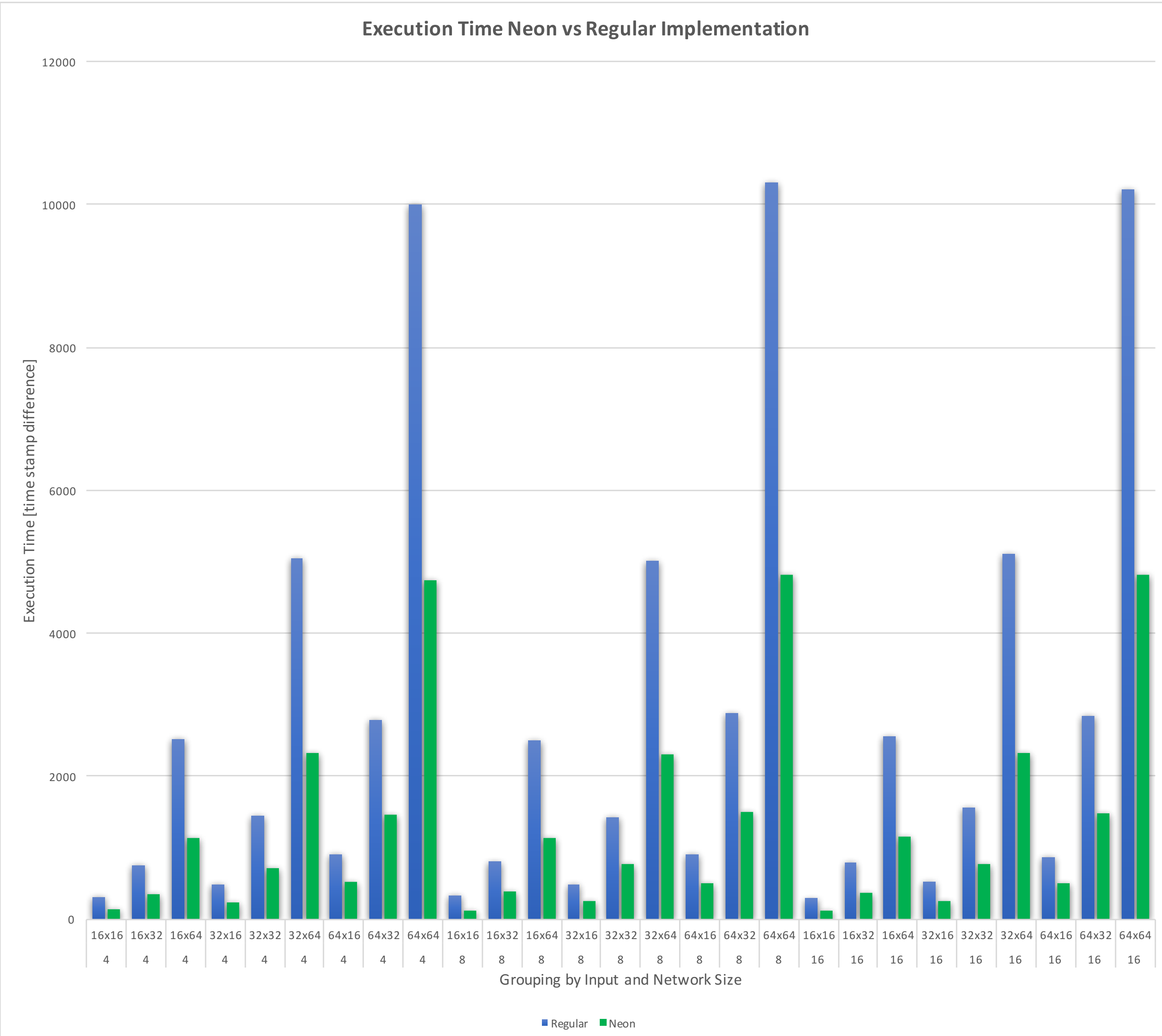
```
gcc DNN_pi.c -lm -o DNN_pi -mfpu=neon
```

Then, enter:

```
Python runExp.py
```

The results of the C implementation and optimization with Neon are shown below. The x-axis is organized according to Network (hidden layers x hidden nodes) and input nodes (4, 8, or 16).

The timing function was used was to take the difference obtained between start and stop with the time.h clock() function.



The results show that there is significant average speed increase when using Neon intrinsics on the ARM architecture, as opposed to using traditional coding methods. The general trend for execution time was for execution time to increase as the number of hidden nodes increased.

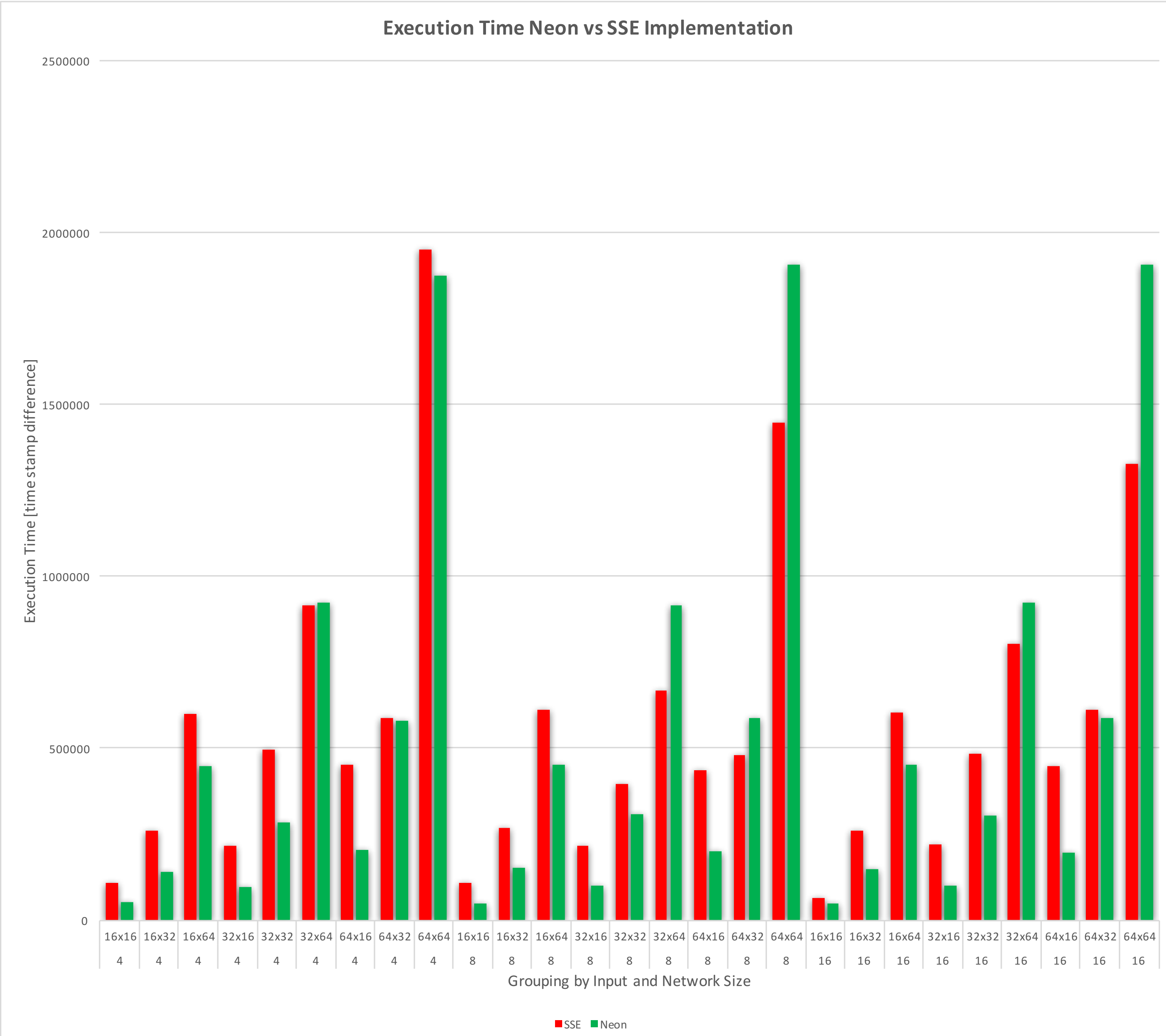
Very clearly, the Neon intrinstics greatly speed up the execution time of a neural network on a RaspberryPi device. There are no anomalies. In most cases, there is a greater than 2x speed increase when measuring execution time. Again, the larger networks benefit more from the usage of the intrinsics.

Much like the SSE implementation, I used the intrinsic data type float32x4_t, which is four aligned floats. With this data type, I separated input and weights into 4 float partitions. Then, I multiplied these together. So, instead of traversing the loops for the number of times that there are numbers of input nodes, I only traversed time/4 of that and used intrinsic multipliers to speed up the process.

Data:

Input	Network Size	Output	Regular Time	Neon Time
4	16x16	4	299	130
4	16x32	4	743	354
4	16x64	4	2523	1134
4	32x16	4	488	239
4	32x32	4	1431	718
4	32x64	4	5046	2328
4	64x16	4	900	519
4	64x32	4	2781	1464
4	64x64	4	10001	4733
8	16x16	4	328	124
8	16x32	4	801	387
8	16x64	4	2499	1137
8	32x16	4	485	256
8	32x32	4	1419	777
8	32x64	4	5017	2312
8	64x16	4	899	504
8	64x32	4	2882	1489
8	64x64	4	10296	4817
16	16x16	4	292	126
16	16x32	4	795	370
16	16x64	4	2551	1146
16	32x16	4	528	251
16	32x32	4	1557	771
16	32x64	4	5109	2332
16	64x16	4	863	493
16	64x32	4	2848	1481
16	64x64	4	10209	4813

Because of different timing functions in the two architectures, I couldn't use the same timing function. However, since the regular execution time was the same in each, I found the multiplicative factor that would be needed to correct for the time difference (SSE_time/Neon_time) = 396.07. Multiplying the Neon values by this number, I was directly able to compare Neon vs SSE. The results are below, same schema:



For the most part, Neon does a much better job on the smaller networks. Whereas, SSE performs equally or better job on the larger networks (except for the strange anomaly mentioned before).

Theoretically, if each node could be executed (weight multiplication + bias) in one clock cycle, we would see an increase in speed of 4x + some constant values using the intrinsics available. Of course, this is not quite possible.

Data:

Input	Network Size	Output	SSE Time	Neon Time
4	16x16	4	109890	51489.1
4	16x32	4	258533	140208.78
4	16x64	4	600504	449143.38
4	32x16	4	217892	94660.73
4	32x32	4	496782	284378.26
4	32x64	4	916868	922050.96
4	64x16	4	451079	205560.33
4	64x32	4	588795	579846.48
4	64x64	4	1951134	1874599.31
8	16x16	4	108718	49112.68
8	16x32	4	268476	153279.09
8	16x64	4	609809	450331.59
8	32x16	4	217423	101393.92
8	32x32	4	394440	307746.39
8	32x64	4	669606	915713.84
8	64x16	4	435948	199619.28
8	64x32	4	480892	589748.23
8	64x64	4	1448736	1907869.19
16	16x16	4	63016	49904.82
16	16x32	4	258274	146545.9
16	16x64	4	601963	453896.22
16	32x16	4	220653	99413.57
16	32x32	4	482873	305369.97
16	32x64	4	805072	923635.24
16	64x16	4	449369	195262.51
16	64x32	4	612403	586579.67
16	64x64	4	1327900	1906284.91