COL730 Assignment 1 - CUDA Implementation of FANN

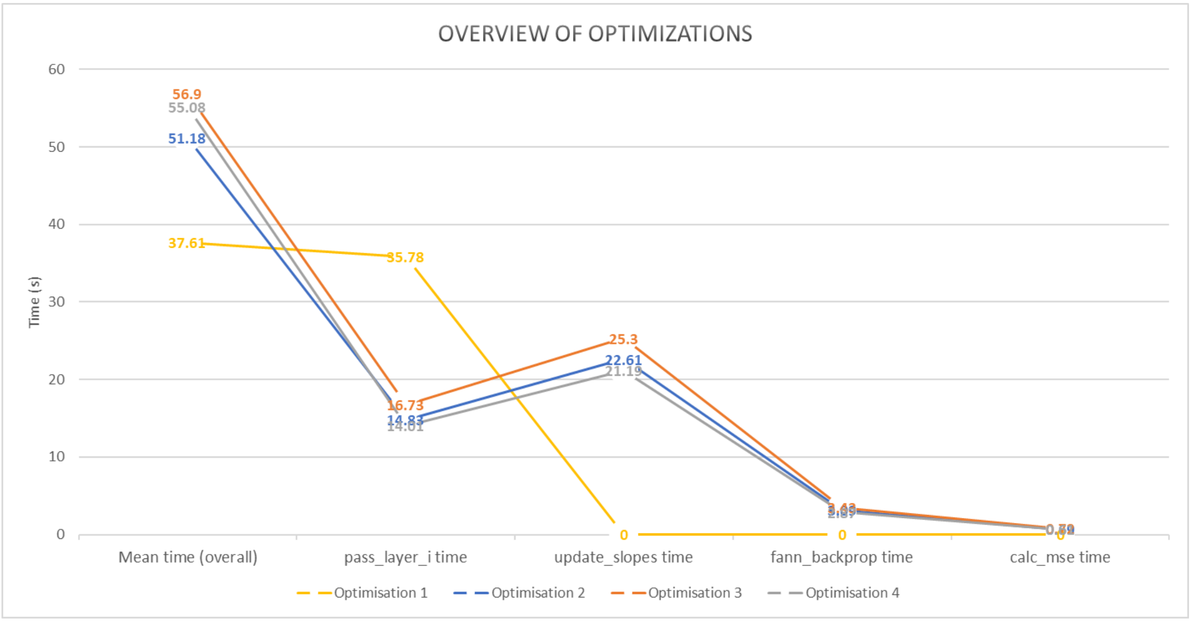
~ Samarth Bhatia, 2019CH10124

The FANN code provided to us contains a feed forward ANN implementation. We can convert the four main functions as mentioned in the problem statement (feed forward run, MSE calc, backprop, weight update) to kernels running on the GPU to parallelize the computation. However the scale of the datasets and the neural networks defined in the examples is so small that the CUDA overhead adds up and makes the CUDA code slower than the CPU code. However, we clearly see that CUDA is essential for deep learning when the networks are large and so are the datasets and libraries like PyTorch make extensive use of this to gain 2x-10x performance improvements than CPU code.

For a sample comparison, a small real-world image dataset (Fashion-MNIST) has 60,000 training + 10,000 testing samples of 28*28 images,spanning across 10 classes. This means that for a very simple 3 layer NN with 512 hidden neurons, in `fann` terms, `num_input = 784`, `num_hidden_neurons = 512` and `num_output = 10`, effectively making `num_total_connections = 406,528`.
In contrast, the largest example in `fann` (on the **mushroom** dataset) has `num_input = 125`, `num_hidden_neurons = 32` and `num_output = 3`, with `num_total_connections = 4098`. That is a difference of `100x`, and if we were to run our CUDA implementation on real-word examples like that, we would be able to see the improvements *more significantly*.

I started with simply parallelizing the given functions into kernels as a starting/naïve implementation (this involves a lot of `cudaMemcpy`s), and built up from there with several optimizations using methods taught in class ranging from scaling number of threads, reducing number of copies by performing more computation on the device, memory coalescing, optimizing the multiplications etc.

Graph comparing all optimizations (brief) and



Optimizations done:

1. Parallelize only the `fann_run` function into a kernel.
2. Parallelize all the functions into kernels.
3. Add scaling (increase number of threads).
4. Use **memory coalescing**.
5. Convert the multiplication of last layer's neuron values and connection weights to a parallel vector multiplication.

These are the kernels that I have defined to take place of the CPU functions:

1. `pass_layer_i` - replaces the `fann_run` forward pass function. This iterates over the layers and is parallelized over neurons in that layer. The vector-vector elementwise multiplication can also be replaced with another kernel.
2. `calc_mse` - replaces `fann_compute_MSE` and is over the final layer (so is parallelized over neurons in the last layer), and calculates the mean squared error loss with the given desired output.
3. `fann_backprop_layer` (and `_complete`) - these two kernels replace the `fann_backpropagate_MSE` function. Iterating over layers, these are again parallelized on neurons and calculate the error of the previous layer (propagated through connections) and the gradient of the `sigmoid` activation.
4. `update_slopes` - replaces the `fann_update_slopes_batch` function which goes over the layers and updates the neuron 'slopes', is parallel over neurons in a layer.
5. `update_weights_parallel` - replaces the `fann_update_weights_irpropm` function (called at the end of an epoch) to update all the weights based on the slopes and the previous slopes and steps. This is parallel over the total connections in the ANN.

and some small helper kernels/ `__device__` functions.

Results

1. `mushroom` dataset

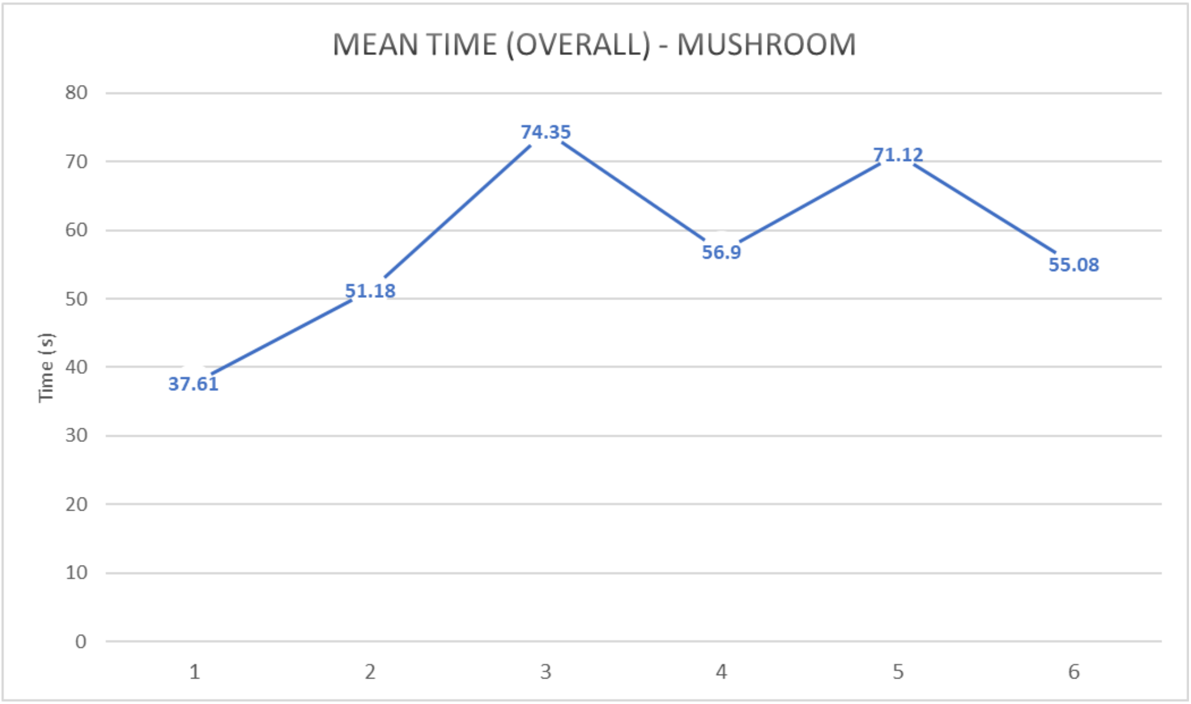
Parameters

```
num_data = 4062;
num_input = 125;
num_output = 2;
num_layers = 3;
num_neurons_hidden = 32;
desired_error = 0.0001;
max_epochs = 30;
```

Name of kernel/function	1	2	3	4
`update_slopes`	-	22.61 s (52.15%)	25.30 s (52.45%)	21.19 s (52.60%)
`pass_layer_i`	35.78 s (97.58%)	14.83 s (34.22%)	16.73 s (34.68%)	14.01 s (34.78%)

Name of kernel/function	1	2	3	4
`fann_backprop_layer`	-	3.09 s (7.12%)	3.43 s (7.11%)	2.87 s (7.14%)
`calc_mse`	-	0.69 s (1.60%)	0.72 s (1.50%)	0.61 s (1.52%)
Mean time to finish	37.61 s	51.18 s	56.90 s	55.08 s

Name of kernel/function	1 (only `fann_run` kernel)	2 (all kernels, max threads)	3 (threads = 64)	4 (threads = 128)	5 (MC, threads = 64)	6 (MC, threads = 128)
Mean time to finish	37.61 s	51.18 s	74.35 s	56.90 s	71.12 s	55.08 s

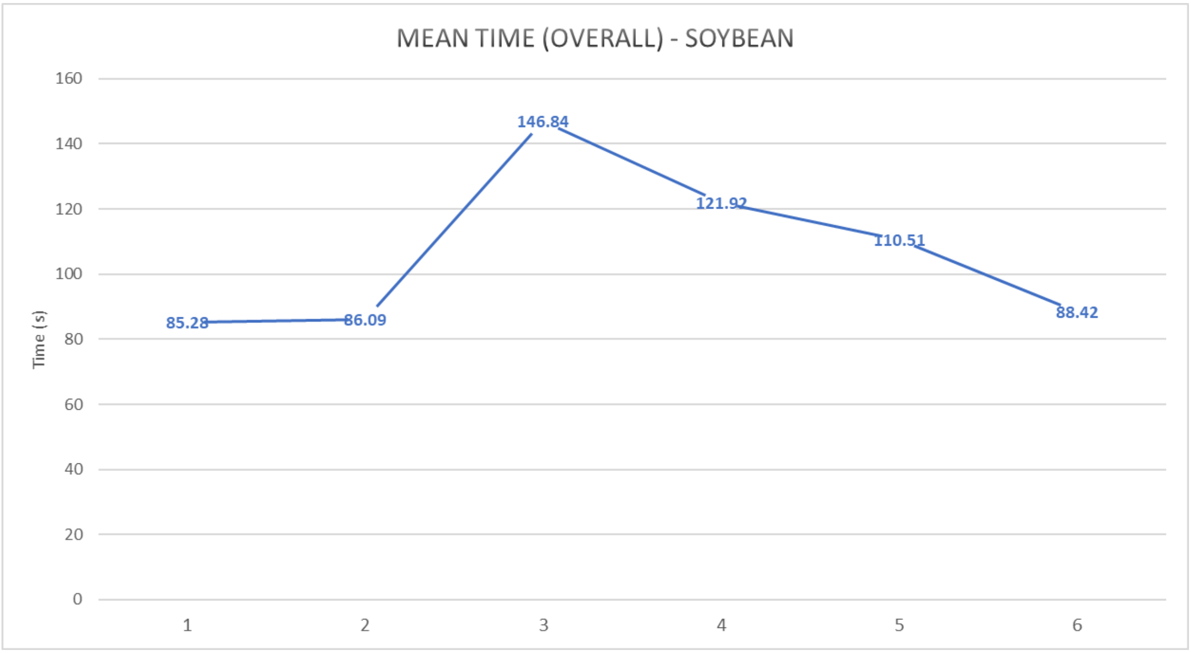


2. `soybean` dataset

Parameters

```
num_data = 342;
num_input = 82;
num_output = 19;
num_layers = 3;
num_neurons_hidden = 128;
desired_error = 0.0001;
max_epochs = 300;
```

Name of kernel/function	1 (only `fann_run` kernel)	2 (all kernels, max threads)	3 (threads = 64)	4 (threads = 128)	5 (MC, threads = 64)	6 (MC, threads = 128)
Mean time to finish	85.28 s	86.09 s	146.84 s	121.92 s	110.51 s	88.42 s

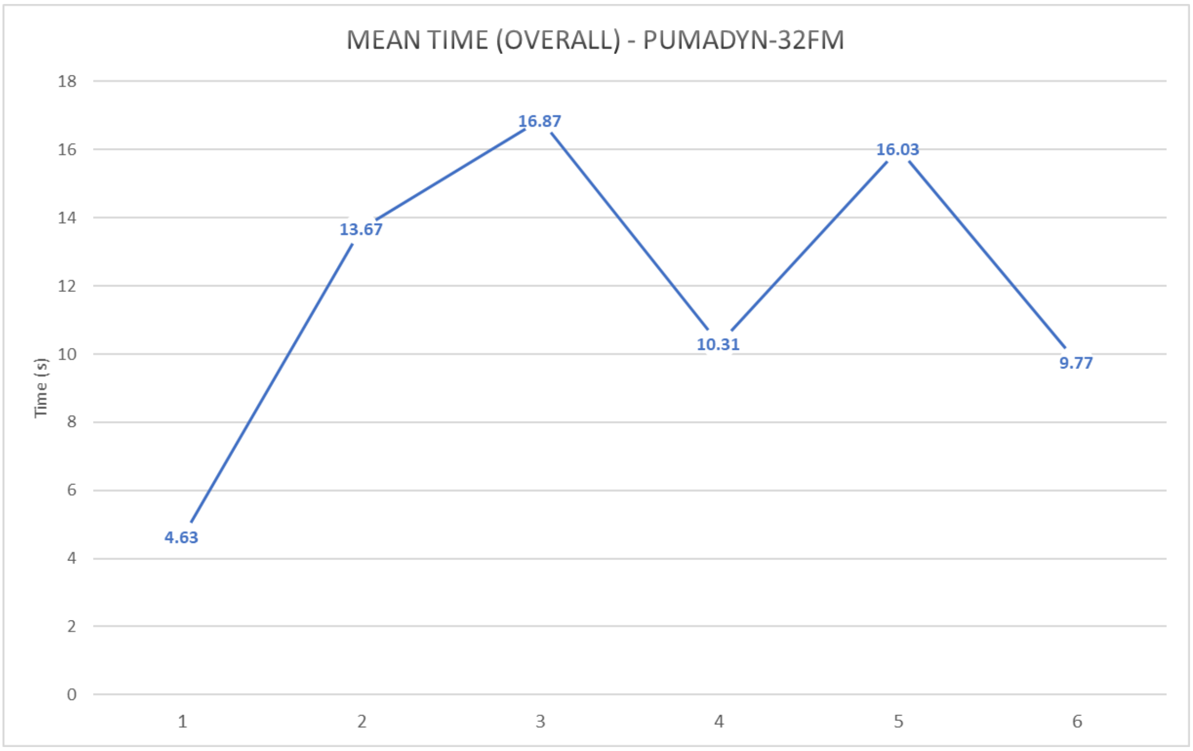


3. `pumadyn-32` dataset

Parameters

```
num_data = 512;
num_input = 32;
num_output = 1;
num_layers = 3;
num_neurons_hidden = 32;
desired_error = 0.01;
max_epochs = 50;
```

Name of kernel/function	1 (only `fann_run` kernel)	2 (all kernels, max threads)	3 (threads = 16)	4 (threads = 32)	5 (MC, threads = 16)	6 (MC, threads = 32)
Mean time to finish	4.63 s	13.67 s	16.87 s	10.31 s	16.03 s	9.77 s



Notes

Since this is an ANN, we can parallelize the forward/backward runs and weight updates on the neurons in a layer and the connections of that neuron, but not on the layers themselves, as that ordering is essential to the logic and working. Also, we come down with the limitation of keeping everything the same between the CPU `ann` variables and GPU `dev_ann` variables, which involves a significant amount of `cudaMemcpy`s, which is also a big portion of the time taken by this implementation. We might still be able to remove some copy overhead by engineering it even more. However, the point is that it would not make a difference and CPU would still be faster than GPU unless we greatly increase the number of parameters in our neural networks. We are only using a 3 layer, 162 neuron (depending on dataset) NN, amounting to just about 4000 connections. In real life, NNs are much bigger with most networks having at least a few hundred thousand parameters.