CUDA Assignment

Yukti Makhija (2019BB10067)

In fann code (sequential), there are four main functions that are called for each data point in every epoch: fann_run(), fann_compute_MSE(), fann_backpropagate_MSE(), fann_update_slopes_batch() (in this order). These can be seen in fann_train_epoch_irpropm(). After this, the model weights get updated in the struct fann using fann_update_weights_irpropm(). The above steps are repeated for all the epochs.

The sequential and parallel implementations were tested on the mushroom, robot, and kin32fm datasets.

Sequential Results

a) Mushroom

Experimental Setup

- num_layers = 3
- num_inputs= 125
- num_neurons_hidden = 32
- desired_error = 0.0001
- num_output=2
- max_epochs = 50
- fann_set_activation_function_hidden = FANN_SIGMOID_SYMMETRIC
- fann_set_activation_function_output = FANN_SIGMOID
- Total training samples = 4062
- Total testing samples = 4062

MSE error on test data: 0.000001

The average time taken by each function per epoch (in seconds)

- fann_run: 0.020000
- fann_compute_MSE: 0.000000
- fann_backpropagate_MSE: 0.010000
- fann_update_slopes_batch: 0.020000
- fann_train_epoch_irpropm: 0.050000 (total time for 1 epoch)

Total training time (fann_train_on_data) : 0.590000

b) Robot

Experimental Setup

- num_layers = 3
- num_inputs= 48
- num_neurons_hidden = 64
- desired_error = 0.001
- num_output=3
- max_epochs = 50

- fann_set_activation_function_hidden = FANN_SIGMOID_SYMMETRIC
- fann_set_activation_function_output = FANN_SIGMOID
- Total training samples = 374
- Total testing samples = 594

MSE error on test data: 0.008569

The average time taken by each function per epoch (in seconds)

- fann_run: 0.010000
- fann_compute_MSE: 0.000000
- fann_backpropagate_MSE: 0.000000
- fann_update_slopes_batch: 0.000000
- fann_train_epoch_irpropm: 0.010000 (total time for 1 epoch)

Total training time (fann_train_on_data) : 0.220000

c) Kin32fm

Experimental Setup

- num_layers = 3
- num_inputs= 32
- num_neurons_hidden = 64
- desired_error = 0.001
- num_output=1
- max_epochs = 50
- fann_set_activation_function_hidden = FANN_SIGMOID_SYMMETRIC
- fann_set_activation_function_output = FANN_SIGMOID
- Total training samples = 2048
- Total testing samples = 2048

MSE error on test data: 0.12768

The average time taken by each function per epoch (in seconds)

- fann_run: 0.010000
- fann_compute_MSE: 0.000000
- fann_backpropagate_MSE: 0.000000
- fann_update_slopes_batch: 0.010000
- fann_train_epoch_irpropm: 0.020000 (total time for 1 epoch)

Total training time (fann_train_on_data) : 1.000

From the results it's clear that only fann_run() and fann_update_slopes_batch() take a large fraction of the total time.

Parallel Program

The for-loops for the epochs and data points can't be parallelised as we want to train a single model. Initially, I studied the four functions mentioned above to find potential sites for parallelisation.

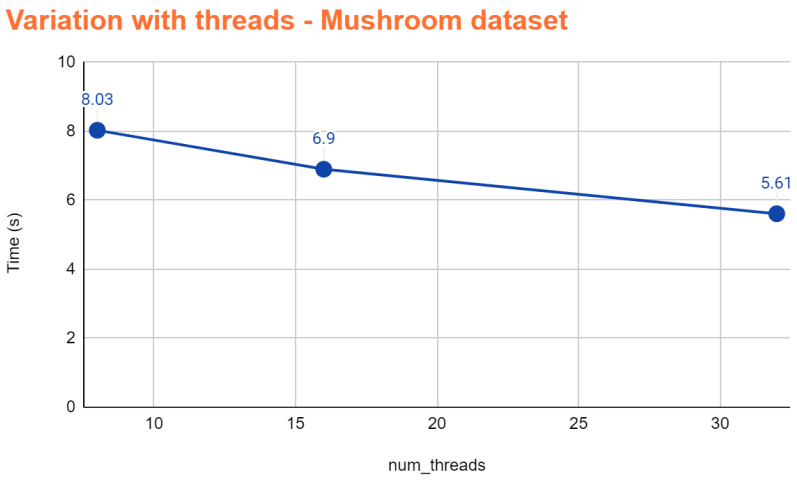
I have made the following changes in the parallel implementation. We have two sets of variables in our code, one on the CPU and their copies on the GPU. These were regularly synced after each kernel using cudaMemcpy.

1. **fann_run() → fann_run_parallel()**
   There were two nested for-loops in this function, one for the layers and the other neuron in a given layer. Here the order of layers in the neural network architecture is significant so that it will remain sequential, but we can parallelise the computation happening on each neuron in a given layer so that they occur simultaneously. This was done by defining a **kernel __global__ void neuron_computation()**. In neuron computation, neuron values are multiplied with corresponding weights using **fann_mult().** This part of the code was further parallelised to reduce runtime. I performed some experiments by varying the number of threads and blocks and the effect on the run time of fann_run_parallel().

   For **__global__ void neuron_computation():**
   Setting: num_blocks =1
   num_threads was varied.

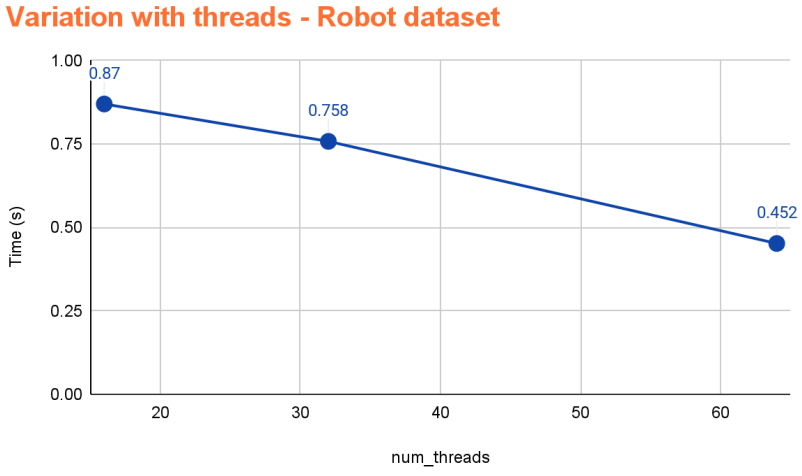
   a) **Mushroom**

num_threads	Time
8	8.03
16	6.9
32	5.61



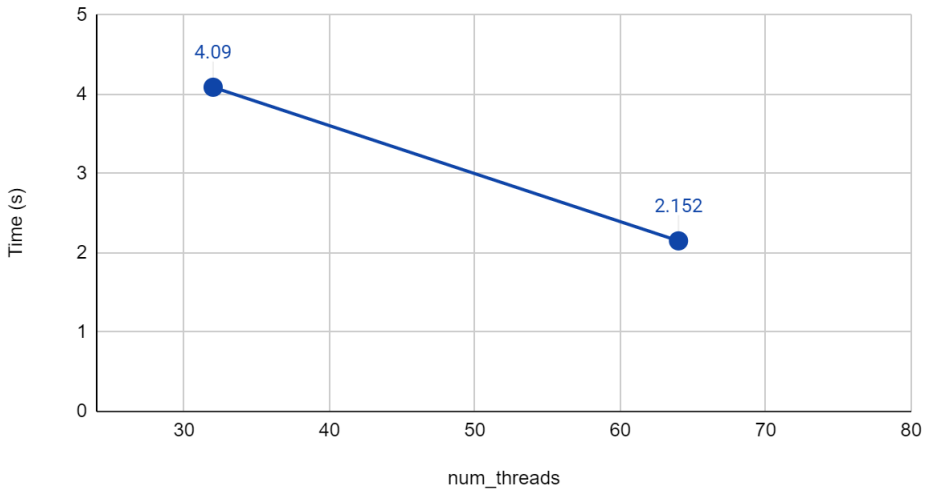
   b) **Robot**

num_threads	Time
16	0.87
32	0.758
64	0.452



c) Kin32fm

num_threads	Time
32	4.09
64	2.152



2. **fann_compute_MSE() → fann_compute_MSE_parallel()**
   This function runs only on the last layer of the neural network, and the mean squared error is calculated for predicted output with the ground truth. Since there is one loop for the neurons present in the last layers, and the execution order is insignificant, we replaced it with a kernel **__global__ void kernel_MSE()**.

   For **__global__ void kernel_MSE():**
   Setting: num_blocks =1
   num_threads = num_output (number of neurons in last layer)

a) Mushroom
Average time taken by fann_compute_MSE_parallel() (per epoch) = 0.421 seconds

b) Robot
Average time taken by fann_compute_MSE_parallel() (per epoch) = 0.053 seconds

c) Kin32fm
Average time taken by fann_compute_MSE_parallel() (per epoch) = 0.18 seconds

3. **fann_backpropagate_MSE() → fann_backpropagate_MSE_parallel()**
   There are two separate for-loops in this function, one for the last layer neurons where the loss is backpropagated and the second for gradient calculation in the second last layer neurons. I wrote two kernel functions for them to replace the loops: **__global__ void neuron_backpropagate()** and **__global__ void backpropagate_accumulate_last()**.

   For **__global__ void neuron_backpropagate():**
   Setting: num_blocks =1
   num_threads = num_output (number of neurons in last layer)

   For **__global__ void backpropagate_accumulate_last():**
   Setting: num_blocks =1
   num_threads = number of neurons in the second last layer

a) **Mushroom**
Average time taken by fann_backpropagate_MSE_parallel() (per epoch) = 0.646 seconds

b) **Robot**
Average time taken by fann_backpropagate_MSE_parallel() (per epoch) = 0.09 seconds

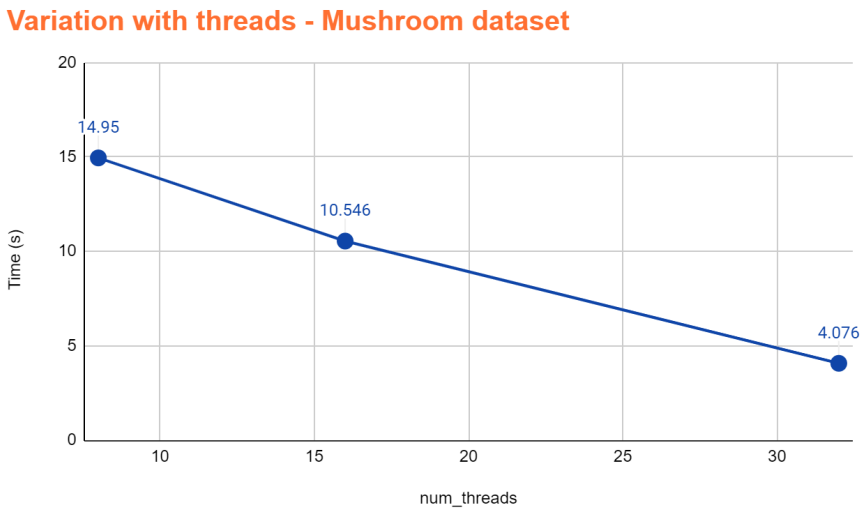
c) **Kin32fm**
Average time taken by fann_backpropagate_MSE_parallel() (per epoch) = 0.39 seconds

4. **fann_update_slopes_batch() → fann_update_slopes_batch()**
This function is to calculate the updated weights for each connection. This was performed using three nested loops: for the layers, neurons in each layer and connections of each neuron. Again, I parallelised the neuron computation by replacing it with a kernel **__global__ void neuron_update()**. This was further optimised by writing a kernel to replace the innermost loop for neuron connections.

For **__global__ void neuron_update():**
Setting: num_blocks =1
num_threads was varied.

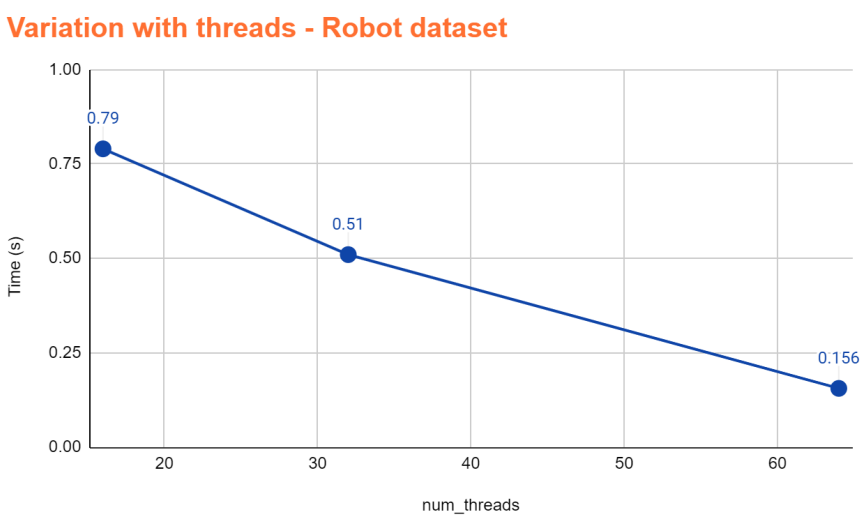
a) **Mushroom**

num_threads	Time
8	14.95
16	10.546
32	4.076



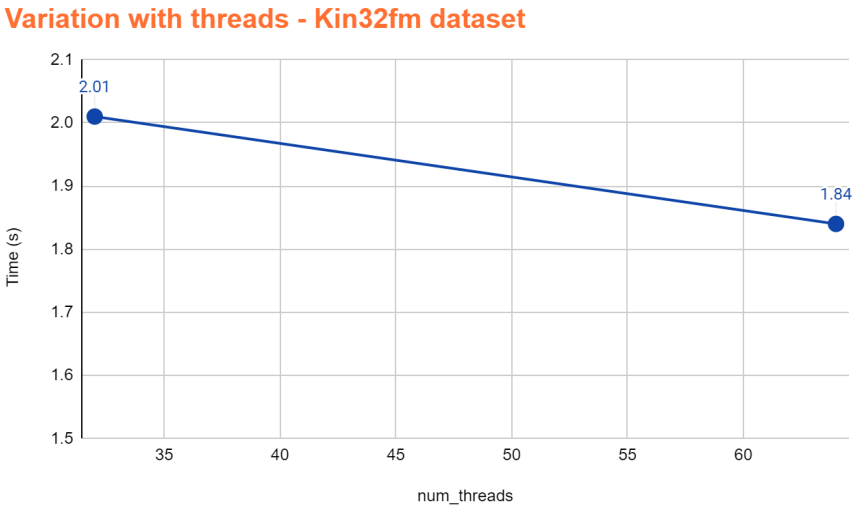
b) **Robot**

num_threads	Time
16	0.79
32	0.51
64	0.156



c) Kin32fm

num_threads	Time
32	2.01
64	1.84



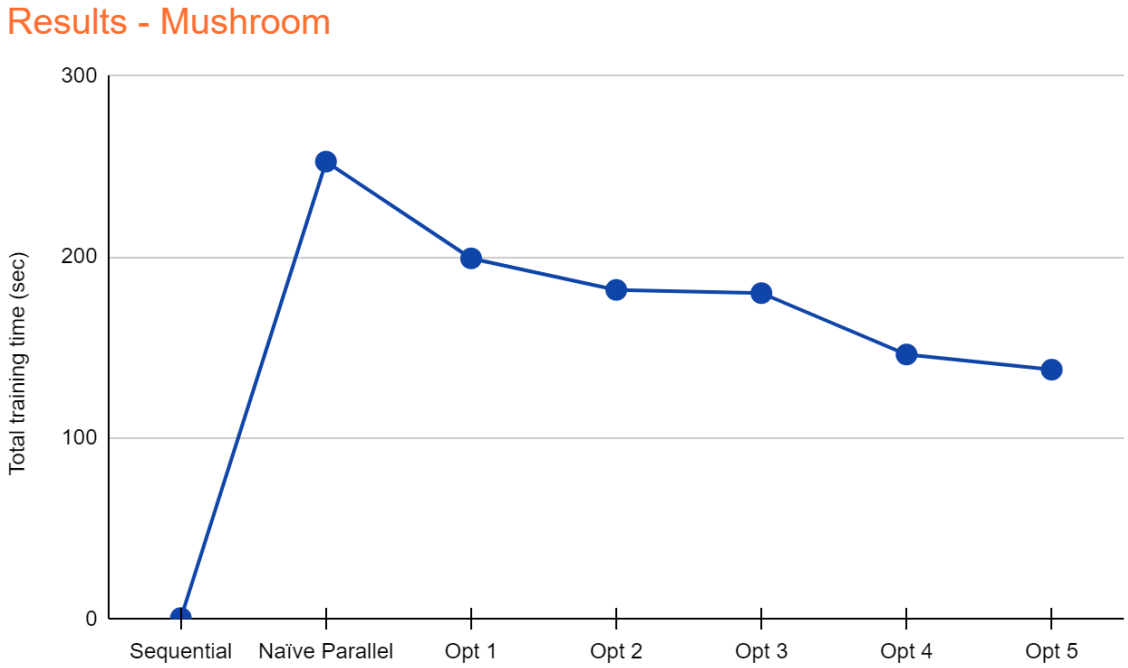
Optimisations

1. Performing cudaMalloc and cudaMemcpy of ann, weights, and train_errors for each epoch instead of each kernel.
2. Doubled the number of threads for **kernel __global__ void neuron_computation().**
3. Memory coalescing in **kernel __global__ void neuron_computation().**
4. Doubled the number of threads for kernel **__global__ void neuron_update().**
5. Memory coalescing in **__global__ void neuron_update().**
6. Replaced the loop for fann_mult of neuron values and weights with vector product using kernels.

Results on Mushroom

MSE error on test data: 0.003719

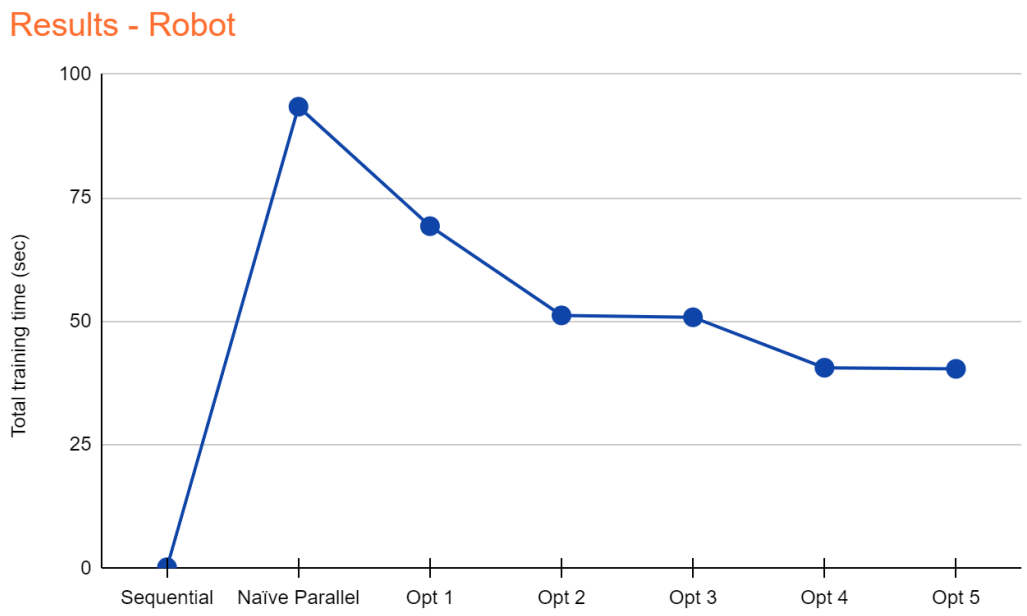
	Sequential	Naïve Parallel	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6
Total training time (sec)	0.59	252.63	199.2	181.8	180.01	146.14	137.75	-



```
==22863== Profiling application: ./opt mushroom
==22863== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   54.99%  50.7330s    113736  446.06us  151.78us  741.51us  neuron_update(fann_neuron*, int, fann_neuron*, float*,
 float*, int, int)
                   35.41%  32.6712s    113736  287.25us  112.23us  464.52us  neuron_computation(fann_layer*, fann*, fann_neuron**,
bool, fann_neuron*, float*, fann_neuron*, int, int)
                    5.80%  5.34865s     56868  94.053us  93.793us  94.817us  neuron_backpropagate(fann_neuron*, float*, int, float*
, int)
                    1.30%  1.20244s     56868  21.144us  20.001us  24.544us  kernel_MSE(fann*, fann_neuron*, float*, float*, int)
                    0.88%  808.41ms    682444  1.1840us     992ns  7.4240us  [CUDA memcpy HtoD]
                    0.84%  776.74ms     56868  13.658us  11.360us  13.920us  backpropagate_accumulate_last(fann_neuron*, int, float
*)
                    0.64%  594.50ms    398104  1.4930us  1.3760us  7.7760us  [CUDA memcpy DtoH]
                    0.13%  124.06ms     56868  2.1810us  2.1120us  8.2240us  [CUDA memset]
      API calls:   47.49%  64.3340s    625618  102.83us  2.8740us  35.671ms  cudaFree
                   45.69%  61.8967s   1080548  57.282us  5.6320us  37.780ms  cudaMemcpy
                    4.02%  5.44962s    398076  13.689us  6.8200us  28.052ms  cudaLaunchKernel
                    2.37%  3.20463s    625618  5.1220us  3.0900us  161.39ms  cudaMalloc
                    0.29%  386.99ms     56868  6.8050us  4.4280us  12.872ms  cudaMemset
                    0.15%  208.84ms    568680     367ns     132ns  15.857ms  cudaGetLastError
                    0.00%  562.44us         1  562.44us  562.44us  562.44us  cuDeviceTotalMem
                    0.00%  318.37us        96  3.3160us     175ns  141.73us  cuDeviceGetAttribute
                    0.00%  38.493us         1  38.493us  38.493us  38.493us  cuDeviceGetName
                    0.00%  7.9530us         1  7.9530us  7.9530us  7.9530us  cuDeviceGetPCIBusId
                    0.00%  4.1900us         2  2.0950us     253ns  3.9370us  cuDeviceGet
                    0.00%  2.3730us         2  1.1860us     772ns  1.6010us  cuDeviceGetCount
                    0.00%     308ns         1     308ns     308ns     308ns  cuDeviceGetUuid
```

Results on Robot

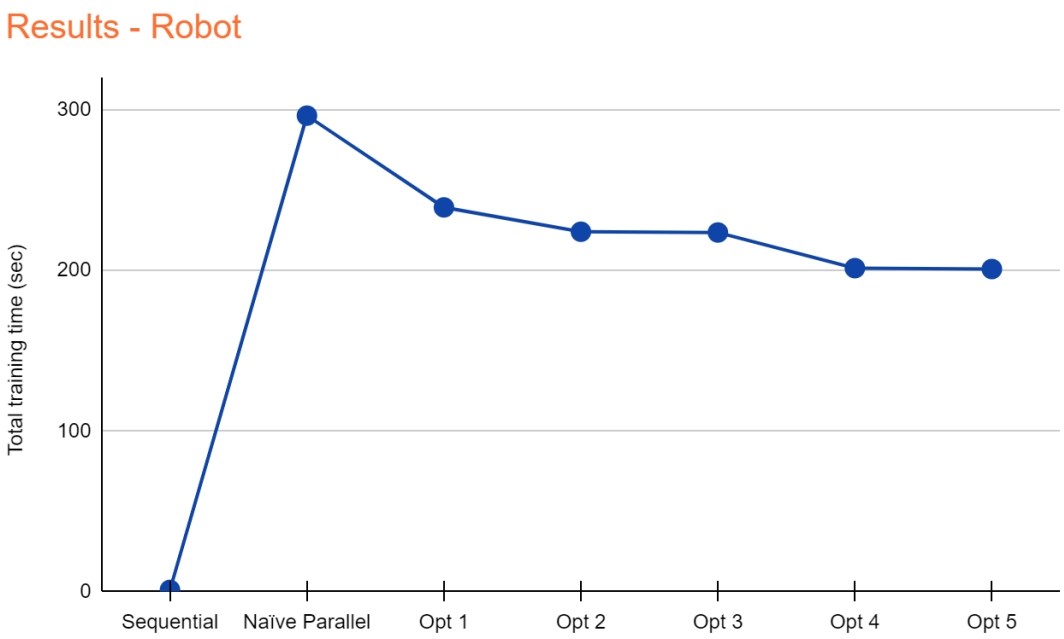
	Sequential	Naïve Parallel	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6
Total training time (sec)	0.22	93.4	69.23	51.18	50.79	40.59	40.36	45.62

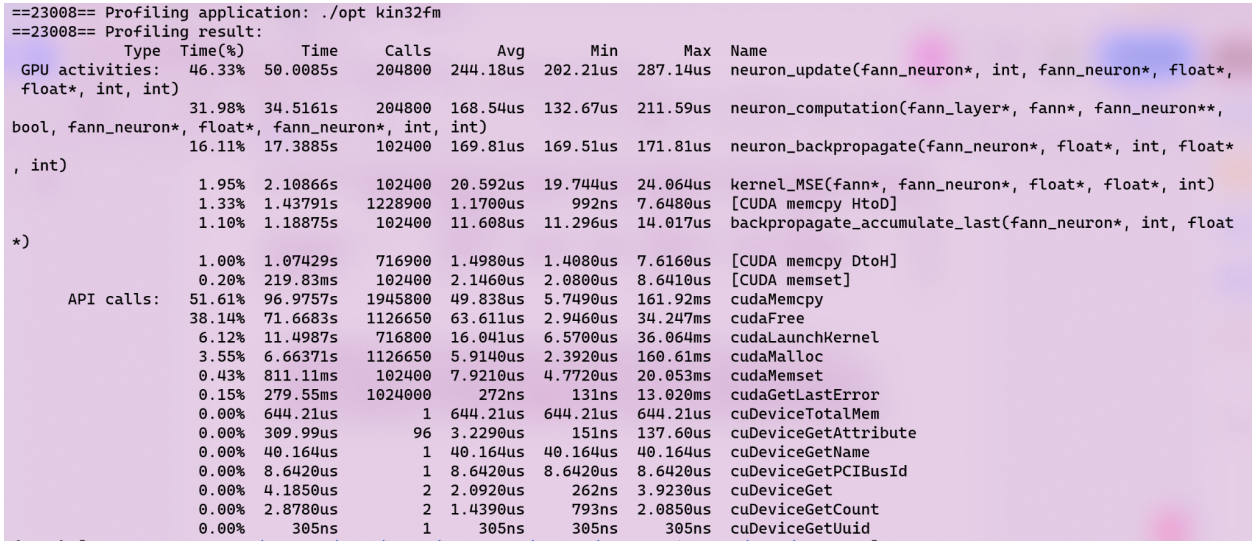


```
==22984== Profiling application: ./opt robot
==22984== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   47.99%  10.9452s     37400  292.65us  289.99us  295.94us  neuron_update(fann_neuron*, int, fann_neuron*, float*,
 float*, int, int)
                   32.57%  7.42853s     37400  198.62us  189.06us  209.79us  neuron_computation(fann_layer*, fann*, fann_neuron**,
bool, fann_neuron*, float*, fann_neuron*, int, int)
                   14.43%  3.29169s     18700  176.03us  175.71us  176.87us  neuron_backpropagate(fann_neuron*, float*, int, float*
, int)
                    1.70%  386.94ms     18700  20.692us  20.032us  24.449us  kernel_MSE(fann*, fann_neuron*, float*, float*, int)
                    1.17%  265.97ms    224500  1.1840us     992ns  6.8480us  [CUDA memcpy HtoD]
                    1.11%  253.15ms     18700  13.537us  8.7680us  13.824us  backpropagate_accumulate_last(fann_neuron*, int, float
*)
                    0.86%  195.86ms    131000  1.4950us  1.3760us  7.1360us  [CUDA memcpy DtoH]
                    0.18%  40.219ms     18700  2.1500us  2.0800us  7.6800us  [CUDA memset]
      API calls:   50.45%  18.5523s    355500  52.186us  6.3900us  36.391ms  cudaMemcpy
                   39.44%  14.5047s    205950  70.428us  3.0620us  25.117ms  cudaFree
                    5.89%  2.16530s    130900  16.541us  6.7580us  23.817ms  cudaLaunchKernel
                    3.62%  1.33043s    205950  6.4590us  3.1670us  160.80ms  cudaMalloc
                    0.45%  165.65ms     18700  8.8580us  5.0110us  13.128ms  cudaMemset
                    0.15%  53.720ms    187000     287ns     131ns  12.974ms  cudaGetLastError
                    0.00%  572.66us         1  572.66us  572.66us  572.66us  cuDeviceTotalMem
                    0.00%  323.19us        96  3.3660us     146ns  145.05us  cuDeviceGetAttribute
                    0.00%  61.051us         1  61.051us  61.051us  61.051us  cuDeviceGetName
                    0.00%  9.4990us         1  9.4990us  9.4990us  9.4990us  cuDeviceGetPCIBusId
                    0.00%  4.2670us         2  2.1330us     274ns  3.9930us  cuDeviceGet
                    0.00%  2.9820us         2  1.4910us     764ns  2.2180us  cuDeviceGetCount
                    0.00%     263ns         1     263ns     263ns     263ns  cuDeviceGetUuid
```

Results on Kin32fm

	Sequential	Naïve Parallel	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6
Total training time (sec)	1	296.51	239.34	224.17	223.69	201.48	200.92	212.53





```
==23008== Profiling application: ./opt kin32fm
==23008== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   46.33%  50.0085s    204800  244.18us  202.21us  287.14us  neuron_update(fann_neuron*, int, fann_neuron*, float*,
 float*, int, int)
                   31.98%  34.5161s    204800  168.54us  132.67us  211.59us  neuron_computation(fann_layer*, fann*, fann_neuron**,
bool, fann_neuron*, float*, fann_neuron*, int, int)
                   16.11%  17.3885s    102400  169.81us  169.51us  171.81us  neuron_backpropagate(fann_neuron*, float*, int, float*
, int)
                    1.95%  2.10866s    102400  20.592us  19.744us  24.064us  kernel_MSE(fann*, fann_neuron*, float*, float*, int)
                    1.33%  1.43791s   1228900  1.1700us     992ns  7.6480us  [CUDA memcpy HtoD]
                    1.10%  1.18875s    102400  11.608us  11.296us  14.017us  backpropagate_accumulate_last(fann_neuron*, int, float
*)
                    1.00%  1.07429s    716900  1.4980us  1.4080us  7.6160us  [CUDA memcpy DtoH]
                    0.20%  219.83ms    102400  2.1460us  2.0800us  8.6410us  [CUDA memset]
      API calls:   51.61%  96.9757s   1945800  49.838us  5.7490us  161.92ms  cudaMemcpy
                   38.14%  71.6683s   1126650  63.611us  2.9460us  34.247ms  cudaFree
                    6.12%  11.4987s    716800  16.041us  6.5700us  36.064ms  cudaLaunchKernel
                    3.55%  6.66371s   1126650  5.9140us  2.3920us  160.61ms  cudaMalloc
                    0.43%  811.11ms    102400  7.9210us  4.7720us  20.053ms  cudaMemset
                    0.15%  279.55ms   1024000     272ns     131ns  13.020ms  cudaGetLastError
                    0.00%  644.21us         1  644.21us  644.21us  644.21us  cuDeviceTotalMem
                    0.00%  309.99us        96  3.2290us     151ns  137.60us  cuDeviceGetAttribute
                    0.00%  40.164us         1  40.164us  40.164us  40.164us  cuDeviceGetName
                    0.00%  8.6420us         1  8.6420us  8.6420us  8.6420us  cuDeviceGetPCIBusId
                    0.00%  4.1850us         2  2.0920us     262ns  3.9230us  cuDeviceGet
                    0.00%  2.8780us         2  1.4390us     793ns  2.0850us  cuDeviceGetCount
                    0.00%     305ns         1     305ns     305ns     305ns  cuDeviceGetUuid
```

From the results, it’s clear that the original sequential implementation is several orders of magnitude faster than the naive parallel implementation and even optimising the code doesn’t help much. Fine-grained analysis with nvprof revealed that the lion’s share of the total time is taken by cudaMemcpy. Another reason why this is slow is that we’ve used atomicAdd(), which makes that portion sequential but is essential to avoid race conditions. The main reason behind this is that this is a very small-scale application, and the reduction in time from parallelisation gets concealed due to overhead because of cudaMemcpy.