CUDA Implementation of ANN

COL730 Assignment 1

Dakshit Babbar | 2020EE30163

September 2022

1 Objective

We are given a sequential implementation of the training of an Artificial Neural Network (ANN) on some given data-sets like mushroom, robot etc. Our objective is:

- To prallelize the sequential implementation of the training of the ANN.
- To further optimize our parallel implementation as much as possible.

2 The Sequential Implementation

We are given an already implemented library in C, FANN, which allows us to implement the training of an artificial neural network just by using a bunch of functions predefined for the user.

The function **`fann_train_on_data`** (fann_train_on_file.c line 265) is used for the training of the ANN. Its definition is fundamentally based on the following four steps and their corresponding defined functions:

- **Feed-forward Run:** Given an input, the output from the ANN is calculated. The function used for this is:
  `fann_run` (`ann, input`) (fann.c line 501)
- **MSE Computation:** The corresponding mean squared error for each neuron in the last layer is calculated. The function used for this is:
  `fann_compute_MSE` (`ann, desired_output`) (fann_train.c line 216)
- **MSE Back Propagation:** Errors of all the other neurons are calculated by propagating the MSE backwards form the last layer to the first layer. The equation used is,

  $$\delta_i^{(l)} = \sum_{j=1}^{n} w_i^{(l)} \delta_j^{(l+1)} f'(z_j^{(l+1)}) \qquad (1)$$

  where $\delta_i^{(l)}$ and $w_i^{(l)}$ is the error and the weight corresponding to the neuron i in the layer l, n is the number of neurons in the layer $l+1$, f is the activation function and $z_j^{(l+1)}$ is the input to the neuron j in the layer $l+1$. The function used for this is:
  `fann_backpropagate_MSE` (`ann`) (fann_train.c line 290)

- **Slope Update:** Gradients of the error function with respect to each weight is calculated. The equation used for this is,

$$\Delta_{ij}^{(l)} = a_i^{(l)} \delta_j^{(l+1)} \qquad (2)$$

where $\Delta_{ij}^{(l)}$ is the gradient of the error w.r.t the weight j corresponding to the neuron i in layer l and $\delta_j^{(l+1)}$ is the error corresponding to the neuron j in the layer $l+1$. The function used for this is:
`fann_update_slopes_batch` (`ann, layer_begin, layer_end`) (fann_train.c line 460)

We aim to parallelize this sequential implementation. To do so, we will parallelize the implementation of the above mentioned four functions.

3 Parallelize and Optimize

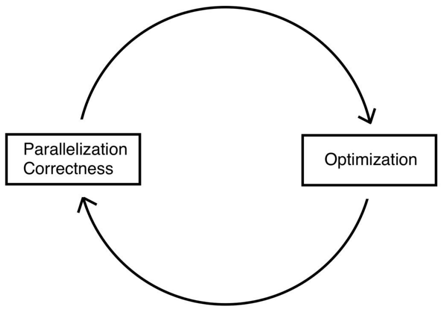
Before we start the parallelization of our program we will make the following assumptions,

- The ANN being trained will not have any shortcuts (connections to a layer other than the just previous layer).
- The ANN being trained will be fully connected i.e. the connection rate defined in the library will always be 1.

In any sequential implementation of a program, only some fraction of the code is parallelizable. Our objective is to paralellize this fraction of the program and further optimize the parallelization.

The parallelizable fraction of the code includes the four functions mentioned in the previous section which we will parallelize.

The parallelization process is a cyclic process. We first define CUDA kernels and call threads to execute them making sure that the correctness of the program is maintained. After doing this we optimize the parallel implementation. Then we start this cycle again.



We will use this approach to parallelize and optimize the above mentioned functions. We will first parallelize the function then make some requisite optimizations and then cumulatively parallelize the remaining functions in a similar manner.

Once all the four functions are parallelized, we will make some more optimizations that will improve the performance. Initially we will be using the mushroom data-set for the training of our ANN.

3.1 Parallelizing Feed-forward run

Trial 1 (Parallelization and Correctness):

The sequential implementation of **fann_run** involves three nested for loops. The first one to iterate over the layers, the second one to iterate over all the neurons in one layer and the third one to iterate over the array of connections and weights of one neuron, to element-wise multiply them and then add all the products together to get the **neuron_sum**.

Here we eliminate the third for loop, and instead of iterating on every connection and weights to multiply them together, we call threads equal to the number of connections and ask each thread to compute the product of its corresponding connection output and its weight. With this we will be executing all the floating operations parallelly. All these threads will then increment a global variable for which we use **atomicAdd** to maintain mutual exclusion. We do this by defining a kernel **compute_neuron_sum_kernel**.(CUDA_OPTIM.cu, line 11)

Parallel Algorithm

```
__global__ void compute_neuron_sum_kernel(...){
    idx = threadIdx.x;

    val = weights[idx]*(neurons[idx].value);
    atomicAdd(sum, val);
}

fann_run{
    layer_it = second_layer;
    for(; layer_it!=last_layer; layer_it++){
        neuron_it = first_neuron;
        for(; neuron_it!=last_neuron; neuron_it++){
            d_weights; //pointer to the weight corresponding to the first connection
            d_neurons; //pointer to the first connection
            d_sum; //pointer to an address initialized to zero
            compute_neuron_sum_kernel<<<1, num_connections>>>(d_weights, d_neurons,
                num_connections, d_sum);
        }
    }
}
```

Time	Name
9m14s	Total
8.42s	cudaMemcpy HToD
7.10s	compute_neuron_sum_kernel
3.78	cudaMemcpy DToh

The training process takes 9min 14sec to run and is now to be optimized. To do so we profile the program to find our bottle-neck.

The profile clearly shows that **cudaMemcpy** from HToD takes most of the time and is called the maximum number of times too. So it makes sense to reduce the number of its calls because we certainly cannot reduce the time of copying the data from host to device.

We can do this by getting rid of the second for loop in the above algorithm.

Trial 2 (Optimization):
Issue:
In the previous trial, to parallelize the third nested for loop, we called **cudaMemcpy** to copy the weights and neuron connections of every neuron from host to device, one at a time in its corresponding iteration. This leads to less memory throughput and hence low performance.

Resolution:
In this trial we will try to utilize all the available memory bandwidth as much as possible i.e. increase memory throughput which will lead to better performance.

To do this, we will copy the weights and the neuron connections of an entire layer together instead of copying it neuron by neuron. This means that we will have to get rid of the second nested for loop and compute the contents of an entire layer together.

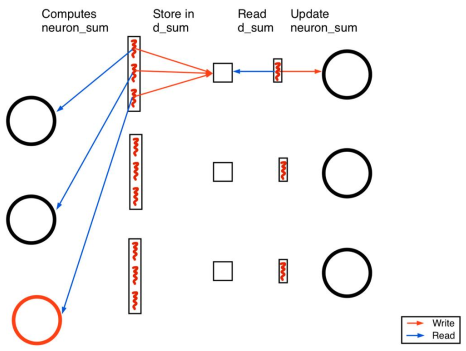
We do this by making two kernels, one **compute_neuron_sum_for_entire_layer_kernel** which updates an array **d_sums** initialised to zero, to store the neuron sum values of all the neurons and the second **compute_entire_layer_kernel** which uses **d_sums** to update **neuron_sum** and **neuron_value** of all the neurons (CUDA_OPTIM.cu, line 62). The first kernel is called with one block for each neuron in the present layer and one thread in each block for each connection. The second kernel is called with threads equal to the number of neurons in the present layer.

Parallel Algorithm

```
__global__ void compute_neuron_sum_for_entire_layer_kernel(...){
    //computes the neuron sum for every neuron in this layer
    //update the corresponding d_sum pointer
}

__global__ void compute_entire_layer_kernel(...){
    //accesses d_sum for every neuron
    //applies activation to the the neuron sum obtained from d_sum
    //updates the neuron's sum attribute
}
fann_run{
    layer_it = second_layer;
    for(; layer_it!=last_layer; layer_it++){
        d_neurons; //pointer to first neuron of this layer
        d_neuron_connections; //pointer to first connection of the first neuron
        d_weights; //pointer to weight corresponding to the first connection of the
            first neuron
        d_sum; //pointer to an array of length of number of neurons in this layer
            initialized to zero
        BLOCKS = num_neurons;
        THREADS = num_neuron_connections;
        compute_neuron_sum_for_entire_layer_kernel<<<BLOCKS, THREADS>>>(d_neurons,
            d_neuron_connections, d_weights, d_sums);

        BLOCKS = 1;
        THREADS = num_neurons;
        compute_entire_layer_kernel<<<BLOCKS, THREADS>>>(d_neurons, d_sums);
    }
}
```

Time	Name
1m31s	Total
1.44s	compute_neuron_sum_for_entire_layer_kernel
829.27ms	compute_entire_layer_kernel
770.41ms	cudaMemcpy HToD
267.23ms	cudaMemset
260.76ms	cudaMemcpy DToH

The algorithm is depicted pictorially with the operations of all the threads in the following figure.



In this trial we obtain a runtime of 1min 31s, which is significantly better than our previous trial. But wish to optimize it even further. Hence we profile our program to find our bottle-neck. The profile shows that we are successful is reducing the latency used by **`cudaMemcpy`** but now our kernels are utilising a large amount of time. We can optimize this by directly updating the **`neuron_sum`** for all the neurons instead of making a separate array **`d_sums`** and then copying these values into **`neuron_sum`**.

Trial 3 (Optimization):
Issue:
In the previous trial we first used cudaMemset to initialize an array **`d_sums`** of length equal to the number of neurons in the present layer. Then we used a kernel to update this array with the neuron sums of the corresponding neurons and another kernel to copy these values from **`d_sums`** to the neuron sum attributes. Doing this, we are utilizing some extra space in every iteration to make the new array, which takes a significant amount of time.

Resolution:
In this trial we will not make a separate array **`d_sums`** to store the **`neuron_sum`** of the neurons. Instead we will make a separate kernel that will initialize the **`neuron_sum`** values of all the neurons parallelly to zero in every iteration while the other two kernels do the same work. The only difference being that the kernels will directly update the **`neuron_sum`** and not some separate array.

The three kernels used in this trial are, **`direct_initialise_neuron_sums`** to initialize the neuron sums, **`direct_compute_neuron_sum_for_entire_layer_kernel`** to update the neuron sums and **`direct_compute_entire_layer_kernel`** to update the neuron values.

Time	Name
47.094s	Total

We get a runtime of 47.094s which is yet another significant increase in our performance. Now we will parallelize MSE computation function

3.2 Parallelizing MSE Computation

The sequential implementation of **`fann_compute_MSE`** involves one for loop which iterates over every neuron in the last layer of the ANN and does the following work,

1. Computes the difference between the neuron value and the desired output.
2. Adds the square of the difference to **`MSE_value`** of the ANN and increments **`num_MSE`**.
3. Multiply the computed difference with the derived activation output of the neuron and updates the error corresponding to the neuron in the ANN.

Here we will get rid of this for loop and call a kernel **`compute_MSE_parallel_kernel`** with a number of threads equal to the number of neurons in the last layer. Where each thread does the above mentioned tasks for their corresponding neuron (CUDA_OPTIM.cu, line 325).

Parallel Algorithm

```
__global__ void compute_MSE_parallel_kernel(...){
    //compute difference
    //update MSE_value and num_MSE
    //update error_it
}

fann_compute_MSE{
    //pointer to first neuron of last layer
    d_neurons;
    //pointer to the first desired output
    d_desired_output;
    //pointer to the initialized error corresponding to the first neuron of the last
        layer
    d_error_it;
    //pointer to the MSE_value of the ANN
    d_ann_MSE;
    //pointer to the num_MSE of the ANN
    d_ann_num_MSE;

    BLOCKS = 1;
    THREADS = num_neurons;
    compute_MSE_parallel_kernel<<<BLOCKS, THREADS>>>(d_neurons, d_desired_output,
        d_error_it, d_ann_MSE, d_ann_num_MSE, ...)
}
```

Time	**Name**
2m25s	Total
1.473s	cudaMemcpy HToD
844.94ms	direct_compute_neuron_sum_for_entire_layer_kernel
645.29ms	direct_compute_entire_layer_kernel
645.12ms	cudaMemcpy DToH
444.07	compute_MSE_parallel_kernel

The total runtime after parallelizing MSE computation is 2min 25sec, which is greater than what we got with just parallelizing feed forward run. This is because when we call new kernels, new threads are made which takes time. But there is less scope of optimization for this particular function. So we now move to parallelizing the MSE backpropagation.

3.3 Parallelizing MSE Backpropagation

In the sequential implementation of **`fann_backpropagate_MSE`** there are three nested for loops. The first to iterate over every layer from last to first, the second for loop to iterate over every neuron in the present layer, and the third for loop to iterate over the connections of the neurons in the previous layer and update their error values with the product of the error of the neuron in the present layer and the weight corresponding to the connection (using (1)). It also multiplies the activation output with this error and finally updates the error values.

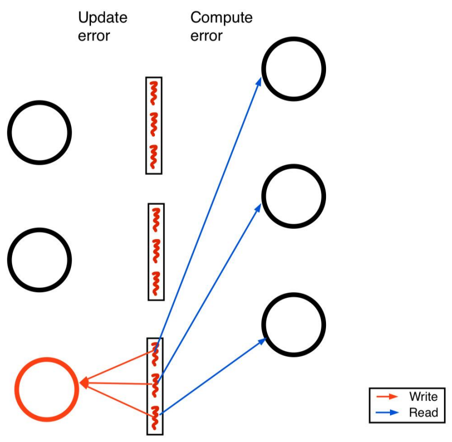
Here we will get rid of the inner two for loops, by calling a kernel **`backpropagate_MSE_parallel_kernel`** with one block for each connection in the previous layer (because all neurons in the present layer has the same number of connections) and one thread in each block for every neuron in the present layer. We will call another kernel **`activate_errors`** with the number of threads equal to the number of neurons in the previous layer that multiplies the errors with the activation outputs.

Parallel Algorithm

```
__global__ void backpropagate_MSE_parallel_kernel(...){
    //compute error using (1) without activation output
    //update the error values using atomicAdd
}

_global__ void activate_errors(...){
    //multiply the errors with the activation outputs
    //finally update the error values
}

fann_backpropagate_MSE{
    d_prev_neurons; //pointer to the first connection in the previous layer
    d_prev_errors; //pointer to the error of the first connection in the previous layer
    d_errors; //pointer to the error of the first neuron in the present layer

    BLOCKS = num_prev_neurons;
    THREADS = num_neurons-1;
    backpropagate_MSE_parallel_kernel<<<BLOCKS, THREADS>>>(d_prev_neurons,
        d_prev_errors, d_errors, d_weights);

    BLOCKS = 1;
    THREADS = num_prev_neurons;
    activate_errors<<<BLOCKS, THREADS>>>(d_prev_neurons, d_prev_errors);

}
```

Time	**Name**
2m3s	Total
1.75s	cudaMemcpy HToD
811.31ms	direct_compute_entire_layer_kernel
710.87ms	cudaMemcpy DToH
615.26ms	direct_compute_neuron_sum_for_entire_layer_kernel
408.6ms	compute_MSE_parallel_kernel
301.83ms	direct_initialise_neuron_sum
268.46ms	activate_errors
156.49ms	backpropagate_MSE_parallel_kernel

The following figure pictorially depicts the algorithm, with the functions of the threads involved,



We obtain a runtime of 2min 3sec, which is better than what we had obtained after paralleizing MSE computation. We now move ahead to parallelize slope updates.

3.4 Parallelizing Slope Updates

In the sequential implementation of **fann_update_slopes_batch** there are three nested for loops. The first to iterate over every layer from second to last, the second for loop to iterate over every neuron in the present layer, and the third for loop to iterate over the connections of the neurons in the previous layer and update the slope of the error w.r.t the weight of the corresponding connection using equation (2).

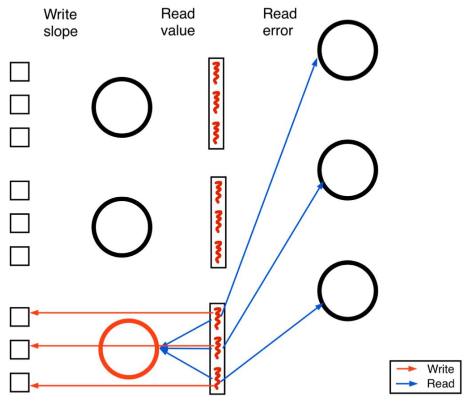
Here we will get rid of the inner two for loops, by calling a kernel **update_slopes_parallel _kernel** with one block for each connection in the previous layer (because all neurons in the present layer has the same number of connections) and one thread for every neuron in the present layer, in each block.

Parallel Algorithm

```
__global__ void update_slopes_parallel_kernel(...){
    //compute slope using (2)
    //update the slope value
}

fann_update_slopes_batch{
    d_prev_neurons; //pointer to the first connection in the previous layer
    d_prev_slopes; //pointer to the slope of the first connection in the previous layer
    d_errors; //pointer to the error of the first neuron in the present layer

    BLOCKS = num_prev_neurons;
    THREADS = num_neurons-1;
    update_slopes_parallel_kernel<<<BLOCKS, THREADS>>>(d_prev_slopes, d_prev_neurons,
        d_errors);

}
```

Time	**Name**
2m17s	Total
2.65s	cudaMemcpy HToD
958.05ms	cudaMemcpy DToH
811.34ms	direct_compute_entire_layer_kernel
622.93ms	direct_compute_neuron_sum_for_entire_layer_kernel
443.44ms	update_slopes_parallel _kernel
421.97ms	compute_MSE_parallel_kernel
298.58ms	direct_initialise_neuron_sum
277.86ms	activate_errors
159.38ms	backpropagate_MSE_parallel_kernel

The following figure pictorially depicts the algorithm, with the functions of the threads involved,



We obtain a runtime of 2min 17sec here, which is not as good as we obtained when we parallelized MSE Backpropagation, this is because we are calling the same number of threads again and copying the same data again back and forth. This takes most of our time. Hence to optimize this we get rid of the slope update function completely and merge it with the MSE Backpropagation itself (Details in next subsection).

3.5 Merging Slope Updates with MSE Backpropagation

Issue:
In the profile of the the slope update function, we learnt that **`cudaMemcpy`** HToD is the bottleneck for our program. To improve our performance we have to reduce the number of times we are copying data from host to device.

Resolution:
Consider the following points regarding the slope update and the MSE backpropagate function,

- Kernels launched by both the functions take the same pointers as inputs except the slopes of the neurons of the previous layer.
- The launch configuration of kernels of both these functions is exactly the same.
- There is no loop dependencies in the slope update function i.e. one iteration of the loop depends neither on the previous iteration nor on the next one. Also this function is order independent i.e. we can start our iteration from the last layer as well as from the first layer.

As these points hold for the above mentioned functions hence we can easily merege these two into one function and reduce the number of times we are copying data from host to device.

If we get rid of the slope update function and merge it with the MSE Backpropagate function then we will have to make the MSE Backpropagate function do the tasks that slope update function was doing. One of which is initializing the slope pointers to zero for all the neuron connections in the entire ANN.

After doing this we can simply call the kernel of MSE Backpropagation but with an extra input of slope pointer this time and with the same launch configuration.

Parallel Algorithm

```
__global__ void backpropagate_MSE_parallel_kernel(...){
    //compute error using (1) without activation output
    //compute slope
    //update slope

    if(layer = second_layer){
       return;
    }
    //update the error values using atomicAdd
}

_global__ void activate_errors(...){
    //multiply the errors with the activation outputs
    //finally update the error values
}

fann_backpropagate_MSE{
    d_prev_neurons; //pointer to the first connection in the previous layer
    d_prev_errors; //pointer to the error of the first connection in the previous layer
    d_errors; //pointer to the error of the first neuron in the present layer
    d_prev_slopes; //pointer to the slope of the first connection in the previous layer

    BLOCKS = num_prev_neurons;
    THREADS = num_neurons-1;
    backpropagate_MSE_parallel_kernel<<<BLOCKS, THREADS>>>(d_prev_neurons,
        d_prev_errors, d_errors, d_weights, d_slopes);

    BLOCKS = 1;
    THREADS = num_prev_neurons;
    activate_errors<<<BLOCKS, THREADS>>>(d_prev_neurons, d_prev_errors);
}
```

Time	**Name**
1m57s	Total
2.61s	cudaMemcpy HToD
1.16s	cudaMemcpy DToH
781.34ms	direct_compute_entire_layer_kernel
619.93ms	direct_compute_neuron_sum_for_entire_layer_kernel
569.58ms	activate_errors
487.05ms	backpropagate_MSE_parallel_kernel
376.69ms	compute_MSE_parallel_kernel
282.42ms	direct_initialise_neuron_sum

Here we obtain a run time of 1min 57sec which is almost 20 times better than the runtime obtained when the two functions where parallelized individually. Hence we are successfull in reducing the time taken by our bottle-neck to a high extent.

4 Conclusion and Results

4.1 Performance on Mushroom Dataset

All through the previous sections we have used mushroom dataset to compare the performance of our different parallel implementations. Along with this it is important to see, how is our parallel implementation working as compared to our sequential implementation. The following table and plot summarises the total execution time (converted to seconds) of our parallel implementations starting from the most naive parallelization to the most optimal one as well as the execution time of the sequential implementation.

Time	**Name**
1.68s	Sequential
559.724s	fann_run_trial_1
91.226s	fann_run_trial_2
47.094s	fann_run_trial_3
145.927s	compute_MSE
123.467s	backpropagate_MSE
137.9s	slopes_update
119.058s	merge_backprop_and_slopes

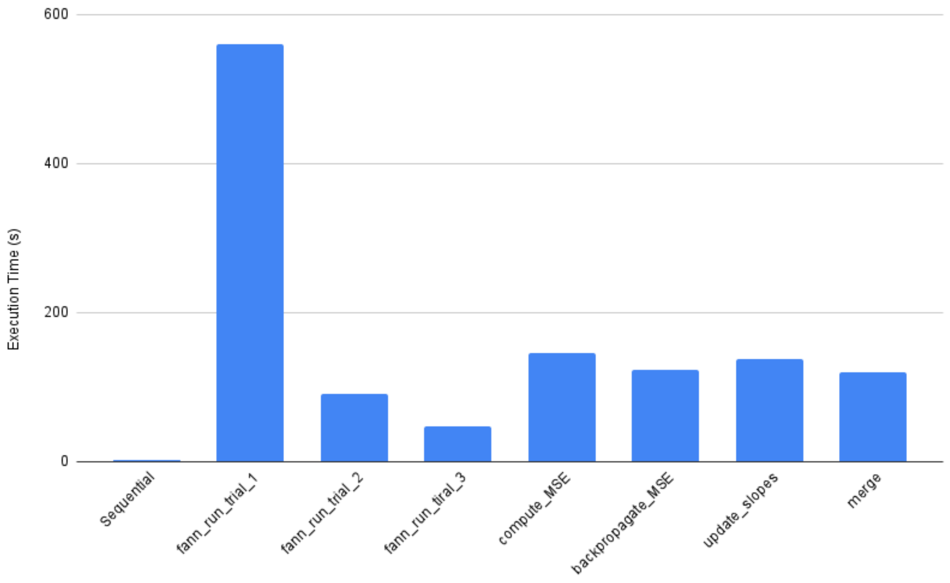


Fig. Execution time throughout the trials

We see that as we kept parallelizing our functions cumulatively, the execution time has decreased and perforance increased. There were times when the performance decreased, the reason for the same has been described in the respective sections and the requisite optimizations have been made.

We also note that our parallel implementation is not optimized enough that it performs better than the sequential implementation. The profile of our final parallel implementation shows that **`cudaMemcpy`** HToD is the bottle neck in terms of the GPU activities and **`cudaMalloc`** is the bottle neck in terms of the API calls made. This is depicted in the following plots of the profile of our program.

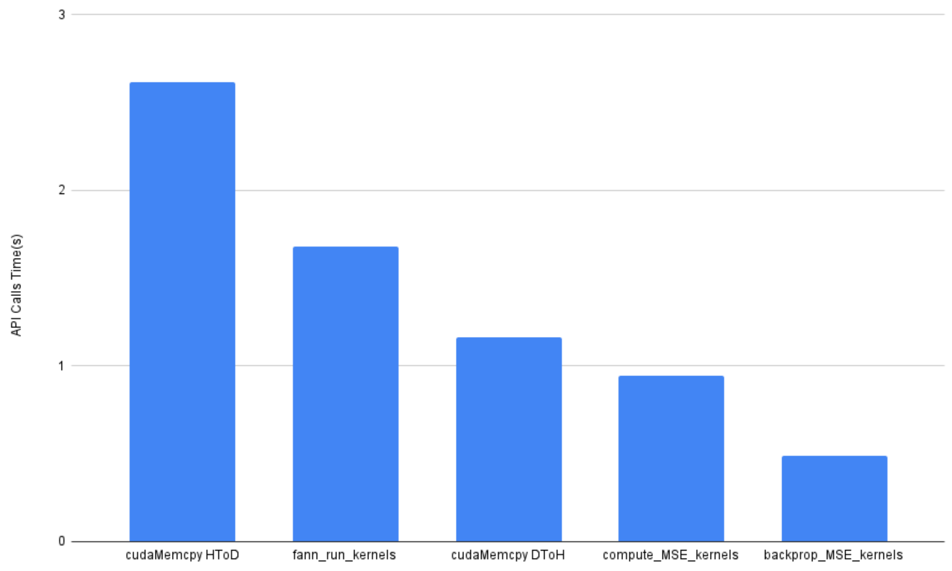


Fig. GPU Activities

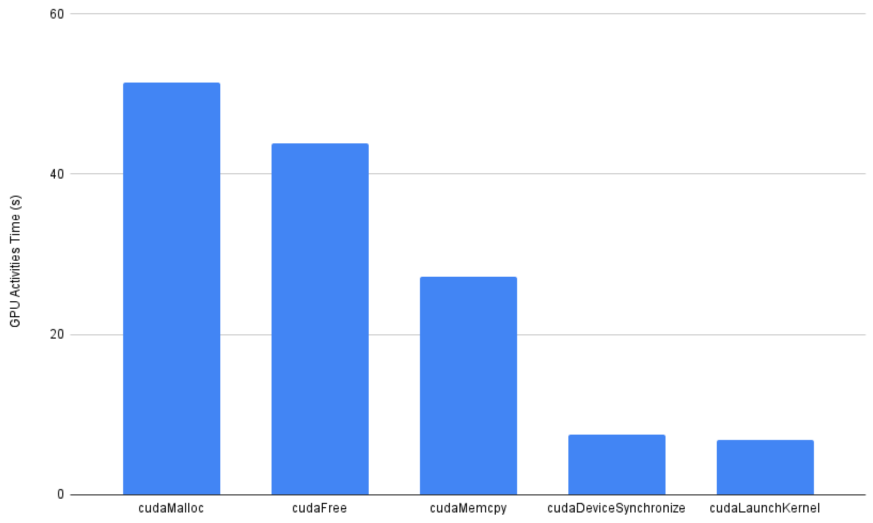


Fig.API Calls

By this we can say that our parallel implementation can be optimized even more if we further reduce the number of times we copy data back and forth. This is discussed briefly in section 5.

4.2 Performance on other Datasets

The following table and plot summarizes the performance of our final parallel implementation on different data sets.

	Robot	**Mushroom**	**Gene**
Sequential	1.059s	1.68s	9.18s
Parallel	47.58s	119.058s	798.048
Speedup	0.02225	0.01411	0.01150

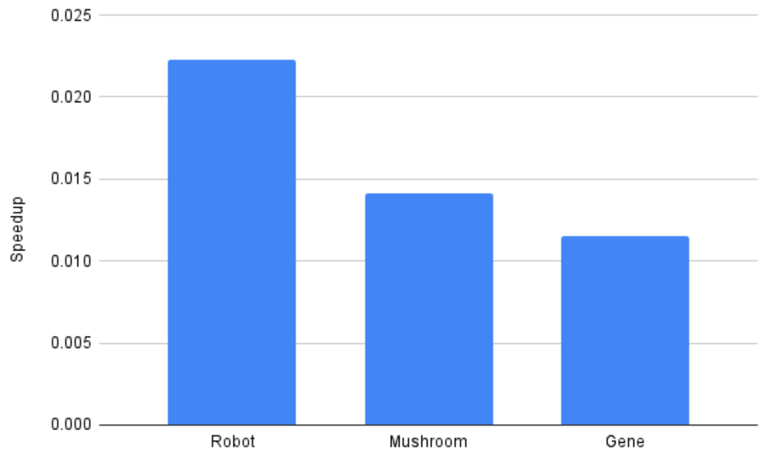


Fig. Speedups for different datasets

This clearly shows that we get maximum speedup for the robot dataset. Presently we are getting a speedup which is less than 1 for all the datasets. This means that our parallel implementation is not as good as the sequential implementation and needs more optimization. So we describe some methods to further optimize our parallel implementation.

5 Further Optimizations

Clearly our model is not performing better than the sequential version, hence we wish to optimize it further. Here we describe some approaches that we can use to further optimize the parallel implementation.

As we know that **`cudaMemcpy`** and **`cudaMalloc`** are the bottle necks fir our program hence we have to reduce the number of calls we make to these functions. We can use the following methods to do this.

- As the kernel launch configuration of feed forward run and Compute MSE is exactly the same, we can merge both of these together just like we did with backpropagate MSE and slopes update.
- We can further merge all the four functions together by calling the **`cudaMemcpy`** and **`cudaMalloc`** functions just once. Copy data to a global device variable and use this in all the four operations. This significantly will reduce the number of calls to these functios.