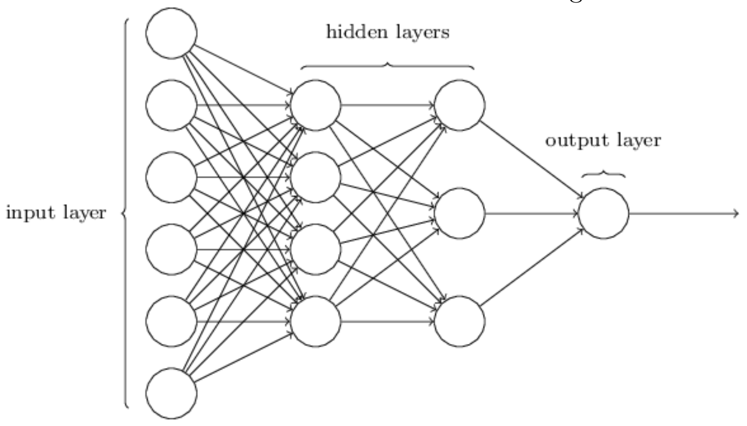
Backpropagation algorithm using GPU

Christophoros Bekos (mpekchri@auth.gr)

October 5, 2017

1 Introduction

In this project we are called to parallelize backpropagation algorithm , using cuda and pthreads . But before we step into the code , it would be useful to gain a better understanding of how a neural network operates , and how backpropagation is used in order to learn such a network .
The most common structure of a neural network is shown in the figure bellow :



As you can see , there are three kind of layers (each layer includes a number of nodes) : input, hidden and output layers . The output of each node of layer i ,is forwarded in the input of all nodes in layer i + 1 . Also each individual input to a node is doesn't affect it's output the same way than rest of inputs , instead , it is being multiplied with a weight factor (which is individual for each input and each node) .
Nodes who belong in layer 1 are called passive nodes , and the just pass their input value to the next layer . The rest of the nodes computes it's output according to :

$$output = sigmoid(\sum_{0}^{N-1}(input * weights + bias))$$

where input and weights are vectors of length N (N = number of inputs to our node) and bias is a single ,individual for each node ,value . Sigmoid function is defined as : $sigmoid(z) = \frac{1}{1+e^{-z}}$.
In summary , if we want to describe such a system we need to define :

- **w** : a 3-D array , where w[i] will contain all the weights we need between layers i and i+1 . Following this logic , w[i][j] contains the vector of weights ,that will be multiplied by inputs in node j (node j belongs to layer i+1) .
- **b** : a 2-D array , where b[i] contains the biases of all nodes that belong to layer i .
- **a** : an 1-D vector , which contains all inputs to our network (length of a equals to N) .

1.1 Network's structure in pseudocode

In order to train such a neural network we may use the following pseudocode :
As you can see here we have also define some new matrices :

Algorithm 1 create-network

```
procedure CREATE-NETWORK
  sizes = [num-of-layers];
  // matrices definitions
  w = [sizes[num-of-layers][][];
  b = [sizes[num-of-layers][];
  a = [sizes[0];
  sigm-der = [sizes[num-of-layers][];
  delta = [sizes[num-of-layers][];
  for(i=0; i<num-of-layers; i++)
    w[i][j] = [sizes[i]*sizes[i+1]];
    b[i] = [sizes[i]];
    a[i] = [sizes[i]];
    sigm-der[i] = [sizes[i]];
    delta[i] = [sizes[i]];
  end-for
```

- **delta** : is a 2-D matrix which contains the error (desired output - our network's output) from each node , in every layer .
- **sigm-der** : is a 2-D matrix which contains sigmoid's function derivative to each input vector $((\sum_0^{N-1}(input * weights + bias))$ in every layer (except the input layer of course) . Sigmoid derivative can easily be computed in two steps :
  1. sigm[i][j] = $(\sum_0^{N-1}(input * weights + bias)$
  2. sigm-der[i][j] = $sigm[i][j] * (1 - sigm[i][j])$

The term simg[i][j] is being computed in feedforward() function (will be discussed later on this document) and since it's already cached (when computed) it would be smart to also compute sigmoid's derivative, and store it back to sigm-der array .

1.2 Train the network

The following pseudocode gives you an abstract view of how our network will be trained .

Algorithm 2 train-network

```
procedure TRAIN-NETWORK
  for(e=0 ; e<epochs; e++)
    for(b=0; b<batch-size; b++)
      (a[0],$y_{desired}$) = get-datasets();
      $\tilde{y}$ = feedforward(a);
      cost = find-cost-derivative($\tilde{y}$,$y_{desired}$);
      $d_L$ = compute-output-layers-error(cost);
      backpropagation($d_L$);
    end-for
    update-weights();
  end-for
```

In the previous algorithm , cost is vector size of (sizes[end-1] x 1) , while d_L has same size and represents the error of each node ,which belongs to output layer .

2 Parallelization of the algorithm using cuda

Implementing training in gpu presupposes data transfer between cpu and gpu. Some of those data (matrices w and b) will already have been transfered in gpu , and since their values remain unaltered until one batch (inner loop in algorithm 2) finishes , we only move those data in the beginning of each inner loop (right after update-weights() is completed). **in order to optimize the hole procedure we may start sending data to gpu (the one that already have been updated) before update-weights() finishes - this can be easily done by creating cuda streams and using cudaMemcpyAsync .** On the other hand in each inner loop's iteration we need to "read" again the new pair of (a[0],$y_{desired}$) , so that add some delay ,since we must move these data before feedforward() runs .

2.1 Important properties of cuda code

- **SM and threads execution**
  Someone who is newbie to cuda programming may suppose that all threads in a block run concurrently . Well that's false of course, only threads that belong in the same **warp** run concurrently , while the rest of the warps ,in the same block , execute in a pseudo-parallel way (actually schedulers in new gpus can pipeline 2 warps per cycle ,which means 64 threads per cycle). All you need to keep in mind is that threads ,which belong in the same warp, must always execute the same code **(avoid if or any other branch command)** . In reality if an SM executes one warp and a branch occurs , it will switch to another branch .
- **Shared memory**
  Since communication with global memory is usually highly inefficient , a cuda application must minimize global memory access : only transfer data once (from global to shared memory) , do all the computations using shared and then transfer back the results to global memory .
- **Efficient memory access**
  Threads must access parts of global memory in a continues way ,avoiding crossed data transfer .
- **Sufficient work on each thread**
  Well this is the real world , starting a kernel, a thread ,or a block has some cost, so we must ensure that we assign sufficient work to thread . Otherwise , even if our code looks (and it is) completely parallel ,it will need way more time to execute .
- **Data transfer from CPU to GPU**
  We always aim to use all the bus capacity , in order to achieve maximum optimization . In order to maximize data speed we may use pinned memory ,which allows faster data transfers between cpu - gpu and vice versa , but it requires not cached memory (in cpu of course) , so in our problem there isn't an obvious way to implement such a solution unless both loops (batch learning and the outer loop) are running on cuda . Unfortunately i ran out of time and didn't try this .

Having all these in mind we might proceed in cuda code (finaly !!) .

3 What has been done so far

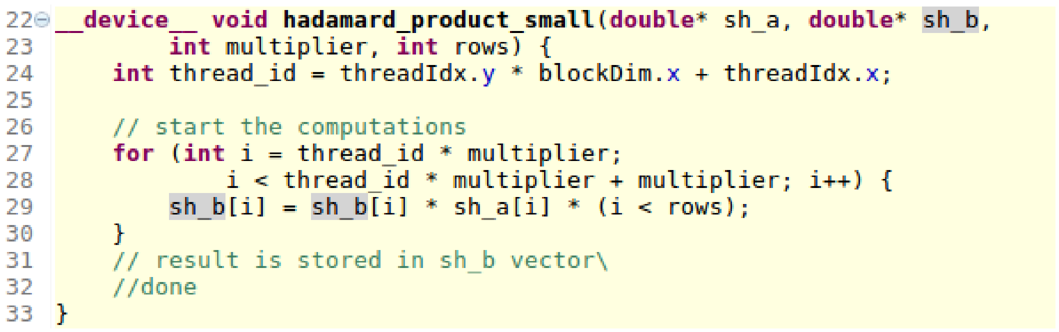
Unfortunately i got confused and in first time i parallelized **only backpropagation function** and not the rest of the training . This work was made in last 3 days, so the only think that has been done so far is :

- c++ definition of the network
- parallel algorithm for backpropagation function(fully debugged , also serial version exists in order to compare results).
- parallel algorithm for feedforward function (fully debugged , also serial version exists in order to compare results).

- a version of train function that is not complete, it only includes function for feedforward . The results that will be showed are based on this algorithm.

A full solution to our problem will be in my github page in less than 10 days ,you may check it out : *https://github.com/mpekchri*

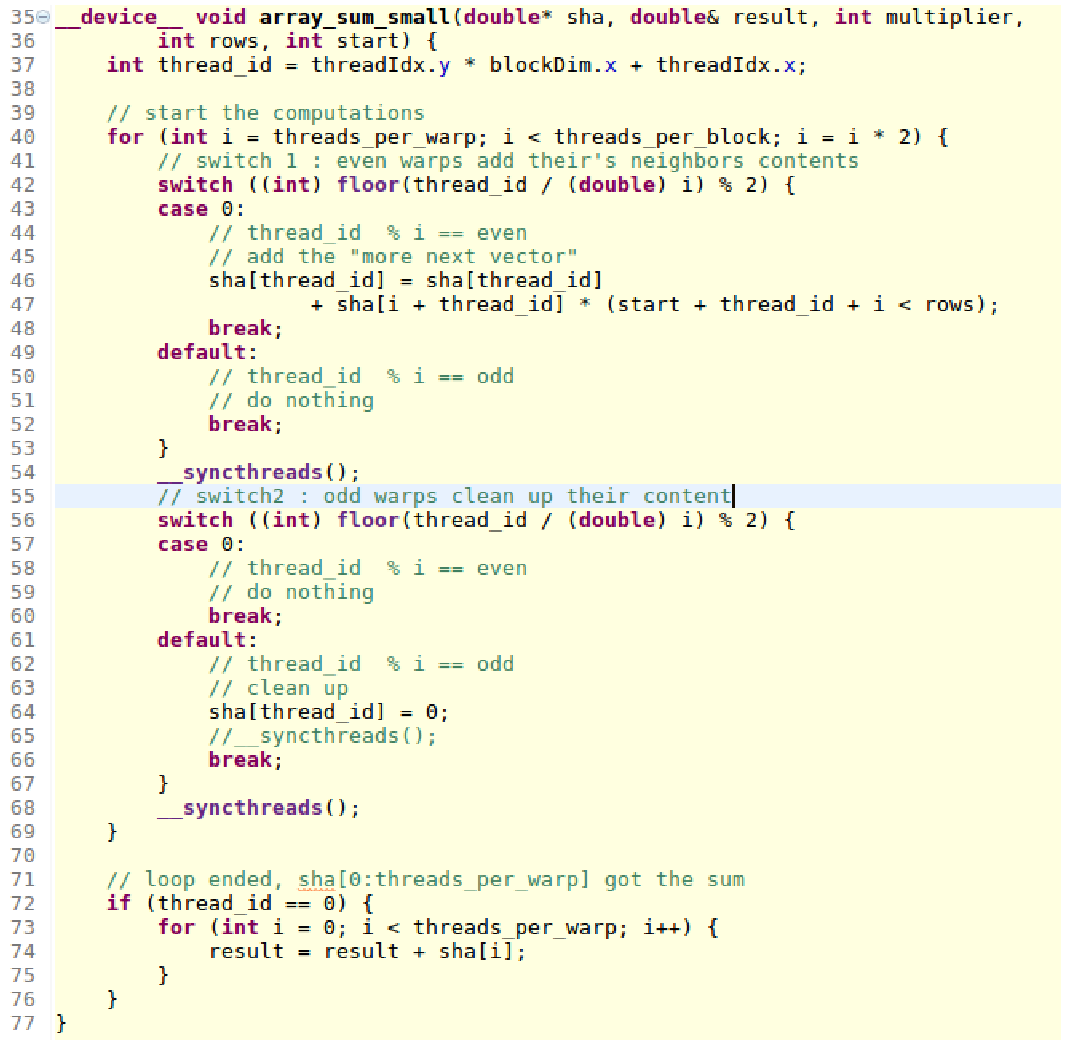
4 hadamard product - cuda function



```
__device__ void hadamard_product_small(double* sh_a, double* sh_b,
        int multiplier, int rows) {
    int thread_id = threadIdx.y * blockDim.x + threadIdx.x;

    // start the computations
    for (int i = thread_id * multiplier;
            i < thread_id * multiplier + multiplier; i++) {
        sh_b[i] = sh_b[i] * sh_a[i] * (i < rows);
    }
    // result is stored in sh_b vector\
    //done
}
```

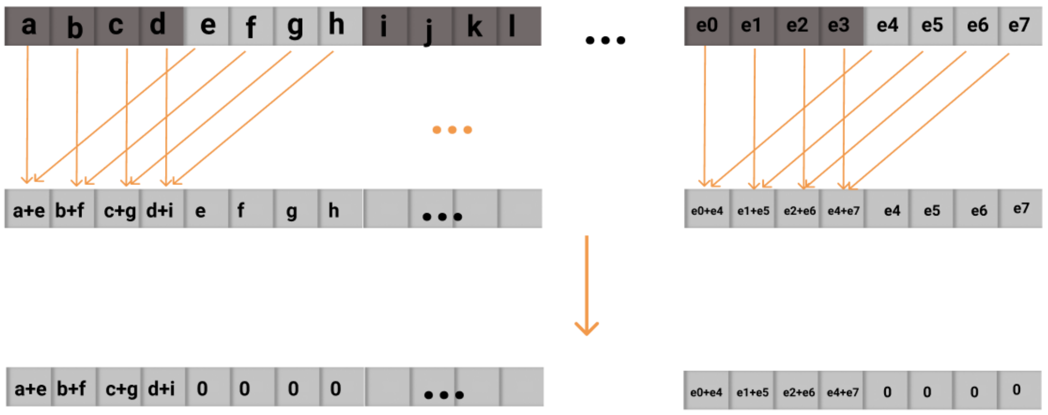
Our function accepts arguments sh-a and sh-b , which are the one part of the vectors , that we want to compute theirs hadamard product . Remember , multiple threads modify the same vector, and each thread must have sufficient work to do . Each thread executes 'multiplier' (third argument) commands and we use thread-id(as defined in code) to modify the correct values . Notice that each thread operates on a number of **continues** memory cells .
Still we got a serious problem : what if one thread tries to access a cell out of vector's length range? Well if we add an if statement to our code we really decrease our code efficiency .
Instead of such a solution , a common trick is to pad the array with zeros , or add a command like " * (i < rows) " , which mean that the computation will only occur for the desired cells ,while the rest will produce zeros (**no use of branch statement!!** Anyway, our shared array must be padded to avoid unexpected behavior.

5 sum vector's elements - cuda function

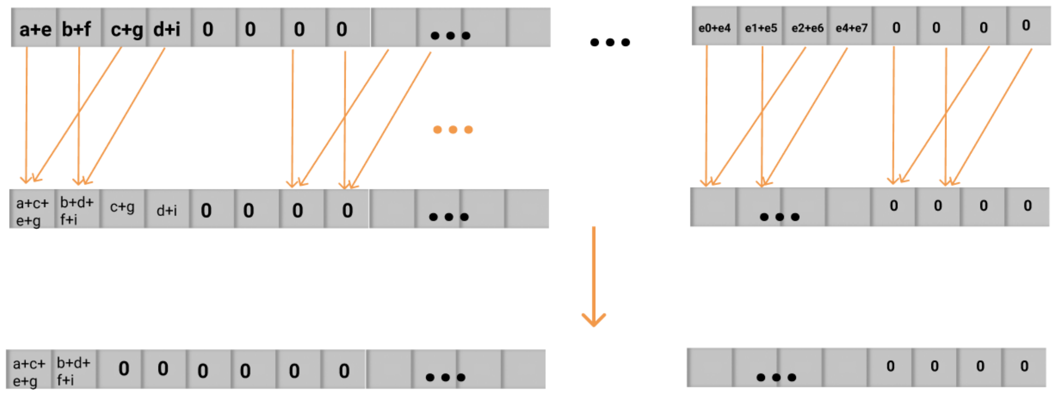


```
__device__ void array_sum_small(double* sha, double& result, int multiplier,
        int rows, int start) {
    int thread_id = threadIdx.y * blockDim.x + threadIdx.x;

    // start the computations
    for (int i = threads_per_warp; i < threads_per_block; i = i * 2) {
        // switch 1 : even warps add their's neighbors contents
        switch ((int) floor(thread_id / (double) i) % 2) {
        case 0:
            // thread_id  % i == even
            // add the "more next vector"
            sha[thread_id] = sha[thread_id]
                    + sha[i + thread_id] * (start + thread_id + i < rows);
            break;
        default:
            // thread_id  % i == odd
            // do nothing
            break;
        }
        __syncthreads();
        // switch2 : odd warps clean up their content
        switch ((int) floor(thread_id / (double) i) % 2) {
        case 0:
            // thread_id  % i == even
            // do nothing
            break;
        default:
            // thread_id  % i == odd
            // clean up
            sha[thread_id] = 0;
            //__syncthreads();
            break;
        }
        __syncthreads();
    }

    // loop ended, sha[0:threads_per_warp] got the sum
    if (thread_id == 0) {
        for (int i = 0; i < threads_per_warp; i++) {
            result = result + sha[i];
        }
    }
}
```

Well this is a bit more complex , since we must take advantage of the concurrent execution of threads in a warp . The idea is explained by the figure below :


a b c d e f g h i j k l
...
e0 e1 e2 e3 e4 e5 e6 e7
a+e b+f c+g d+i e f g h ...
e0+e4 e1+e5 e2+e6 e4+e7 e4 e5 e6 e7
a+e b+f c+g d+i 0 0 0 0 ...
e0+e4 e1+e5 e2+e6 e4+e7 0 0 0 0

Imagine that each thread must compute 'multiplier' = 4 commands . We define a loop ,at each step of which, we divide threads in two teams : the one that will add it's own contents with the one of it's "neighbor" and the other that won't execute any commands . Then the first team doesn't do any computations , while the second one zeroes it's own elements .



In the very next iteration of our loop we define different thread teams as shown in the figure above
.
It must be mentioned that all "teams" consist of 32 or 64 or 128 , ... etc , threads .This way we ensure that all threads in a warp will execute the same code ,increasing our code efficiency .
When only one "team" remains we stop the iteration and only one thread will add the 32 first values of the result vector .

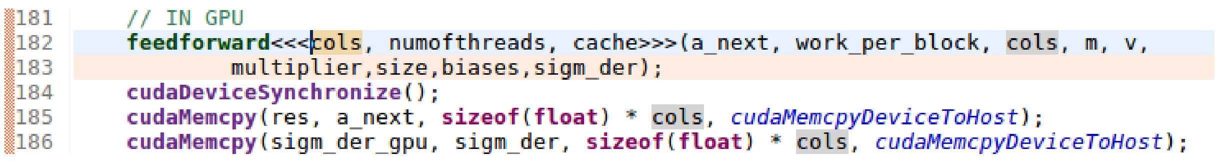
6 Handling data in a continues way

You may notice the variable **multiplier** in my scripts . Since a lot of times we need one thread to do more than one "job" , we use this variable to define how much tasks a thread must execute . Be aware that all these tasks must relate to continuous memory segments . So when by multiplier and the excising indexing,our threads transfer data from/to global memory / shared in a continues way ,and also refer to shared's indexes in a continues way (although indexing in shared is of less importance).

7 Final code

As said before the initial attempt was a parallelization of just Backpropagation function and this one (including initial report) are in "old-project" folder . The network's structure written in cpp is included in "network-in-cpp" . The following scripts are in "correct-version" folder.

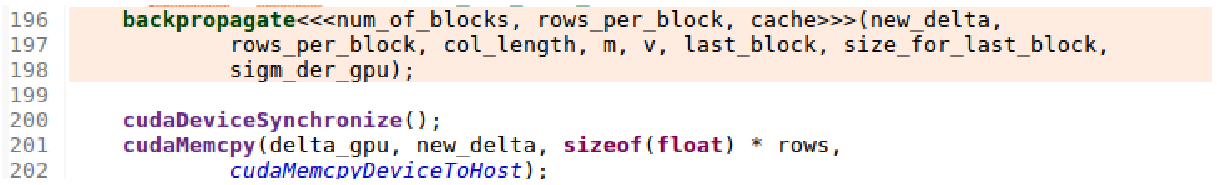
7.1 Feedforward function



```
    // IN GPU
    feedforward<<<cols, numofthreads, cache>>>(a_next, work_per_block, cols, m, v,
            multiplier,size,biases,sigm_der);
    cudaDeviceSynchronize();
    cudaMemcpy(res, a_next, sizeof(float) * cols, cudaMemcpyDeviceToHost);
    cudaMemcpy(sigm_der_gpu, sigm_der, sizeof(float) * cols, cudaMemcpyDeviceToHost);
```

Feedforward will use one block for each column of w (that will be multiplier by vector a) . All blocks will produce a single result and since we got number of blocks = number of columns we construct a vector size of num-of-columns. We will add this vector with biases and we got the next a (input to next layer).
Code can be found in **feedforward.cu**

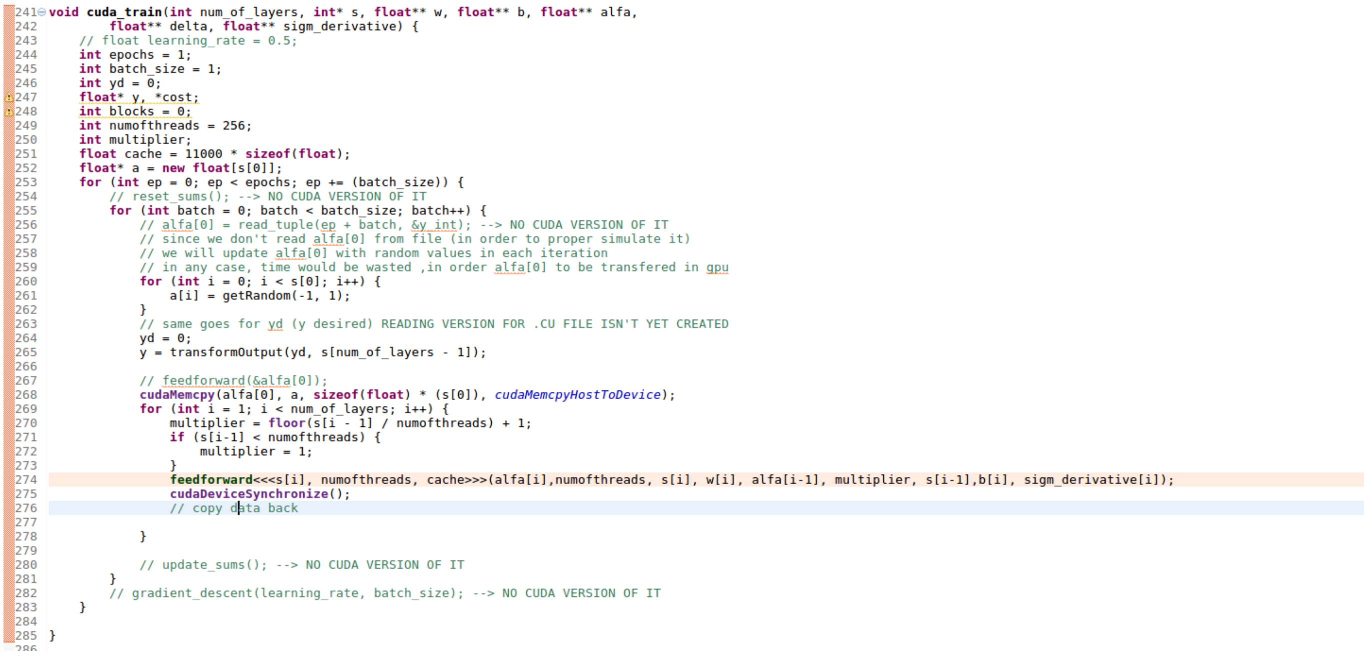
7.2 Backpropagation function



```
    backpropagate<<<num_of_blocks, rows_per_block, cache>>>(new_delta,
            rows_per_block, col_length, m, v, last_block, size_for_last_block,
            sigm_der_gpu);

    cudaDeviceSynchronize();
    cudaMemcpy(delta_gpu, new_delta, sizeof(float) * rows,
            cudaMemcpyDeviceToHost);
```

In backpropagation we use a different number of threads (may vary according to researchers choice) . As a result of this, one block will deal with multiple w's rows . In this approach each thread of one block becomes responsible to multiply a single row of w , with vector delta (same for all threads).
Code can be found in **backpropagate.cu**

7.3 Train function



```
void cuda_train(int num_of_layers, int* s, float** w, float** b, float** alfa,
        float** delta, float** sigm_derivative) {
    // float learning_rate = 0.5;
    int epochs = 1;
    int batch_size = 1;
    int yd = 0;
    float* y, *cost;
    int blocks = 0;
    int numofthreads = 256;
    int multiplier;
    float cache = 11000 * sizeof(float);
    float* a = new float[s[0]];
    for (int ep = 0; ep < epochs; ep += (batch_size)) {
        // reset_sums(); --> NO CUDA VERSION OF IT
        for (int batch = 0; batch < batch_size; batch++) {
            // alfa[0] = read_tuple(ep + batch, &y_int); --> NO CUDA VERSION OF IT
            // since we don't read alfa[0] from file (in order to proper simulate it)
            // we will update alfa[0] with random values in each iteration
            // in any case, time would be wasted ,in order alfa[0] to be transfered in gpu
            for (int i = 0; i < s[0]; i++) {
                a[i] = getRandom(-1, 1);
            }
            // same goes for yd (y desired) READING VERSION FOR .CU FILE ISN'T YET CREATED
            yd = 0;
            y = transformOutput(yd, s[num_of_layers - 1]);

            // feedforward(&alfa[0]);
            cudaMemcpy(alfa[0], a, sizeof(float) * (s[0]), cudaMemcpyHostToDevice);
            for (int i = 1; i < num_of_layers; i++) {
                multiplier = floor(s[i - 1] / numofthreads) + 1;
                if (s[i-1] < numofthreads) {
                    multiplier = 1;
                }
                feedforward<<<s[i], numofthreads, cache>>>(alfa[i],numofthreads, s[i], w[i], alfa[i-1], multiplier, s[i-1],b[i], sigm_derivative[i]);
                cudaDeviceSynchronize();
                // copy data back

            }

            // update_sums(); --> NO CUDA VERSION OF IT
        }
        // gradient_descent(learning_rate, batch_size); --> NO CUDA VERSION OF IT
    }

}
```

As said before ,while the serial code is complete , parallel code is not . I had time only to apply (without bugs) feedforward function (as defined before) in our train loop . So we actually compare times according to serial and parallel feedforward .
Running **ergasia4-final.cu** we get the results :

- acceleration 165.993989 %
- acceleration 168.351284 %
- acceleration 166.933783 %

Even if such an acceleration is not enough for a cuda project, since we have only completed one part of it, we may expect better results by a fully parallelized code.