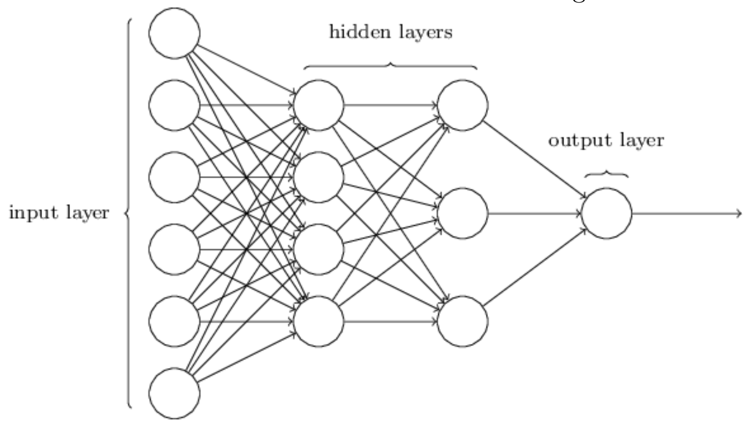
Parallelization of Backpropagation algorithm in cuda

Christophoros Bekos (mpekchri@auth.gr)

October 2, 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 .
There is no need to further explain most of functions in algorithm 2 , because this project's task concerns only backpropagation algorithm :

So all the work is done in w[i]*delta[i+1] (where w[i] is a sizes[i] x sizes[i+1] matrix ,while delta[i+1] is a size of sizes[i+1] vector) . Having all these in mind you can easily understand that temp will be a vector of length sizes[i] , so does sigm-der[i] and this way (hadamard product means pointwise

Algorithm 3 backpropagation

```
1: procedure BACKPROPAGATE-ERROR
2:     delta[end-1] = $d_L$ ;
3:     for(i=num-of-layers-2; i>=0; i++)
4:         temp = w[i]*delta[i+1];        delta[i] = hadamard-product(temp,sigm-der[i]);    end-for
```

multiplication) we compute (backpropagate) the error delta[i] ,for the rest of the layers .

2 Parallelization of the algorithm using cuda

Implementing backpropagation in cuda presupposes data transfer between cpu and gpu . Since backpropagation() function does require w, b and sigm-der matrix ,as well as d_L we will need to transfer those data . It must be mentioned , that w and b matrices contents don't change until we reach update-weights() function (in algorithm 3) .
This is a very useful observation , since we may start transferring w and b right after we finish the execution of update-weights() and by the time we reach backpropagation() function data will have already been transfered .
On the other side d_L and sigm-der vectors change their values in each batch-size iteration , so we need to wait a few time until their transfer is completed .
Also when we finish our job in cuda (kernel finishes) the only data we require back to cpu is the delta vector (which is not considered big).
So , as far as we consider data transfers the updated (algorithm 2) looks like :

Algorithm 4 train-network-using-gpu

```
1: procedure TRAIN-NETWORK-USING-GPU
2:     for(e=0 ; e<epochs; e++)
3:         for(b=0; b<batch-size; b++)
4:             (a[0],$y_{desired}$ = get-datasets();
5:             $\tilde{y}$ = feedforward(a);
6:             cost = find-cost-derivative($\tilde{y}$,$y_{desired}$);
7:             $d_L$ = compute-output-layers-error(cost);
8:             start-timer();
9:             send-$d_L$-sigmder-in-gpu() − synchronous ;
10:            make sure that w-b have arrived in gpu ;
11:            backpropagation-in-gpu() − kernel call ;
12:            wait until kernel finishes ;
13:            retrieve-$d_L$-back-to-cpu ;
14:            stop-timer();
15:        end-for
16:        update-weights();
17:        send-w-b-in-gpu() − asynchronous ;
18:    end-for
```

Functions start and stop timer may be used (here are defined in an abstract way) to count kernel's execution time and compare it to the serial code .

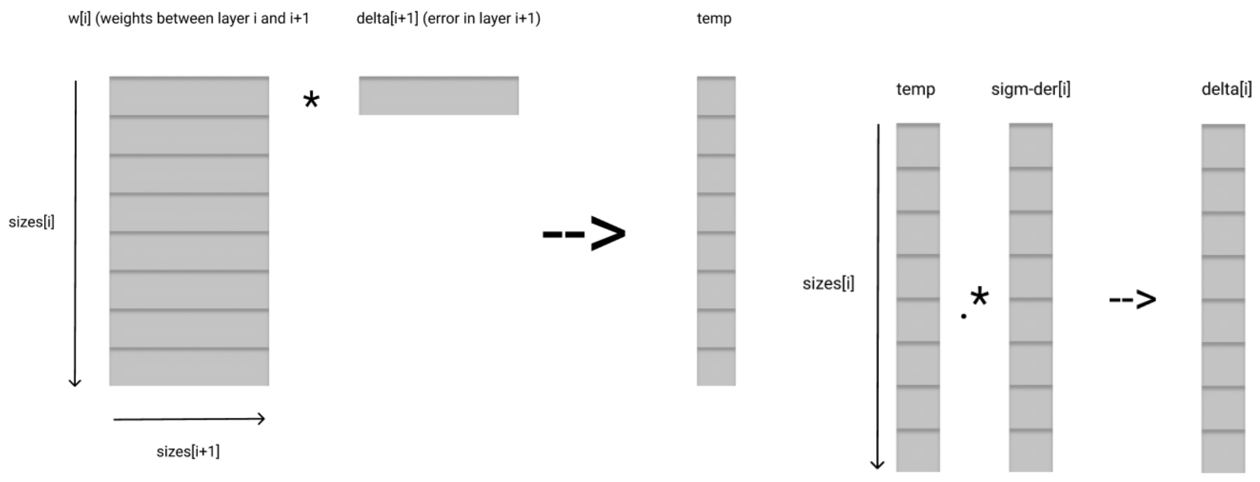
3 Implementing pthreads to accelerate data transfer

– TO BE FILLED –

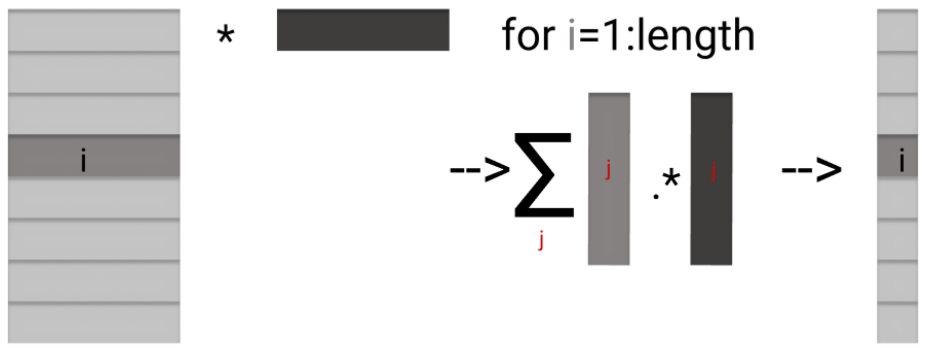
4 Cuda code for backpropagation

As we have already discussed , backpropagation is actually the combination of two operations : one array * vector multiplication and one hadamard product between two vectors . Actually array-vector multiplication can be further analyzed in even simpler operations : one hadamard product (which we would already have defined) and a summarize of the previous operation into one single value , in order to compute one element,of the result (result here is a vector) .
Just because it's easier to describe the hole procedure with images rather than words , we provide the following figures :

4.1 backpropagation procedure


w[i] (weights between layer i and i+1
delta[i+1] (error in layer i+1)
temp
sizes[i]
sizes[i+1]
*
-->
temp
sigm-der[i]
delta[i]
sizes[i]
.*
-->

4.2 vector matrix multiplication procedure



Summarizing all we need is to define two cuda functions (kernels) : one that computes **hadamard product (.*) between two vectors** , and one that **summarizes (in parallel) all elements of a vector** .

4.3 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 .

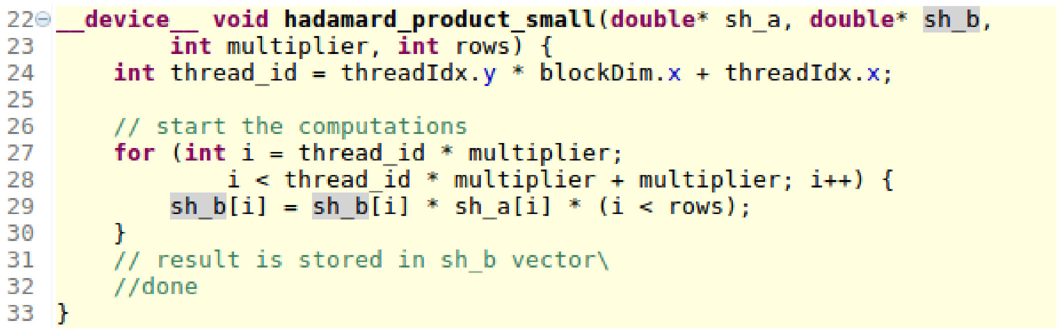
- **Efficient memory access**
  Threads must access parts of global memory in a continues way ,avoiding crossed data transfer .

- **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 .

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

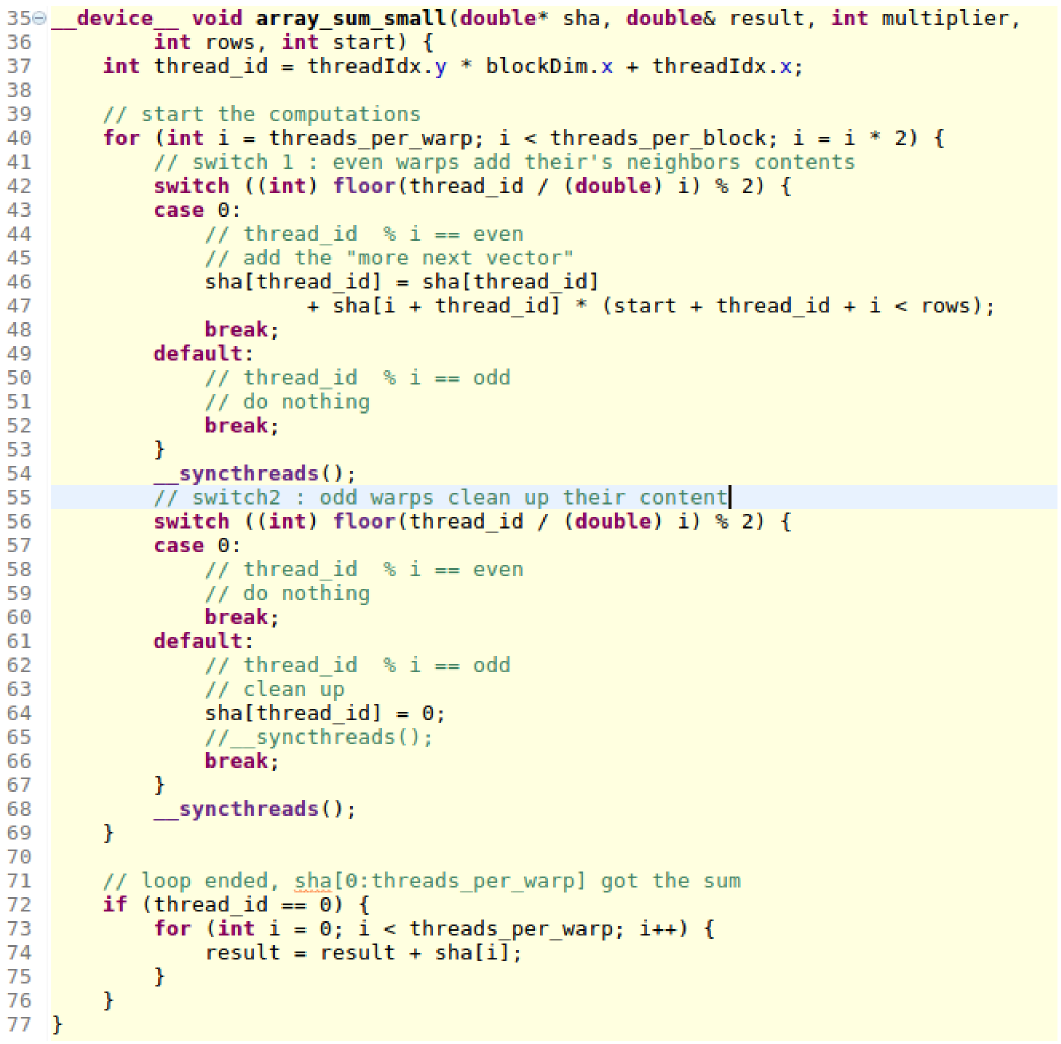
5 hadamard product - cuda function



```
__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
}
```

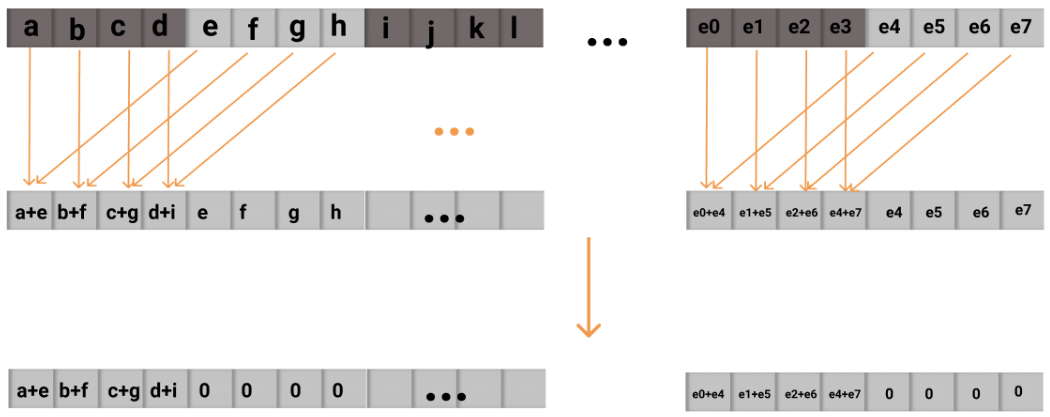
Our function accepts arguments sh-a and sh-b , which are the one part of the vectors , that we want to compute theirs hadamard product . Remember , multiple threads modify the same vector, and each thread must have sufficient work to do . Each thread executes 'multiplier' (third argument) commands and 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.

6 sum vector’s elements - cuda function



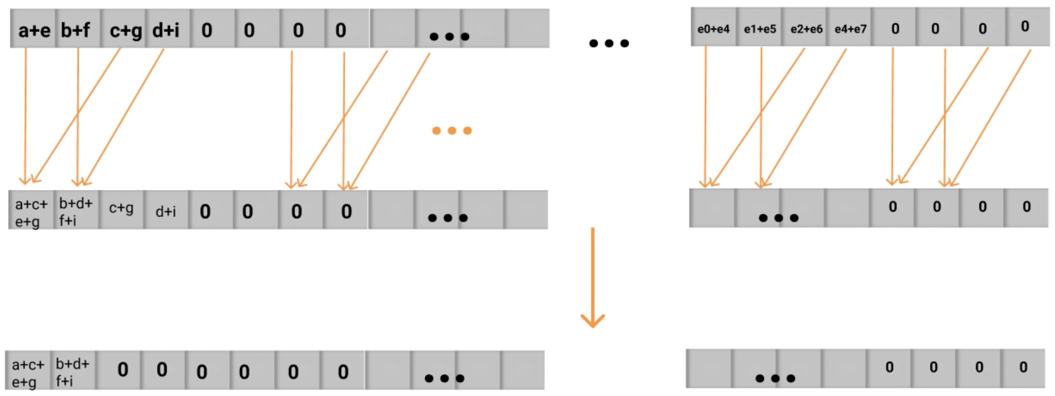
```
__device__ void array_sum_small(double* sha, double& result, int multiplier,
        int rows, int start) {
    int thread_id = threadIdx.y * blockDim.x + threadIdx.x;

    // 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 .



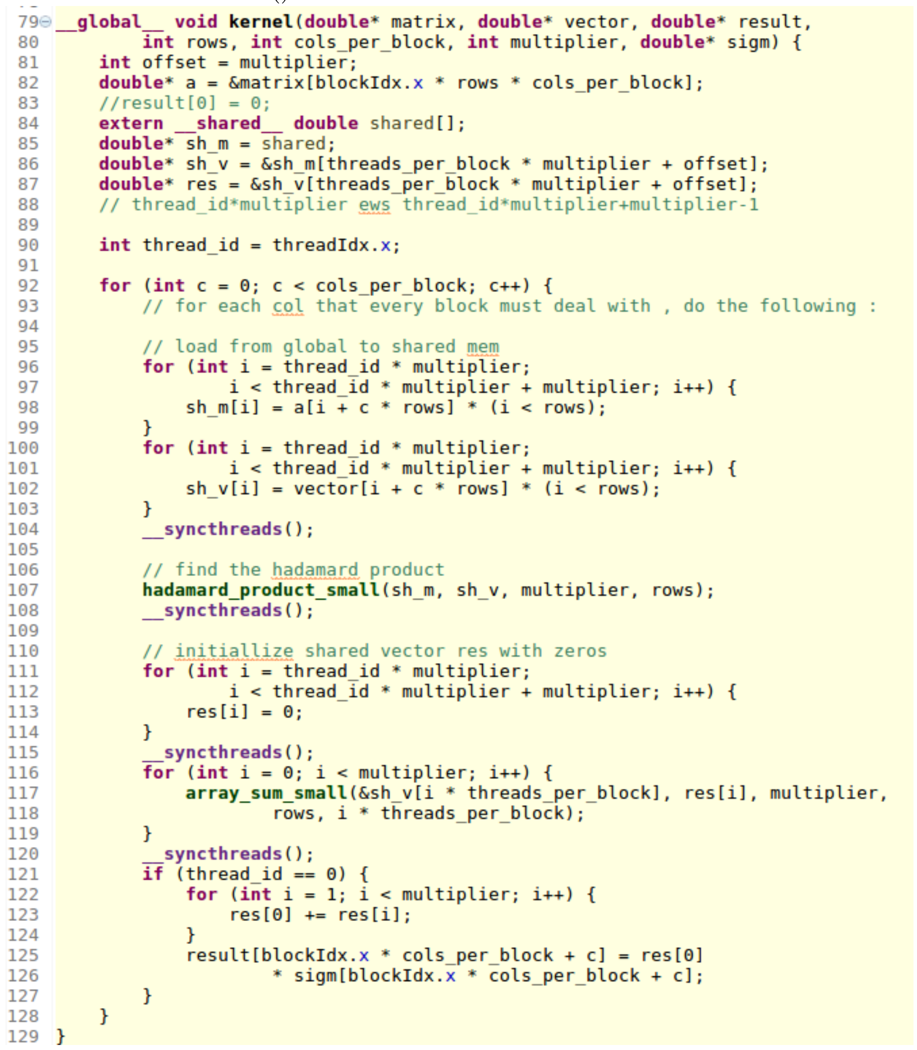
In the very next iteration of our loop we define different thread teams as shown in the figure above .

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 .

7 Our cuda kernel

Our cuda kernel is named kernel() and it is shown bellow :

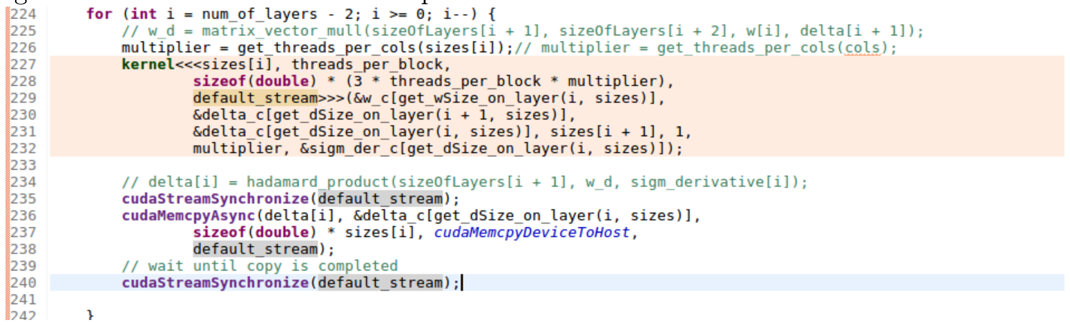


```
__global__ void kernel(double* matrix, double* vector, double* result,
        int rows, int cols_per_block, int multiplier, double* sigm) {
    int offset = multiplier;
    double* a = &matrix[blockIdx.x * rows * cols_per_block];
    //result[0] = 0;
    extern __shared__ double shared[];
    double* sh_m = shared;
    double* sh_v = &sh_m[threads_per_block * multiplier + offset];
    double* res = &sh_v[threads_per_block * multiplier + offset];
    // thread_id*multiplier ews thread_id*multiplier+multiplier-1

    int thread_id = threadIdx.x;

    for (int c = 0; c < cols_per_block; c++) {
        // for each col that every block must deal with , do the following :

        // load from global to shared mem
        for (int i = thread_id * multiplier;
                i < thread_id * multiplier + multiplier; i++) {
            sh_m[i] = a[i + c * rows] * (i < rows);
        }
        for (int i = thread_id * multiplier;
                i < thread_id * multiplier + multiplier; i++) {
            sh_v[i] = vector[i + c * rows] * (i < rows);
        }
        __syncthreads();

        // find the hadamard product
        hadamard_product_small(sh_m, sh_v, multiplier, rows);
        __syncthreads();

        // initiallize shared vector res with zeros
        for (int i = thread_id * multiplier;
                i < thread_id * multiplier + multiplier; i++) {
            res[i] = 0;
        }
        __syncthreads();
        for (int i = 0; i < multiplier; i++) {
            array_sum_small(&sh_v[i * threads_per_block], res[i], multiplier,
                    rows, i * threads_per_block);
        }
        __syncthreads();
        if (thread_id == 0) {
            for (int i = 1; i < multiplier; i++) {
                res[0] += res[i];
            }
            result[blockIdx.x * cols_per_block + c] = res[0]
                    * sigm[blockIdx.x * cols_per_block + c];
        }
    }
}
```

Since it's the global function (called from cpu) is his own responsibility to arrange data transfer from global to shared memory (notice the offset in shared memory declaration for the reasons we mentioned above) .

8 The big picture of our algorithm

We do transfer all data in an asynchronous way , and when it's computed , we pass d_L to kernel and start computations . **One weakness of our algorithm is the fact that kernel it is not called only once ,but a few times in a loop , which of course is not optimal .** This will change in the next version ,that will be published in my github page (*https://github.com/mpekchri/backpropagation-in-cuda*) . Until now , even if its not optimal , results were way too good . So here is kernel call from cpu :



```
    for (int i = num_of_layers - 2; i >= 0; i--) {
        // w_d = matrix_vector_mull(sizeOfLayers[i + 1], sizeOfLayers[i + 2], w[i], delta[i + 1]);
        multiplier = get_threads_per_cols(sizes[i]);// multiplier = get_threads_per_cols(cols);
        kernel<<<sizes[i], threads_per_block,
                sizeof(double) * (3 * threads_per_block * multiplier),
                default_stream>>>(&w_c[get_wSize_on_layer(i, sizes)],
                &delta_c[get_dSize_on_layer(i + 1, sizes)],
                &delta_c[get_dSize_on_layer(i, sizes)], sizes[i + 1], 1,
                multiplier, &sigm_der_c[get_dSize_on_layer(i, sizes)]);

        // delta[i] = hadamard_product(sizeOfLayers[i + 1], w_d, sigm_derivative[i]);
        cudaStreamSynchronize(default_stream);
        cudaMemcpyAsync(delta[i], &delta_c[get_dSize_on_layer(i, sizes)],
                sizeof(double) * sizes[i], cudaMemcpyDeviceToHost,
                default_stream);
        // wait until copy is completed
        cudaStreamSynchronize(default_stream);

    }
```

Asynchronous data transfers are made using cudaMemcpyAsync function . Of course the script only compares backpropagation algorithm and the rest of code (for the neural network) is not yet created .

9 Results

Algorithm turns out to be completely inefficient for small data (because of multiple kernel calls) , but behaves better for big data (many inputs per layer) giving approximately 65 % acceleration (which is considered bad result for a cuda app) .
In this approach we choose to use 1 block for each w[i]'s row , which turned out to be a bad choice , we must modify it in the future . Algorithm will be furthermore optimized after the deadline :(

9.1 A last minute's improvement

The way algorithm (in folder slow-version) is written , only a few blocks are used in each iteration . Despite the fact that we also have to remove iterations (all must be computed in gpu) , changing this , gives us better results . **Please use scripts in folder last-version in order to evaluate this project .**
Indicative results are provided ,for the following networks: number of layers = 4
sizes[0] = 90000
sizes[1] = 90
sizes[2] = 90
sizes[3] = 10
accelaration –> 353.7 % number of layers = 3
sizes[0] = 783;
sizes[1] = 30;
sizes[2] = 10;
accelaration is -31.7 %
In the end code turns out to be good (but not good enough) and one of the most significant factors for performance is the choice of block number .

10 Further parallelization using pthreads

Well since profiling showed us that most of computational time is wasted in gpu calculations , first think we need to do is modify the code .
Pthreads could be used later, if there is bottleneck in data transfer . It would be really easy to implement them ,just split our data in k teams, create k+1 threads and whild main thread was executing some part of train function ,the rest of the threads would pass their data (continues in order to avoid cache misses) in gpu (using all avaiable busses) .
Of course the hole network can be further parallelized , but not in this project !!