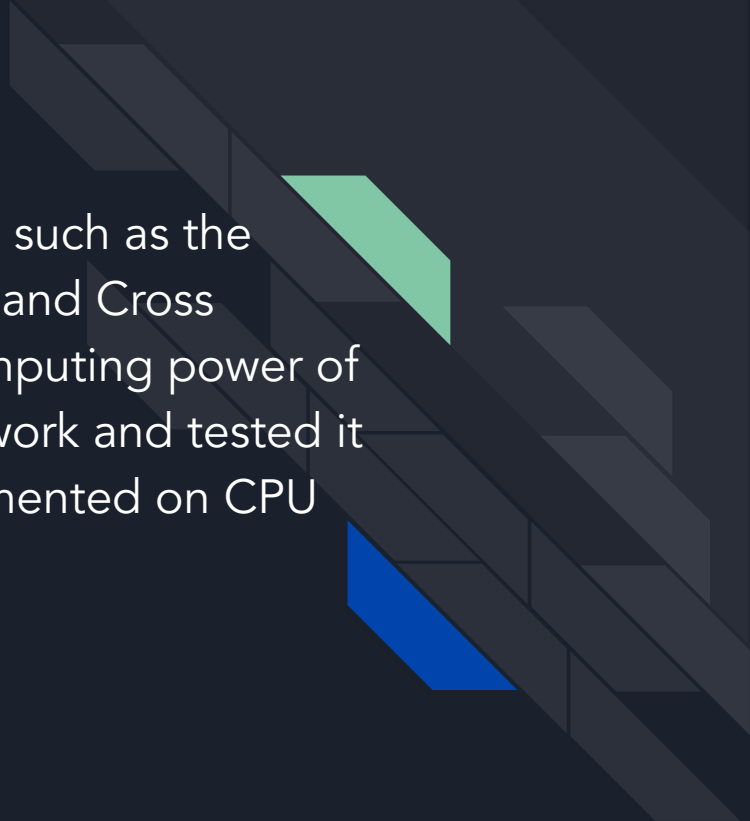


A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

Parallelizing Neural Networks

Neural networks are inherently parallel.

We implemented neural network layers such as the Linear(fully connected), ReLU, Sigmoid and Cross entropy loss to harness the parallel computing power of a GPU. We put them together in a network and tested it on a simple dataset. Layers also implemented on CPU for comparisons.



Code Structure





Classes

Matrix — Most of the neural network boils down to tensor operations, for which we just need a matrix(2nd order tensor).

Layers:

- 1) **Linear**
- 2) **ReLU(Rectified linear unit)**
- 3) **Sigmoid**

Cross-entropy loss: This will compute binary cross-entropy loss.

Forward and backward pass for all layers and the cost function.

FFN: Feed forward network, this is the class where we will put all layers together to construct and train a network.



Matrix Class

We want the matrix to be available both on host(CPU) and the device(GPU) memory. Most operations will be performed on device but we need to initialize the matrix on host, just because it's easier. Matrix class will manage memory allocation and make data transfer between host and device easier.

- 1) `AllocateMemory()`
- 2) `CopyHostToDevice()`
- 3) `CopyDeviceToHost()`

`cudaMalloc()` is called in 1)

`cudaMemcpy()` function is called for 2) & 3) in the required direction.

This all is managed within the Matrix class so that we don't have to worry about it otherwise.



Linear Layer

We implemented the Forward and Backward functions where most of the work(matrix multiplication) is done by CUDA kernels.

W is weights matrix, b is bias vector and A is input to this layer.

Forward: $Z = W.A + b$

Backward: Update the parameters by gradient descent using the backpropagated error from next layer and pass on the error to previous layer.

- 1) $dL/dA = [dL/dZ] \cdot [dZ/dA] = W^T \cdot dZ$
- 2) $dL/dW = [dL/dZ] \cdot [dZ/dW] = 1/m \cdot dZ \cdot A^T$
- 3) $dL/db = [dL/dZ] \cdot [dZ/db] = 1/m \cdot \sum_{i=0}^{m-1} \{ dZ^i \}$, m is batch size

Separate kernels for 1) and 2),3).



CUDA Kernel Implementation

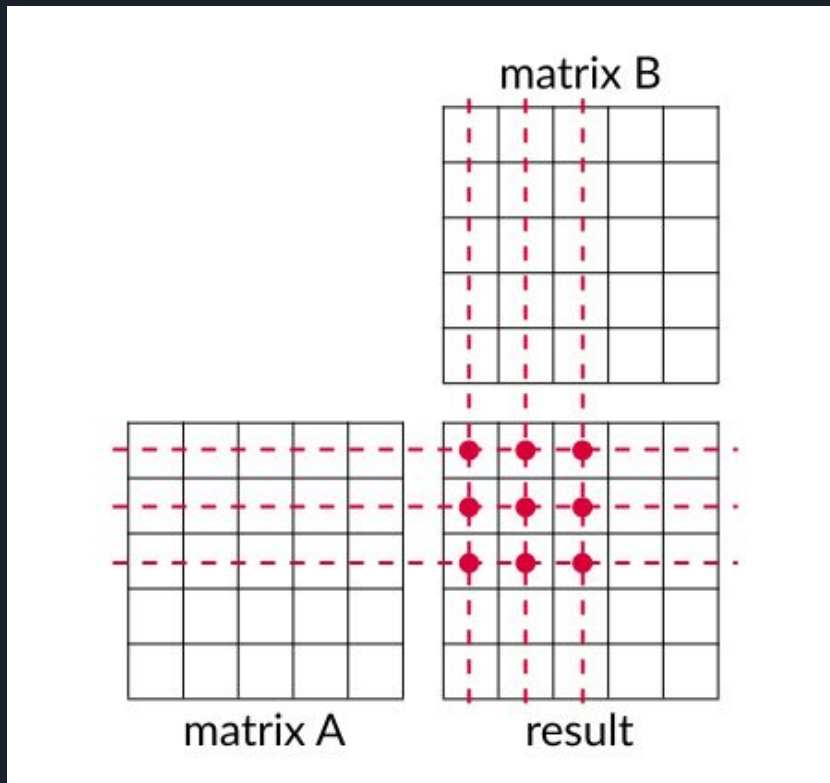
Most of computations in this layer are matrix multiplications.

We didn't want race conditions because it slows the kernel down, so two threads cannot write to the same location. But, the threads can read from the same location simultaneously, because they won't change during computations.

So, every thread calculates only a single element of the output matrix.

We fix the number of threads per block to be a 16x16 grid and decide the number of blocks according to the size of the output matrix.

Matrix Multiplication





CUDA Kernel Implementation

In the Backward() functions, we need to do matrix multiplication with transpose of matrices. We don't have to transpose them, we can just manage the indices accordingly when multiplying.

Example, for calculating $W^T \cdot dZ$, instead of taking a row of W and column of dZ , we take a column of W and a column of dZ .

When we update the bias, we have to sum over all rows of dZ . We can do this using only one thread, or our other option is the `atomicAdd()` function.

Both are undesirable and slow down the kernel as a lot of the threads need to wait. But, we don't have any option here.

In this case, we didn't use `atomicAdd()` and just used one thread to loop over a row.



ReLU Layer

Both activation layers we are implementing are very simple with element-wise operations.

Forward:

$$\begin{aligned}\text{ReLU}(x) &= 0 && \text{if } x < 0, \\ &= x && \text{if } x > 0;\end{aligned}$$

Backward:

For backpropagation, using the chain rule, we get the following formula:

$$\begin{aligned}dZ &= 0 && \text{if } x < 0, \\ &= dA && \text{if } x > 0;\end{aligned}$$



Sigmoid Layer

Sigmoid layer should compute sigmoid function for every matrix element.

Forward:

$$\text{sigmoid}(x) = \{e^x\} / \{1 + e^x\}$$

Backward:

According to the chain rule(dA is the error from next layer):

$$dZ = [\{dL\} / \{\text{sigmoid}(x)\}] [\{\text{sigmoid}(x)\} / \{dx\}] = dA \cdot \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$$

Where L is a cost function and $\{dL\} / \{\text{sigmoid}(x)\}$ is the error introduced by sigmoid layer.



CUDA Kernel Implementation

First we calculate index for current thread, then we check if this index is within matrix bounds.

Every CUDA thread compute a single output.

We have fixed the number of threads per block to 64 and decide the number of blocks accordingly.

Both the activation functions are element-wise so each thread just computes for one element of the output.



Cross-Entropy

We have decided to use binary cross-entropy as cost function.

It returns gradient according to network output and our target labels. Binary cross-entropy is defined with following equation:

$$\text{BCE} = \frac{1}{m} \cdot \sum_{i=0 \text{ to } m} \{ y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y}) \}$$

And by calculating its derivative we compute gradient as:

$$\text{Gradient BCE} = [y \cdot 1y] / [\hat{y} \cdot 1\hat{y}]$$

Where by \hat{y} we denote output values and by y the ground truth labels.

We use a built-in math function `logf` here.



FFN Class

Feed forward network, this is the class where we will put all layers together to construct and train a network.

We add new layers using Add() function. The pointers to all the layer objects are stored in a vector in this class.

The most important functions here are Forward() and Backward() which will just call the Forward() and Backward() functions of all the layers sequentially.

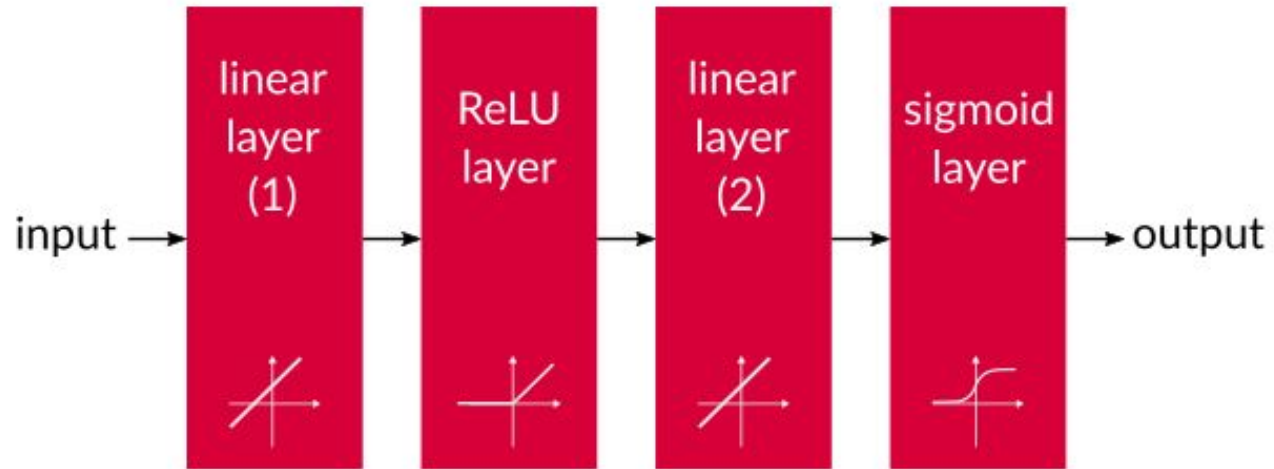
A Train() function has been implemented in this class which takes an instance of the Dataset class, learning rate, number of epochs and a boolean which tells whether to run the CPU only version of all functions.

We have implemented the CPU only functions for all layers as well.

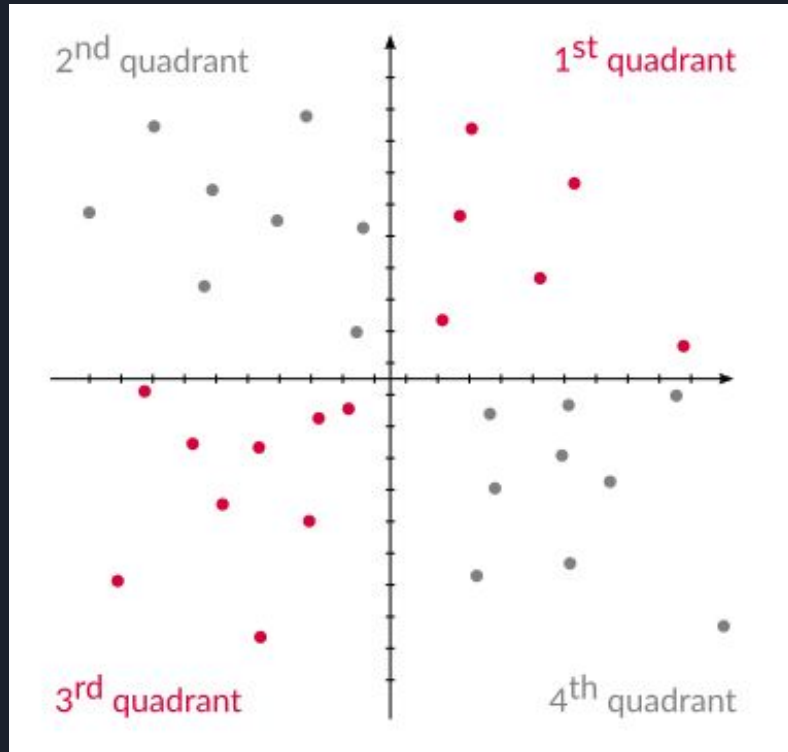
Training



Network architecture



Dataset



```
atharva@Atharva-PC:~/Documents/HPSC/cudaNeuralNets$ nvcc -std=c++11 train.cu -o train
atharva@Atharva-PC:~/Documents/HPSC/cudaNeuralNets$ ./train
Epoch: 0
Test Loss: 0.692727
Test Accuracy: 0.480469
Epoch: 1
Test Loss: 0.685889
Test Accuracy: 0.734375
Epoch: 2
Test Loss: 0.627985
Test Accuracy: 0.882812
Epoch: 3
Test Loss: 0.422742
Test Accuracy: 0.963867
Epoch: 4
Test Loss: 0.258762
Test Accuracy: 0.987305
Epoch: 5
Test Loss: 0.189306
Test Accuracy: 0.99707
Epoch: 6
Test Loss: 0.15398
Test Accuracy: 0.999023
Epoch: 7
Test Loss: 0.132437
Test Accuracy: 0.999023
Epoch: 8
Test Loss: 0.117702
Test Accuracy: 1
Epoch: 9
Test Loss: 0.106858
Test Accuracy: 1
```

Profiling



CPU only code profiling in callgrind(valgrind)


callgrind.out.29201 [./train]

Open < Back > Forward ^ Up % Relative Cycle Detection Relative to Parent <> S

Flat Profile

Search:

Incl.	Self	Called	Function	Location
99.99	0.00	(0)	0x00000000000000c...	ld-2.23.so
99.96	0.00	11	_dl_runtime_resolv...	ld-2.23.so: dl-trampoline.h
99.96	0.00	1	_start	train
99.96	0.00	(0)	(below main)	libc-2.23.so: libc-start.c
99.95	0.00	1	main	train
99.07	0.00	1	FFN::Train(Dataset, ...	train
66.11	50.16	171 380 979	Matrix::operator()(i...	train
57.70	0.00	1 990	FFN::Backward(Mat...	train
48.67	8.41	3 980	Linear::BackwardC...	train
41.24	0.00	2 000	FFN::Forward(Matri...	train
33.26	9.67	4 000	Linear::ForwardCP...	train
20.41	6.50	3 980	Linear::UpdatePara...	train
16.48	16.48	176 960 862	std::__shared_ptr<...	train
7.09	2.17	1 990	ReLU::BackwardCP...	train
7.06	2.12	2 000	ReLU::ForwardCPU...	train
1.70	1.18	5 565 085	Matrix::operator[](...	train
1.53	0.22	1 990	Sigmoid::Backward...	train
0.86	0.00	4 404	cudaMemcpy	train
0.86	0.00	4 404	cudart::cudaApiMe...	train
0.85	0.11	2 000	Sigmoid::ForwardC...	train
0.83	0.00	4 404	cudart::driverHeloe...	train



Total percent of time serial code spends in functions which are almost completely parallelizable

Linear: Backward: 48.67 Forward: 33.26 Total: 81.93 (expected, most of our computation are matrix multiplications in the linear layer)

ReLU: Backward: 7.09 Forward: 7.06 Total: 14.15

Sigmoid: Backward: 1.03 Forward: 0.6 Total: 1.63

Grand total: 98.01% of the code is parallelizable.

ReLU takes a lot longer than Sigmoid even when they basically both are simple element-wise operations. This is probably because of lot of if statements in ReLU.



Theoretical limit using Amdahl's Law

$$\text{Speedup} = 1 / ((1 - p) + p / ps)$$

where p is the fraction of the code that is parallelizable and ps is the number of processors.

Even if this is for processors and not GPU cores, we think we can at least get an upper limit of how much speedup we can get compared to CPU only code.

Using $p = 0.9801$ and $ps = 768$ (number of CUDA cores on GTX 1050ti),

Max speedup achievable is 45.45 times faster than the serial code.

We get 32.09 times faster than the serial code.



Profiling using NVIDIA Visual Profiler

On the next slide, we see a screenshot from the NVIDIA Visual Profiler, which shows time spent in each kernel.

