‘Train multi-layer Neural network using Backpropagation’

December, 5, 2016

Department of Electrical Engineering,

University of California, Riverside

EE217-Fall’16

TEAM MEMBERS

Ramkumar Subramanian (SID: 861255897)

Introduction:

Neural networks are computational models which works like the way the brain solves the problem. Each node has multiple neurons connected to it. Each connected neuron has a weight associated with it. In this backward propagation, neural networks scheme, based on the inputs provided we calculate the intermediate layer values and then we follow through the chain and calculate the results for all stages till output stage. Then we compare the output with the expected result and then we follow the chain backwards to update all the weights per the expected output results. Once the training is completed with the weights being updated at all levels, the neural network is ready to be used for any application. These are especially useful in various machine learning/artificial intelligence techniques. There is so much of data computation happening at each node which presents us with an opportunity to parallelize through GPU computing.

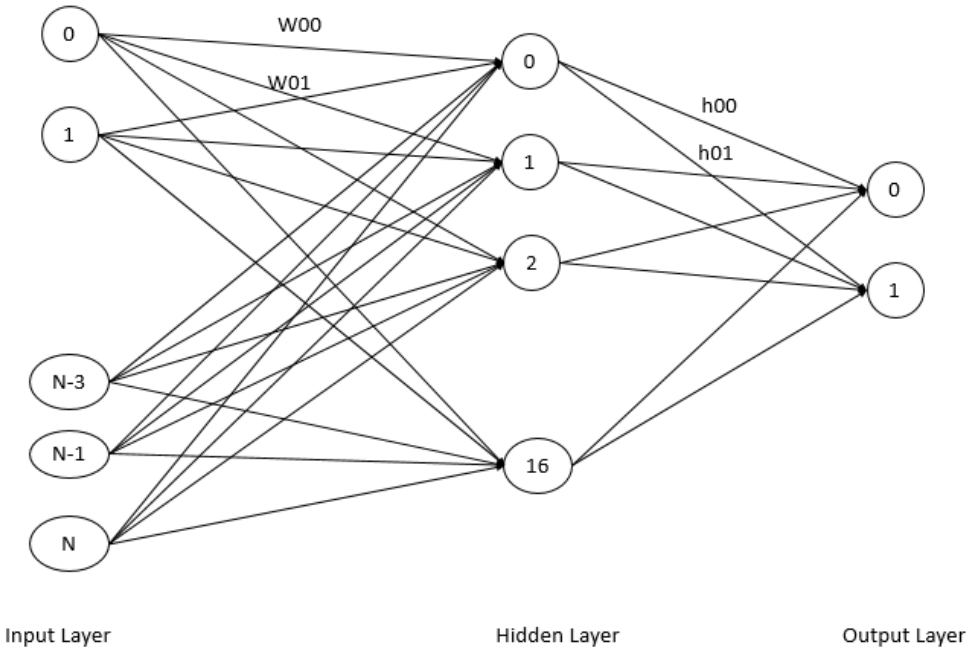

Figure 1: Architecture of 3-layer Neural network

Output Calculation:

The calculations are done at two stages, one in hidden layer and other in output layer. The sample calculations are shown for one node below,

Hidden Layer:

H01 = Inp[0]*W10 + Inp[1] * W11 + Inp[2] * W12 + …. + Inp[N] * W1N

Output Layer:

O0 = H0 * h00 + H1 * h01 + H2 * h02 + … + H16 * h016

Similar calculations are done at each node in this neural network. Once the final results are computed the weights are updated from output stage to input stage. This refers to the training phase of neural network. Once the neural network is warmed up its ready to process the real inputs.

Program Flow:

Two functions have been introduced in the backprop_kernel.c code.

1. gpu_kernel_wrapper - Does forward path calculations for both input to hidden as well as hidden to output layer.
2. gpu_output_error – Calculates error in hidden layer and output layer and then updates the weights according to the calculated error.

Many optimizations were tried, while some were successful while the others din't quite come up as expected. Listing them below.

Shared memory Optimization:

Intuition: When there are 1000's of inputs and 17 hidden layers and when we launch a thread for each input element, there is a need for us to use atomicAdd to update 17 hidden layers which results in heavy serialization of flow (AtomicAdd conflicts are resolved by serialization).

Changes done: In shared memory created a hidden layer results container. Each thread block will update this hidden layer container using atomicAdd private to each thread block. Once all the threads in the thread block are over the flow will update the global hidden layer result using atomicAdd. This method reduces contention between atomicAdd's in great extent. Tabulating metrics between two methods below.

	Run time (us)
All threads directly perform AtomicAdd into Global memory	1681
When all threads in thread block perform AtomicAdd to shared memory	181

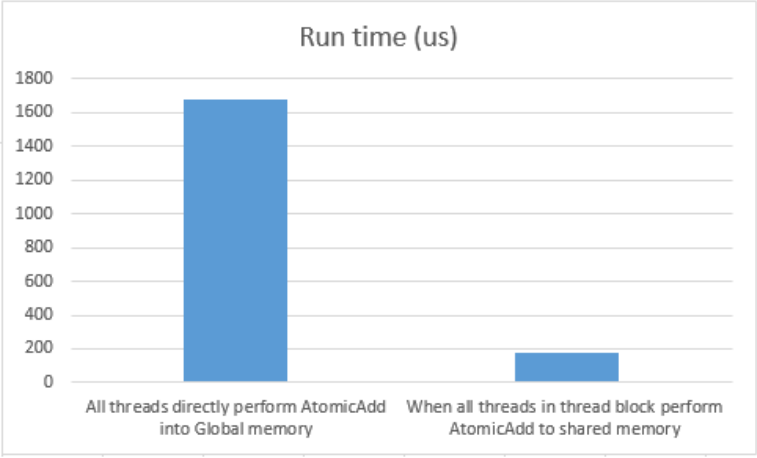

Figure 2: Tabulating the run time comparison between naïve atomicAdd and Shared memory atomicAdd

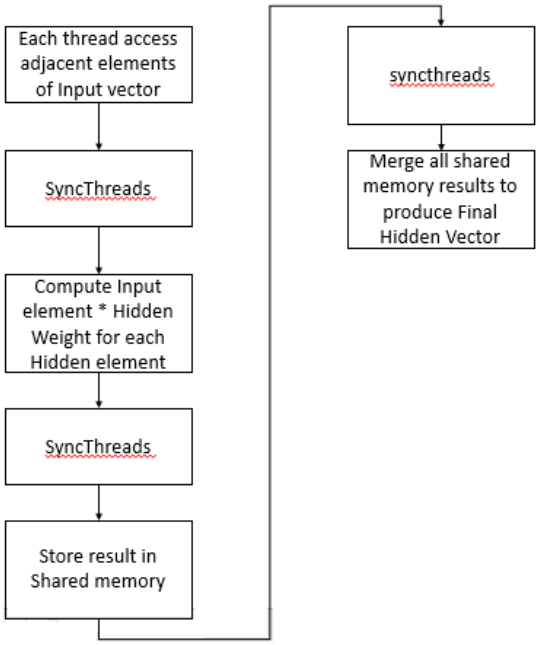

Figure 3: Showing the work flow for shared memory atomicAdd

Cuda Streams Optimization (Pipelining):

intuition: There are multiple kernels that are being launched to complete each computation stage. So we can ideally start data transfer for the current kernel perform cudaDeviceSynchronize, then again start the data transfer required for the next kernel, then we launch the current kernel, so in this way we will be maximizing the utilization of PICE, GPU and all associated hardware.

Challenge faced: Couldn't perform this enhancement for all kernels, when included this for forward computation kernel the code crashed for large number of inputs.

Effort: Spent more than a day to perform this update which couldn't be done across for all kernels.

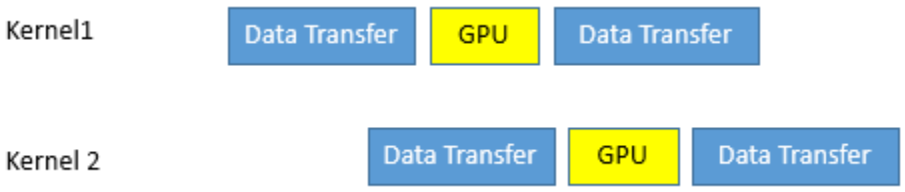

Figure 4: Showing cudaStream copy for multiple kernels

Transferring 2-D matrix directly to kernel:

intuition: The input/hidden weights are in 2-D matrices, but cudaMemCpy can't handle 2-D matrices. Tried to use cudaMemcpy2D() function listed in CUDA user guide.

Challenge faced: was getting compilation error which couldn't be resolved.

Distribution of time across various stages for 1000 inputs:

As its evident from the below chart that the forward computation takes maximum amount of time. Any optimization targeted towards the forward computation stage will greatly reduce the run time.

Stage	Time in us
kernel_bpnn_layerforward(float*, float*, float*, int, int)	181.259
kernel_squash(float*, int)	2.601
kernel_bpnn_layerforward(float*, float*, float*, int, int)	4.832
kernel_squash(float*, int)	2.129
gpu_output_error_kernel_function(float*, float*, float*, int, float*)	1.704
gpu_hidden_error_kernel_function(float*, int, float*, int, float*, float*)	1.802
gpu_weight_adjust_function(float*, int, float*, int, float*)	1.771
D2H	8.096
H2D	21.568

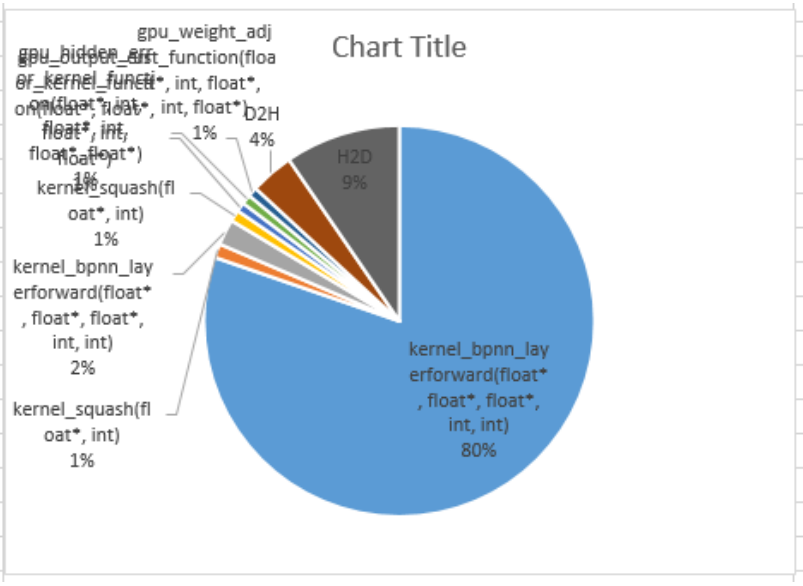

Figure 5: Showing Runtime distribution among the various stages of GPU computation