HACETTEPE UNIVERSITY

COMPUTER ENGINEERING

CMP674 - GPU PARALLEL PROGRAMMING

Text Classification with Multilayer Perceptron

N12165419 Mustafa KÖSE

O18148203 Saim SUNEL

Table of Contents

MLP Implementation In CUDA

MLPs are inspired by neurons in brain [1]. Multilayer Perceptrons are universal function approximators [2] [3]. This feature makes them applicable and attractive for the problem domains in which a mapping needs to be constructed between given input data to output data. They can be thought as a black box which maps given input to output data. Multi layer perceptrons consist of one input, one or more hidden layers and one output. The information is supplied from input and then flows through hidden layers and finally reaches to output layer. Therefore this type of architecture is classified as feed-forward neural network architecture. The hidden layer number, node number's in layer's are up to the designer. The output layer node number depends on the problem to be solved. If the mapped function is so complicated the hidden layer number could be high. The hidden layers provide information to be extracted hierarchically. First hidden layer extracts useful knowledge from inputted information. Second hidden layer extracts knowledge from first hidden layer and so on. The knowledge is extracted from knowledge. At the output highest level information is obtained. Another important feature of the MLPs is the ability of generalization. It can estimate the target values of samples that are not shown previously in learning algorithm.

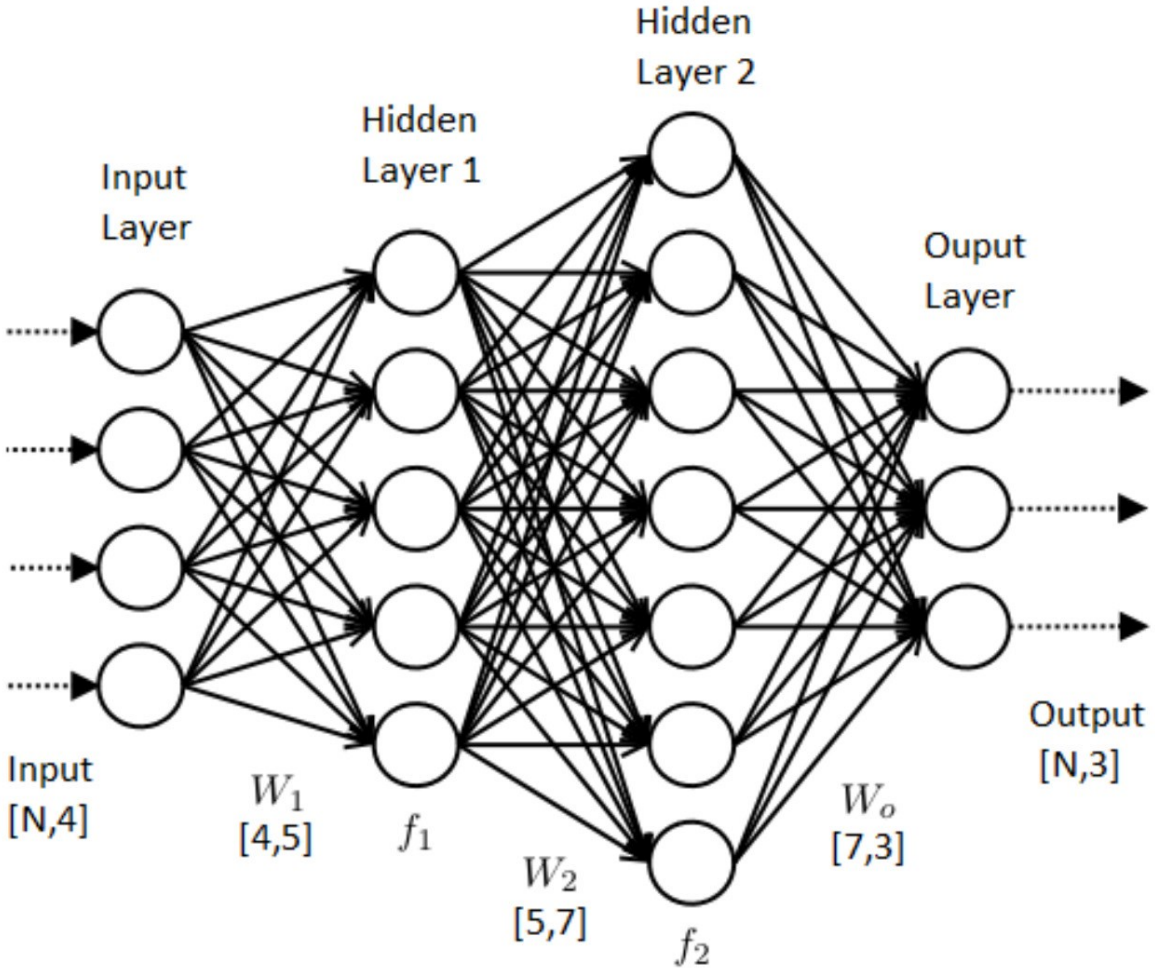
Hidden layers' nodes accumulate information from previous layer and applies a nonlinearity to this information. This is what enables MLPs to learn nonlinear mappings. There are several nonlinearity functions that could be used for this purpose. Main characteristic of the activation is being differentiable. Differentiability is critical in training algorithms. The activation function selection is also up to designer. By considering the features of the problem he/she must decide which function to use. Different activation functions possess different advantages and disadvantages.
MLPs are used in supervised learning tasks in which the target values are provided by the designer. These tasks are classified as regression and classification. MLP tries to match each given data point to given target. In regression problems network tries to learn continuous numerical values. When the information is provided to inputs, the network outputs one or more numerical values. In classification tasks network learns to separate one or more classes from each other. When the data is supplied to network, it outputs the probability of given data belonging to each class. In unsupervised learning no target values is given [4]. The aim of the MLP here is to obtain the structure of data such as cluster, hierarchies lye in data. In reinforcement learning problems, value functions or policy functions are approximated. In this domain MLPs are used as function approximators [5]. They approximate the value function or policy function of the problem.

Learning in MLP means that mapping capability of the MLP between inputs and target is getting better and better progressively. A cost function is defined how well MLP performs the mapping task. A set of learning algorithms exists. They try minimize the error(cost) function by tweaking parameters of the network. Learning is realized in hidden layers. Learning simply involves two phases. In the first one, the data is supplied to network and the output is observed and we have the correct target values. We obtain an error metric using the target values and the network output. In second step we modify the weight vectors so that the new values of the weight will produce less error. This modification part could be performed by using different number of samples. The update of weights could be modified by using only one sample. The error is calculated for this sample and the weights are updated by using this error. Learning with this technique could be quite slow because using individual sample, learning algorithm might converge with oscillation. Another way is to modify the weights using the whole data set. The problem with this technique is that if we have thousands of samples, only one weight updating will take so long. There is also another technique in between these two extremes.

A predefined number of samples are used to train the network in each step.Our project purposes a simple Multilayer Perceptron implementation in CUDA for classification and regression tasks in machine learning. In serial programming, the outputs of layers in MLP are produced sequentially. A matrix of inputs is given, first layer output is calculated by performing required weight multiplications and nonlinear transformation for each sample in order. By using parallel computing, for individual sample the outputs are computed parallelly. All MLP operations can
be deduced to matrix operations. GPUs are capable of running thousands of concurrent threads. Matrix operations can be performed on GPU quite fast. So implementing a MLP for GPU will result in huge performance gains.

With the usage flexibility of MLP in many areas (bioinformatic, scene classification, text categorization etc.) we will try to implement it for text categorization on Reuter-21578 data set. Our main focus is to monitor the success of MLP implementation on single-label document classification. Data set details can be found on this source [6]. Also results of this work can lead us to improve the method for categorizing the multi-label documents. [7], [8], [9] these are the techniques that are used for both categorization approaches. NLP methods (POS, stemming, NER) will not be applied to our project.

MLP Operations



(Figure 1: Example of a Multilayer Perceptron. Input, output and layer weights' sizes are shown. [6])
All related operations are shown below. Hidden layers first calculate linear combination of previous layers' then applies a nonlinear function. The output layer, if the problem is a regression problem, just performs weighted summation of last hidden layer. If classification problem, it performs again weighted summation and applies softmax function. Softmax function turns this summation into probability distribution.

$$InputData = X^{(0)} = \begin{vmatrix} X_1^{(0)0} & X_2^{(0)0} & X_3^{(0)0} \\ X_1^{(0)1} & X_2^{(0)1} & X_3^{(0)2} \\ X_1^{(0)2} & X_2^{(0)2} & X_3^{(0)2} \\ X_1^{(0)3} & X_2^{(0)3} & X_3^{(0)3} \\ X_1^{(0)4} & X_2^{(0)4} & X_3^{(0)4} \end{vmatrix} \qquad TargetData = R = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \\ R_1^1 & R_2^1 & R_3^1 \\ R_1^2 & R_2^2 & R_3^2 \\ R_1^3 & R_2^3 & R_3^3 \\ R_1^4 & R_2^4 & R_3^4 \end{vmatrix}$$

$$X_{input\ node\ index}^{(0)sample\ index} \qquad R_{output\ node\ index}^{sample\ index}$$

$$Each\ sample\ is\ represented\ as\ row\ vector$$

$$First\ sample\ is\ X^{(0)0} = \begin{vmatrix} X_1^{(0)0} & X_2^{(0)0} & X_3^{(0)0} \end{vmatrix}$$

$$Second\ sample\ is\ X^{(0)1} = \begin{vmatrix} X_1^{(0)1} & X_2^{(0)1} & X_3^{(0)1} \end{vmatrix}$$

$$First\ sample's\ target\ is\ R^0 = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \end{vmatrix}$$

$$Second\ sample's\ target\ is\ R^1 = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \end{vmatrix}$$

$$W^{(1)} = \begin{vmatrix} W_{1,1}^{(1)} & W_{1,2}^{(1)} & W_{1,3}^{(1)} & W_{1,4}^{(1)} & W_{1,5}^{(1)} \\ W_{2,1}^{(1)} & W_{2,2}^{(1)} & W_{2,3}^{(1)} & W_{2,4}^{(1)} & W_{2,5}^{(1)} \\ W_{3,1}^{(1)} & W_{3,2}^{(1)} & W_{3,3}^{(1)} & W_{3,4}^{(1)} & W_{3,5}^{(1)} \\ W_{4,1}^{(1)} & W_{4,2}^{(1)} & W_{4,3}^{(1)} & W_{4,4}^{(1)} & W_{4,5}^{(1)} \end{vmatrix} \qquad W^{(2)} = \begin{vmatrix} W_{1,1}^{(2)} & W_{1,2}^{(2)} & W_{1,3}^{(2)} & W_{1,4}^{(2)} & W_{1,5}^{(2)} & W_{1,6}^{(2)} & W_{1,7}^{(2)} \\ W_{2,1}^{(2)} & W_{2,2}^{(2)} & W_{2,3}^{(2)} & W_{2,4}^{(2)} & W_{2,5}^{(2)} & W_{2,6}^{(2)} & W_{2,7}^{(2)} \\ W_{3,1}^{(2)} & W_{3,2}^{(2)} & W_{3,3}^{(2)} & W_{3,4}^{(2)} & W_{3,5}^{(2)} & W_{3,6}^{(2)} & W_{3,7}^{(2)} \\ W_{4,1}^{(2)} & W_{4,2}^{(2)} & W_{4,3}^{(2)} & W_{4,4}^{(2)} & W_{4,5}^{(2)} & W_{4,6}^{(2)} & W_{4,7}^{(2)} \\ W_{5,1}^{(2)} & W_{5,2}^{(2)} & W_{5,3}^{(2)} & W_{5,4}^{(2)} & W_{5,5}^{(2)} & W_{5,6}^{(2)} & W_{5,7}^{(2)} \end{vmatrix}$$

$$B^{(1)} = \begin{vmatrix} B_1^{(1)} & B_2^{(1)} & B_3^{(1)} & B_4^{(1)} & B_5^{(1)} \end{vmatrix}$$

$$B^{(2)} = \begin{vmatrix} B_1^{(2)} & B_2^{(2)} & B_3^{(2)} & B_4^{(2)} & B_5^{(2)} & B_6^{(2)} & B_7^{(2)} \end{vmatrix}$$

$$B^{(3)} = \begin{vmatrix} B_1^{(3)} & B_2^{(3)} & B_3^{(3)} \end{vmatrix}$$

$$X_0^{(0)} = X_0^{(1)} = X_0^{(2)} = +1$$

$$W_o = W^{(3)} = \begin{vmatrix} W_{1,1}^{(3)} & W_{1,2}^{(3)} & W_{1,3}^{(3)} \\ W_{2,1}^{(3)} & W_{2,2}^{(3)} & W_{2,3}^{(3)} \\ W_{3,1}^{(3)} & W_{3,2}^{(3)} & W_{3,3}^{(3)} \\ W_{4,1}^{(3)} & W_{4,2}^{(3)} & W_{4,3}^{(3)} \\ W_{5,1}^{(3)} & W_{5,2}^{(3)} & W_{5,3}^{(3)} \\ W_{6,1}^{(3)} & W_{6,2}^{(3)} & W_{6,3}^{(3)} \end{vmatrix}$$

$$Softmax_i(a) = \frac{\exp a_i}{\sum \exp a_i}$$

$$O^{(l)} = X^{(l-1)}.W^{(l)} + B^{(l)} \qquad X^{(l)} = \sigma(O^{(l)})$$

$$O^{(1)} = X^{(0)}.W^{(1)} + B^{(1)} \qquad X^{(1)} = \sigma(O^{(1)})$$

$$O^{(2)} = X^{(1)}.W^{(2)} + B^{(2)} \qquad X^{(2)} = \sigma(O^{(2)})$$

$$O^{(3)} = X^{(2)}.W^{(3)} + B^{(3)}$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$if\ regression\ Output = O^{(3)}\ if classification\ Output = Softmax(O^{(3)})$$

Learning Algorithm (Backpropagation)

Learning algorithm adjusts network's weights to minimize error (cost) function. While training the network we choose to use backpropagation algorithm. Backpropagation algorithm adjusts weights of network by following gradient direction in which the error is reduced on error surface. It is a gradient based algorithm. It calculates a gradient on surface of error function. Then it modifies the weights by adding the calculated gradient in reverse direction to weights.

$$\Delta W = -\eta . \frac{\partial E}{\partial W} \qquad \frac{\partial E}{\partial W} \text{ is the gradient in which error increases.}$$

$$\eta \text{ controls magnitude of the gradient. Inverse gradient reduces the error.}$$

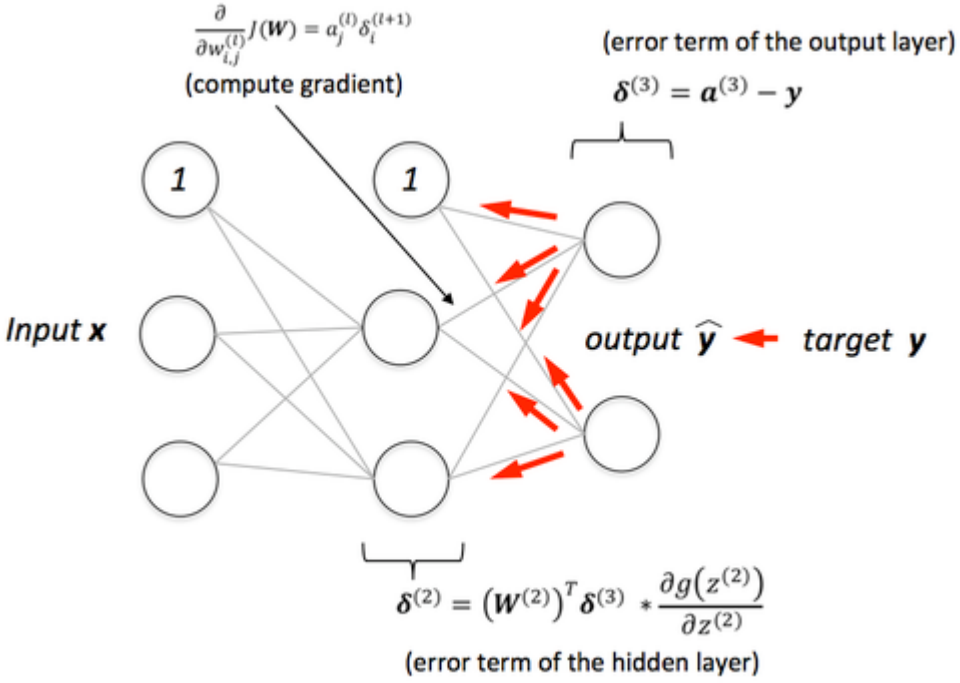
In classification tasks cross entropy function, in regression tasks summed squared error function is used.

$$\text{if regression Summed Squared Loss } E = \frac{1}{2} \sum_{t(\text{sample index})=0}^{n} \sum_{m(\text{output node index})=1}^{M} (R_m^t - O_m^{(3)t})^2$$

$$Error = R - Output$$

$$\text{if classification Cross Entropy Loss} = E = \sum_{t(\text{sample index})=0}^{n} \sum_{m(\text{output node index})=1}^{M} (-R_m^t * log(O_m^{(3)t}))$$

Backpropagation algorithm relies on chain rule. By starting from output is distributes error from output layer to input layer. Each weight is changed depending on how much it contributes to error.



$$\Delta W^{(3)}_{(k,m)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . X^{(2)t}$$

$$\Delta W^{(2)}_{(j,k)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . W^{(3)}_{(k,m)} . \sigma(O^{(2)t}_k) . (1 - \sigma(O^{(2)t}_k)) . X^{(1)t}_j$$

$$\Delta W^{(1)}_{(i,j)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . \sum_{k=1} . W^{(3)}_{(k,m)} . \sigma(O^{(2)t}_k) . (1 - \sigma(O^{(2)t}_k) . W^{(2)}_{(j,k)} . \sigma(O^{(1)t}_j) . (1 - \sigma(O^{(1)t}_j) . X^{(0)t}_i$$

$\odot$ is Ha

$$\begin{gathered}
In\ matrix\ operations\ form: \\
Output\ Error = Error = R - Output \\
\Delta W^{(3)} = \eta . (X^{(2)})^T . Output\ Error \\
\Delta W^{(2)} = \eta . (X^{(1)})^T . ((Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) . (1 - \sigma(O^{(2)}))) \\
\Delta W^{(1)} = \eta . (X^{(0)})^T . ((((Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) . (1 - \sigma(O^{(2)}))) . (W^{(2)})^T) \odot (\sigma(O^{(1)}) . (1 - \sigma(O^{(1)})))) \\
Hidden\ Layer\ 2\ Error = (Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) . (1 - \sigma(O^{(2)})) \\
Hidden\ Layer\ 1\ Error = (Hidden\ Layer\ 2\ Error . (W^{(2)})^T) \odot (\sigma(O^{(1)}) . (1 - \sigma(O^{(1)})) \\
\Delta W^{(3)} = \eta . (X^{(2)})^T . Output\ Error \\
\Delta W^{(2)} = \eta . (X^{(1)})^T . Hidden\ Layer\ 2\ Error \\
\Delta W^{(1)} = \eta . (X^{(0)})^T . Hidden\ Layer\ 1\ Error \\
LayerError(l) = (LayerError(l+1) . (W^{(l+1)})^T) \odot (\sigma(O^{(l)}) . (1 - \sigma(O^{(l)})))
\end{gathered}$$

$$\Delta B^{(3)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Output\ Layer\ Error$$

(Figure 2: It shows how backpropagation distributes error originated at output to inner layers. [7])

$$\Delta B^{(2)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Hidden\ Layer\ 2\ Error$$

$$\Delta B^{(1)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Hidden\ Layer\ 1\ Error$$

$$Dimension\ of\ column\ vector\ of\ +1s\ is\ (sample\ number, 1)$$

$$\frac{\partial(\frac{1}{2}\sum_t \sum_m (R^t_m - O^{(3)t}_m)^2)}{\partial w} = \sum_t \sum_m (R^t_m - O^{(3)t}_m) . -1 . \frac{\partial O^{(3)t}_m}{\partial w}$$

$$\frac{\partial(\sum_t \sum_m (-R^t_m * log(O^{(3)t}_m)))}{\partial w} = \sum_t \sum_m (R^t_m - O^{(3)t}_m) . -1 . \frac{\partial O^{(3)t}_m}{\partial w}$$

For classification and regression cost functions we obtain same gradient formulas.

All these learning operations can be represented with matrices operations. Matrix multiplication, Hadamard multiplication, transpose operation, matrix subtraction and addition, scalar-matrix multiplication.

Related Works

Multilayer Perceptrons can be trained via parallel technologies due to their nature. Its fundamental operations are matrix operations. GPUs are very efficient in calculating matrices because of their design concerns in graphical industry. MLPs can be parallelized via thread parallelizing. But CPUs with more than one cores are not dedicated to only one task. They perform context switching between threads, wait for other threads to finish their job. Their concurrently runnable threads number is also limited. Their multitask nature brings some flaws to efficiency. On the other hand, if they are implemented in GPU hardware, there will be thousands of threads running for only one dedicated task. All bandwidth of the device will be allocated to current task. CUDA is relatively a new technology which enables programmers to benefit of parallelism and computation power of GPUs. GPUs' design concern was to perform graphical computations in high speeds. And graphical operations are programmed to GPUs via program elements called shaders. Before CUDA technology, If the programmers were to use GPU for parallel tasks, they had to write their program as shader. [8] research implements a neural network in shader programs to enhance performance of a text detection system. They could obtain 2 0-fold performance enhancement using an ATI RADEON 9700 PRO boar. With the emergence of CUDA it has become easy to write programs for GPU. And hence MLP implementations become more easier.

[9] in this work, the authors tried to solve a problem of training multilayer perceptron simultaneously on a given specific problem called The inverse problem in geophysics using CUDA technology. On this problem they created and trained lots of MLPs at the same time. They used standard backpropagation with momentum with half batch training. Each MLP has only one single layer. Each MLP consists 1,648 inputs, 8 neurons in the hidden layer and one neuron in the output layer. They trained 75 MLPs simultaneously. They implemented normal CPU code and GPU code for this task. They obtained 50x speed increase compared to optimized CPU-based computer program and 150x compared to a commercially available CPU-based program.

[10] in this work authors try to minimize MNIST (hand written character set) error by designing multiple MLPs. The trained MLPs were 2-9 layer long with different layer node numbers. These architectures resulted in millions of parameters. Training these networks would take so long time. They implemented MLPs on GPU to reduce this training time. They used backpropagation algorithm with decaying learning rate. While they were classifying the digits, they sent the character to have been recognized to a set of MLPs. Each network produced its output and final decision was made in a committee manner. They could obtain very low error 0.35%. (35 out of 10,000 digits). Implementation detail of GPU can be inspected in paper.

[11] in this work, authors try to compare highly optimized CPU implementation of MLP and their GPU implementation on variaous data sets. For comparison they used the fastest implementation known to the authors the Fast Artificial Neural Network Library (FANN) by Steffen Nissen. This library owes its speed to optimized inner loops and cache optimization and a technique to calculating sigmoids. They used parallelized matrix operations (BLAS) provided by NVIDIA's CUDA (CUBLAS). They constructed both in CPU implementation and in GPU a MLP consisting 378 nodes of input layer, a

single hidden layer consisting of 2048 neurons and output layer of 71 neurons. In small sized problems CUDA implementation falled back due to data transfers from CPU/GPU to GPU/CPU, kernelinitializations, although it was faster than the CPU implementation. To get a better view on the performance of CUDA agains CPU they used relatively large data sets. Their results show that CUDA accelerates computation quite significantly.

[12] in this work, authors compare performance of CPU implemented MLP and GPU implemented MLP. They introduce CUDA technology, Multilayer perceptron and backpropagation algorithm. They talk about the matrix operations used in MLP. They implemented several MLPs in C language and at the same time on GPU. They constructed MLPs for two problems. For each problem they wrote CPU program and GPU program. The first problem was classical XOR problem in MLPs. They implemented several MLPs consisting different number of hidden layers. They compared C language implementation and GPU implementation performance. They showed that as the hidden layer number increased the duration of computation increased with a high slop with C implementation. On the other hand on GPU implementation the duration increased with a small slope. The second problem was classification problem which aims at identifying a number of heart diseases in an ECG. They also did the same thing with this problem and showed that on GPU side duration time of calculation increased with less slope compared to C implementation as the hidden layer number increased. They also mentioned that the problem to be solved must be computationally heavy to compensate GPU initialization and operation overhead.

[13] in this work, authors aimed to accelerate the training time of an MLP for handwritten digit recongition system. For recognition to be flawless, the MLP architecture should be enough deep and the data to be used should be huge to cover the variations in different writing styles. This requirement exposes a problem of high computation time to standard CPU based MLP. They aimed to overcome this problem by implementing MLP on GPU. They used standard backpropagation algorithm. They explained their GPU program in paper. They implemented a MLP with 784 inputs 533 hidden neurons and 10 nodes at output layer. They also spoke about that the results of computations must be stored row-oriented version to get higher performance.

[14] in this work, authors compare CPU implementation and GPU implementation of MLPs for a face recognition task. They compare MLP implemented with CPU parallelizing library OpenMP and CUDA implementation. They created MLPs with different number of hidden neurons using OpenMP and CUDA. The face images were emotional expressions of faces. In their programs they tried to find out the emotional status of the image currently supplied to neural network. Also for OpenMP they searched for the optimal number of threads required to obtain best performance for a given MLP architecture. They showed that after a certain number of hidden neurons the GPU outperforms. If the hidden neuron number is less than this threshold, CPU implementation outperforms due to GPU initialization and running overhead.

[15] in this works, the authors combine OpenMP with CUDA implementation for further performance gain. Their aim was to accelerate text detection problem. The problem is to find the locations in which text exists. To solve this problem, they came up with a two phase of architecture. The first step consists of feature extraction from the data. A predefined window was run over the image to find candidate regions. They implemented this task on CPU with OpenMP to accelerate feature extraction. Later two classify the regions they sent this extracted features to GPU implemented MLP with 30 nodes in hidden

layer. By combining thread level parallelism and GPU based parallelism, they could come up with a program 5 times faster than implementation using CPU and about 4 times faster than implementation on only GPU without OpenMP.[16] in this work, authors claim that they implemented first artificial neural network with Levenberg-Marquardt (LM) training method on GPU. They claim that this technique was not applied previously due to inverse matrix operation involved in this algorithm. They adopted a solution to this problem in their work. At that CUDA was not present. Instead they used Brook GPU program language to create GPU implementation. They also implemented the same network solely on CPU and made comparisons between CPU and GPU implementations. They compared different MLPs with different hidden layer numbers and showed that GPU implementations overperformed CPU implementations.

[17] in this work, the authors implement multilayer perceptron with backpropagation algorithm with CUDA. They explain some insights of multilayer perceptron, backpropagation algorithm and CUDA BLAS library in the paper. They mentioned some useful functions of the library used in backpropagation. They created several MLPs with different number of hidden neurons. They tested these MLPs on two dataset. They also showed that GPU implementation on these sets outperformed CPU implementations.

[18] in this work, authors aimed to construct and human pose estimator system which is invariant to view points and can run in real time. To provide real time behavior they implemented one hidden layer MLP in CUDA to process data. To obtain invariance in view, they benefited from generalization capability of neural networks. They used 5 cameras from different viewpoints while constructing training set. To handle high dimensional data from cameras they used PCA algorithm to reduce the input data dimension. They extracted location invariant features by using described techniques in paper from images by performing preprocessing on images. They constructed several MLPs to find out with which number of hidden neurouns the system delivered best accuracy. They showed that CUDA and PCA delivers quite significant speed performance.

References

[1] Warren S.McCulloch and Walter H. Pitt’s 1943 paper, ‘‘A Logical Calculus of the Ideas Immanent in Nervous Activity,’’
[2] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2(4), 303-314.
[3] Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251–257
[4] Hinton, G. E.; Salakhutdinov, R.R. (28 July 2006). "Reducing the Dimensionality of Data with Neural Networks". Science. 313 (5786): 504–507.
[5] Tesauro, Gerald (March 1995). "Temporal Difference Learning and TD-Gammon"
[6] http://ana.cachopo.org/datasets-for-single-label-text-categorization
[7] A. Cardoso-Cachopo, A. Oliveira, Semi-supervised single-label text categorization using centroid-based classifiers, in: Proceedings of the ACM Symposium on Applied Computing, Seoul, Korea, March 11–15, 2007, pp. 844–851.
[8] M. Zhang, Z. Zhou, Multi-Label Neural Networks with Applications to Functional Genomics and Text Categorization
[9] L. Lenc, P. Kral, Ensemble of Neural Networks for Multi-label Document Classification
[10] https://techblog.viasat.com/wp-content/uploads/2017/07/ANN-Diagram.png
[11] https://sebastianraschka.com/images/faq/visual-backpropagation/backpropagation.png

[12] Kyoung-Su Oh, K.S., Jung, K.: GPU implementation of neural networks. Pattern Recognition 37, 1311–1314 (2004)
[13] Multifold Acceleration of Neural Network Computations Using GPU Alexander Guzhva, Sergey Dolenko, and Igor Persiantsev D.V. Skobeltsyn
[14] Deep Big Multilayer Perceptrons For Digit Recognition Dan Claudiu Cireşan 1,2 , Ueli Meier 1,2 , Luca Maria Gambardella 1,2 , and Jürgen Schmidhuber 1,2
[15] Parallel Training of a Multi-Layer Perceptron on a GPU Chris Oei, Gerald Friedland, Adam Janin October 7, 2009
[16] TRAINING AND APPLYING A FEEDFORWARD MULTILAYER NEURAL NETWORK IN GPU Klaus Raizer, Hugo Sakai Idagawa, Prof. Dr. Eur ıı pedes Guilherme de Oliveira Nóbrega, Prof. Dr.
Luiz Otávio Saraiva Ferreira
[17] Fast Efficient Artificial Neural Network for Handwritten Digit Recognition Viragkumar N. Jagtap , Shailendra K. Mishra[18] Multicore and GPU Parallelization of Neural Networks for Face Recognition Altaf Ahmad
Huqqani, Erich Schikuta, Sicen Ye, Peng Chen
[19] Neural Network Implementation using CUDA and OpenMP Honghoon Jang, Anjin Park, Keechul Jung
[20] Graphics Processor Unit Hardware Acceleration of Levenberg-Marquardt Artificial Neural Network Training David Scanlan, David Mulvaney
[21] Parallel Training of a Back-Propagation Neural Network using CUDA
Xavier Sierra-Canto, Francisco Madera-Ram ıı rez, V ıı ctor Uc-Cetina
[22] View-Point Insensitive Human Pose Recognition using Neural Network and CUDA Sanghyeok Oh, Keechul Jung

Serial and Parallel Code Inspection

In our project primarily, we would like implement a document classifier via multi-layer perceptron. For this task we had referenced some papers in related work section. But currently we don’t have these papers’ implementations and data sets used. For our task we would like prepare our dataset. What we want is to accelerate training procedure with the help of GPU computation capabilities. So, our main objective is to have a fast neural network.

We have implemented our neural network in C language and Python language. Primarily C language is used. All functions related to neural network computation have been implemented from scratch in C side and in Python side numpy library’s functions for matrix operations have been used directly. Classification side has been tested via MNIST dataset in C implementation. Due to computation time, a small part of it has been used to verify our implementation. In Python side it is tested via our own data set of characters. Regression part has been tested in Python side due to data set loading opportunities.

In our implementation, user can easily create an MLP structure with many hidden layers as she/he would like. She/he can declare as many output and input nodes as she/he wishes. Our current implementation is using backpropagation implementation with no momentum, adaptive learning rate based approaches like AdaGrad, RMSProp, AdamOptimizer etc... If needed we consider to add one of these tools due to their benefit. Because our network that will be used in documentation task will not be very deep and hence decaying gradient or exploding gradient problems are less likely to occur, we don’t see any benefit of using batch normalization technique.

To accelerate neural network computation currently we have implemented thread parallelization. The data is partitioned among the threads, each thread performs forward propagation(data is fed to input layer and computations related are carried out from input to output) and backpropagation (error occurred at output due to forward propagation is used to just the weights of network). After each thread finishes its job, the computed adjustments of weights are summed and applied to network. This procedure is repeated many times till sufficient error rate is obtained. Currently our implementation used batch learning. In this technique whole data is used while adjusting the weights of the network. If needed, mini batch training or stochastic learning could be used in later steps of the project.

Our project source code could be accessed via (https://github.com/SaimSUNEL/MLPForRegressionandClassification).

We present some multi-layer perceptron implementations in this part.

[1] in this work authors try to minimize MNIST (hand written character set) error by designing multiple MLPs. The trained MLPs were 2-9 layer long with different layer node numbers. These architectures resulted in millions of parameters. Training these networks would take so long time. They implemented MLPs on GPU to reduce this training time. They used backpropagation algorithm with decaying learning rate. While they were classifying the digits, they sent the character to have been recognized to a set of MLPs. Each network produced its output and final decision was made in a committee manner. They could obtain very low error 0.35%. (35 out of 10,000 digits detected false). Implementation detail of GPU can be inspected in paper.

[2] in this work, the authors implement multilayer perceptron with backpropagation algorithm with CUDA. They explain some insights of multilayer perceptron, backpropagation algorithm and CUDABLAS library in the paper. They mentioned some useful functions of the library used in backpropagation. They created several MLPs with different number of hidden neurons. They tested these MLPs on two datasets; cancer dataset, mushroom data set. They also showed that GPU implementation on these sets outperformed CPU implementations with increase in speed of 46x and 63x with one hidden layer in their experiments.

[3] in this work, authors aimed to accelerate the training time of an MLP for handwritten digit recongition system. For recognition to be flawless, the MLP architecture should be enough deep and the data to be used should be huge to cover the variations in different writing styles. This requirement exposes a problem of high computation time to standard CPU based MLP. They aimed to overcome this problem by implementing MLP on GPU. They used standard backpropagation algorithm. They explained their GPU program in paper. They implemented a MLP with 784 inputs 533 hidden neurons and 10 nodes at output layer. They also spoke about that the results of computations must be stored row-oriented version to get higher performance. They share their implementation partially in their paper. They used MNIST dataset for experiments. They obtained %98 accuracy on test set.

[4] A Neural Network in 10 lines of CUDA C++ Code – in this website authors implemented a one hidden layer multilayer perceptron to classify Iris data set in CUDA for education purposes.

[5] Neural Network in C++ (Part 2: MNIST Handwritten Digits Dataset) – In this work author implements a serial multilayer perceptron with 2 hidden layers. They use MNIST data for

testing.

[6] MNIST training with Multi-Layer Perceptron – in this implementation authors built one layer MLP to train for MNIST dataset in Python. They used a library called Chainer. Chainer library can be made to run on GPU with simple functions. They compared implementations GPU enabled and not enabled. They obtained 6.9x and 18.6x speed differences compared to CPU with different hidden layer nodes.

[7] CUDA/OpenCL Sample OCR Project – In this project authors implemented a single hidden layer multilayer perceptron using Levenberg-Marquardt (LM) learning algorithm in by using OpenCL library. They used The Letter Recognition Data Set(Data set courtesy of UCI Machine Learning Repository).

[8] Multi-Layer Perceptron MNIST – On this page a multi-layer perceptron with 2 hidden layers was implemented in Tensorflow library for character recognition by using MNIST dataset.

[9] cuANN – In this work author implemented a multilayer perceptron framework for CUDA. Custom MLP structure can be constructed as in our code.

[10] CudaNeuralNetworks – In this project author implemented serial and GPU parallel multilayer perceptron with one hidden layer.[1] Deep Big Multilayer Perceptrons For Digit Recognition Dan Claudiu Cireşan 1,2 , Ueli Meier 1,2 ,
Luca Maria Gambardella 1,2 , and Jürgen Schmidhuber 1,2

[11] A CUDA Parallel Implementation of Feed Forward Neural Networks for MNIST Recognition on GPU – In this work authors implemented a feedforward neural network on GPU with Cublas library for hand writen digit recognition.

References

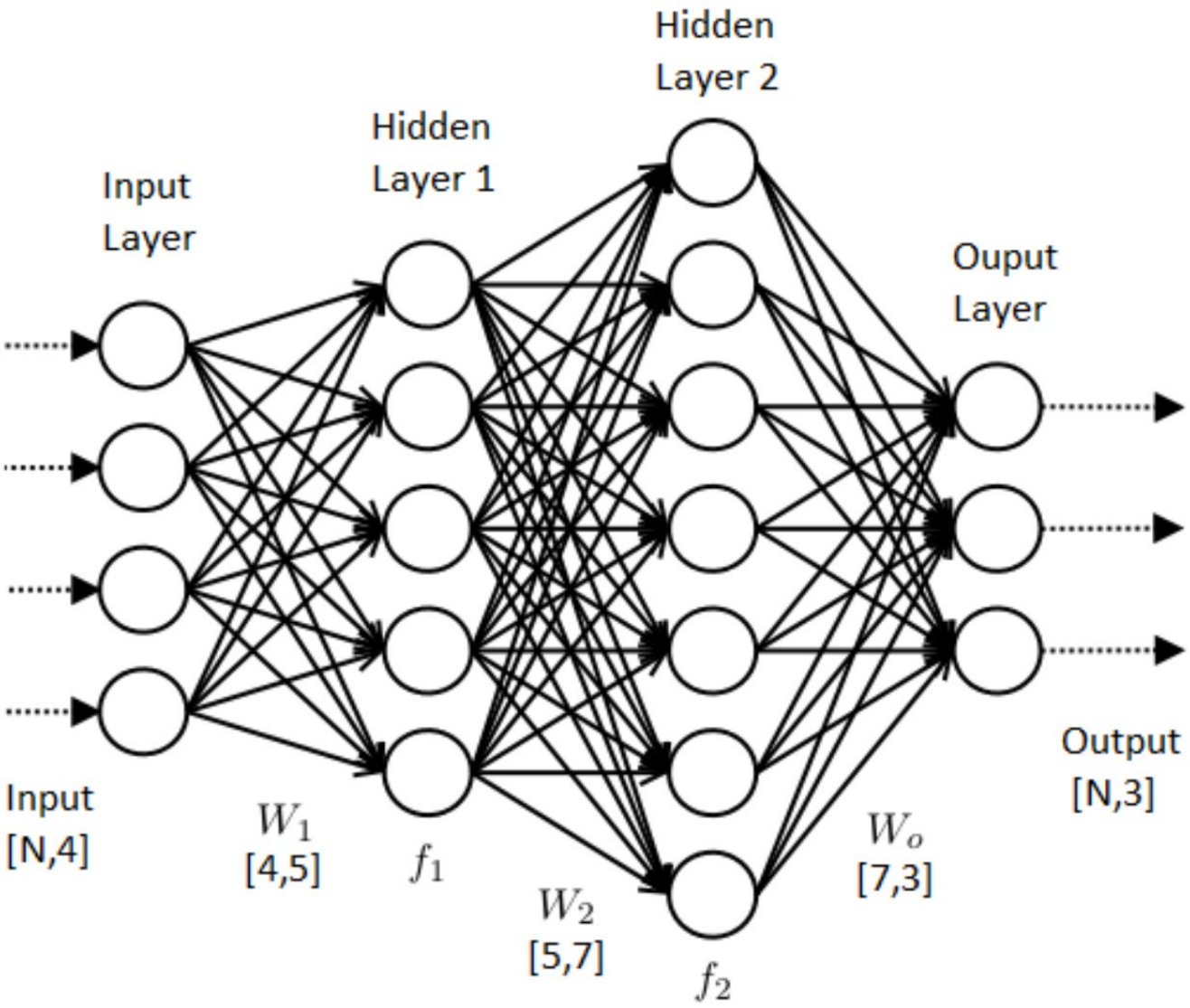
[1] Deep Big Multilayer Perceptrons For Digit Recognition Dan Claudiu Cireşan 1,2 , Ueli Meier 1,2 , Luca Maria Gambardella 1,2 , and Jürgen Schmidhuber 1,2

[2] Parallel Training of a Back-Propagation Neural Network using CUDA
Xavier Sierra-Canto, Francisco Madera-Ramirez, Victor Cetina
[3] Fast Efficient Artificial Neural Network for Handwritten Digit Recognition Viragkumar N. Jagtap , Shailendra K. Mishra
[4] https://cognitivedemons.wordpress.com/2017/09/02/a-neural-network-in-10-lines-of-cuda-c-code/
[5] https://cognitivedemons.wordpress.com/2018/06/08/neural-network-in-c-part-2-mnist-handwritten-digits-dataset/
[6] https://corochann.com/mnist-training-with-multi-layer-perceptron-1149.html
[7] http://www.neurosolutions.com/products/cuda/sample.html
[8] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/multi_layer_perceptron_mnist.html
[9] https://github.com/alexge233/cuANN
[10] https://github.com/PirosB3/CudaNeuralNetworks
[11] https://github.com/xqding/NeuralNetwork_GPU-CUDA-_MNIST

GPU Design

Our project constructs a Multilayer-Perceptron to classify text documents on GPU hardware. MLP has two main operations: forward pass and backward pass. In forward forward pass, the data is supplied to neural network input and the required operations are performed and at output layer the answers/results of the network is observed. In backward pass, the neural network's weights are adjusted to correct the neural network output depending on given target values. By performing backpropagation many times, the neural networks become able to output correct values by the time.

MLP Structure and Operations



(Figure 1: Example of a Multilayer Perceptron. Input, output and layer weights' sizes are shown. [6])

All related operations are shown below. Hidden layers first calculate linear combination of previous layers' then applies a nonlinear function. The output layer, if the problem is a regression problem, just performs weighted summation of last hidden layer. If classification problem, it performs

again weighted summation and applies softmax function. Softmax function turns this summation into probability distribution.

$$InputData = X^{(0)} = \begin{vmatrix} X_1^{(0)0} & X_2^{(0)0} & X_3^{(0)0} \\ X_1^{(0)1} & X_2^{(0)1} & X_3^{(0)2} \\ X_1^{(0)2} & X_2^{(0)2} & X_3^{(0)2} \\ X_1^{(0)3} & X_2^{(0)3} & X_3^{(0)3} \\ X_1^{(0)4} & X_2^{(0)4} & X_3^{(0)4} \end{vmatrix} \qquad TargetData = R = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \\ R_1^1 & R_2^1 & R_3^1 \\ R_1^2 & R_2^2 & R_3^2 \\ R_1^3 & R_2^3 & R_3^3 \\ R_1^4 & R_2^4 & R_3^4 \end{vmatrix}$$

$$X_{input\ node\ index}^{(0)sample\ index} \qquad R_{output\ node\ index}^{sample\ index}$$

$$Each\ sample\ is\ represented\ as\ row\ vector$$

$$First\ sample\ is\ X^{(0)0} = \begin{vmatrix} X_1^{(0)0} & X_2^{(0)0} & X_3^{(0)0} \end{vmatrix}$$

$$Second\ sample\ is\ X^{(0)1} = \begin{vmatrix} X_1^{(0)1} & X_2^{(0)1} & X_3^{(0)1} \end{vmatrix}$$

$$First\ sample's\ target\ is\ R^0 = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \end{vmatrix}$$

$$Second\ sample's\ target\ is\ R^1 = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \end{vmatrix}$$

$$W^{(1)} = \begin{vmatrix} W_{1,1}^{(1)} & W_{1,2}^{(1)} & W_{1,3}^{(1)} & W_{1,4}^{(1)} & W_{1,5}^{(1)} \\ W_{2,1}^{(1)} & W_{2,2}^{(1)} & W_{2,3}^{(1)} & W_{2,4}^{(1)} & W_{2,5}^{(1)} \\ W_{3,1}^{(1)} & W_{3,2}^{(1)} & W_{3,3}^{(1)} & W_{3,4}^{(1)} & W_{3,5}^{(1)} \\ W_{4,1}^{(1)} & W_{4,2}^{(1)} & W_{4,3}^{(1)} & W_{4,4}^{(1)} & W_{4,5}^{(1)} \end{vmatrix} \qquad W^{(2)} = \begin{vmatrix} W_{1,1}^{(2)} & W_{1,2}^{(2)} & W_{1,3}^{(2)} & W_{1,4}^{(2)} & W_{1,5}^{(2)} & W_{1,6}^{(2)} & W_{1,7}^{(2)} \\ W_{2,1}^{(2)} & W_{2,2}^{(2)} & W_{2,3}^{(2)} & W_{2,4}^{(2)} & W_{2,5}^{(2)} & W_{2,6}^{(2)} & W_{2,7}^{(2)} \\ W_{3,1}^{(2)} & W_{3,2}^{(2)} & W_{3,3}^{(2)} & W_{3,4}^{(2)} & W_{3,5}^{(2)} & W_{3,6}^{(2)} & W_{3,7}^{(2)} \\ W_{4,1}^{(2)} & W_{4,2}^{(2)} & W_{4,3}^{(2)} & W_{4,4}^{(2)} & W_{4,5}^{(2)} & W_{4,6}^{(2)} & W_{4,7}^{(2)} \\ W_{5,1}^{(2)} & W_{5,2}^{(2)} & W_{5,3}^{(2)} & W_{5,4}^{(2)} & W_{5,5}^{(2)} & W_{5,6}^{(2)} & W_{5,7}^{(2)} \end{vmatrix}$$

$$B^{(1)} = \begin{vmatrix} B_1^{(1)} & B_2^{(1)} & B_3^{(1)} & B_4^{(1)} & B_5^{(1)} \end{vmatrix}$$

$$B^{(2)} = \begin{vmatrix} B_1^{(2)} & B_2^{(2)} & B_3^{(2)} & B_4^{(2)} & B_5^{(2)} & B_6^{(2)} & B_7^{(2)} \end{vmatrix}$$

$$B^{(3)} = \begin{vmatrix} B_1^{(3)} & B_2^{(3)} & B_3^{(3)} \end{vmatrix}$$

$$X_0^{(0)} = X_0^{(1)} = X_0^{(2)} = +1$$

$$W_o = W^{(3)} = \begin{vmatrix} W_{1,1}^{(3)} & W_{1,2}^{(3)} & W_{1,3}^{(3)} \\ W_{2,1}^{(3)} & W_{2,2}^{(3)} & W_{2,3}^{(3)} \\ W_{3,1}^{(3)} & W_{3,2}^{(3)} & W_{3,3}^{(3)} \\ W_{4,1}^{(3)} & W_{4,2}^{(3)} & W_{4,3}^{(3)} \\ W_{5,1}^{(3)} & W_{5,2}^{(3)} & W_{5,3}^{(3)} \\ W_{6,1}^{(3)} & W_{6,2}^{(3)} & W_{6,3}^{(3)} \\ W_{7,1}^{(3)} & W_{7,2}^{(3)} & W_{7,3}^{(3)} \end{vmatrix}$$

$$O^{(l)} = X^{(l-1)}.W^{(l)} + B^{(l)} \qquad X^{(l)} = \sigma(O^{(l)})$$

$$O^{(1)} = X^{(0)}.W^{(1)} + B^{(1)} \qquad X^{(1)} = \sigma(O^{(1)})$$

$$O^{(2)} = X^{(1)}.W^{(2)} + B^{(2)} \qquad X^{(2)} = \sigma(O^{(2)})$$

$$O^{(3)} = X^{(2)}.W^{(3)} + B^{(3)}$$

$$if\ regression\ Output = O^{(3)}\ if classification\ Output = Softmax(O^{(3)})$$

$$\Delta W^{(3)}_{(k,m)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . X^{(2)t}$$

$$\Delta W^{(2)}_{(j,k)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . W^{(3)}_{(k,m)} . \sigma(O^{(2)t}_k) . (1 - \sigma(O^{(2)t}_k)) . X^{(1)t}_j$$

$$\Delta W^{(1)}_{(i,j)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m) . \sum_{k=1} . W^{(3)}_{(k,m)} . \sigma(O^{(2)t}_k) . (1 - \sigma(O^{(2)t}_k) . W^{(2)}_{(j,k)} . \sigma(O^{(1)t}_j) . (1 - \sigma(O^{(1)t}_j) . X^{(0)t}_i$$

$\odot$ *is Hadamard multiplication*

$$\begin{gathered}
In\ matrix\ operations\ form: \\
Output\ Error = Error = R - Output \\
\Delta W^{(3)} = \eta . (X^{(2)})^T . Output\ Error \\
\Delta W^{(2)} = \eta . (X^{(1)})^T . ((Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))) \\
\Delta W^{(1)} = \eta . (X^{(0)})^T . ((((Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))) . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)})))) \\
Hidden\ Layer\ 2\ Error = (Output\ Error . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)})) \\
Hidden\ Layer\ 1\ Error = (Hidden\ Layer\ 2\ Error . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)})) \\
\Delta W^{(3)} = \eta . (X^{(2)})^T . Output\ Error \\
\Delta W^{(2)} = \eta . (X^{(1)})^T . Hidden\ Layer\ 2\ Error \\
\Delta W^{(1)} = \eta . (X^{(0)})^T . Hidden\ Layer\ 1\ Error \\
LayerError(l) = (LayerError(l+1) . (W^{(l+1)})^T) \odot (\sigma(O^{(l)}) \odot (1 - \sigma(O^{(l)})))
\end{gathered}$$

$$\Delta B^{(3)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Output\ Layer\ Error \qquad \Delta B^{(2)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Hidden\ Layer\ 2\ Error$$

$$\Delta B^{(1)} = \begin{vmatrix} +1 \\ +1 \\ +1 \\ \vdots \end{vmatrix} . Hidden\ Layer\ 1\ Error$$

$$Dimension\ of\ column\ vector\ of\ +1s\ is\ (sample\ number, 1)$$

W1 is the weight matrix between input and first hidden layers, W2 between first hidden layer and second hiddenlayer, W3 between second hidden layer and output.

Forward pass:

At input, the data matrix and W1 are multiplied. This multiplication's result(O1) is applied a nonlinear function(sigmoid) and the first layer's output(sigmoid(O1)) is obtained. Then sigmoid(O1) matrix and W2 are multiplied. This result(O2) is applied a nonlinear function(sigmoid) and the second hidden layer(sigmoid(O2)) output is obtained. Then sigmoid(O2) matrix and W3 are multiplied. If the problem is regression problem the resulted matrix(O3) is output of the neural network. If it is classification, softmax function is applied to resulted matrix and this constitudes the output(softmax(O3)) of neural network.

Backward pass:

At output layer the output matrix of neural network(O3 or softmax(O3)) is subtracted from the target value (R) matrix. This constitudes the error matrix(Err(output)) of output layer. To obtain update amount of W3, the Err(output) should be multiplied via the transpose of sigmoid(O2). To obtain second hidden layer error matrix(Error(second hidden layer)), the Err(output) should be multiplied with transpose of W3 and (sigmoid(O2)(1-sigmoid(O2))). Update of W2 is obtained by multiplying the Error(second hidden layer) with transpose of sigmoid(O1). First hidden layer's error matrix (Error(first hidden layer)) is obtained by multiplying Error(second hidden layer) with transpose of W2 and sigmoid(O1)(1-sigmoid(O1)). And finally the update amount of W1 is obtained by multiplying Error(first hidden layer) with transpose of input data matrix.

So simply as shown above in forward pass and backward pass we are performing these matrix operations:

- Matrix Multiplication, Matrix Addition and Subtraction, Matrix Transpose, Hadamard Multiplication, Scalar Matrix Multiplication, Scalar Matrix Subtraction, Sigmoid function, Softmax function, matrix element summation.

-Matrix element summation function sums all elements of matrix and can be implemented via reduction technique.

Parallelization of Computations

Unfortunately, the forward pass operation must be performed sequentially. We can not obtain second hidden layer's output before the first hidden layer. First first hidden layer output later second hidden layer and later output layer result must be calculated. This sequential computation is also valid for backpropagation. First output layer error must be calculated later second hidden layer error.

In forward we can not calculate two distinct layer outputs of the network at the same time. In backward propagation, we have shown that to obtain a weight's update we must know its next layer error value. Error values of layers must be calculated sequentially. But weight updates of network after obtaining each layer's error value, can be calculated simultaneously. Because error values and output of each layer have already obtained.

So outputs and error values of each layer must be calculated sequentially. This is due to nature of the operation. But after obtaining the error values and output of layer, the update of the whole weights can be calculated simultaneously.

$$\Delta W^{(3)} = \eta . (X^{(2)})^T . Output\ Error$$
$$\Delta W^{(2)} = \eta . (X^{(1)})^T . Hidden\ Layer\ 2\ Error$$
$$\Delta W^{(1)} = \eta . (X^{(0)})^T . Hidden\ Layer\ 1\ Error$$

The description of sequential and parallelization so far has been for the algorithms used in MLP. The main operations that are performed are simple matrix operations. In our project our weights and the inputs are represented as 2D matrices. So multiplication, addition, subtraction etc. operations on these matrices can be parallelized easily with GPU hardware.

Operations

In this part we represent the operations that are used for MLP and their naive implementation and discuss their improvement via available hardware features and techniques.

Matrix Multiplication

```
__device__ float result[RESULT_ROW][RESULT_COLUMN];
__global__ void MatrixMultiplication(float * m1, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;
        if(thx < M2_COLUMN && thy < M1_ROW)
        {
                float toplam = 0;
                for(int h = 0; h < column_count; h++)
                {
                        toplam += m1[thy*column_count+h] * m2[h*row_count+thx];
                }
        result[thy][thx] = toplam;
        }
}
```

All operations represented via “.” symbol in matrix operations shown above are matrix multiplication(except “η.” operation).
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Matrix Addition

```
__global__ void MatrixAddition(float * m1, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        result__[thy][thx] = m1[thx+column_count*thy] + m2[thx+column_count*thy];

        }

}
```

All operations represented via ”+” symbol in matrix operations shown above are matrix addition.
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Matrix Subtraction

```
__global__ void MatrixSubtraction(float * m1, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        result__[thy][thx] = m1[thx+column_count*thy] - m2[thx+column_count*thy];

        }

}
```

All operations represented via ”-” symbol in matrix operations shown above are matrix subtraction(except “1-$\sigma(O)$”).
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Matrix Transpose

```
__global__ void TransposeMatrix(float * res, float * m1, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        res[thx*row_count+thy] = m1[thx+column_count*thy] ;

        }

}
```

All operations represented via ”T” symbol in matrix operations shown above are matrix transpose operation.
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Hadamard Multiplication

```
__device__ float result[ROW][COLUMN];
__global__ void HadamardMultiplication(float * m1, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        result[thy][thx] = m1[thx+column_count*thy] * m2[thx+column_count*thy];

        }

}
```

All operations represented via ”⊙” symbol in matrix operations shown above are hadamard multiplication.
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Scalar Matrix Subtraction

```
__global__ void ScalarMatrixSubtraction(float value, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        result__[thy][thx] = value - m2[thx+column_count*thy];

        }

}
```

1 – σ(O) operation represents a scalar matrix subtraction.
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Scalar Matrix Multiplication

```
__global__ void ScalarMatrixMultiplication(float value, float * m2, int column_count, int row_count)
{
        int thx = blockIdx.x*blockDim.x+ threadIdx.x;
        int thy = blockIdx.y*blockDim.y+threadIdx.y;

        if(thx < column_count && thy < row_count)
        {

        result__[thy][thx] = value * m2[thx+column_count*thy];

        }

}
```

“η.” operation represents scalar matrix multiplication.
Possible optimization :
Testing different kernel and grid sizes.
Shared memory usage
Unrolling
L1 cache usage
Read-only memory usage

Implementation Considerations

In GPU, allocations of memory for data matrix, weight matrices, bias matrices, layer outputs, layer error matrices will be allocated. These memory allocations are not fixed size because user can define any custom MLP structure. But fixing the architecture required values could be obtained.

Block and Grid Size

-Because our neural network structure is not static we can not decide exact grid and dimension sizes of the kernels but we will consider these advices from textbook.

For a given kernel, trying different grid and block dimensions may yield better performance. Shared memory and registers are precious resources in an SM. Shared memory is partitioned among thread blocks resident on the SM and registers are partitioned among threads.

GUIDELINES FOR GRID AND BLOCK SIZE

Using these guidelines will help your application scale on current and future devices:

Keep the number of threads per block a multiple of warp size (32).
Avoid small block sizes: Start with at least 128 or 256 threads per block.
Adjust block size up or down according to kernel resource requirements.
Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism to your device.
Conduct experiments to discover the best execution confi guration and resource usage.

nvvp is a Visual Profiler, which helps you to visualize and optimize the performance of your CUDA program. This tool displays a time-line of program activity on both the CPU and GPU, helping you to identify opportunities for performance improvement. In addition, nvvp analyzes your application for potential performance bottlenecks and suggests actions to take to eliminate or reduce those bottlenecks.

The CUDA Toolkit includes a spreadsheet, called the CUDA Occupancy Calculator, which assists you in selecting grid and block dimensions to maximize occupancy for a kernel.

- Our weights matrices are not so huge generally. So to obtain more parallesism we plan to create as much as kernel we can to keep busy all the Sms. As book stated we plan to keep the block size more than the SMs on GPU. Huge block count is also not applicable because total number of blocks that can be run on GPU is limited. As book stated nvidia tools could be used to find optimal values for block and grid sizes.

Global Memory

Aligned coalesced memory accesses are ideal: A wrap accessing a contiguous chunk of memory starting at an aligned memory address. To maximize global memory throughput, it is important to organize memory operations to be both aligned and coalesced.

All accesses to global memory go through the L2 cache. Many accesses also pass through the L1 cache, depending on the type of access and your GPU's architecture. If both L1 and L2 caches are used, a memory access is serviced by a 128-byte memory transaction. If only the L2 cache is used, a memory access is serviced by a 32-byte memory transaction.

Loop unrolling is a technique that attempts to optimize loop execution by reducing the frequency of branches and loop maintenance instructions. In loop unrolling, rather than writing

the body of a loop once and using a loop to execute it repeatedly, the body is written in code multiple times. The goal is improving performance by reducing instruction overheads and creating more independent instructions to schedule. As a result, more concurrent operations are added to the pipeline leading to higher saturation of instruction and memory bandwidth. This provides the warp scheduler with more eligible warps that can help hide instruction or memory latency. The GPU has more flexibility in scheduling them concurrently, potentially leading to better global memory utilization.

There are two major factors that influence the performance of device memory operations:

➤ Efficient use of bytes moving between device DRAM and SM on-chip memory: To avoid wasting device memory bandwidth, memory access patterns should be aligned and coalesced.

➤ Number of memory operations concurrently in-flight: Maximizing the number of in-flight memory operations is possible through either 1) unrolling, yielding more independent memory accesses per thread, or 2) modifying the execution configuration of a kernel launch to expose more parallelism to each SM.

Much like instruction latency, you can increase the available parallelism by either creating more independent memory operations within each thread/warp, or creating more concurrently active threads/warps.

A “thin” block improves the effectiveness of store operations by increasing the number of consecutive elements stored by a thread block as a result of a larger value in the innermost dimension of the thread block.

However, on the GPU only memory load operations can be cached; memory store operations cannot be cached.

Global memory’s latency is the biggest bottleneck for our application. Because all the resulted matrices should be stored in there. To get optimum performance we need to finish storing and loading data in minimum number of transfers. So to achieve this coalesced and aligned memory transfers will be used as much as it can be possible. L1 cache usage may help to minimize to access memory accesses so we plan to use and test it. So one way to optimize the bandwidth usage of the global memory is to request as many independent memory transfer as possible. So with this, CUDA may optimize this transfers to minimize the total memory accesses. One way to achieve is to use unrolling technique. We are planning to use unrolling technique for optimization. For data transfer between CPU and GPU the pinned memory will be tried for optimization. In actual kernels we will be using structures because all weights are a part of structure. So rather than using array of structures we are planning to use structure of arrays.

Note that shared memory and L1 cache are physically closer to the SM than both the L2 cache and global memory. As a result, shared memory latency is roughly 20 to 30 times lower than global memory, and bandwidth is nearly 10 times higher.

Shared memory can be used to hide the performance impact of global memory latency and bandwidth.

It is possible to improve global memory coalesced access using shared memory in many cases. Shared memory is a key enabler for many high-performance computing applications.

Shared memory bank width defines which shared memory addresses are in which shared memory banks. Memory bank width varies for devices depending on compute capability. There are two different bank widths:

➤ 4 bytes (32-bits) for devices of compute capability 2.x

➤ 8 bytes (64-bits) for devices of compute capability 3.x

You can configure how much L1 cache and how much shared memory will be used by kernels

Memory padding is one way to avoid bank conflicts.

To avoid strided global memory access, 2D shared memory can be used to cache data from the original matrix. A column read from 2D shared memory can be transferred to a transposed matrix row stored in global memory.

It is best to have threads with consecutive values of threadIdx.x accessing consecutive locations in shared memory.

Shared memory can help to our program to eliminate the global memory access overhead. Because the weigths are not huge, they can be stored in shared memory for temporarily.
Some functions like transpose performs column write which is very inefficient patter for global memory. We can overcome this problem by using shared memory via padding. At this time the writing operation can be done as it were column-wise. After creating transpose on shared and copying it coalesced and aligned to global memory will result in performance.

On the other hand usage of L1 cache also be beneficial. We plan to try different configurations of shared memory like access bank width, different memory allocations for L1 and shared memory. Too much use of shared memory may also become a bottleneck because it is shared among the thread blocks. The lower amount of shared memory usage, the more concurrent threads reside in a SM.

Constant Memory

It is best if all threads in a warp access the same location in constant memory. Accesses to different addresses by threads within a warp are serialized.

- Learning rate of neural network is a constant. By storing this value in constant memory will benify its broadcasting feature. By this feature one value is delivered to multiple threads with on memory transition.

Texture Memory

Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance.

Kepler GPUs add the ability to use the GPU texture pipeline as a read-only cache for data stored in global memory. Because this is a separate read-only cache with separate memory bandwidth from normal global memory reads, use of this feature can yield a performance benefit for bandwidth-limited kernels.

Generally, the read-only cache is better for scattered reads than the L1 cache, and should not be used when threads in a warp all read the same address. The granularity of the read-only cache is 32 bytes.

In some cases usage of this read-only memory to access global memory data may be benefical. We plan to test whether this feature helps us or not.

If data transfer from CPU to GPU operation causes a bottleneck for the application, Cuda streams feature could be used while obtaining the neural network output. So while performing some computation, the data set could be transferred at the same time.

Code Optimization

For optimization simplification we have used our OneHiddenClassification.cu program. This code constructs a one hidden layer neural network (32754 input nodes 128 hidden layer nodes, 4 output nodes) for our text classification task. By using Tensorflow (A computation library) we have confirmed that our data can be correctly classified with high accuracy with only one hidden layer.

Our optimization steps of GPU code consist of following steps:

Combination of some sequential operations(functions) into one function to reduce data retrieval overhead. These consecutive functions use their input output sequentially. Two or more times the data is retrieved from global memory. Instead we can apply these operations in one retrieval.

Some kernel executions have dependencies but some don’t. We can run non-dependent kernels at the same because they have no input output dependency to each other. By executing these kinds of kernels at the same time we can gain more parallelism on GPU and consequently speed gain.

Unrolling technique: In first implementation our kernels is responsible for only one data point.

By introducing more data point calculation, we can increase the total operations on fly on GPU resulting in more optimization and speed gain.

Shared Memory usage with padding: In first implementation our kernels retrieve all data directly from global memory. Most of our kernels obtain data and perform one operation on it and save it back to global memory. The main consideration here is to make all these global data load and store operations aligned and coalesced. Every thread in one warp is executed at the same time. These 32 threads request data from the global memory. Ideally when the data reads are coalesced and aligned, this demand from one warp can be fulfilled with only one data transaction. If the data is not aligned the data retrieval operation is performed more than one transaction, which results in performance degrade. Most of our kernel performs coalesced and aligned data reading. But transpose and matrix multiplication operations include column read which results uncoalesced data retrieval transactions. One remedy to this problematic reading is to load data to first shared memory. Shared memory is a on chip memory which is much more faster than global memory in reading and storing operations. So, reading the data from global memory in coalesced and aligned manner and storing it to shared memory saves us from performing more data transactions than needed. We are eliminating uncoalesced and aligned data requests. Shared memory has a special data reading mechanism. The shared memory is divided into banks. If two requests correspond to the same bank, the back confliction occurs and shared memory services this two requests sequentially. When we upload the data to shared memory and perform transpose operation or matrix multiplication, here we face the block confliction problem. Because kernels will try to read or write column vectors which are mapped to same banks. Of course, bank confliction overhead is less than the global memory's unaligned and uncoalesced memory accesses. To fix this confliction problem a padding technique is used. This technique simply adds a blank column vector to original data to shift overlapping banks and any accesses to the same bank with padding will not cause bank confliction therefore the store or load operations will be performed in optimum manner.

Texture memory cache: Texture memory has a cache optimized for 2D spatial data. CUDA framework allows programmers to use this cache while accessing the global data. Global data can be cached on this cache memory. We can use it like a L1 cache. Before going to directly to global memory the data load request will be directed to texture cache, it the requested data rest there, the data will be retrieved from here if not cache miss occurs and from global data the needed data is loaded to cache. By using this cache, the memory request number from global memory could be lessened. Because the global memory accesses are time consuming, we expect some gain from usage of this cache.

Batch Size and Thread Block number: Different thread block numbers effect the GPU utilization rate. Finding correct thread block number requires some trail and error process. Batch size of the training data defines sample number that is used in forward and back propagation. Sending more number of samples to training leads early finishing of whole data pass.

We have test these optimization techniques by applying every one of them to unoptimized code. Only one feature is implemented at a time.

Combination of Some Sequential Operations

```
MatrixAdd<<<grid, block>>>(output_2, output_2, bias_result_2);
cudaDeviceSynchronize();
Exponential<<<grid, block>>>(output_2, output_2);
cudaDeviceSynchronize();
```

These two functions are fused into one function.

```
AddandExponential<<<grid, block>>>(output_2, output_2, bias_result_2);
cudaDeviceSynchronize();
```

Original forms:

```
__global__ void MatrixAdd(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{

	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] + vec2->data[tid];
	}
}

__global__ void Exponential(Vector2D * result, Vector2D * vec1)
{


	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = exp(vec1->data[tid]);
	}
}
```

Combined form:

```
__global__ void AddandExponential(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
```

```
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = exp(vec1->data[tid] + vec2->data[tid]);
	}
}

	MatrixPairwiseProduct<<<grid6, block>>>(layer_1_error, layer_1_error, output_1);
	cudaDeviceSynchronize();
	ScalarMinusVector2D<<<grid6, block>>>(scalar_minus, 1.0, output_1);
	cudaDeviceSynchronize();
	MatrixPairwiseProduct<<<grid6, block>>>(layer_1_error, layer_1_error, scalar_minus);
	cudaDeviceSynchronize();
```

These three functions are fused into one.

```
	LayerErrorCalculate<<<grid6, block>>>(layer_1_error, layer_1_error, output_1);
```

Original forms:

```
__global__ void MatrixPairwiseProduct(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * vec2->data[tid];
	}
}
__global__ void ScalarMinusVector2D(Vector2D * result, float value, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;


	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = 1-vec1->data[tid];
	}
}
```

Combined form:

```
//Combination of matrixpairwise-Scalarminus-matrixpairwise in backpropagte....
```

```
__global__ void LayerErrorCalculate(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{

	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid + < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * vec2->data[tid]*(1-vec2->data[tid]) ;
	}
}

	dim3 grid10((OUTPUT_NODE_COUNT+block.x-1)/block.x,
	(HIDDEN_LAYER_NODE_COUNT+block.y-1)/block.y);
	ScalarMatrixProduct<<<grid10, block>>>(w2_update, learning_rate, w2_update);
	cudaDeviceSynchronize();
	//Apply w2 update
	MatrixAdd<<<grid10, block>>>(w2, w2, w2_update);
	cudaDeviceSynchronize();
```

These two functions are fused into one function.

```
	dim3 grid10((OUTPUT_NODE_COUNT+block.x-1)/block.x,
	(HIDDEN_LAYER_NODE_COUNT+block.y-1)/block.y);
	ApplyWeightChange<<<grid10, block>>>(w2, learning_rate, w2_update);
```

Original forms:

```
__global__ void ScalarMatrixProduct(Vector2D * result, float scalar, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = scalar*vec1->data[tid];
	}
}

__global__ void MatrixAdd(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
```

```
	{
		result->data[tid] = vec1->data[tid] + vec2->data[tid];
	}
}
```

Combined form:

```
__global__ void ApplyWeightChange(Vector2D * result, float learning_rate, Vector2D * source)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*source->width+tx;

	if(tid < source->width*source->height)
	{
		result->data[tid] += learning_rate*source->data[tid];	}}


			dim3 gridd((OUTPUT_NODE_COUNT+block.x-1)/block.x,
			(batch_size+block.y-1)/block.y);
			Log2D<<<gridd, block>>>(output_2, output_2);
			cudaDeviceSynchronize();
			MatrixPairwiseProduct<<<gridd, block>>>(layer_2_error, batch_label, o
			utput_2);
			cudaDeviceSynchronize();

		dim3 gridd((OUTPUT_NODE_COUNT+block.x-1)/block.x, (batch_size+block.y-1
		)/block.y);
		calculateCrossEntropyLoss<<<gridd, block>>>(layer_2_error, batch_label,
		output_2);
```

Original forms:

```
__global__ void Log2D(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	float val;
	if(tid < vec1->width*vec1->height)
	{	val = log(vec1->data[tid]);
		result->data[tid] = val;
	}
}

__global__ void MatrixPairwiseProduct(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
```

```
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * vec2->data[tid];
	}
}
```

Combined form:

```
__global__ void calculateCrossEntropyLoss(Vector2D * __restrict__ result, Vector2D * __restrict__ vec1, Vector2D * __restrict__ vec2)
{	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
		if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * log(vec2->data[tid]);
	}
}
```

Unrolling Technique

Original form:

```
__global__ void MatrixAdd(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] + vec2->data[tid];
	}
	}
```

Unrolled form:

```
__global__ void MatrixAdd(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid +blockDim.x*3< vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] + vec2->data[tid];
		result->data[tid+blockDim.x] = vec1->data[tid+blockDim.x] + vec2->data[tid+blockDim.x];
		result->data[tid+2*blockDim.x] = vec1->data[tid+2*blockDim.x] + vec2->data[tid+2*blockDim.x];
		result->data[tid+3*blockDim.x] = vec1->data[tid+3*blockDim.x] + vec2->data[tid+3*blockDim.x];
```

```
	}

}
```

Original form:

```
__global__ void MatrixSubtract(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] - vec2->data[tid];

	}
}
```

Unrolled form:

```
__global__ void MatrixSubtract(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] - vec2->data[tid];
		result->data[tid+blockDim.x] = vec1->data[tid] - vec2->data[tid+blockDim.x];
		result->data[tid+2*blockDim.x] = vec1->data[tid+2*blockDim.x] - vec2-
>data[tid+2*blockDim.x];
		result->data[tid+2*blockDim.x] = vec1->data[tid+3*blockDim.x] - vec2-
>data[tid+3*blockDim.x];
	}


}
```

Original form:

```
__global__ void TransposeVector2D(Vector2D * res, Vector2D * m1)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;
	int tid = thx + thy*m1->width;

	if(tid < m1->width*m1->height)
	{
```

```
	res->data[thy+thx*m1->height] = m1->data[tid] ;
	}

}
```

Unrolled form:

```
__global__ void TransposeVector2DUnroll4(Vector2D * res, Vector2D * m1)
{

	int thx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;
	int tid = thx + thy*m1->width;

	if(tid +3*blockDim.x< m1->width*m1->height)
	{

	res->data[thy+thx*m1->height] = m1->data[tid] ;
	res->data[thy+(thx+blockDim.x)*m1->height] = m1->data[tid+blockDim.x] ;
	res->data[thy+(thx+2*blockDim.x)*m1->height] = m1->data[tid+2*blockDim.x] ;
	res->data[thy+(thx+3*blockDim.x)*m1->height] = m1->data[tid+3*blockDim.x] ;

	}

}
```

Original form:

```
__global__ void MatrixProduct(Vector2D * result, Vector2D * m1, Vector2D * m2)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;

	if(thx < result->width && thy < result->height)
	{
		float toplam = 0;
		for(int h = 0; h < m1->width; h++)
		{
			toplam += m1->data[thy*m1->width+h] * m2->data[h*m2->width+thx];
		}
	result->data[thy*result->width + thx] = toplam;
	}

}
```

Unrolled form:

```
__global__ void MatrixProduct(Vector2D * result, Vector2D * m1, Vector2D * m2)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
```

```
	int thy = blockIdx.y*blockDim.y+threadIdx.y;

	if(thx < result->width && thy < result->height)
	{
		float toplam = 0;

		#pragma unroll 4
		for(int h = 0; h < m1->width; h++)
		{
			toplam += m1->data[thy*m1->width+h] * m2->data[h*m2->width+thx];

		}
	result->data[thy*result->width + thx] = toplam;

	}

}
```

Original form:

```
__global__ void ScalarMinusVector2D(Vector2D * result, float value, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = 1-vec1->data[tid];
	}

}
```

Unrolled form:

```
__global__ void ScalarMinusVector2D(Vector2D * result, float value, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;



	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = 1-vec1->data[tid];
		result->data[tid+blockDim.x] = 1-vec1->data[tid+blockDim.x];
		result->data[tid+2*blockDim.x] = 1-vec1->data[tid+2*blockDim.x];
		result->data[tid+3*blockDim.x] = 1-vec1->data[tid+3*blockDim.x];
	}
```

```
}
```

Original form:

```
__global__ void ScalarMatrixProduct(Vector2D * result, float scalar, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = scalar*vec1->data[tid];
	}
}
```

Unrolled form:

```
__global__ void ScalarMatrixProduct(Vector2D * result, float scalar, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = scalar*vec1->data[tid];
		result->data[tid+blockDim.x] = scalar*vec1->data[tid+blockDim.x];
		result->data[tid+2*blockDim.x] = scalar*vec1->data[tid+2*blockDim.x];
		result->data[tid+3*blockDim.x] = scalar*vec1->data[tid+3*blockDim.x];
	}
}
```

Original form:

```
__global__ void MatrixPairwiseProduct(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * vec2->data[tid];
	}
}
```

Unrolled form:

```
__global__ void MatrixPairwiseProduct(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
```

```
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = vec1->data[tid] * vec2->data[tid];
		result->data[tid+blockDim.x] = vec1->data[tid+blockDim.x] * vec2-
>data[tid+blockDim.x];
		result->data[tid+2*blockDim.x] = vec1->data[tid+2*blockDim.x] * vec2-
>data[tid+2*blockDim.x];
		result->data[tid+3*blockDim.x] = vec1->data[tid+3*blockDim.x] * vec2-
>data[tid+3*blockDim.x];
	}
}
```

Original form:

```
__global__ void Softmax(Vector2D * result, Vector2D * vec1)
{
	int tid = blockIdx.y*blockDim.y + threadIdx.y;

	if(tid < vec1->height)
	{
		float toplam = 0;
		for(int a = 0; a < vec1->width;a++)
		{
			toplam += vec1->data[a+tid*vec1->width];

		}
		for(int a = 0; a < vec1->width;a++)
		{
			result->data[a+tid*vec1->width] = vec1->data[a+tid*vec1->width]/toplam;

		}
	}
}
```

Unrolled form:

```
__global__ void Softmax(Vector2D * result, Vector2D * vec1)
{
	int tid = blockIdx.y*blockDim.y + threadIdx.y;

	if(tid < vec1->height)
	{
		float toplam = 0;
		#pragma unroll OUTPUT_NODE_COUNT
		for(int a = 0; a < vec1->width;a++)
		{
```

```
                toplam += vec1->data[a+tid*vec1->width];

            }
            #pragma unroll OUTPUT_NODE_COUNT
            for(int a = 0; a < vec1->width;a++)
            {
                result->data[a+tid*vec1->width] = vec1->data[a+tid*vec1->width]/toplam;

            }
        }
}
```

Original form:

```
__global__ void Sum2D(Vector2D * vec)
{
    int tid = threadIdx.y;
    float val = 0;
    int width = vec->width;
    for(int a = 0; a < width; a++)
    {
        val += vec->data[a+tid*width];
    }
    error_sum[tid] = val;
}
```

Unrolled form:

```
__global__ void Sum2D(Vector2D * vec)
{
    int tid = threadIdx.y;
    float val = 0;
    int width = vec->width;
    #pragma unroll 4
    for(int a = 0; a < width; a++)
    {
        val += vec->data[a+tid*width];
    }
    error_sum[tid] = val;
}
```

Original form:

```
__global__ void ArgMax2D(Vector2D * vec1)
{
    int tid = blockIdx.y*blockDim.y + threadIdx.y;
    if(tid < vec1->height)
    {
        float max = -100000;
        int max_index = 0;
```

```
		for(int a = 0; a < vec1->width;a++)
		{
			if(vec1->data[tid*vec1->width+a]>max)
			{
				max = vec1->data[tid*vec1->width+a];
				max_index = a;
			}}
	arg_max_result[tid] = max_index;
	}
}
```

Unrolled form:

```
__global__ void ArgMax2D(Vector2D * vec1)
{
	int tid = blockIdx.y*blockDim.y + threadIdx.y;
	if(tid < vec1->height)
	{
		float max = -100000;
		int max_index = 0;
		#pragma unroll 4
		for(int a = 0; a < vec1->width;a++)
		{
			if(vec1->data[tid*vec1->width+a]>max)
			{
				max = vec1->data[tid*vec1->width+a];
				max_index = a;
			}
		}
		arg_max_result[tid] = max_index;
	}
}
```

Original form:

```
__global__ void Log2D(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;

	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = log(vec1->data[tid]);
	}
}
```

Unrolled form:

```
__global__ void Log2D(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid + 3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = log(vec1->data[tid]);
		result->data[tid+blockDim.x] = log(vec1->data[tid+blockDim.x]);
		result->data[tid+2*blockDim.x] = log(vec1->data[tid+2*blockDim.x]);
		result->data[tid+3*blockDim.x] = log(vec1->data[tid+3*blockDim.x]);
	}
}
```

Original form:

```
__global__ void Exponential(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = exp(vec1->data[tid]);
	}
}
```

Unrolled form:

```
__global__ void Exponential(Vector2D * result, Vector2D * vec1)
{	int tx = blockIdx.x*blockDim.x*4*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = exp(vec1->data[tid]);
		result->data[tid+blockDim.x] = exp(vec1->data[tid+blockDim.x]);
		result->data[tid+2*blockDim.x] = exp(vec1->data[tid+2*blockDim.x]);
		result->data[tid+3*blockDim.x] = exp(vec1->data[tid+3*blockDim.x]);
	}

}
```

Original form:

```
__global__ void Sigmoid(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
```

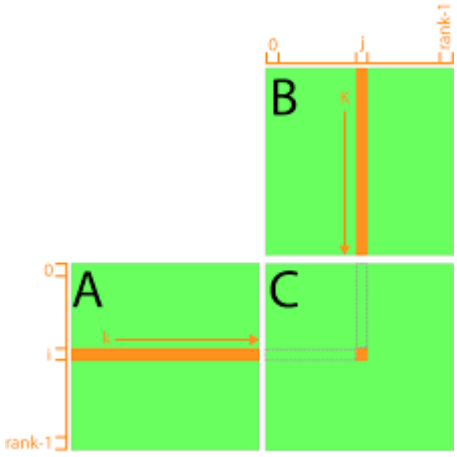
```
	int tid = ty*vec1->width+tx;
	if(tid < vec1->width*vec1->height)
	{
		result->data[tid] = 1.0/(1.0 + exp(-(vec1->data[tid])));
	}
}
```

Unrolled form:

```
__global__ void Sigmoid(Vector2D * result, Vector2D * vec1)
{
	int tx = blockIdx.x*blockDim.x*4+ threadIdx.x;
	int ty = blockIdx.y*blockDim.y + threadIdx.y;
	int tid = ty*vec1->width+tx;
	if(tid +3*blockDim.x< vec1->width*vec1->height)
	{
		result->data[tid] = 1.0/(1.0 + exp(-(vec1->data[tid])));
		result->data[tid+blockDim.x] = 1.0/(1.0 + exp(-(vec1->data[tid+blockDim.x])));
		result->data[tid+2*blockDim.x] = 1.0/(1.0 + exp(-(vec1->data[tid+2*blockDim.x])));
		result->data[tid+3*blockDim.x] = 1.0/(1.0 + exp(-(vec1->data[tid+3*blockDim.x])));
	}
}
```

Shared Memory with Padding

Without shared memory:
Each thread of the kernel calculates the multiplication of one row vector of first matrix with one column vector of second matrix. Second matrix data reading from global memory is not coalesced. It brings more transaction for data retrieval overhead.



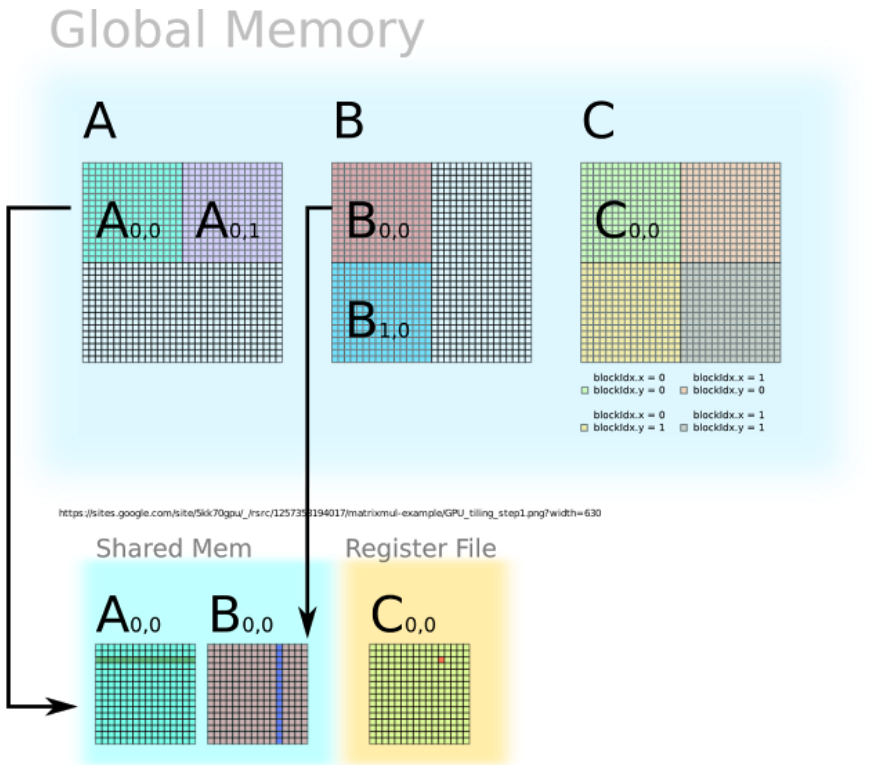
(https://www.3dgep.com/cuda-memory-model/)

```
__global__ void MatrixProduct(Vector2D * result, Vector2D * m1, Vector2D * m2)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;
```

```
        if(thx < result->width && thy < result->height)
        {
                float toplam = 0;
                for(int h = 0; h < m1->width; h++)
                {
                        toplam += m1->data[thy*m1->width+h] * m2->data[h*m2->width+thx];
                }
        result->data[thy*result->width + thx] = toplam;

        }
}
```

With shared memory:
Each matrix in matrix multiplication is divided into blocks(tiles). Thread blocks are responsible for each part. Each thread block uses shared memory with tile size. First each block reads matrix data from global to shared memory for each tile. Because the data transactions fall into the warp boundaries, of thread block the transactions of reading data from global to shared are aligned and coalesced. While reading the column vector of tile in B matrices normally bank conflictions occur. To eliminate that the required padding of shared memory is applied.



(http://www.cstechera.com/2016/03/tiled-matrix-multiplication-using-shared-memory-in-cuda.html)

```
__global__ void MatrixProductShared(  Vector2D * result, Vector2D * m1, Vector2D * m2 )//float *A, float *B, float *C ) {
 {
  __shared__ float A_tile[TILE_HEIGHT][TILE_WIDTH];
   __shared__ float B_tile[TILE_HEIGHT][TILE_WIDTH+1];
```

```
   int numARows = m1->height, numAColumns= m1->width, numBRows = m2->height,
numBColumns = m2->width, numCRows = result->height, numCColumns = m2->width;
  float * A = m1->data, * B = m2->data, * C = result->data;
  float sum = 0.0;
  // tx for thread_x or tile_x
  int tx = threadIdx.x; int ty = threadIdx.y;
  // cx for top left corner of tile in C
  int cx = blockIdx.x * blockDim.x; int cy = blockIdx.y * blockDim.y;
  // Cx for cell coordinates in C
  int Cx = cx + tx; int Cy = cy + ty;

  int total_tiles = (numAColumns + TILE_WIDTH - 1) / TILE_WIDTH;

  for (int tile_idx = 0; tile_idx < total_tiles; tile_idx++) {
    // the corresponding tiles' top left corners are:
    // for A: row = blockIdx.y * blockDim.y, col = tile_idx * TILE_WIDTH
    // for B: row = tile_idx * TILE_WIDTH, col = blockIdx.x * blockDim.x
     // loading tiles
    int Ax = tile_idx * TILE_WIDTH + tx; int Ay = cy + ty;
    int Bx = cx + tx; int By = tile_idx * TILE_WIDTH + ty;
   if (Ax < numAColumns && Ay < numARows) {
      A_tile[ty][tx] = A[Ay * numAColumns + Ax];
    }
    else {
      A_tile[ty][tx] = 0.0;
    }
    if (Bx < numBColumns && By < numBRows) {
      B_tile[ty][tx] = B[By * numBColumns + Bx];
    }
    else {
      B_tile[ty][tx] = 0.0;
    }
    __syncthreads();
    // multiplying tiles
    for (int i = 0; i < TILE_WIDTH; i++) {
      sum += A_tile[ty][i] * B_tile[i][tx];
    }
    __syncthreads();
  }
  // saving result (discarded if we're in the wrong thread)
  if (Cx < numCColumns && Cy < numCRows) {
    C[Cy * numCColumns + Cx] = sum;
  }
}
```

Without shared memory:
Kernel reads one data point from the source matrix and stores it to target matrix. The data readings are aligned and coalesced. But storing operation is performed in column vector manner. Therefore, more storing operations will be performed writing data rather than only one write transaction.

```
__global__ void TransposeVector2D(Vector2D * res, Vector2D * m1)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;
	int tid = thx + thy*m1->width;
	if(tid < m1->width*m1->height)
	{
	res->data[thy+thx*m1->height] = m1->data[tid] ;
	}
}
```

With Shared Memory:
First matrix data is brought to shared memory. All readings are coalesced and aligned. The transpose operation is performed on shared memory. Each thread block brings it data to shared memory and writes to target location in shared memory. This writing operations cause bank confliction. To eliminate this flaw, on the target shared memory matrix padding technique is applied. After each block performs transpose operation, the storing this transposed data on shared memory to global memory is carried out in coalesced and aligned manner. So, we eliminate uncoalesced memory accesses by using shared memory.

```
__global__ void TransposeVector2DShared(Vector2D * res, Vector2D * m1)
{
	int thx = blockIdx.x*blockDim.x+ threadIdx.x;
	int thy = blockIdx.y*blockDim.y+threadIdx.y;

	int tid = thx + thy*m1->width;

	 __shared__ float ordered_data[BLOCK_Y][BLOCK_X+1];
	 __shared__ float transposed_data[BLOCK_Y][BLOCK_X+1];

	int j = threadIdx.x+blockDim.x*blockIdx.y;

	int k = threadIdx.y + blockDim.y*blockIdx.x;

	int target = j + res->width*k;

	if(tid < m1->width*m1->height)
	{
	//padded
	ordered_data[threadIdx.y][threadIdx.x] = m1->data[tid] ;

	}
	__syncthreads();
```

```
        if(thx < m1->width && thy< m1->height)
        {

        transposed_data[threadIdx.x][threadIdx.y]  = ordered_data[threadIdx.y][threadIdx.x];

        }
__syncthreads();

if(thx < m1->width && thy< m1->height)
        {
        res->data [target] = transposed_data[threadIdx.y][threadIdx.x] ;

        }}
```

Texture memory cache

No texture cache is used:

```
__global__ void MatrixAdd(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{
        int tx = blockIdx.x*blockDim.x+ threadIdx.x;
        int ty = blockIdx.y*blockDim.y + threadIdx.y;
        int tid = ty*vec1->width+tx;

        if(tid < vec1->width*vec1->height)
        {
              result->data[tid] = vec1->data[tid] + vec2->data[tid];
        }
}
```

Texture cache is used:

```
__global__ void MatrixAdd(Vector2D * __restrict__ result, Vector2D * __restrict__  vec1, Vector2D *
__restrict__  vec2)
{
        int tx = blockIdx.x*blockDim.x+ threadIdx.x;
        int ty = blockIdx.y*blockDim.y + threadIdx.y;
        int tid = ty*vec1->width+tx;
        if(tid < vec1->width*vec1->height)
        {
              result->data[tid] = vec1->data[tid] + vec2->data[tid];
        }
}
```

Texture memory is not used:

```
__global__ void MatrixSubtract(Vector2D * result, Vector2D * vec1, Vector2D * vec2)
{       int tx = blockIdx.x*blockDim.x+ threadIdx.x;
        int ty = blockIdx.y*blockDim.y + threadIdx.y;
```

```
        int tid = ty*vec1->width+tx;
        if(tid < vec1->width*vec1->height)
        {
            result->data[tid] = vec1->data[tid] - vec2->data[tid];
        }
}
```

Texture memory is used:

```
__global__ void MatrixSubtract(Vector2D * __restrict__ result, Vector2D * __restrict__  vec1,
Vector2D *  __restrict__  vec2)
{
        int tx = blockIdx.x*blockDim.x+ threadIdx.x;
        int ty = blockIdx.y*blockDim.y + threadIdx.y;
        int tid = ty*vec1->width+tx;
        if(tid < vec1->width*vec1->height)
        {
            result->data[tid] = vec1->data[tid] – vec2→data[tid];}}
```

The remaining functions are in the same form.

Comparison Results

For comparison, we have tested every code with 1 iteration 9 batch training 10 batch testing. During this execution the metrics of nvprof have been obtained. To find out whether the technique improved execution time upon the unoptimized code, we have run all code with 100 iteration whole training data and whole testing data. By doing so we have been able to see the effect of change in long run.

Unoptimized OneHiddenClassification Program Results

First program execution time : 212.580296 second program execution time : 208.290466
Average execution time : 210,435381

Kernel: Log2D(Vector2D*, Vector2D*)		Min	Max	Avg
gld_transactions	Global Load Transactions	3394	3394	3394
gst_transactions	Global Store Transactions	128	128	128
gld_efficiency	Global Memory Load Efficiency	53.79%	53.79%	53.79%
gst_efficiency	Global Memory Store Efficiency	89.06%	89.06%	89.06%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	611	611	611
global_store_requests	Total number of global store requests from Multiprocessor	118	118	118
sm_efficiency	Multiprocessor Activity	7.85%	8.44%	8.04%
achieved_occupancy	Achieved Occupancy	0.485524	0.489766	0.487825
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixProduct(Vector2D*, Vector2D*, Vector2D*)				
gld_transactions	Global Load Transactions	1074	157221906	32781061
gst_transactions	Global Store Transactions	1	524064	39199
gld_efficiency	Global Memory Load Efficiency	22.16%	82.50%	48.11%

gst_efficiency Global Memory Store Efficiency 50.00% 100.00% 76.86%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 232 37208995 8038451
global_store_requests Total number of global store requests from Multiprocessor 1 524064 39199
sm_efficiency Multiprocessor Activity 7.67% 99.89% 37.57%
achieved_occupancy Achieved Occupancy 0.111670 0.904035 0.462840
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: ScalarMinusVector2D(Vector2D*, float, Vector2D*)
gld_transactions Global Load Transactions 20482 20482 20482
gst_transactions Global Store Transactions 512 512 512
gld_efficiency Global Memory Load Efficiency 56.94% 56.94% 56.94%
gst_efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total number of global load requests from Multiprocessor 3072 3072 3072
global_store_requests Total number of global store requests from Multiprocessor 512 512 512
sm_efficiency Multiprocessor Activity 30.02% 35.13% 32.02%
achieved_occupancy Achieved Occupancy 0.456350 0.475226 0.466024
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)
Kernel: Sigmoid(Vector2D*, Vector2D*)
gld_transactions Global Load Transactions 14346 14346 14346
gst_transactions Global Store Transactions 512 512 512
gld_efficiency Global Memory Load Efficiency 59.34% 59.34% 59.34%
gst_efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
shared_efficiency Shared Memory Efficiency 0.00%
global_load_requests Total number of global load requests from Multiprocessor 2561 2561 2561
global_store_requests Total number of global store requests from Multiprocessor 512 512 512
sm_efficiency Multiprocessor Activity 35.96% 41.40% 39.54%
achieved_occupancy Achieved Occupancy 0.461936 0.476044 0.467086
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: ScalarMatrixProduct(Vector2D*, float, Vector2D*)
gld_transactions Global Load Transactions 2582 20967042 5250117
gst_transactions Global Store Transactions 1 524064 131160
gld_efficiency Global Memory Load Efficiency 21.21% 56.94% 38.92%
gst_efficiency Global Memory Store Efficiency 50.00% 100.00% 84.73%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 389 4193184 512019
global_store_requests Total number of global store requests from Multiprocessor 1 524064 63962
sm_efficiency Multiprocessor Activity 6.04% 99.73% 30.87%
achieved_occupancy Achieved Occupancy 0.352943 0.795298 0.472345
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: Sum2D(Vector2D*)

gld_transactions	Global Load Transactions	114	114	114
gst_transactions	Global Store Transactions	4	4	4
gld_efficiency	Global Memory Load Efficiency	24.81%	24.81%	24.81%
gst_efficiency	Global Memory Store Efficiency	100.00%	100.00%	100.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	24	24	24
global_store_requests	Total number of global store requests from Multiprocessor	4	4	4
sm_efficiency	Multiprocessor Activity	4.86%	7.36%	6.35%
achieved_occupancy	Achieved Occupancy	0.015616	0.015631	0.015622
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: Softmax(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	518	518	518
gst_transactions	Global Store Transactions	64	64	64
gld_efficiency	Global Memory Load Efficiency	24.56%	24.56%	24.56%
gst_efficiency	Global Memory Store Efficiency	25.00%	25.00%	25.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	89	89	89
global_store_requests	Total number of global store requests from Multiprocessor	16	16	16
sm_efficiency	Multiprocessor Activity	10.73%	12.00%	11.30%
achieved_occupancy	Achieved Occupancy	0.015623	0.015627	0.015624
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixSubtract(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	6338	6338	6338
gst_transactions	Global Store Transactions	128	128	128
gld_efficiency	Global Memory Load Efficiency	60.71%	60.71%	60.71%
gst_efficiency	Global Memory Store Efficiency	89.06%	89.06%	89.06%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	974	974	974
global_store_requests	Total number of global store requests from Multiprocessor	118	118	118
sm_efficiency	Multiprocessor Activity	4.21%	8.91%	7.76%
achieved_occupancy	Achieved Occupancy	0.468496	0.489397	0.483328
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixPairwiseProduct(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	6338	26626	19863
gst_transactions	Global Store Transactions	128	512	384
gld_efficiency	Global Memory Load Efficiency	60.71%	66.96%	64.88%
gst_efficiency	Global Memory Store Efficiency	89.06%	100.00%	96.35%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	974	4096	3055
global_store_requests	Total number of global store requests from Multiprocessor	118	512	380
sm_efficiency	Multiprocessor Activity	6.37%	38.30%	25.93%
achieved_occupancy	Achieved Occupancy	0.409192	0.479844	0.446007
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: TransposeVector2D(Vector2D*, Vector2D*)

gld_transactions Global Load Transactions 18234 4718574 1585080
gst_transactions Global Store Transactions 3984 1048562 352214
gld_efficiency Global Memory Load Efficiency 57.14% 62.50% 59.35%
gst_efficiency Global Memory Store Efficiency 12.50% 12.50% 12.50%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 2530 655357 220149
global_store_requests Total of global store requests from Multiprocessor 502 131071 44028
sm_efficiency Multiprocessor Activity 28.80% 98.91% 53.97%
achieved_occupancy Achieved Occupancy 0.382613 0.759363 0.521375
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: ArgMax2D(Vector2D*)
gld_transactions Global Load Transactions 134 134 134
gst_transactions Global Store Transactions 4 4 4
gld_efficiency Global Memory Load Efficiency 24.81% 24.81% 24.81%
gst_efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total number of global load requests from Multiprocessor 25 25 25
global_store_requests Total number of global store requests from Multiprocessor 4 4 4
sm_efficiency Multiprocessor Activity 6.40% 8.40% 7.44%
achieved_occupancy Achieved Occupancy 0.015621 0.015632 0.015626
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: Exponential(Vector2D*, Vector2D*)
gld_transactions Global Load Transactions 3394 3394 3394
gst_transactions Global Store Transactions 128 128 128
gld_efficiency Global Memory Load Efficiency 53.79% 53.79% 53.79%
gst_efficiency Global Memory Store Efficiency 89.06% 89.06% 89.06%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 611 611 611
global_store_requests Total of global store requests from Multiprocessor 118 118 118
sm_efficiency Multiprocessor Activity 6.43% 8.26% 7.50%
achieved_occupancy Achieved Occupancy 0.481851 0.488086 0.484999
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

A long run nvprof output :
==10654== Profiling application: ./OneHiddenClassification
==10654== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 79.26% 137.571s 194772 706.32us 2.2070us 15.877ms
MatrixProduct(Vector2D*, Vector2D*, Vector2D*)
8.87% 15.4011s 129786 118.67us 1.5680us 1.1177ms MatrixAdd(Vector2D*,
Vector2D*, Vector2D*)
8.09% 14.0482s 86400 162.59us 1.5680us 1.0158ms
ScalarMatrixProduct(Vector2D*, float, Vector2D*)

```
            3.26%  5.66601s     64800  87.438us  2.3360us  620.51us  TransposeVector2D(Vector2D*,
Vector2D*)
            0.14%  246.88ms        94  2.6264ms     512ns  167.73ms  [CUDA memcpy HtoD]
            0.07%  128.28ms     64800  1.9790us  1.6000us  17.888us
MatrixPairwiseProduct(Vector2D*, Vector2D*, Vector2D*)
            0.07%  114.24ms     21693  5.2660us  4.9600us  177.60us  Softmax(Vector2D*,
Vector2D*)
            0.04%  71.715ms     21693  3.3050us  2.8160us  18.144us  Sigmoid(Vector2D*,
Vector2D*)
            0.04%  61.940ms     43386  1.4270us  1.1830us  16.096us  PointerSet(Vector2D*,
Vector2D*, int, int)
            0.03%  53.689ms     21600  2.4850us  2.3360us  19.776us  Log2D(Vector2D*, Vector2D*)
            0.03%  50.599ms     21600  2.3420us  2.1760us  15.968us  MatrixSubtract(Vector2D*,
Vector2D*, Vector2D*)
            0.03%  47.755ms     21600  2.2100us  1.8880us  15.712us  Sum2D(Vector2D*)
            0.03%  46.123ms     21693  2.1260us  1.9840us  15.808us  Exponential(Vector2D*,
Vector2D*)
            0.03%  45.961ms     21600  2.1270us  2.0470us  20.544us
ScalarMinusVector2D(Vector2D*, float, Vector2D*)
            0.01%  17.283ms     21693     796ns     320ns  17.344us  [CUDA memcpy DtoH]
            0.00%  228.80us        93  2.4600us  2.2720us  3.2000us  ArgMax2D(Vector2D*)
      API calls:   96.81%  177.041s    735633  240.66us  1.6290us  15.902ms  cudaDeviceSynchronize
            2.21%  4.04183s    735516  5.4950us  4.2550us  45.583ms  cudaLaunch
            0.48%  880.29ms   2055069     428ns     122ns  544.72ms  cudaSetupArgument
            0.16%  295.20ms     21693  13.607us  8.9500us  90.177us  cudaMemcpyFromSymbol
            0.14%  248.01ms        94  2.6384ms  3.9640us  167.86ms  cudaMemcpy
            0.08%  149.70ms    735516     203ns     145ns  305.62us  cudaConfigureCall
            0.08%  144.57ms        56  2.5816ms  3.3160us  141.24ms  cudaMalloc
            0.04%  74.445ms         1  74.445ms  74.445ms  74.445ms  cudaDeviceReset
            0.00%  1.0087ms        86  11.729us     447ns  454.83us  cuDeviceGetAttribute
            0.00%  183.45us         1  183.45us  183.45us  183.45us  cuDeviceTotalMem
            0.00%  129.73us         1  129.73us  129.73us  129.73us  cuDeviceGetName
            0.00%  5.9250us         1  5.9250us  5.9250us  5.9250us  cuDeviceGetPCIBusId
            0.00%  4.4700us         3  1.4900us     428ns  3.3060us  cuDeviceGetCount
            0.00%  2.7730us         2  1.3860us     427ns  2.3460us  cuDeviceGet
            0.00%  2.0310us         1  2.0310us  2.0310us  2.0310us  cudaGetDeviceCount



79.26%  137.571s    194772  706.32us  2.2070us  15.877ms  MatrixProduct(Vector2D*, Vector2D*,
Vector2D*)
            8.87%  15.4011s    129786  118.67us  1.5680us  1.1177ms  MatrixAdd(Vector2D*,
Vector2D*, Vector2D*)
            8.09%  14.0482s     86400  162.59us  1.5680us  1.0158ms
ScalarMatrixProduct(Vector2D*, float, Vector2D*)
            3.26%  5.66601s     64800  87.438us  2.3360us  620.51us  TransposeVector2D(Vector2D*,
Vector2D*)
```

These three functions are the ones that must be optimized for performance gain.

Function combination and independent kernel execution.

Several functions are fused into one. We don’t put the metric results because the functions are different from the original one. Also, we have changed the execution order of independent kernels.

First program execution time : 163.206515 Second Program execution time : 170.467363
Average execution time : 166.836939

Long run nvprof output:
Accuracy : 1.444892
Tamam
Program execution time : 163.206515

==10774== Profiling application: ./OneHiddenClassificationFunctionCombination
==10774== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	82.90%	118.540s	194772	608.61us	2.1760us	4.3354ms	MatrixProduct(Vector2D*, Vector2D*, Vector2D*)
	12.30%	17.5821s	86400	203.50us	1.5990us	1.2613ms	ApplyWeightChange(Vector2D*, float, Vector2D*)
	4.28%	6.11896s	64800	94.428us	2.5280us	909.08us	TransposeVector2D(Vector2D*, Vector2D*)
	0.17%	240.27ms	94	2.5560ms	512ns	166.63ms	[CUDA memcpy HtoD]
	0.10%	143.21ms	21693	6.6010us	5.4400us	215.13us	Softmax(Vector2D*, Vector2D*)
	0.05%	67.843ms	21693	3.1270us	2.4320us	23.296us	AddandSigmoid(Vector2D*, Vector2D*, Vector2D*)
	0.04%	64.029ms	43386	1.4750us	1.1830us	18.464us	PointerSet(Vector2D*, Vector2D*, int, int)
	0.04%	59.791ms	21600	2.7680us	2.3680us	20.896us	calculateCrossEntropyLoss(Vector2D*, Vector2D*, Vector2D*)
	0.04%	55.511ms	21600	2.5690us	2.2080us	18.079us	Sum2D(Vector2D*)
	0.04%	55.309ms	21600	2.5600us	2.2400us	17.952us	MatrixSubtract(Vector2D*, Vector2D*, Vector2D*)
	0.03%	45.331ms	21600	2.0980us	1.8880us	20.992us	LayerErrorCalculate(Vector2D*, Vector2D*, Vector2D*)
	0.01%	18.707ms	21693	862ns	352ns	18.784us	[CUDA memcpy DtoH]
	0.00%	301.63us	93	3.2430us	2.9760us	3.6480us	ArgMax2D(Vector2D*)
API calls:	97.64%	144.716s	324768	445.60us	1.4850us	25.401ms	cudaDeviceSynchronize
	1.77%	2.62084s	519237	5.0470us	3.6120us	4.0409ms	cudaLaunchKernel
	0.20%	298.50ms	21693	13.760us	8.8550us	331.97us	cudaMemcpyFromSymbol
	0.18%	260.94ms	56	4.6597ms	3.3870us	254.76ms	cudaMalloc
	0.16%	241.31ms	94	2.5671ms	3.6260us	166.76ms	cudaMemcpy
	0.05%	76.838ms	1	76.838ms	76.838ms	76.838ms	cudaDeviceReset
	0.00%	740.16us	96	7.7100us	444ns	314.83us	cuDeviceGetAttribute

0.00%	208.83us	1	208.83us	208.83us	208.83us	cuDeviceTotalMem
0.00%	119.54us	1	119.54us	119.54us	119.54us	cuDeviceGetName
0.00%	4.7560us	3	1.5850us	413ns	3.5020us	cuDeviceGetCount
0.00%	2.5610us	1	2.5610us	2.5610us	2.5610us	cuDeviceGetPCIBusId
0.00%	2.4820us	2	1.2410us	445ns	2.0370us	cuDeviceGet
0.00%	2.3500us	1	2.3500us	2.3500us	2.3500us	cudaGetDeviceCount

So, before function combination original code has run in average 210,435381 seconds. Function combined version runs 166.836939 seconds. %20 speed gain has been obtained.

Unrolling technique

First program execution time : 177.111507 Second program execution time : 177.466072
Average : 177,2887895

Kernel: Log2D(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	2794	2794	2794
gst_transactions	Global Store Transactions	80	80	80
gld_efficiency	Global Memory Load Efficiency	46.05%	46.05%	46.05%
gst_efficiency	Global Memory Store Efficiency	90.00%	90.00%	90.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total number of global load requests from Multiprocessor	521	521	521
global_store_requests	Total number of global store requests from Multiprocessor	88	88	88
sm_efficiency	Multiprocessor Activity	10.32%	11.54%	10.66%
achieved_occupancy	Achieved Occupancy	0.223332	0.248030	0.234191
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixProduct(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	1074	157221906	32780924
gst_transactions	Global Store Transactions	1	524064	39199
gld_efficiency	Global Memory Load Efficiency	22.16%	82.50%	48.11%
gst_efficiency	Global Memory Store Efficiency	50.00%	100.00%	76.86%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	232	37208995	8038374
global_store_requests	Total number of global store requests from Multiprocessor	1	524064	39199
sm_efficiency	Multiprocessor Activity	8.21%	99.89%	36.90%
achieved_occupancy	Achieved Occupancy	0.123302	0.941063	0.462160
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: ScalarMinusVector2D(Vector2D*, float, Vector2D*)

gld_transactions	Global Load Transactions	12802	12802	12802
gst_transactions	Global Store Transactions	512	512	512
gld_efficiency	Global Memory Load Efficiency	68.98%	68.98%	68.98%
gst_efficiency	Global Memory Store Efficiency	100.00%	100.00%	100.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	1920	1920	1920
global_store_requests	Total of global store requests from Multiprocessor	512	512	512
sm_efficiency	Multiprocessor Activity	9.22%	10.96%	10.37%

achieved_occupancy	Achieved Occupancy	0.437226	0.487806	0.470328
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: Sigmoid(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	11274	11274	11274
gst_transactions	Global Store Transactions	512	512	512
gld_efficiency	Global Memory Load Efficiency	70.14%	70.14%	70.14%
gst_efficiency	Global Memory Store Efficiency	100.00%	100.00%	100.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	1793	1793	1793
global_store_requests	Total of global store requests from Multiprocessor	512	512	512
sm_efficiency	Multiprocessor Activity	12.53%	13.23%	13.02%
achieved_occupancy	Achieved Occupancy	0.454561	0.472193	0.465716
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: ScalarMatrixProduct(Vector2D*, float, Vector2D*)

gld_transactions	Global Load Transactions	2562	13102722	3287722
gst_transactions	Global Store Transactions	0	524064	131472
gld_efficiency	Global Memory Load Efficiency	20.83%	68.98%	46.04%
gst_efficiency	Global Memory Store Efficiency	0.00%	100.00%	72.23%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	384	1965408	493158
global_store_requests	Total of global store requests from Multiprocessor	0	524064	131426
sm_efficiency	Multiprocessor Activity	5.78%	99.50%	38.57%
achieved_occupancy	Achieved Occupancy	0.231258	0.909328	0.515578
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixAdd(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	2562	19391490	2372921
gst_transactions	Global Store Transactions	0	524064	64111
gld_efficiency	Global Memory Load Efficiency	20.83%	75.80%	58.07%
gst_efficiency	Global Memory Store Efficiency	0.00%	100.00%	83.92%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	384	3013536	368756
global_store_requests	Total of global store requests from Multiprocessor	0	524064	64091
sm_efficiency	Multiprocessor Activity	5.76%	99.63%	23.72%
achieved_occupancy	Achieved Occupancy	0.236834	0.926100	0.453854
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: Sum2D(Vector2D*)

gld_transactions	Global Load Transactions	114	114	114
gst_transactions	Global Store Transactions	4	4	4
gld_efficiency	Global Memory Load Efficiency	24.81%	24.81%	24.81%
gst_efficiency	Global Memory Store Efficiency	100.00%	100.00%	100.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	24	24	24
global_store_requests	Total of global store requests from Multiprocessor	4	4	4

sm_efficiency	Multiprocessor Activity	5.54%	7.87%	6.65%
achieved_occupancy	Achieved Occupancy	0.015616	0.015631	0.015624
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: Softmax(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	518	518	518
gst_transactions	Global Store Transactions	64	64	64
gld_efficiency	Global Memory Load Efficiency	24.56%	24.56%	24.56%
gst_efficiency	Global Memory Store Efficiency	25.00%	25.00%	25.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	89	89	89
global_store_requests	Total of global store requests from Multiprocessor	16	16	16
sm_efficiency	Multiprocessor Activity	11.18%	12.10%	11.65%
achieved_occupancy	Achieved Occupancy	0.015623	0.015627	0.015626
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixSubtract(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	5378	5378	5378
gst_transactions	Global Store Transactions	80	80	80
gld_efficiency	Global Memory Load Efficiency	53.41%	53.41%	53.41%
gst_efficiency	Global Memory Store Efficiency	90.00%	90.00%	90.00%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	824	824	824
global_store_requests	Total of global store requests from Multiprocessor	88	88	88
sm_efficiency	Multiprocessor Activity	9.57%	10.45%	10.02%
achieved_occupancy	Achieved Occupancy	0.253425	0.293739	0.274066
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: MatrixPairwiseProduct(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	5378	18946	14423
gst_transactions	Global Store Transactions	80	512	368
gld_efficiency	Global Memory Load Efficiency	53.41%	75.80%	68.33%
gst_efficiency	Global Memory Store Efficiency	90.00%	100.00%	96.67%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%
global_load_requests	Total of global load requests from Multiprocessor	824	2944	2237
global_store_requests	Total of global store requests from Multiprocessor	88	512	370
sm_efficiency	Multiprocessor Activity	8.63%	11.51%	10.50%
achieved_occupancy	Achieved Occupancy	0.249967	0.489317	0.408073
shared_utilization	Shared Memory Utilization	Idle (0)	Idle (0)	Idle (0)

Kernel: TransposeVector2D(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	13826	3538854	1199408
gst_transactions	Global Store Transactions	4096	1048520	355160
gld_efficiency	Global Memory Load Efficiency	58.98%	65.09%	61.30%
gst_efficiency	Global Memory Store Efficiency	12.50%	12.50%	12.50%
shared_efficiency	Shared Memory Efficiency	0.00%	0.00%	0.00%

global_load_requests Total of global load requests from Multiprocessor 2176 557041 188777
global_store_requests Total of global store requests from Multiprocessor 512 131068 44401
sm_efficiency Multiprocessor Activity 11.95% 98.40% 51.14%
achieved_occupancy Achieved Occupancy 0.413317 0.870657 0.576137
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: ArgMax2D(Vector2D*)
gld_transactions Global Load Transactions 134 134 134
gst_transactions Global Store Transactions 4 4 4
gld_efficiency Global Memory Load Efficiency 24.81% 24.81% 24.81%
gst_efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 25 25 25
global_store_requests Total of global store requests from Multiprocessor 4 4 4
sm_efficiency Multiprocessor Activity 6.34% 8.54% 7.08%
achieved_occupancy Achieved Occupancy 0.015618 0.015632 0.015626
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Kernel: Exponential(Vector2D*, Vector2D*)
gld_transactions Global Load Transactions 2794 2794 2794
gst_transactions Global Store Transactions 80 80 80
gld_efficiency Global Memory Load Efficiency 46.05% 46.05% 46.05%
gst_efficiency Global Memory Store Efficiency 90.00% 90.00% 90.00%
shared_efficiency Shared Memory Efficiency 0.00% 0.00% 0.00%
global_load_requests Total of global load requests from Multiprocessor 521 521 521
global_store_requests Total of global store requests from Multiprocessor 88 88 88

sm_efficiency Multiprocessor Activity 6.37% 9.97% 9.34%
achieved_occupancy Achieved Occupancy 0.257498 0.283186 0.267128
shared_utilization Shared Memory Utilization Idle (0) Idle (0) Idle (0)

Long run nvprof output :
Program execution time : 177.466072

==17408== Profiling application: ./OneHiddenClassificationUnroll
==17408== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 80.64% 112.595s 194772 578.09us 2.1440us 3.8781ms MatrixProduct(Vector2D*, Vector2D*, Vector2D*)
8.31% 11.6006s 129786 89.382us 1.2790us 949.91us MatrixAdd(Vector2D*, Vector2D*, Vector2D*)
6.37% 8.89828s 86400 102.99us 1.2790us 872.60us ScalarMatrixProduct(Vector2D*, float, Vector2D*)
3.83% 5.34408s 64800 82.470us 6.3040us 625.37us TransposeVector2D(Vector2D*, Vector2D*)

```
                0.22%  307.21ms        94  3.2682ms     512ns  220.42ms  [CUDA memcpy HtoD]
                0.14%  195.23ms     64800  3.0120us  2.4640us  171.49us
MatrixPairwiseProduct(Vector2D*, Vector2D*, Vector2D*)
                0.11%  155.41ms     21693  7.1630us  6.9440us  236.89us  Sigmoid(Vector2D*,
Vector2D*)
                0.08%  112.62ms     21693  5.1910us  4.9280us  172.25us  Softmax(Vector2D*,
Vector2D*)
                0.07%  92.583ms     21600  4.2860us  4.0320us  324.35us  Log2D(Vector2D*, Vector2D*)
                0.05%  73.142ms     21600  3.3860us  3.1680us  222.53us  MatrixSubtract(Vector2D*,
Vector2D*, Vector2D*)
                0.05%  65.097ms     21600  3.0130us  2.8480us  16.928us
ScalarMinusVector2D(Vector2D*, float, Vector2D*)
                0.05%  65.017ms     21693  2.9970us  2.7840us  18.496us  Exponential(Vector2D*,
Vector2D*)
                0.04%  60.867ms     43386  1.4020us  1.1520us  18.112us  PointerSet(Vector2D*,
Vector2D*, int, int)
                0.03%  41.195ms     21600  1.9070us  1.7280us  18.432us  Sum2D(Vector2D*)
                0.01%  16.281ms     21693     750ns     639ns  15.200us  [CUDA memcpy DtoH]
                0.00%  227.26us        93  2.4430us  2.2400us  10.080us  ArgMax2D(Vector2D*)
    API calls:   96.37%  141.745s    735633  192.68us  1.6030us  3.8846ms  cudaDeviceSynchronize
                2.72%  3.99738s    735516  5.4340us  4.2290us  4.1265ms  cudaLaunch
                0.24%  358.64ms        94  3.8153ms  4.1040us  220.54ms  cudaMemcpy
                0.21%  310.34ms   2055069     151ns     122ns  395.86us  cudaSetupArgument
                0.21%  304.90ms     21693  14.055us  9.3090us  227.66us  cudaMemcpyFromSymbol
                0.10%  147.97ms        56  2.6423ms  3.7540us  144.14ms  cudaMalloc
                0.10%  139.94ms    735516     190ns     138ns  432.63us  cudaConfigureCall
                0.05%  80.881ms         1  80.881ms  80.881ms  80.881ms  cudaDeviceReset
                0.00%  998.84us        86  11.614us     674ns  429.04us  cuDeviceGetAttribute
                0.00%  251.23us         1  251.23us  251.23us  251.23us  cuDeviceTotalMem
                0.00%  149.55us         1  149.55us  149.55us  149.55us  cuDeviceGetName
                0.00%  7.3260us         3  2.4420us     857ns  4.5690us  cuDeviceGetCount
                0.00%  4.3150us         1  4.3150us  4.3150us  4.3150us  cuDeviceGetPCIBusId
                0.00%  3.8930us         2  1.9460us  1.0890us  2.8040us  cuDeviceGet
                0.00%  3.0110us         1  3.0110us  3.0110us  3.0110us  cudaGetDeviceCount
```

If we focus on four bottleneck functions MatrixProduct, MatrixAdd, ScalarMatrixProduct, TransposeVector2D,

Matrix product unrolled:

```
 gld_requested_throughput Requested Global Load Throughput  129.48MB/s  207.33GB/s  152.35GB/s
gst_requested_throughput  Requested Global Store Throughput  1.9325MB/s  6.2180GB/s  1.4865GB/s

 gld_throughput  Global Load Throughput  525.63MB/s  259.60GB/s  186.69GB/s
  gst_throughput Global Store Throughput  3.8649MB/s  6.2180GB/s  1.4869GB/s
```

Matrix product rolled:

gld_requested_throughput Requested Global Load Throughput 203.49MB/s 232.36GB/s 180.50GB/s
gst_requested_throughput Requested Global Store Throughput 3.0372MB/s 6.9685GB/s 1.7613GB/s

gld_throughput Global Load Throughput 826.11MB/s 290.94GB/s 221.20GB/s
gst_throughput Global Store Throughput 4.9869MB/s 6.9685GB/s 1.7617GB/s

MatrixAdd unrolled:
gld_requested_throughput Requested Global Load Throughput 177.00MB/s 55.968GB/s 44.923GB/s
gst_requested_throughput Requested Global Store Throughput 4.0690MB/s 23.879GB/s 19.165GB/s

gld_throughput Global Load Throughput 821.94MB/s 83.585GB/s 67.109GB/s
gst_throughput Global Store Throughput 8.1380MB/s 23.879GB/s 19.168GB/s

MatrixAdd rolled:
gld_requested_throughput Requested Global Load Throughput 232.96MB/s 47.692GB/s 46.075GB/s
gst_requested_throughput Requested Global Store Throughput 0.00000B/s 21.420GB/s 20.691GB/s

gld_throughput Global Load Throughput 908.26MB/s 62.922GB/s 60.821GB/s
gst_throughput Global Store Throughput 0.00000B/s 21.420GB/s 20.699GB/s

ScalarMatrixProduct unrolled:
gld_requested_throughput Requested Global Load Throughput 169.45MB/s 34.955GB/s 30.203GB/s
gst_requested_throughput Requested Global Store Throughput 4.0346MB/s 27.281GB/s 23.568GB/s
gld_throughput Global Load Throughput 798.85MB/s 61.390GB/s 53.060GB/s
gst_throughput Global Store Throughput 8.0692MB/s 27.281GB/s 23.571GB/s

ScalarMatrixProduct rolled:
gld_requested_throughput Requested Global Load Throughput 232.96MB/s 47.692GB/s 46.075GB/s
gst_requested_throughput Requested Global Store Throughput 0.00000B/s 21.420GB/s 20.691GB/s
gld_throughput Global Load Throughput 908.26MB/s 62.922GB/s 60.821GB/s
gst_throughput Global Store Throughput 0.00000B/s 21.420GB/s 20.699GB/s

TransposeVector2D unrolled:
gld_requested_throughput Requested Global Load Throughput 3.0488GB/s 19.200GB/s 18.534GB/s
gst_requested_throughput Requested Global Store Throughput 2.4391GB/s 15.360GB/s 14.827GB/s
gld_throughput Global Load Throughput 4.8781GB/s 33.601GB/s 32.422GB/s
gst_throughput Global Store Throughput 19.513GB/s 122.88GB/s 118.62GB/s

TransposeVector2D rolled:
gld_requested_throughput Requested Global Load Throughput 232.96MB/s 47.692GB/s 46.075GB/s
gst_requested_throughput Requested Global Store Throughput 0.00000B/s 21.420GB/s 20.691GB/s
gld_throughput Global Load Throughput 908.26MB/s 62.922GB/s 60.821GB/s
gst_throughput Global Store Throughput 0.00000B/s 21.420GB/s 20.699GB/s

As we can observe from metrics above by unrolling more data requests and store operations are being demanded. When the number of instructions increases GPU has a better ability to handle instruction optimization, fullfilling the requested demands.

So before function combination original code has run in average 210,435381 seconds. Unrolled technique combined version runs 177,2887895 seconds. %15,7 speed gain has been obtained.

Shared Memory with Padding

There are two functions where shared memory is used. MatrixProduct and TransposeVector.

First program execution time : 140.887181 Second program execution time : 132.795527
Average : 136,841354

Kernel: MatrixProductShared(Vector2D*, Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	5766	27262082	2701366	
gst_transactions	Global Store Transactions	1	524064	39199	
gld_efficiency	Global Memory Load Efficiency	18.87%	91.37%	47.60%	
gst_efficiency	Global Memory Store Efficiency	50.00%	100.00%	76.86%	
shared_efficiency	Shared Memory Efficiency	70.00%	70.00%	70.00%	
global_load_requests	Total of global load requests from Multiprocessor	929	4718368	518055	
global_store_requests	Total of global store requests from Multiprocessor	1	524064	39199	
sm_efficiency	Multiprocessor Activity	4.48%	99.69%	35.17%	
achieved_occupancy	Achieved Occupancy	0.490692	0.942265	0.527091	
shared_utilization	Shared Memory Utilization	Low (1)	High (7)	Low (2)	

Kernel: TransposeVector2DShared(Vector2D*, Vector2D*)

gld_transactions	Global Load Transactions	18322	6290942	2111283	
gst_transactions	Global Store Transactions	128	131040	43893	
gld_efficiency	Global Memory Load Efficiency	42.16%	44.78%	43.16%	
gst_efficiency	Global Memory Store Efficiency	50.00%	100.00%	83.33%	
shared_efficiency	Shared Memory Efficiency	33.69%	100.00%	77.89%	
global_load_requests	Total of global load requests from Multiprocessor	3437	1179551	395865	
global_store_requests	Total of global store requests from Multiprocessor	128	131040	43893	
sm_efficiency	Multiprocessor Activity	9.07%	98.50%	53.66%	
achieved_occupancy	Achieved Occupancy	0.485462	0.926145	0.632801	
shared_utilization	Shared Memory Utilization	Low (1)	Low (1)	Low (1)	

Long term execution in nvprof:
Program execution time : 140.887181

==12736== Profiling application: ./OneHiddenClassificationSharedMem
==12736== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name

```
 GPU activities:   62.82%  59.8691s    194772  307.38us  2.5920us  2.4894ms
MatrixProductShared(Vector2D*, Vector2D*, Vector2D*)
                  16.60%  15.8225s    129786  121.91us  1.5360us  1.2451ms  MatrixAdd(Vector2D*,
Vector2D*, Vector2D*)
                  14.97%  14.2692s     86400  165.15us  1.5360us  1.2639ms
ScalarMatrixProduct(Vector2D*, float, Vector2D*)
                   4.64%  4.41929s     64800  68.198us  2.1440us  676.76us
TransposeVector2DShared(Vector2D*, Vector2D*)
                   0.28%  267.90ms        94  2.8499ms     512ns  184.99ms  [CUDA memcpy HtoD]
                   0.14%  132.01ms     64800  2.0370us  1.6630us  20.928us
MatrixPairwiseProduct(Vector2D*, Vector2D*, Vector2D*)
                   0.12%  114.77ms     21693  5.2900us  4.8960us  286.17us  Softmax(Vector2D*,
Vector2D*)
                   0.08%  73.697ms     21693  3.3970us  2.9760us  284.32us  Sigmoid(Vector2D*,
Vector2D*)
                   0.07%  65.025ms     43386  1.4980us  1.1520us  23.072us  PointerSet(Vector2D*,
Vector2D*, int, int)
                   0.06%  53.529ms     21600  2.4780us  2.3040us  18.272us  Log2D(Vector2D*, Vector2D*)
                   0.05%  50.352ms     21600  2.3310us  2.1760us  16.640us  MatrixSubtract(Vector2D*,
Vector2D*, Vector2D*)
                   0.05%  47.695ms     21600  2.2080us  1.8880us  22.848us  Sum2D(Vector2D*)
                   0.05%  46.108ms     21693  2.1250us  1.9520us  17.759us  Exponential(Vector2D*,
Vector2D*)
                   0.05%  45.188ms     21600  2.0920us  1.8880us  18.048us
ScalarMinusVector2D(Vector2D*, float, Vector2D*)
                   0.02%  19.372ms     21693     892ns     320ns  19.296us  [CUDA memcpy DtoH]
                   0.00%  207.49us        93  2.2310us  1.9830us  5.6960us  ArgMax2D(Vector2D*)
     API calls:   94.95%  104.730s    735633  142.37us  1.6200us  7.2172ms  cudaDeviceSynchronize
                   3.79%  4.17586s    735516  5.6770us  4.1260us  4.0064ms  cudaLaunch
                   0.34%  372.96ms     21693  17.192us  9.3710us  160.71us  cudaMemcpyFromSymbol
                   0.31%  342.73ms   2055069     166ns     126ns  426.67us  cudaSetupArgument
                   0.24%  269.58ms        94  2.8679ms  4.0420us  185.11ms  cudaMemcpy
                   0.16%  178.02ms    735516     242ns     156ns  399.27us  cudaConfigureCall
                   0.14%  150.39ms        56  2.6856ms  3.5320us  146.60ms  cudaMalloc
                   0.07%  76.008ms         1  76.008ms  76.008ms  76.008ms  cudaDeviceReset
                   0.00%  817.88us        86  9.5100us     443ns  359.22us  cuDeviceGetAttribute
                   0.00%  180.99us         1  180.99us  180.99us  180.99us  cuDeviceTotalMem
                   0.00%  111.38us         1  111.38us  111.38us  111.38us  cuDeviceGetName
                   0.00%  4.7350us         1  4.7350us  4.7350us  4.7350us  cuDeviceGetPCIBusId
                   0.00%  4.5910us         3  1.5300us     427ns  3.4610us  cuDeviceGetCount
                   0.00%  2.8570us         2  1.4280us     508ns  2.3490us  cuDeviceGet
                   0.00%  1.9700us         1  1.9700us  1.9700us  1.9700us  cudaGetDeviceCount

MatrixProduct without shared memory:
gld_transactions                        Global Load Transactions        1074   157221906    32781061
gst_transactions                        Global Store Transactions          1      524064       39199
```

gld_throughput Global Load Throughput 525.63MB/s 259.60GB/s 186.69GB/s
gst_throughput Global Store Throughput 3.8649MB/s 6.2180GB/s 1.4869GB/s

global_load_requests Total of global load requests from Multiprocessor 232 37208995 8038451
global_store_requests Total of global store requests from Multiprocessor 1 524064 39199

gld_efficiency Global Memory Load Efficiency 22.16% 82.50% 48.11%
gst_efficiency Global Memory Store Efficiency 50.00% 100.00% 76.86%
MatrixProduct with shared memory:
gld_transactions Global Load Transactions 5766 27262082 2701366
gst_transactions Global Store Transactions 1 524064 39199

gld_throughput Global Load Throughput 1.6716GB/s 58.979GB/s 29.479GB/s
gst_throughput Global Store Throughput 6.6343MB/s 15.723GB/s 3.5334GB/s

global_load_requests Total of global load requests from Multiprocessor 929 4718368 518055
global_store_requests Total of global store requests from Multiprocessor 1 524064 39199

gld_efficiency Global Memory Load Efficiency 21.21% 56.94% 38.92%
gst_efficiency Global Memory Store Efficiency 50.00% 100.00% 84.73%

TransposeVector2D without shared memory:
gld_transactions Global Load Transactions 18234 4718574 1585080
gst_transactions Global Store Transactions 3984 1048562 352214

gld_throughput Global Load Throughput 4.8781GB/s 33.601GB/s
32.422GB/s
gst_throughput Global Store Throughput 19.513GB/s 122.88GB/s 118.62GB/s

gld_throughput Global Load Throughput 4.8781GB/s 33.601GB/s 32.422GB/s
gst_throughput Global Store Throughput 19.513GB/s 122.88GB/s 118.62GB/s

gld_efficiency Global Memory Load Efficiency 57.14% 62.50% 59.35%
gst_efficiency Global Memory Store Efficiency 12.50% 12.50% 12.50%

TransposeVector2D with shared memory:
gld_transactions Global Load Transactions 18322 6290942 2111283
gst_transactions Global Store Transactions 128 131040 43893

gld_throughput Global Load Throughput 10.722GB/s 67.067GB/s 64.456GB/s
gst_throughput Global Store Throughput 927.63MB/s 21.036GB/s 20.165GB/s

global_load_requests Total of global load requests from Multiprocessor 3437 1179551 395865
global_store_requests Total of global store requests from Multiprocessor 128 131040 43893

gld_efficiency Global Memory Load Efficiency 42.16% 44.78% 43.16%
gst_efficiency Global Memory Store Efficiency 50.00% 100.00% 83.33%

From above, we can deduce that shared memory usage leads to coalesced and aligned reading and storing operations. In unoptimized form some readings and storings were unaligned. GPU performs more memory transaction to fulfill this demand. On the other hand, the reading is performed in coalesced and aligned manner GPU needed less memory transactions to fulfill this demand. Shared memory usage lessened total transactions to global memory.

So before shared memory usage original code has run in average 210,435381 seconds. Shared memory with padding runs 136,841354 seconds. %35 speed gain has been obtained.

Texture Cache

First program execution time : 208.765870 Second program execution time : 207.500291
Average : 208,1330805

Speed gain : %1,1 The effect of it is so small, we just show the most bottleneck function.

Texture not used:
Kernel: MatrixProduct(Vector2D*, Vector2D*, Vector2D*)

gst_transactions	Global Store Transactions	1	524064	39199
l2_write_transactions	L2 Write Transactions	14	524097	39213
gst_requested_throughput	Requested Global Store Throughput	1.9325MB/s	6.2180GB/s	1.4865GB/s
tex_cache_transactions	Unified Cache Transactions	268	39305476	8195230
global_store_requests	Total of global store requests from Multiprocessor	1	524064	39199

Texture used:
Kernel: MatrixProduct(Vector2D*, Vector2D*, Vector2D*)

gst_transactions	Global Store Transactions	1	524064	10952
l2_write_transactions	L2 Write Transactions	14	524077	10966
gst_requested_throughput	Requested Global Store Throughput	2.5718MB/s	6.2308GB/s	383.33MB/s
tex_cache_transactions	Unified Cache Transactions	269	39305477	8336842
global_store_requests	Total of global store requests from Multiprocessor	1	524064	10952

Usage of texture cache lessened write transactions and performed the same work in less number of store transactions.

Batch Size and Thread Block number:

We have tested several block sizes and batch sizes.

BATCH_SIZE	block_x	block_y	feed_block_x	feed_block_y	back_block_x	back_block_y	error	time	accuracy
32	32	32					1792	66.366664	76.713711
32	64	16							
32	16	64							
32	16	16					0	51.657997	1.444892
32	32	32	32	32	32	32	1792	66.366664	76.713711
32	64	16	32	32	32	32	0	67.016877	76.713711
32	16	64	32	32	32	32	503	66.146933	76.713711
32	16	16	32	32	32	32	461	66.406175	76.680106
32	32	32	32	32	16	16	3105	64.312346	52.016127
32	64	16	32	32	16	16	0	63.489224	51.276881
32	16	64	32	32	16	16	839	63.825981	52.184141
32	16	16	32	32	16	16	804	65.789716	52.284944
32	32	32	16	16	32	32	0	51.348242	1.444892
32	64	16	16	16	32	32	0	53.553986	1.444892
32	16	64	16	16	32	32	0	53.570938	1.444892
32	16	16	16	16	32	32	0	53.886724	1.444892
64	32	32	32	32	32	32	0	44.531389	1.42663
64	64	16	32	32	32	32	0	44.431587	1.42663
64	16	64	32	32	32	32	0	44.511612	1.42663
64	16	16	32	32	32	32	0	44.443287	1.42663
64	32	32	32	32	16	16	0	43.043412	1.42663
64	64	16	32	32	16	16			
64	16	64	32	32	16	16			
64	16	16	32	32	16	16			
64	32	32	16	16	32	32		40.474114	
64	64	16	16	16	32	32			
64	16	64	16	16	32	32			
64	16	16	16	16	32	32			
128	32	32	32	32	32	32		37.01504	
128	64	16	32	32	32	32			
128	16	64	32	32	32	32			
128	16	16	32	32	32	32			
128	32	32	32	32	16	16			
128	64	16	32	32	16	16			
128	16	64	32	32	16	16			
128	16	16	32	32	16	16			
128	32	32	16	16	32	32			
128	64	16	16	16	32	32			
128	16	64	16	16	32	32			
128	16	16	16	16	32	32			

256	35.088176
512	30.903113
1024	25.800068
2048	22.170835

We have used batch size of 1024 and thread block as 32 x 32.

Serial Code Comparisons

We have tested our serial, parallel and GPU code with the same amount of data and the same task. Each code is given 1600 data point and perform operation on that data.
Serial code run 1: 233.059089 run 2: 232.267144 Average: 232.6631165
Parallel(16 thread) code run 1: 103.712619 run 2: 111.021819 Average: 107.367219
Parallel(8 thread) code run 1: 103.926238 run 2: 103.134728 Average: 103.530483
Unoptimized code run 1: 0.635689 run 2: 0.633862 Average : 0.6347755
Tensorflow code run1: 0.404778003693 Run2: 0.412333011627 Average : 0.40855550766
Optimized code run 1: 0.430240 Run2: 0.441005 Average : 0.4356225

Unoptimized whole data run : (191.457468 + 190.580037)/2 = 191.0187525

Optimized whole data run: (38.038570+38.054162)/2 = 38.046366

Comparison with Other Project

https://github.com/xqding/NeuralNetwork_GPU-CUDA-_MNIST

Other project code execution times first run Program execution time : 10.119466 second Program execution time : 10.064995 Average : 10.0922305
Our unoptimized code execution time – First run Program execution time : 76.402052 Program execution time : 76.213952 Average: 76.308002

Optimized(32 batch) code run 1: 43.530097 run 2: 43.498238 Average : 43.5141675

Hacettepe Üniversitesi
Bilgisayar Mühendisliği
Bölümü

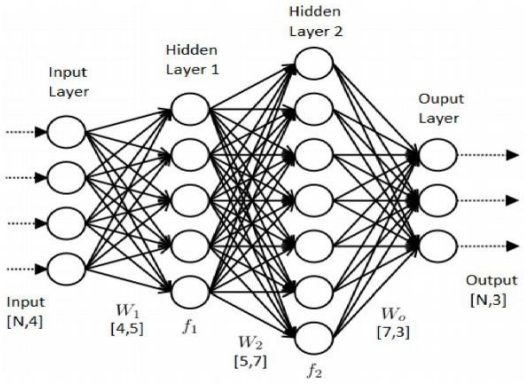
Text Classification with Multilayer Perceptron

BİL674 – GPU Programming
N12165419 Mustafa KÖSE
O18148203 Saim SUNEL

Outline

1. **What is MLP?**
   - MLP features and usage
   - Operations on MLP
   - MLP Training
2. **Optimization**
   - Function combination and execution order
   - Shared Memory with Padding
   - Unrolling technique
   - Texture Cache
   - Batch Size and Thread Block number:
3. **Dataset**
   - Reuters Data Set
4. **Comparisons**
   - Serial Code Comparisons
   - Comparison with Other Project

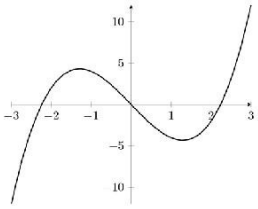
What is MLP?

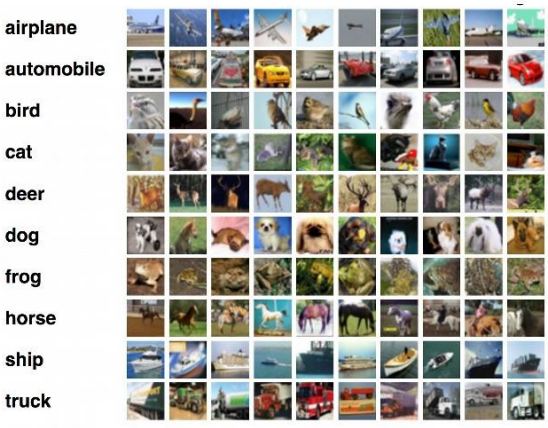


MLP Features and Usage

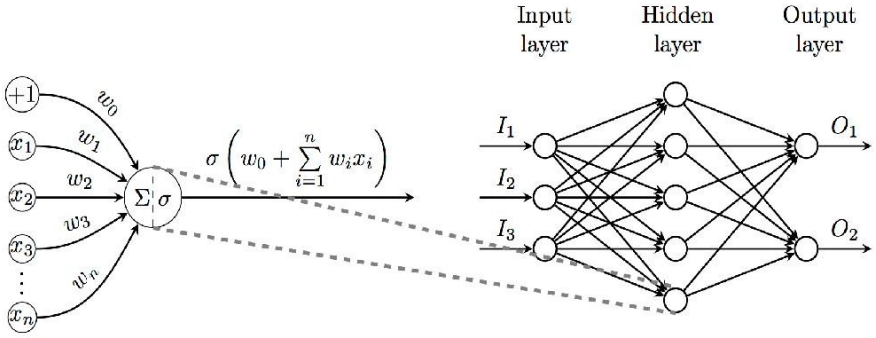
Universal Function Approximator
Automatic feature extractors
Generalization capability

Supervised Learning
Unsupervised Learning
Reinforcement learning





Operations on MLP



$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

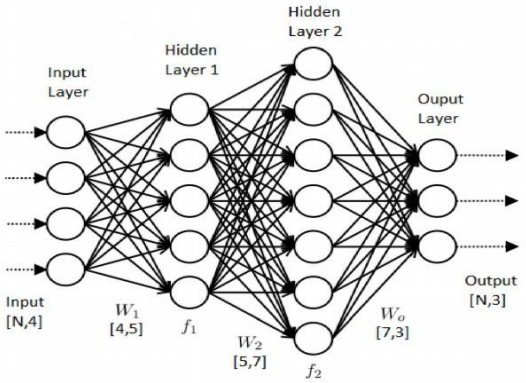
Operations on MLP

$$InputData = X^{(0)} = \begin{vmatrix} X_1^{(0)0} & X_2^{(0)0} & X_3^{(0)0} \\ X_1^{(0)1} & X_2^{(0)1} & X_3^{(0)2} \\ X_1^{(0)2} & X_2^{(0)2} & X_3^{(0)2} \\ X_1^{(0)3} & X_2^{(0)3} & X_3^{(0)3} \\ X_1^{(0)4} & X_2^{(0)4} & X_3^{(0)4} \end{vmatrix} \qquad TargetData = R = \begin{vmatrix} R_1^0 & R_2^0 & R_3^0 \\ R_1^1 & R_2^1 & R_3^1 \\ R_1^2 & R_2^2 & R_3^2 \\ R_1^3 & R_2^3 & R_3^3 \\ R_1^4 & R_2^4 & R_3^4 \end{vmatrix}$$

$$W^{(1)} = \begin{vmatrix} W_{1,1}^{(1)} & W_{1,2}^{(1)} & W_{1,3}^{(1)} & W_{1,4}^{(1)} & W_{1,5}^{(1)} \\ W_{2,1}^{(1)} & W_{2,2}^{(1)} & W_{2,3}^{(1)} & W_{2,4}^{(1)} & W_{2,5}^{(1)} \\ W_{3,1}^{(1)} & W_{3,2}^{(1)} & W_{3,3}^{(1)} & W_{3,4}^{(1)} & W_{3,5}^{(1)} \\ W_{4,1}^{(1)} & W_{4,2}^{(1)} & W_{4,3}^{(1)} & W_{4,4}^{(1)} & W_{4,5}^{(1)} \end{vmatrix} \qquad W^{(2)} = \begin{vmatrix} W_{1,1}^{(2)} & W_{1,2}^{(2)} & W_{1,3}^{(2)} & W_{1,4}^{(2)} & W_{1,5}^{(2)} & W_{1,6}^{(2)} & W_{1,7}^{(2)} \\ W_{2,1}^{(2)} & W_{2,2}^{(2)} & W_{2,3}^{(2)} & W_{2,4}^{(2)} & W_{2,5}^{(2)} & W_{2,6}^{(2)} & W_{2,7}^{(2)} \\ W_{3,1}^{(2)} & W_{3,2}^{(2)} & W_{3,3}^{(2)} & W_{3,4}^{(2)} & W_{5,5}^{(2)} & W_{3,6}^{(2)} & W_{3,7}^{(2)} \\ W_{4,1}^{(2)} & W_{4,2}^{(2)} & W_{4,3}^{(2)} & W_{4,4}^{(2)} & W_{4,5}^{(2)} & W_{4,6}^{(2)} & W_{4,7}^{(2)} \\ W_{5,1}^{(2)} & W_{5,2}^{(2)} & W_{5,3}^{(2)} & W_{5,4}^{(2)} & W_{5,5}^{(2)} & W_{5,6}^{(2)} & W_{5,7}^{(2)} \end{vmatrix}$$

$$B^{(.)} = \begin{vmatrix} B_1^{(1)} & B_2^{(1)} & B_3^{(1)} & B_4^{(1)} & B_5^{(1)} \end{vmatrix}$$
$$B^{(2)} = \begin{vmatrix} B_1^{(2)} & B_2^{(2)} & B_3^{(2)} & B_4^{(2)} & B_5^{(2)} & B_6^{(2)} & B_7^{(2)} \end{vmatrix}$$
$$B^{(3)} = \begin{vmatrix} B_1^{(3)} & B_2^{(3)} & B_3^{(3)} \end{vmatrix}$$
$$X_0^{(0)} = X_0^{(.)} = X_0^{(2)} = +1$$

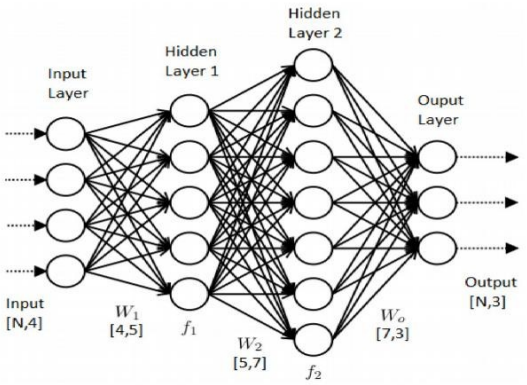
$$W_3 = W^{(3)} = \begin{vmatrix} W_{1,1}^{(3)} & W_{1,2}^{(3)} & W_{1,3}^{(3)} \\ W_{2,1}^{(3)} & W_{2,2}^{(3)} & W_{2,3}^{(3)} \\ W_{3,1}^{(3)} & W_{3,2}^{(3)} & W_{3,3}^{(3)} \\ W_{4,1}^{(3)} & W_{4,2}^{(3)} & W_{4,3}^{(3)} \\ W_{5,1}^{(3)} & W_{5,2}^{(3)} & W_{5,3}^{(3)} \\ W_{6,1}^{(3)} & W_{6,2}^{(3)} & W_{6,3}^{(3)} \\ W_{7,1}^{(3)} & W_{7,2}^{(3)} & W_{7,3}^{(3)} \end{vmatrix}$$



Operations on MLP

$$O^{(l)} = X^{(l-1)}.W^{(l)} + B^{(l)} \qquad X^{(l)} = \sigma(O^{(l)})$$
$$O^{(1)} = X^{(0)}.W^{(1)} + B^{(1)} \qquad X^{(1)} = \sigma(O^{(1)})$$
$$O^{(2)} = X^{(1)}.W^{(2)} + B^{(2)} \qquad X^{(2)} = \sigma(O^{(2)})$$
$$O^{(3)} = X^{(2)}.W^{(3)} + B^{(3)}$$

$$if\ regression\ Output = O^{(3)}\ if classification\ Output = Softmax(O^{(3)})$$



MLP Training

BackPropagation

A gradient descent algorithm

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

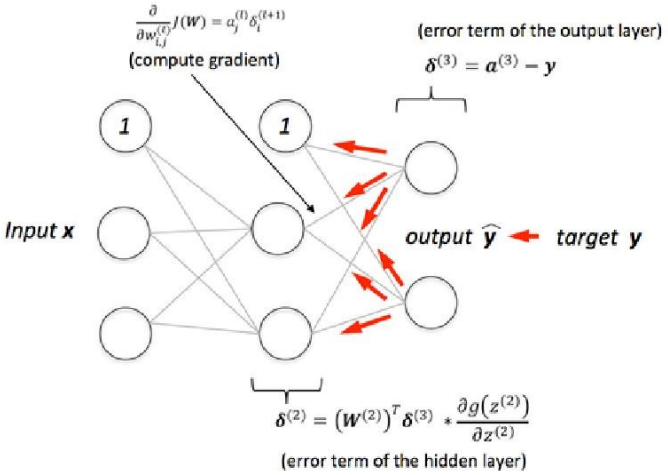
$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

$$\mathrm{MSE} = \frac{1}{n}\sum_{i-1}^{n}(Y_i - \hat{Y}_i)^2.$$

$$\Delta W^{(3)}_{(k,m)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m).X^{(2)t}$$

$$\Delta W^{(2)}_{(j,k)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m).W^{(3)}_{(k,m)}.\sigma(O^{(2)t}_k).(1-\sigma(O^{(2)t}_k)).X^{(1)t}_j$$

$$\Delta W^{(1)}_{(i,j)} = \eta . \sum_t \sum_m (R^t_m - O^{(3)t}_m).\sum_{k=1}.W^{(3)}_{(k,m)}.\sigma(O^{(2)t}_k).(1-\sigma(O^{(2)t}_k)).W^{(2)}_{(j,k)}.\sigma(O^{(1)t}_j).(1-\sigma(O^{(1)t}_j)).X^{(0)t}_i$$



MLP Training

$\odot$ *is Hadamard multiplication*

$$\textit{In matrix operations form:}$$
$$\textit{Output Error} = \textit{Error} = R - \textit{Output}$$
$$\Delta W^{(3)} = \eta . (X^{(2)})^T . \textit{Output Error}$$
$$\Delta W^{(2)} = \eta . (X^{(1)})^T . ((\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))$$
$$\Delta W^{(1)} = \eta . (X^{(0)})^T . ((((\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))) . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)}))))$$
$$\textit{Hidden Layer 2 Error} = (\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))$$
$$\textit{Hidden Layer 1 Error} = (\textit{Hidden Layer 2 Error} . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)}))$$
$$\Delta W^{(3)} = \eta . (X^{(2)})^T . \textit{Output Error}$$
$$\Delta W^{(2)} = \eta . (X^{(1)})^T . \textit{Hidden Layer 2 Error}$$
$$\Delta W^{(1)} = \eta . (X^{(0)})^T . \textit{Hidden Layer 1 Error}$$
$$\textit{LayerError}(l) = (\textit{LayerError}(l+1) . (W^{(l+1)})^T) \odot (\sigma(O^{(l)}) \odot (1 - \sigma(O^{(l)})))$$

Involving Matrix Operations

- Matrix Multiplication
- Matrix Addition
- Sigmoid Operation
- Matrix Subtraction
- Matrix Scalar Multiplication
- Matrix Transpose
- Matrix Pairwise Multiplication
- Matrix Scalar Subtraction
- Softmax Operation
- Log operation
- Matrix Element Sum

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

$$O^{(l)} = X^{(l-1)} . W^{(l)} + B^{(l)} \qquad X^{(l)} = \sigma(O^{(l)})$$
$$O^{(1)} = X^{(0)} . W^{(1)} + B^{(1)} \qquad X^{(1)} = \sigma(O^{(1)})$$
$$O^{(2)} = X^{(1)} . W^{(2)} + B^{(2)} \qquad X^{(2)} = \sigma(O^{(2)})$$
$$O^{(3)} = X^{(2)} . W^{(3)} + B^{(3)}$$

$$if\ regression\ Output = O^{(3)}\ if classification\ Output = Softmax(O^{(3)})$$

$$\textit{In matrix operations form:}$$
$$\textit{Output Error} = \textit{Error} = R - \textit{Output}$$
$$\Delta W^{(3)} = \eta . (X^{(2)})^T . \textit{Output Error}$$
$$\Delta W^{(2)} = \eta . (X^{(1)})^T . ((\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))$$
$$\Delta W^{(1)} = \eta . (X^{(0)})^T . ((((\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))) . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)}))))$$
$$\textit{Hidden Layer 2 Error} = (\textit{Output Error} . (W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1 - \sigma(O^{(2)}))$$
$$\textit{Hidden Layer 1 Error} = (\textit{Hidden Layer 2 Error} . (W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1 - \sigma(O^{(1)}))$$
$$\Delta W^{(3)} = \eta . (X^{(2)})^T . \textit{Output Error}$$
$$\Delta W^{(2)} = \eta . (X^{(1)})^T . \textit{Hidden Layer 2 Error}$$
$$\Delta W^{(1)} = \eta . (X^{(0)})^T . \textit{Hidden Layer 1 Error}$$
$$\textit{LayerError}(l) = (\textit{LayerError}(l+1) . (W^{(l+1)})^T) \odot (\sigma(O^{(l)}) \odot (1 - \sigma(O^{(l)})))$$

Optimization
Function combination and Execution order

```
MatrixAdd<<<grid, block>>>(output_2, output_2, bias_result_2);
Exponential<<<grid, block>>>(output_2, output_2);

AddandExponential<<<grid, block>>>(output_2, output_2, bias_result_2);
*********************************************************************************
MatrixSubtract<<<grid, block>>>(layer_2_error, label, output_2);
cudaDeviceSynchronize();
TransposeVector2D<<<grid2, block>>>(output_1_transpose, output_1);
cudaDeviceSynchronize();

MatrixSubtract<<<grid, block>>>(layer_2_error, label, output_2);
TransposeVector2DShared<<<grid2, block>>>(output_1_transpose,output_1);
cudaDeviceSynchronize();
```

Unoptimized code average run time 210,435381.

Optimized code average run time 166.836939.

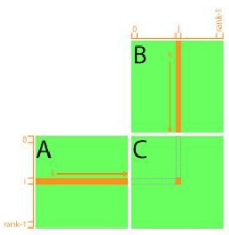
Speed gain : 20%

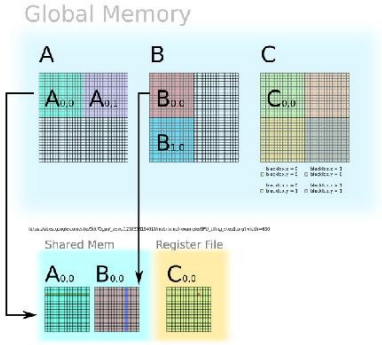
Optimization
Shared Memory with Padding

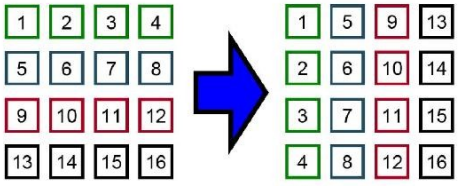
Matrix Product / Transpose

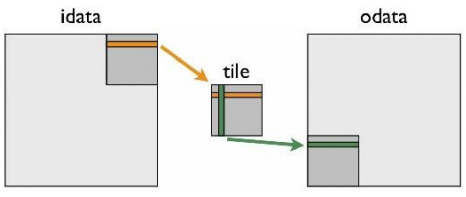
Unoptimized
code run time : 210,435
Optimized
code run time : 136,841

Speed gain : 35%









Shared Memory Metrics

	w/ shared memory (Matrix Product)			w/o shared memory (Matrix Product)		
Metric Name	Min	Max	Avg	Min	Max	Avg
gld_transactions Global Load Transactions	5766	27262082	2701366	1074	157221906	32781061
gst_throughput Global Store Throughput	6.6343MB/s	15.723GB/s	3.5334GB/s	3.8649MB/s	6.2180GB/s	1.4869GB/s
global_load_requests Total of global load requests from Multiprocessor	929	4718368	518055	232	37208995	8038451
gst_efficiency Global Memory Store Efficiency	50.00%	100.00%	84.73%	50.00%	100.00%	76.86%

Optimization
Unrolling technique

```
#pragma unroll 4
for(int a = 0; a < vec1->width;a++)
{toplam += vec1->data[a+tid*vec1->width];}
```

Unoptimized code run time : 210,435381

Optimized code run time : 177,2887895

Speed gain : 15,7%

```
if(tid +3*blockDim.x< source->width*source->height)
{
result->data[tid] += learning_rate*source->data[tid];
result->data[tid+blockDim.x] += learning_rate*source->data[tid+blockDim.x];
result->data[tid+2*blockDim.x] += learning_rate*source->data[tid+2*blockDim.x];
result->data[tid+3*blockDim.x] += learning_rate*source->data[tid+3*blockDim.x];
}
```

Unrolling Metrics

	unRolled (Matrix Product)			Rolled (Matrix Product)		
Metric Name	Min	Max	Avg	Min	Max	Avg
gld_requested_throughput Requested Global Load Throughput	129.48MB/s	207.33GB/s	152.35GB/s	203.49MB/s	232.36GB/s	180.50GB/s
gst_requested_throughput Requested Global Store Throughput	1.9325MB/s	6.2180GB/s	1.4865GB/s	3.0372MB/s	6.9685GB/s	1.7613GB/s
gld_throughput Global Load Throughput	525.63MB/s	259.60GB/s	186.69GB/s	826.11MB/s	290.94GB/s	221.20GB/s
gst_throughput Global Store Throughput	3.8649MB/s	6.2180GB/s	1.4869GB/s	4.9869MB/s	6.9685GB/s	1.7617GB/s

Optimization
Texture Cache

```
__global__ void MatrixAdd(Vector2D * __restrict__ result, Vector2D * __restrict__ vec1, Vector2D *
__restrict__ vec2)
{
        int tx = blockIdx.x*blockDim.x+ threadIdx.x;
        int ty = blockIdx.y*blockDim.y + threadIdx.y;
        int tid = ty*vec1->width+tx;
        if(tid < vec1->width*vec1->height)
        {
                result->data[tid] = vec1->data[tid] + vec2->data[tid];
        }
}
```

Unoptimized code run time : 210,435381

Optimized code run time : 208,1330805

Speed gain : 1,1%

Texture Cache Metrics

Kernel: MatrixProduct(Vector2D*, Vector2D*, Vector2D*)

	Texture Used			Texture Not Used		
Metric Name	Min	Max	Avg	Min	Max	Avg
gst_transactions Global Store Transactions	1	524064	10952	1	524064	39199
l2_write_transactions L2 Write Transactions	4	524077	10966	14	524097	39213
gst_requested_throughput Requested Global Store Throughput	2.5718MB/s	6.2308GB/s	383.33MB/s	1.9325MB/s	6.2180GB/s	1.4865GB/s
tex_cache_transactions Unified Cache Transactions	269	39305477	8336842	268	39305476	8195230
global_store_requests Total of global store requests from Multiprocessor	1	524064	10952	1	524064	39199

Optimization
Batch Size and Thread Block number

Batch Size	Main Block X	Main Block Y	Feed Block X	Feed Block Y	Back Block X	Back Block Y	Time
32	32	32	32	32	32	32	66.366664
32	64	16	32	32	32	32	67.016877
32	16	64	32	32	32	32	66.146933
32	16	16	32	32	32	32	66.406175
64	32	32	32	32	32	32	44.531389
64	64	16	32	32	32	32	44.431587
64	16	64	32	32	32	32	44.511612
64	16	16	32	32	32	32	44.443287
128	32	32	32	32	32	32	37.01504
512	32	32	32	32	32	32	30.903113
1024	32	32	32	32	32	32	25.800068
2048	32	32	32	32	32	32	22.170835

Iteration : 100

Avg. Accuracy : 76.7

Iteration : 300

Avg. Accuracy : 94.31

Data

Reuters RCV1/RCV2 Multilingual, Multiview Text Categorization Test collection

- Reuters-21578 text categorization test collection is a resource for research in information retrieval, machine learning, and other corpus-based research.

- data analysis

  http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf

- data link

  http://www.daviddlewis.com/resources/testcollections/reuters21578/

  https://link.springer.com/content/pdf/bbm%3A978-3-642-04533-2%2F1.pdf

  http://ama.liglab.fr/~amini/DataSets/Classification/Multiview/ReutersMutliLingualMultiView.htm

Data

Language	# Documents	Percentage	Vocabulary Size
English	18,758	16.78	21,531
French	26,648	23.45	24,893
German	29,953	26.80	34,279
Italian	24,039	21.51	15,506
Spanish	12,342	11.46	11,547
TOTAL:	111,740	100.00	

Class	# Documents	Percentage
C15	18,816	16.84
CCAT	21,426	19.17
E21	13,701	12.26
ECAT	19,198	17.18
GCAT	19,178	17.16
MCAT	19,421	17.39
TOTAL:	111,740	100.00

Data

Language : ENG

Vocab Size : 32754

- CCAT (Corporate/Industrial)
- ECAT (Economics)
- GCAT (Government/Social)
- MCAT(Markets)

Class	# Documents	Percentage
CCAT	144	1.5
ECAT	1784	17.9
GCAT	3188	32.0
M11	4838	48.6
TOTAL :	9954	100.00

Serial Code Comparisons

Each code is given 1600 data point (1600x32754) and perform backpropagation and error calculation on that data in 1 iteration.

Whole data + 100 iteration
Unoptimized whole data run : 191.0187525
Optimized whole data run : 38.046366

Text data accuracy Tensorflow 95.0418760469

Our code accuracy 94.318181

CODE	1st Run (sec)	2nd Run (sec)	Avg. (sec)
Serial	233.51	232.26	232.66
Parallel(16 thread)	103.71	111.02	107.36
Parallel(8 thread)	103.92	103.13	103.53
Tensorflow	0.40	0.42	0.41
Unoptimized	0.63	0.63	0.63
Optimized	0.43	0.44	0.43

Other Project Comparison

MNIST
https://github.com/xqding/NeuralNetwork_GPU-CUDA-_MNIST

Dataset,

28x28 handwritten images,

50000 training data,

10000 test data,

Batch 32,

Iteration 100

CODE	1st Run	2nd Run	Avg.
MNIST	10.119	10.064	10.092
unOptimized	76.402	76.213	76.308
Optimized	43.530	43.498	43.514

All codes related to our project can be found at
https://github.com/SaimSUNEL/MLPForRegressionandClassification

Important Correction

We have realized mathematical notation misusage in our formulas after sending first assignment. In this formulate Hadamard Multiplication operation was not represented correctly in formula.

Wrong part sent :

$$
\begin{gathered}
In\ matrix\ operations\ form: \\
Output\ Error = Error = R - Output \\
\Delta W^{(3)} = \eta.\,(X^{(2)})^T.\,Output\ Error \\
\Delta W^{(2)} = \eta.\,(X^{(1)})^T.\,((Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}).\,(1-\sigma(O^{(2)})) \\
\Delta W^{(1)} = \eta.\,(X^{(0)})^T.\,((((Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}).\,(1-\sigma(O^{(2)}))).\,(W^{(2)})^T) \odot (\sigma(O^{(1)}).\,(1-\sigma(O^{(1)})))) \\
Hidden\ Layer\ 2\ Error = (Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}).\,(1-\sigma(O^{(2)})) \\
Hidden\ Layer\ 1\ Error = (Hidden\ Layer\ 2\ Error.\,(W^{(2)})^T) \odot (\sigma(O^{(1)}).\,(1-\sigma(O^{(1)})) \\
\Delta W^{(3)} = \eta.\,(X^{(2)})^T.\,Output\ Error \\
\Delta W^{(2)} = \eta.\,(X^{(1)})^T.\,Hidden\ Layer\ 2\ Error \\
\Delta W^{(1)} = \eta.\,(X^{(0)})^T.\,Hidden\ Layer\ 1\ Error \\
LayerError(l) = (LayerError(l+1).\,(W^{(l+1)})^T) \odot (\sigma(O^{(l)}).\,(1-\sigma(O^{(l)})))
\end{gathered}
$$

Corrected version:

$$
\begin{gathered}
In\ matrix\ operations\ form: \\
Output\ Error = Error = R - Output \\
\Delta W^{(3)} = \eta.\,(X^{(2)})^T.\,Output\ Error \\
\Delta W^{(2)} = \eta.\,(X^{(1)})^T.\,((Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1-\sigma(O^{(2)}))) \\
\Delta W^{(1)} = \eta.\,(X^{(0)})^T.\,((((Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1-\sigma(O^{(2)}))).\,(W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1-\sigma(O^{(1)})))) \\
Hidden\ Layer\ 2\ Error = (Output\ Error.\,(W^{(3)})^T) \odot (\sigma(O^{(2)}) \odot (1-\sigma(O^{(2)})) \\
Hidden\ Layer\ 1\ Error = (Hidden\ Layer\ 2\ Error.\,(W^{(2)})^T) \odot (\sigma(O^{(1)}) \odot (1-\sigma(O^{(1)})) \\
\Delta W^{(3)} = \eta.\,(X^{(2)})^T.\,Output\ Error \\
\Delta W^{(2)} = \eta.\,(X^{(1)})^T.\,Hidden\ Layer\ 2\ Error \\
\Delta W^{(1)} = \eta.\,(X^{(0)})^T.\,Hidden\ Layer\ 1\ Error \\
LayerError(l) = (LayerError(l+1).\,(W^{(l+1)})^T) \odot (\sigma(O^{(l)}) \odot (1-\sigma(O^{(l)})))
\end{gathered}
$$