Optimizing Feed-forward Neural Network Performance on GPU using CUDA

I. Problem Description

Artificial neural network is a computational model inspired by the structure of central nervous system of animal or human. The most widely applications which use artificial neural network are applications that are related to pattern recognition and machine learning application. However, the research of implementing artificial neural network in many other applications are being explored. Artificial neural network is compelling for use in many applications because many tasks which are hard to solve using rule-based programming, such as image recognition or computer vision, can be solved using artificial neural network model.

Neural network is usually represented as interconnected neurons and there are many kinds of neural network models that have been proposed to solve some real life problems. Feedforward neural network and Restricted Boltzmann Machine (RBM) are two of the most popular neural network models.

Fully connected (between neurons in adjacent layers) neural network and convolutional neural network are two types of feed-forward neural network. The earlier is a generic representation of feed-forward neural network and useful as starting point of implementing neural network to solve a new problem. The later is a tuned feed-forward neural network for certain applications, in particular this works really well for image recognition related applications but not necessarily works well for any other applications.

Restricted Boltzmann Machine is different from feedforward neural network where there is no connections between visible neurons as well as between hidden neurons. Since the structure of RBM is different with feed-forward neural network, this might offer unique behavior in solving problem. Different type of artificial neural network offers different capability in solving problems as well as different implementation complexity. So, which model suitable for solving problem depends on the kinds of problem that we are going to solve.

In this project, we focus on feed-forward artificial neural network with full-connections between neurons in the adjacent layers. This model is a generic representation of feed-forward artificial neural network and this model is usually used as a generic template of any feed-forward artificial neural network applications which later will be configured into a more specific type of neural network for performance, in term of correct output as well as execution speed. The fully connected feedforward neural network is chosen because this generic model is potentially more useful for many other applications that are currently being explored than any other more specialized type of neural network.

II. Fully-connected Feed-forward Artificial Neural Network

A graphical representation of a fully-connected feed-forward artificial neural network is depicted in Figure 1. It is called fully-connected because each neurons in a given layer is connected to all neurons in its adjacent layers, both before and after except for input and output layers. Input of this neural network is directly assigned to each of its input neurons. The value of each neurons on hidden layers as well as output layer are computed based on the value of its previous neurons. Mathematically, with an example of illustration in Figure 1, to calculate the value of neuron *H*, we can use the Equation 1.

$$H = f(Ax+By+Cz) \quad (1)$$

For Equation 1, *A*, *B*, and *C* are the value of the neurons in the previous layer of *H*. There is an associated weight of each connections between neuron *H* and neurons in its previous layer and they are *x*, *y*, and *z* for weight of connection between *H* and *A*, *B*, and *C*, respectively. The *f(n)* in that equation is an activation function with input *n*. This computation can continue until all the output neurons values are computed and those values are the output of the neural network model.

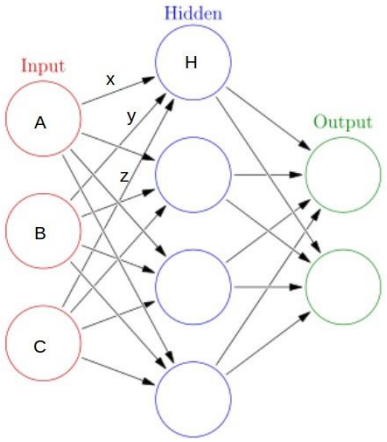


Fig. 1. Illustration of fully-connected feed-forward artificial neural network[2].

There are two modes of feed-forward neural network execution: feed-forward run mode and train mode. For feed-forward run, the values of input neurons is fed into the next layer until all the output neurons values are computed. For train run, it is the same as feed-forward run but after all the output values are computed, the delta of each output values with the desired values are computed and used to compute mean squared error (MSE). The delta can be used again as an input of another run, called back-propagation run, to distribute the delta into each of neurons in hidden layers. The back-propagation run is similar with the feed-forward run but it propagates backward and activation derived function is used instead of activation function. Finally, once all the delta are distributed to each of neurons, all the weight of connections connected to it is updated based on certain algorithm, such as gradient descent.

III. Related Work

There are some libraries which implement artificial neural network. One of them is Fast Artificial Neural Network (FANN)[4] which implement basic feed-forward neural network library. This library doesn't have CUDA version, so we use this library as our baseline configuration.

There are some other libraries which implement CUDA version of feed-forward artificial neural network. Most of them, if not all, are optimizing convolutional neural network which are widely used in image recognition type of applications. One of the library is called NeuralNetwork CUDA library [3] which could be found in code project web site. We observe that this library uses CUDA to optimize convolutional neural network and they implement similar to our naive parallel implementation. There is also a neural network library provided by NVIDIA called cuDNN [5]. Based on its documentation, it is highly optimized for convolutional type of neural network. Another library which is doing similarly is called Caffe [6]. This is a huge type of library which allows the user to customize many kinds of neural network. We observe that all of the libraries mentioned here are optimized for convolutional neural network.

Our optimization is targeting the fully-connected feedforward neural network, the generic feed-forward neural network which could be useful for exploring the neural network for new applications. In addition, we observe that most of the libraries, if not all, are optimizing the computation within a layer transfer, that is computation from a given layer to the next layer. Our optimizations include multiple-layers optimization which improves the performance further.

IV. Serial Solution

For As mentioned before in the previous section, there are one execution part for feed-forward run and four execution parts for train run. The pseudo-code for feed-forward run, delta and MSE computation, back-propagation run, and update weight are shown in Figure 2, Figure 3, Figure 4, and Figure 5, respectively.

The feed-forward and back-propagation runs are essentially similar to vector-matrix multiplication but we apply activation or activation derived function before storing the value of each element in the output vector.

```
Parameter: N (number of layers), E (vectors of neurons for each layer)
Variable: e, e', Value of each neuron
Variable: nSumNeuron, Intermediate value of a targeted neuron
Function: activation(), Activation function

for each n layers, n, in 0:N-2 do
  for each e in E[n+1] do
    nSumNeuron = 0
    for each e' in E[n] do
      nSumNeuron += (weight between e' and e) x e'
    end for
    e = activation(nSumNeuron)
  end for
end for
```

Fig. 2. Feed-forward run pseudo-code.

```
Parameter: N (number of outputs)
Parameter: E (vector of neurons for output layer)
Parameter: E' (vector of neurons for desired output)
Output: nMSE = 0, Value of Mean Square Error
Output: D (vector of delta for outputs)
Function: activation_derived(), Activation function

for each n, in 0:N-1 do
  nMSE += (E[n] - E'[n])^2
  D[n] = (E[n] - E'[n]) x activation_derived(E[n]);
end for
nMSE = nMSE/N
```

Fig. 3. Delta and MSE computation pseudo-code.

```
Parameter: N (number of layers), E (vectors of neurons for each layer)
Parameter/Output: D (vectors of delta)
Variable: e, e', Value of each neuron
Function: activation_derived(), Activation function

for each n layers, n, in N-1:1 do
  for each e in E[n] do
    for each e' in E[n-1] do
      D[n-1] += (weight between e' and e) x D[n]
    end for
    D[n] x= activation_derived(e)
  end for
end for
```

Fig. 4. Back-propagation run pseudo-code.

```
Parameter: N (number of layers)
Parameter: D (vectors of delta)
Variable: e, e', Value of each neuron

for each n layers, n, in 0:N-2 do
  for each e in E[n+1] do
    for each e' in E[n] do
      update (weight between e' and e) based on D[n+1]
    end for
  end for
end for
```

Fig. 5. Update weight pseudo-code.

V. Parallelization Strategy

Before going into more detail about parallelizing the neural network implementation, we analyze the implementation of neural network in FANN library. We observe that the data structure of FANN is stored in one big fann object which is quite hard to directly parallelize. So, the first step is to make a

customized CPU version of FANN which uses data structures that is easier to parallelize, such as making weight matrices in a big array and use an array of pointer to point to the start offset of weight matrix on a particular location in

Fig. 6. Data and task decomposition for feed-forward run naive parallelization.

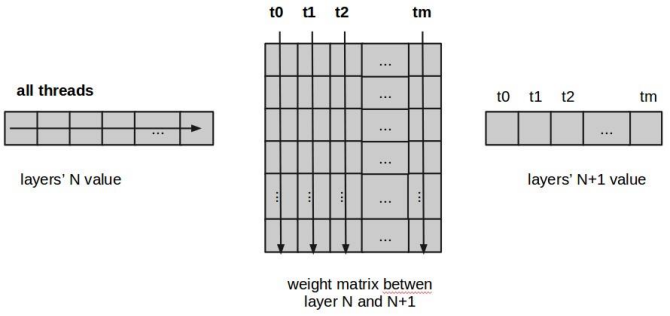
```
Parameter: N (number of layers), E (vectors of neurons for each layer)
Variable: e, e', Value of each neuron
Variable: nSumNeuron, Intermediate value of a targeted neuron
Function: activation(), Activation function

for each n layers, n, in 0:N-2 do
  invoke GPU kernel
    nSumNeuron = 0
    for each neurons e' in E[n] do
      nSumNeuron += (weight between e' and e) x e'
    end for
    e = activation(nSumNeuron)
  end invoke
end for
```

Fig. 7. Feed-forward run naive parallelization pseudo-code.

neural network. Once we restructurize the data structure, we then proceed with parallelizing the code.

For parallel implementation, since most of the computation is done during feed-forward and back-propagation runs, we will focus on those two runs. However, since back-propagation run is essentially similar to feed-forward run, the parallelization strategy applied to feed-forward run could also be applied for back-propagation run. Thus, for the rest of the report, we will show the parallelization strategy and optimization for feedforward run only.

Task decomposition. Naively, to parallelize the feedforward run could be done by dividing the feed-forward run into several steps. Each step will compute all the values in a layer from the values in its previous layer. Since each of this step is inherently parallel, we can decompose the task so that each thread is responsible to compute the value of one neuron. This simple technique doesn't require atomic operation because none of



the threads will write the same memory location.

Data decomposition. As the computation of neurons are evenly distributed to each thread, each thread will access the matrix weight elements associated with the assigned neuron. Figure 6 shows the way feed-forward neural network is naively computed in parallel. From the figure, we can see that each thread will access all the data in one column of weight matrix and all the threads will access the same input values from its previous layer.

Pseudo-code of naive parallel implementation. The pseudo-code of naive parallel implementation is shown in Figure 7.

Fig. 8. Illustration of parallel optimization #1.

```
For each layer transfer
  Invoke GPU kernel single step
    collectively read hidden_prev data into shared memory
    clear sum
    __synchthreads()
    for each element in a column:
      compute sum
    neuron[x] = f(sum)
```

Fig. 9. Feed-forward run with parallel optimization #1 pseudo-code.

VI. PARALLEL OPTIMIZATION

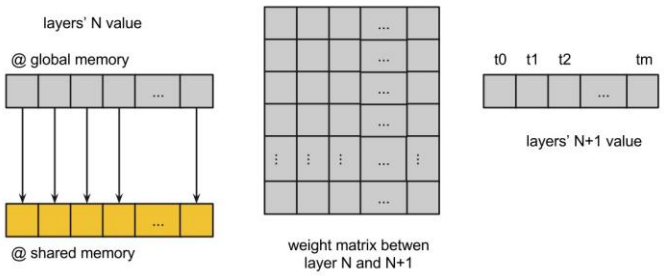
There are four attempts of parallel optimizations developed in this project. All of them will be described in detail in the following subsections.

A. Use shared memory to store previous layer's values

The first optimization is based on the fact that the input values from the previous layer is used by all the threads. So, to reduce the global memory traffic due to accessing input values, we could modify the kernel code to collaboratively load all input values into shared memory and all the threads can just access the shared memory to get the input data instead of going to the global memory to get it. Figure 8 shows the way feed-forward neural network is optimized by using shared memory to store all the input values. The pseudo-code is shown in Figure 9.

B. Finer-grained parallel optimization

The second optimization is based on the idea of reducing the amount of work of each thread in computing the output value of one neuron. Since all the thread will need to do addition as



many as the number of neurons on the previous layer, we could split the work so that those could be done by multiple threads. Later when all the addition parts are done, some those threads can execute a reduction add operation which is likely to be more efficient. Figure 10 depicts this operation.

To do this, multiple threads are now assigned to the same output neuron. For easy implementation, the thread block is organized as two dimensional block where the x value refers to

the output neurons and the y value refers to the part of addition need to be done. When the size of y dimension is one, it is essentially the same as the first optimization.

After all addition are done, the partial sum is stored into sum_s array which is located in shared memory. All the threads with the same x value will then collaboratively do

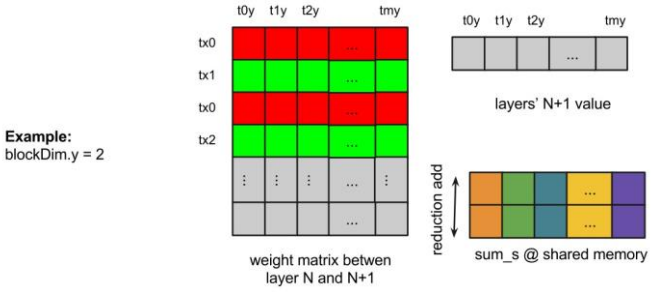


Fig. 10. Illustration of parallel optimization #2.

```
For each layer transfer
  Invoke GPU kernel single step
    collectively read hidden_prev data into shared memory
    clear sum_s
    __synchthreads()
    for each part in column:
      compute partial_sum
    sum_s[idx] = partial_sum
    __syncthreads();
    reduction add operation on sum_s[]
    __syncthreads();
    neuron[x] = f(sum_s[x])
```

Fig. 11. Feed-forward run with parallel optimization #2 pseudo-code.

the reduction add operations. After that, one thread with the same x will apply the activation function to it and store the result into the corresponding address in the global memory. The pseudo-code of this optimization is shown in Figure 11.

The performance after implementing this optimization, however, is not improved greatly and this issue will be discussed in more detail later in Section VIII-B.

C. Multiple layers execution in one GPU kernel call

In the first two optimization, the focus of performance improvement is only within one layer to another layer computation. This single part of operation is executed using one kernel call and we need to invoke another kernel call to operate the next layer computation. This operation is depicted in Figure 12.

The idea of the third optimization is to see if we could avoid calling multiple kernel call to run all the computation in feedforward run. Since the layer sizes could be similar, we might not need to exit the kernel execution from GPU to CPU and get back from CPU to GPU again. Instead, we might be able to run all the computation in one run as shown in Figure 13.

The pseudo-code of this optimization is shown briefly in Figure 7. There is a concern about scalability of this optimization because of the requirement of having global barrier which could not be achieved when there are multiple thread block allocated in a kernel call. This concern will be discussed in more detail in the later section.

D. Multiple layers execution in one GPU kernel call and store hidden values in shared memory

Another optimization idea, which is based on optimization 3, is based on a fact that for normal feed-forward run, the values

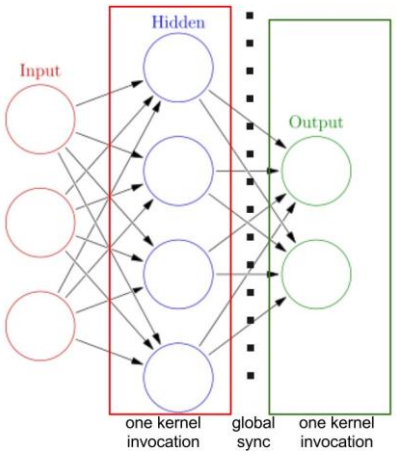


Fig. 12. Single kernel call executes computation from one layer to another layer.

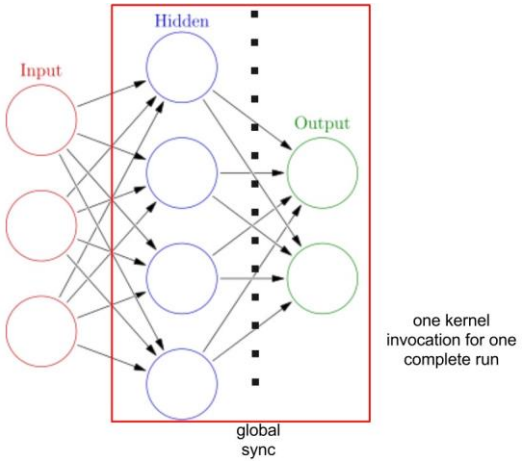


Fig. 13. Single kernel call execute all the computations in a single feedforward run.

```
Invoke one GPU kernel
  For each layer transfer
    __synchthreads()
    collectively read hidden_prev data into shared memory
    clear sum
    __synchthreads()
    for each element in a column:
      compute sum
    neuron[x] = f(sum)
```

Fig. 14. Feed-forward run with parallel optimization #3 pseudo-code.

of hidden layers are not needed by external system. The hidden layers values are needed for train run, but not for normal operation run. So, if the hidden layers values are not needed, we can improve the performance further by avoiding accessing global memory access to store hidden layer's values and store it in shared memory instead. This technique is illustrated in Figure 15. This technique improves the performance by further reduce the traffic of accessing global memory. The pseudocode is shown in Figure 16.

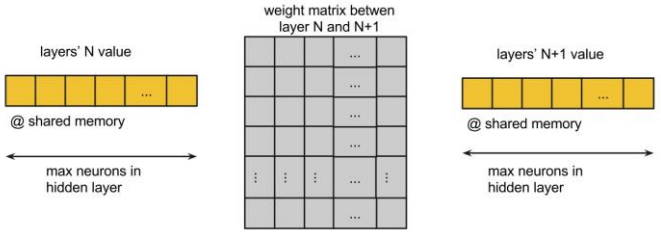


Fig. 15. Illustration of parallel optimization #4.

```
Invoke one GPU kernel
  For each layer transfer
    __synchthreads()
    update pointer to previous layer (input layer/shared memory)
    update pointer to current layer (shared memory/output layer)
    clear sum
    __synchthreads()
    for each element in a column:
      compute sum
    pointer_to_current_layer[x] = f(sum)
```

Fig. 16. Feed-forward run with parallel optimization #4 pseudo-code.

VII. Experimental Methodology

We use GPU Education Machine (GEM) to run all of our experiment. To compare performance, we run the several neural network configurations with varying number hidden layers as well as hidden layer size. The list of neural network configurations we use is shown in Table I. Essentially, the different configurations is intended to see the performance of various optimization on different setting as well as to see how scalable the GPU optimization is. The configurations have different maximum size of neural network, ranging from 500 neurons in a layer up until 4,000 neurons in a layer. The number of hidden layers of each configurations is fixed to 10 layers, except for the last configuration. Note that the memory footprint is quadratically proportional to the width of layers. So, due to large memory usage for very wide layers, we limit the number of hidden layers to only 5 layers for Config-6.

For experiment, we compare the performance of various CPU versions – the original FANN implementation and another version with modified data structure – and GPU versions – naive and four optimized implementations – while running feed-forward execution. The following describe the kernel configurations used in GPU naive and optimized versions.

GPU naive implementation is configured to execute 512 threads in one block and the number of thread blocks are depending on the number of neurons in a layer so that all the output neurons are computed by one thread. The dimension of a thread block is configured to only have one dimension.

GPU optimized version 1 is configured similar to naive version where 512 threads in one block allocated and the number of thread blocks also depends on the number of neurons in a layer so that one thread executes for one output neuron.

GPU optimized version 2 requires multiple threads to execute one output neuron. A thread block is configured as two-dimensional block where threads with the same index in *x* dimension will collaboratively execute the same output. Thread index in *y* dimension is used to decide which row of the weight matrix the thread should read from and to decide which element of sum should be accessed for reduction operation. The number of thread blocks allocated is decided similar to the previous two GPU versions. Unless otherwise stated, the thread block is configured to have 8 threads in *x* dimension and 128 threads in *y* dimension, a total of 1024 threads in a block.

TABLE I. Neural Network Configurations

Config	# Input Neurons	# Hidden Layers	# Hidden Neurons	# Output Neurons
Config-1	200	10	400, 500, 500, 500, 500, 500, 500, 500, 500, 400	100
Config-2	200	10	500, 750, 750, 750, 750, 750, 750, 750, 750, 500	100
Config-3	200	10	500, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 500	100
Config-4	200	10	1000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 1000	100
Config-5	200	10	1000, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 1000	100
Config-6	200	5	1000, 4000, 4000, 4000, 1000	100

GPU optimized version 3 and 4 are configured to use a fixed number of threads and use only one thread block to allow global barrier achieved by only calling __synchthread() function. In our experiment, we set the number of threads to be 1024, the maximum number of threads in a block supported by the GPU hardware in GEM.

VIII. Results and Evaluation

A. Comparison of CPU and GPU versions

Figure 17 shows the total execution time comparison between original FANN CPU version, modified CPU version, and various GPU versions. The speedup comparison for those versions is shown in Figure 18, normalized to original FANN CPU version performance. We can see that all GPU versions consistently perform better than CPU versions. It is not

surprising because neural network computation for one layer to another layer is inherently parallel as there is no data dependency between neurons in the same layer. The more number of neurons, the higher opportunity of making the computation in parallel.

The CPU custom version, that is the CPU implementation with a modified data structure, is 2-8 times slower than the original FANN implementation. This is likely because of the way data is accessed in FANN implementation is much different compared to CPU custom, in particular FANN implementation increment pointer to data and dereference the pointer to get the data whereas in CPU custom the data is accessed using a base pointer and an offset which is calculated using multiple arithmetic operations such as additions and

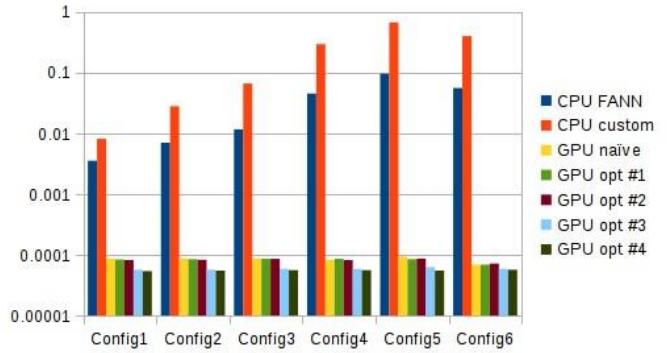


Fig. 17. Execution time (in second) comparison between CPU and GPU versions on various neural network configurations.

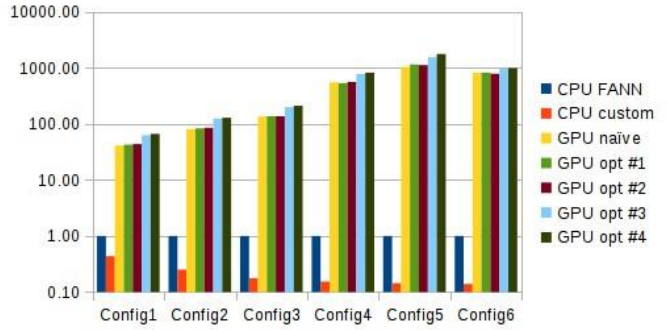


Fig. 18. Speed-up comparison between CPU and GPU versions on various neural network configurations.

multiplications. This implementation is intended to make it easier to be implemented in parallel in GPU.

B. Comparison of various GPU versions

Figure 19 shows the performance comparison of various GPU optimizations, normalized to the GPU naive version. The performance of GPU optimized version 1 and 2 are slightly better than the GPU naive version. On average, the GPU optimized version 1 and 2 is about 2.5% and 3% faster than the naive counterpart, respectively.

Analyzing the performance improvement of GPU optimization version 2, it turns out that the optimization is not giving much performance improvement compared to optimization 1. The more detailed study on optimization version 2 is summarized in Table II. In the table, we compare the performance of varying number of threads collaboratively executing the same neuron, that is from 256 threads per neuron until only 1 threads per neuron (i.e. without finer-grained parallel optimization). Based on the data collected, it shows that the performance of doing finer-grained parallel optimization is roughly similar to the performance of optimization version 1. Neural network computation requires a huge memory footprint to store weight matrix whose dimension is the product of the number of neurons of the consecutive layers. For hidden layer that has 4,000 neurons each, there will be 16 million connections and each connection has an associated weight value. Unfortunately,

TABLE II. FINER-GRAINED PARALLEL OPTIMIZATION EXECUTION TIME (IN MICROSECONDS) FOR VARIOUS NUMBER OF THREADS EXECUTING THE SAME OUTPUT NEURONS

Config	256	128	64	32	16	8	4	2	1
Config-1	82	84	81	86	85	86	82	82	82
Config-2	82	88	83	85	82	83	86	86	86
Config-3	85	85	83	83	82	82	83	84	82
Config-4	87	85	81	86	83	85	84	81	83
Config-5	84	84	82	85	87	86	83	84	83
Config-6	69	73	76	72	70	77	70	68	78

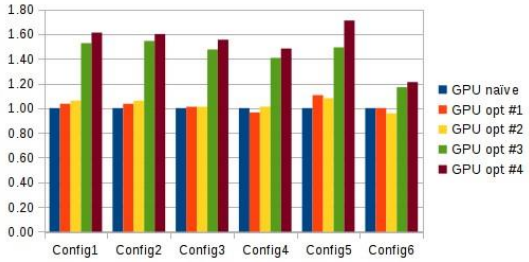


Fig. 19. Speed-up comparison between GPU versions on various neural network configurations.

even though there are more threads executing on the same output, there is no data reuse in weight matrix and thus the memory bandwidth usage is not reduced. This leads to the performance limited by the memory bandwidth. This is likely the reason why increasing the number of threads to work on the same output doesn't help much to improve overall performance. To relax this bottleneck, an improvement on global memory bandwidth usage is prefered.

More performance boost is achieved in optimization 3 and 4, where the speed up reaches 43.6% and 52.9% faster on average, respectively, compared to GPU naive version. It shows

that reducing the kernel invocation overhead helps improving the performance which are limited to global memory bandwidth as discussed earlier. In optimization 4, it reduce the global memory bandwidth further by removing the traffic of loading and storing of hidden layer's values and use shared memory instead which has larger bandwidth and better latency.

Scalability evaluation. In optimization 3 and 4, the number of threads that can execute are limited to the maximum number of threads in a thread block supported by the hardware, in Tesla C2050, it is 1024 threads. This might limit the scalability of those two optimizations. However, since the performance of is limited to the global memory bandwidth due to lack of global memory reuse – especially while accessing the weight matrix, the largest data structure in neural network – the performance of optimization 3 and 4 for the large neural network still better than any other versions, indicating that there's no scalability issue for those optimizations. This is also shown in Figure 19 where optimization 3 and 4 are constantly better than the others in all neural network size studied in this project.

C. Comparison of GPU versions with other existing library

As mentioned in the previous section, there are many other libraries which implement an optimized GPU version of neural network using CUDA. The NeuralNetwork CUDA library [3] provides a simple implementation of neural network for number image recognition. This library uses 3 hidden layers and the first two layers implement a convolutional neural network whereas the last hidden layer and output layer are fully-connected. They do provide source code and according to their code, we observe that their code uses the naive parallel implementation. Trying to change its implementation to compare against us would lead to similar implementation to our naive parallel implementation. Beside that, their implementation is not pure fully-connected neural network. So, we don't compare our performance with their.

Another library, called cuDNN [5], is provided by NVIDIA and they claimed that they use advanced GEMM technique. They don't provide the source code and based on its documentation, it requires GPU with compute capability version 3.0 or higher. So, we don't compare our performance against them.

Caffe [6] is another library which offers neural network optimization using CUDA. This library provide a wide range of customization of neural network and depends on some other libraries. Because of these dependencies, we were having difficulties on using this library in the environment we have and thus we were not able to compare our performance with theirs.

IX. CONCLUSION

In this report, we show that we are successfully optimize the feed-forward computation performance by allowing multilayer execution optimization by allowing only one kernel call to execute multiple layer computation to reduce kernel invocation overhead, improving the overall performance of neural network in GPU. We also show that, based on scalability comparison, even for a large size neural network application which requires huge memory resources, our optimization still works better than optimization within a layer due to memory bandwidth bottleneck.

Based on our experiment, we achieve GPU performance three order of magnitudes speed up over the original FANN CPU version. Compared to GPU naive version, the performance of our optimization achieves up to more than 1.6x faster.

REFERENCES

[1] Kevin L. Priddy and Paul E. Keller, *Artificial neural networks: an introduction*, 3rd ed. Bellingham, Wash. : SPIE, c2005.

[2] http://en.wikipedia.org/wiki/Artificial neural network

[3] http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU

[4] http://leenissen.dk/fann/wp/

[5] https://developer.nvidia.com/cuDNN [6] http://caffe.berkeleyvision.org/