Scalability Analysis of Synchronous Data-Parallel Artificial Neural Network (ANN) Learners

Chang Sun

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Paul E. Plassmann, Chair
Cameron D. Patterson
Mark T. Jones

August 9, 2018
Blacksburg, VA

Keywords: artificial neural networks, machine learning, heterogenous computing, parallel computing, object-oriented programming

Scalability Analysis of Synchronous Data-Parallel Artificial Neural Network (ANN) Learners

Chang Sun

Abstract

(Academic)

Machine learning and artificial intelligence have been some of the most prominent recent topics in the realm of computer science and engineering. After several decades of back and forth in learning model developments, recent advances in deep learning has led to a resurgence of interest within the machine learning community of artificial neural networks (ANNs) models.

Despite the vast enhancements in learning models, current research interest is mostly on improving training accuracies, whereas a fine-grained timing analysis model for the training procedure has been less addressed. Mainstream deep learning frameworks typically have large, complex codes bases; thus, it is difficult to profile the running of these codes in order to generate a fine-grained execution time model.

In this thesis, I present a feedforward ANN model, and implement this model using open-standard packages. A key contribution of this work is to exploit the inherent data parallelism in this model, and develop a software implementation that can take advantage of a heterogeneous computing environment. This computing environment may consist of both Graphics Processing Units (GPUs) and distributed multiprocessors communicating via message-passing. The ANN model implementation is benchmarked on state-of-the-art compute clusters and performance results a detailed timing analysis is presented in this thesis.

The main two contributions of this thesis are: (1) to create an open-source software framework for portable machine learning on heterogeneous computing clusters; and (2) to analyze and generate timing and throughput models for the ANN learner in order to understand

and predict the scaling of these models as a function of problem parameters. This scaling analysis can be used to predict the performance of the software implementation on future heterogenous computing platforms.

Scalability Analysis of Synchronous Data-Parallel Artificial Neural Network (ANN) Learners

Chang Sun

Abstract

(General Audience)

Artificial Neural Networks (ANNs) have been established as one of the most important algorithmic tools in the Machine Learning (ML) toolbox over the past few decades. ANNs' recent rise to widespread acceptance can be attributed to two developments: (1) the availability of large-scale training and testing datasets; and (2) the availability of new computer architectures for which ANN implementations are orders of magnitude more efficient. In this thesis, I present research on two aspects of the second development. First, I present a portable, open source implementation of ANNs in OpenCL and MPI. Second, I present performance and scaling models for ANN algorithms on state-of-the-art Graphics Processing Unit (GPU) based parallel compute clusters.

Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor Dr. Paul Plassmann for the constant support of my master's study and related research, for his patience, flexibility, motivation and immense knowledge.

Besides my advisor, I would also like to thank the rest of my committee members, Dr. Cameron Patterson and Dr. Mark Jones, for their insightful comments and discussions that broadened my horizons and helped me overcome obstacles.

In addition, I am grateful to the ECE department at Virginia Tech for funding my two years of study and research. It is an absolutely fantastic and meaningful two-year experience serving as a Graduate Teaching Assistant for the Embedded System Design course. Not only has it strengthened my technical knowledge and problem-solving skills, but it also makes me a better person in planning logistics, managing resources and leading junior developers. In particular, I want to thank Bob Lineberry for his mentorship and his wisdom on education and lab management.

Last but not least, I would like to thank my family and friends who supported me throughout the time, who gave me the courage to keep marching forward when I was feeling down.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

1.1 Neural networks and deep learning[1]

Deep learning, and in particular deep neural networks (DNN), have become the hottest topic in the field of contemporary machine learning. Contrary to most who would think of deep learning as a brand-new technology, the original ideas of neural networks were introduced in the late 1950s, when Frank Rosenblatt introduced the idea of the perceptron for solving pattern recognition problems [1]. Progress from this early work stagnated because the original perceptron model is limited to a single layer and thus incapable of classifying a variety of simple patterns, such as the XOR function [2]. Multilayer perceptron (MLP) models overcame this limitation, and later proved to be capable of computing any function [3, 4]. Historically, the generalization of MLPs to neural network models have experienced multiple periods of stagnation in research progress. However, since the back-propagation technique was widely adopted through the survey publication by Geoffrey Hinton at el., training a neural network has been widely accepted an iterative procedure of computing and applying partial derivatives for some differentiable function as part of a stochastic gradient descent algorithm [5].

Figure 1-1 illustrates a neural network in the widely accepted, modern sense. It consists of the input layer, the output layer, and two layers in between, namely *hidden layers*. Each node in the input layer typically connects to a portion of a data example such as a pixel in an image, and each node in the output layer usually presents a value indicating the confidence of the network's prediction against a particular input vector (the combined set of inputs) associating with a category or a classification label. The keyword "deep" in deep learning essentially means more than one hidden layer in the neural network.

---

[1] Heading title credits to Michael Nielsen's book *Neural Networks and Deep Learning*.

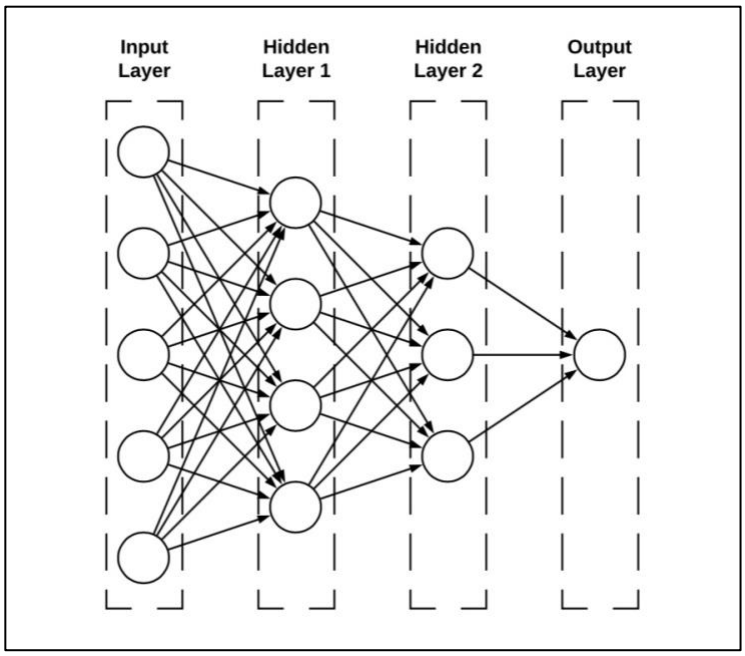


Figure 1-1. A neural network in the modern sense

The illustrated neural network in Figure 1-1 is also an example of a *dense feedforward neural network*. A dense feedforward neural network is the most fundamental form of MLP where the adjacent layers are fully connected, meaning every node is connected to all nodes in the previous layer and the next layer, and the output from one layer is used as the input to the next layer. That is, there is no "backwards" loop in a feedforward network [6]. In this model, each neuron takes the outputs of all neurons in the preceeding layer as inputs, performs a linear transformation over these inputs, and then applies a non-linear squashing function to compute its output. In mathematical terms, the output of any particular neuron is defined as

$$y = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

where $\boldsymbol{x}$ is the input vector denoting all outputs of neurons from the previous layer, $\boldsymbol{w}$ is the weight vector denoting the *weights* from every previous layer node to the current node, b is the *bias*, and finally f is a non-linear squashing function.

In this work presented in this thesis, the *mini-batch stochastic gradient descent* algorithm is used to train the neural network model. In the stochastic gradient decent algorithm, training of

the model is performed over a number of *epochs*, where an epoch consists of visiting the entire training dataset by considering it as a sequence of mini-batch subsets. In this manner, an epoch updates the neural network with the entire training dataset. The reason mini-batches are used is because for larger datasets and larger network models, looping through the entire training dataset in one trace is not practical, because it is computationally intractable, and because it may lead to overfitting of the model. Hence a common strategy is to use *mini-batches*, which means only a subset of the entire training dataset is trained for each *iteration*. The number of iterations in each training epoch is therefore given by the following formula.

$$\text{number of iterations} = \frac{\text{number of training examples in dataset}}{\text{number of examples per mini-batch}}$$

In each iteration, the input examples from a randomly shuffled mini-batch are first fed to the input layer. The values are then propagated through the network in the *forward propagation* procedure. After all values have been computed for the (final) output layer, these predicted output values are compared with the ground truth (known outputs from the training dataset), following which the partial derivatives (also known as *gradients*) are computed for the parameters (i.e., weights and biases) in each layer from the output layer back to the input layer. This procedure of computing partial derivatives in the reverse order of layers is called *back propagation*. Ultimately, the partial derivatives are applied to the parameters and thus the update for a training iteration is complete.

1.2 Opportunities for parallelism

In the past decade, artificial neural network (ANN) learners continue have made significant progress and have become the dominant model for machine learning. This progress has been a cumulative result of the inherent biological relevance and the computational

feasibility of these models due to the vast improvements in available computing power, especially through the exploitation of data parallelism via heterogeneous computer architectures.

The wide adoption of the graphics processing unit (GPU) by itself is arguably the most dominant factor in the resurgence of machine learning. The majority of the work for training a neural network can be broken down into fundamentally parallelizable parts. In particular, the backpropagation training algorithm can be efficiently expressed as a sequence of matrix computations. As an computer architecture that utilizes a single instruction, multiple data (SIMD) paradigm, GPUs can effectively exploit the data parallelism inherent in computation-intensive jobs such as matrix multiplication [7].

Recent research in hardware acceleration for machine learning demonstrates considerable interests in architectures that are power efficient, customizable, and highly specialized in task-specific computations such as field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). As the early adopters, Xilinx and Google already offer services using these alternative architectures, namely the Xilinx ML Suite of the former and Google Cloud TPU of the latter [8, 9]. However, GPUs are still currently the top choice for large-scale machine learning because they are more cost-beneficial in most scenarios, and because of the wide availability of GPUs—they are included in every PC available today [10].

There is additional parallelism is available on today's CPUs as well. Modern hardware provides support for Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX), and these extensions can be used to speed up the learning via vector instructions [11, 12]. For the results presented in this thesis, a heterogeneous system with both CPUs and GPUs is employed in the experimental results.

Prior research on the performance analysis of machine learning algorithms has mainly focused on the performance of software on problems of fixed size. However, in this thesis the goal is not to claim any particular speed-up resulting from code optimization on a specific hardware platform. Rather, the parallel computing scaling inherent in deep learning run on the above-mentioned SIMD architectures is analyzed, and the speed-up capabilities (as a function of problem size) that may be achieved by future hardware advances are predicted.

1.3 Focus of this thesis

The subject of this thesis is the implementation and analysis parallel algorithms for training neural networks. More specifically, the focus is on the impacts of varying hardware specifications on the execution time of training the most canonical type of neural network—fully connected feedforward neural network—using the stochastic gradient descent algorithm.

Questions that we would hope to answer by the work presented herein include: "How much speed-up would be gained by stacking more processors onto a single machine?" or "How would varying the structure of neural network or varying other parameters in learning affect the total runtime?"

1.3.1 Dense feedforward neural network as the learning model

In order to simplify the program for more explicit timing analysis, a dense feedforward neural network model (rather than a sparse interlayer connection structure) is implemented. Although state-of-the-art deep learning models commonly include additional structures such as convolutional layers (with pooling), dropouts and recurrent neural networks, fully-connected layers are still routinely utilized as the final layers (e.g. the decision layer) in the network.

During the time the research in this thesis was conducted, Geoffrey Hinton et al. introduced the Capsule Neural Network (CapsNet)—an ANN that utilizes a genuinely new structure called capsule. With the help of these new structures, that more closely mimic biological neural organization, Hinton's team not only managed to both reduce the error rates and the number of training examples on the MNIST dataset, but they also achieved significantly better results on highly overlapped digits. A crucial point that Hinton et al. made was discouraging pooling layers for creating translational invariance in image recognition [13]. The details of CapsNet implementation are beyond the scope of this thesis. However, the debate over the validity and lack of proof of these new layer structures narrows down the selection to fundamental network structures only.

1.3.2 Optimization using stochastic gradient descent

The crux of deep learning is optimizing the (often) non-convex cost function between the predicted output from training and the ground truth. By the time of this paper, stochastic gradient descent (SGD) is one of the canonical approaches in the machine learning community to train deep neural networks. The major advantages of SGD are that it is a simple and efficient numerical operation, and that it scales well even with large network models and datasets [14].

In his efforts to create a universal and reader-friendly guide of neural networks and deep learning for the general programming community, Michael Nielson has provided a simple and straight-forward Python implementation for mini-batch stochastic gradient decent [6]. Using Nielson's implementation as a starting point, one can investigate the computational cost of scaling dense neural network learning models.

1.3.3 Synchronous data-parallel approach for distributed learning

In addition to utilizing hardware on a single computer, recent advances in high-performance computing (HPC) have shown the advantages of using *compute clusters*, networked computers consisting of multiple compute nodes, to simultaneously train the deep learning model. This parallelized approach is called *distributed learning*. By convention, each node in the cluster is called a *worker*. In the work presented in this thesis, the software implementation is *synchronous data-parallel* in order to maintain a data flow that is equivalent to the basic sequential version.

In the distributed learning setting, there are three major potential sources of parallelism in deep learning: task parallelism, data parallelism, and model parallelism. When training a neural network, task parallelism usually dispatches the different tasks/operations such as matrix multiplication, squashing, and derivative computation onto different workers (e.g., individual computer cluster nodes) in order to achieve a parallelization speed-up. This approach keeps the parallelized implementation functionally equivalent as the sequential implementation. An example of a task-parallel implementation is the in-graph replication approach used in distributed Tensorflow [15]. However, because the operations differ in the amounts of workloads, and typically have inter-task dependencies, task-parallel implementations may suffer from high synchronization costs and, as a result, may not scale well.

Model parallelism typically updates only a subset of all parameters (i.e., the weights and biases in the neural network) on a single worker using all data using some particular algorithm that ensures correctness [16]. In essence, model parallelism aims at splitting the parameters, whereas data parallelism aims at splitting the training examples. As illustrated in Figure 1-2, the data-parallel implementation simply splits the mini-batch examples among all workers, each

worker executing the forward and backward process, and then the global gradient sum (using the results from all the works) is used to update the parameters. This approach is known as a *synchronous* approach because all workers have the same parameters at the beginning of each mini-batch iteration, and the same parameter updates at the end of the iteration.

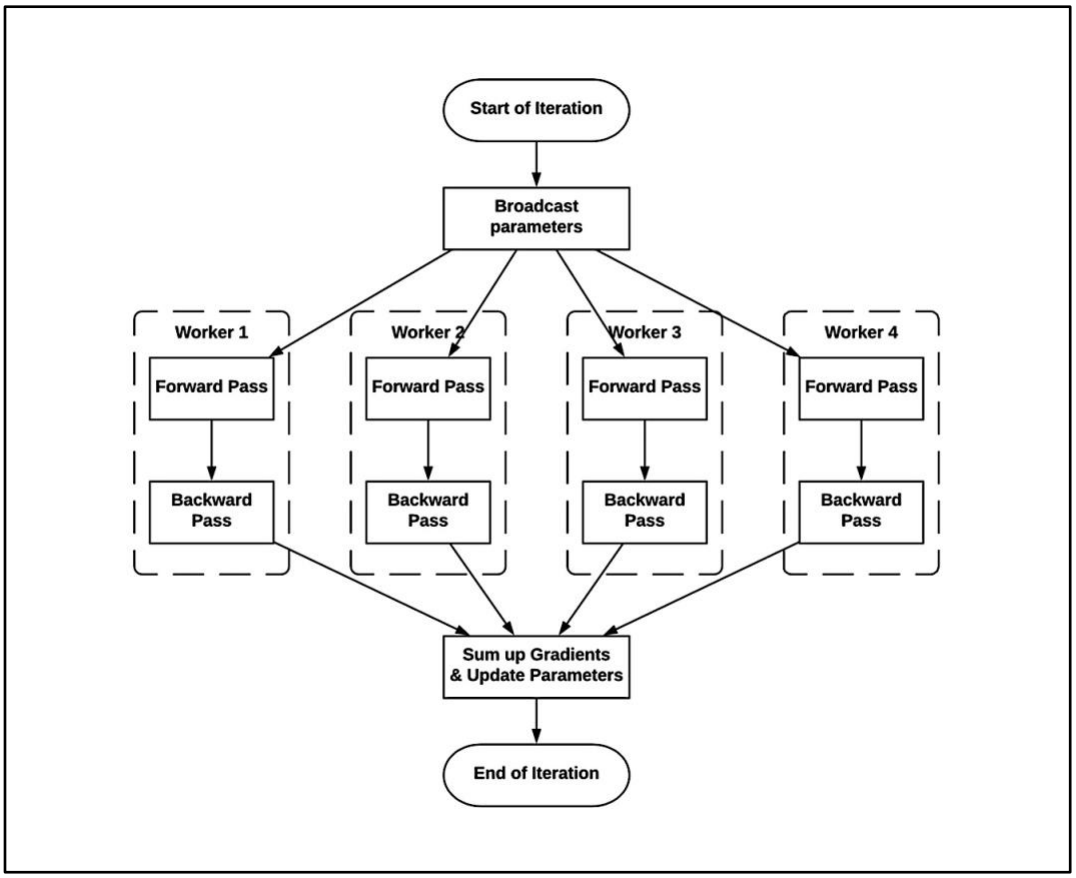


Figure 1-2. Synchronous data-parallel distributed learner

The synchronous data-parallel approach preserves the same data flow as its sequential counterpart. In contrast, for an *asynchronous* data-parallel implementation, the training loops of each worker is independent and no coordination is involved [15].

1.3.4 Universality

An additional research objective for this thesis is to implement a non-platform specific, universal, and easy-to-understand model, so that the implemented framework can be of educational value to the general programming community. State-of-the-art machine learning

frameworks such as TensorFlow, Caffe, and Torch already have good support on a variety of platforms. However, because the code bases for these libraries are large and complex, it is a complicated task to formulate fine-grained timing analysis. In addition, these main-stream frameworks are frequently updated and implementation details are constantly changing.

In the software for distributed learning developed for this thesis, only standard packages are used—data structures from the C++ standard library, OpenCL (which is non-proprietary and cross-platform), and the Message Passing Interface (MPI). A good sandbox game provides just enough of a foundation for players to express their imagination. Simplicity and standardization help the code base be accepted by the majority. We follow the following maxim said about software implementations: "Simple, correct, fast: in that order" [17].

1.4 Organization of this thesis

In this chapter, I have reviewed the background of neural networks and deep learning. I have introduced the goal of this thesis, that is to analyze the scaling of the data-parallel training algorithm for ANN learners. Chapter 2 of this thesis covers the parallel implementation of the ANN learner model, starting from the base case of a sequential implementation. This parallel implementation exploits both fine-grained data-parallelism (using OpenCL) and coarse-grained data-parallelism (using MPI). Chapter 3 of this thesis details the Application Programming Interfaces (APIs) for using the implemented software frameworks. Chapter 4 presents experimental results, including the experimental setup and the benchmark results. Chapter 5 contains a discussion on the benchmark results, formulating a detailed training throughput model under different hardware and problem configurations. Finally, Chapter 6 summarizes the thesis contributions and provides concluding remarks.

Chapter 2 – Implementation of the Neural Network Model

2.1 High-level design choices

2.1.1 C++ as the programming language for dense neural network implementation

In this thesis C++ is selected as the programming language for the implementation of all neural network components. The most prominent reason for this choice is that C++ a mature languge with stable (and usually faster) performance (when compared to other languages), along with explicit memory control. These factors are crucial to the timing analysis. For example, languages using a virtual machine (e.g., Java or Python) may utilize runtime optimization and garbage collection, which would cause considerable variations in measured execution times. Note that for the timing analysis, the neural network implementation is compiled using G++ with the *-O3* flag, which enables the optimizations *-ftree-loop-vectorize* and *-ftree-slp-vectorize*. Among all the enhancements enabled by the high level of compiler optimization, these vectorization flags are especially important for SIMD architectures, since the code loops are then vectorized, and will use AVX YMM vector registers [18, 19].

The software implementation additionally employs the OpenCL standard to exploit data parallelism in neural network training. The OpenCL 1.2 standard utilizes a subset of ISO C99 with extensions for parallelism. Programming in C++ for the project provides close to native compatibility to the official OpenCL APIs [20]. In addition, when using ISO C++ standards, C++ is one of most compatible and universally portable programming languages for modern computing platform. As long as the target architecture (e.g., x86, ARM, MIPS, etc.) has compiler support, the software implementation presented in this thesis may be seamlessly ported.

Last but not least, some additional data structures in the C++11 standard template library (STL) are used. The C11 standard also provides extensive timing utilities defined in *time.h*, upon which a software timer is implemented capable of nanosecond-resolution timing accuracy.

2.1.2 Python as the programming language for timing analysis

Python is selected as the programming language for timing analysis and timing model generation, mainly because of the fast prototyping capabilities and ease of use for mathematical computations. In addition, the visualization framework provided by the Matplotlib provides a very intuitive, MATLAB-alike, yet powerful programming interface [21]. The visualization framework becomes particularly handy in the timing analysis, including the need for plotting the projected speed-up as a function of different parameters.

2.1.3 OpenCL 1.2 for interfacing compute devices

OpenCL is chosen over CUDA because it is an open standard and the computing platform is not limited to the GPU vendor (i.e., CUDA only works on NIVIDIA GPUs). At the time this thesis was written, the latest stable version of OpenCL is 2.2 [22]. Version 1.2 is used because it is the most compatible version across all supported platforms, and because the implementation does not require the advanced features made available in later versions.

2.1.4 MPI for cross-node data passing

The Message Passing Interface (MPI) for is utilized for communication between nodes in a cluster setting for distributed learning. This choice was made because MPI is the standard message-passing scheme available on the clusters from Virginia Tech Advanced Research Computing (ARC). MPI remains the dominant message-passing standard used in high-performance computing today [23].

2.2 Key components for sequential full-matrix-batched training

2.2.1 Principle storage components

At a minimum, the following three major storage components are necessary in order to store the dense neural network model:

1. The sizes of each layer, i.e. the number of neurons in each layer. In Nielson's Python implementation, the sizes are stored in a 1-D *NumPy* array with N integers (N denoting the total number of layers) [6]; whereas in the C++ implementation, the sizes are stored in an *int* array, namely *sizes[]*. To avoid ambiguity, let layer 0 denote the input layer, let layer 1 denote the hidden layer closest to the input layer, and so on.
2. The biases of all neurons in all layers. In Nielson's implementation, the biases are stored in a list of $N-1$ 1-D *NumPy* arrays, where each array represents the biases of the neurons in that particular layer [6]. Note that the input layer does not have biases, so the first array of the list represents the biases of the first hidden layer. In the C++ implementation, the biases are stored in an array of *float* (or *double*) arrays, namely *biases[]*.
3. The weights connecting nodes in adjacent layers. In Nielson's Python implementation, the weights are stored in a list of 2-D *NumPy* arrays, where each array represents the weights of neurons from the previous layer to the current layer. In particular, the first array in the list represents the weights of neurons from the input layer to the first hidden layer [6]. In the C++ implementation, the weights are stored in an array of *float* (or *double*) arrays, namely *weights[]*. Note that the *float* (or *double*) arrays in the C++ implementation are 1-D arrays, meaning that the weight

from neuron *i* in layer *k* to neuron *j* in the current layer is addressed as *weights[i * sizes[k + 1] + j]*.

By convention, the values of biases and weights are initialized as a normal distribution before the stochastic gradient descent training process.

2.2.2 Optimize reusable storage components

Apart from the network structure and the parameters, the dataset, the gradients, the activations and sigmoid primes of all neurons are stored. This is mainly due to the frequent reuse in the training process. Storing these variables as consistent storage such as member variables in the class eliminates the need of repeating memory allocation and de-allocation.

Table 2-1 lists the definitions and size dimensions of these additional storage components.

Variable Name	Definition	Size
train_data_x	training data inputs	*sizes[0] * train_data_size*
train_data_y	training data outputs	*sizes[num_layers - 1] * train_data_size*
test_data_x	test data inputs	*sizes[0] * test_data_size*
test_data_y	test data outputs	1 * *test_data_size*
activations	activation of each neuron	consists of *num_layers* arrays, size of *activations[i] = sizes[i] * mini_batch_size*
primes	squashing function derivative of each neuron	same as *activations*, except for the input layer
nabla_w	partial derivatives of weights	same as *weights*
nabla_b	partial derivatives of biases	same as *biases*

Table 2-1. Function signature of sigmoid

The dataset is split into the training set and the test set. For the MNIST dataset, this means 50,000 image examples for the training set and 10,000 image examples for the test set. Note that the outputs of the training set are stored as “1-hot”, meaning a single 1 for the true label and 0 for every else, whereas the test set outputs are stored as the classification labels.

It is also worth noting that in a mini-batch training, the activations for each layer is stacked with *mini_batch_size* number of rows, with each row representing the values for a particular training example entry. This applies to the squashing function derivatives *primes* as well.

2.2.3 Compute kernels

The compute intensive and data parallelizable parts of the stochastic gradient decent (SGD) algorithm are summarized into 8 functions, noted as *compute kernels*.

sigmoid

Figure 2-1 presents the function signature of the *sigmoid* compute kernel.

```
void sigmoid(data_t* ret_val, int size, OclWorkEngine* engine = NULL)
```

Figure 2-1. Function signature of sigmoid

The sigmoid function is used as the squashing function[2]. The *sigmoid* compute kernel takes in the activations and performs the sigmoid squashing operation

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

Note that the engine argument is used for parallelization using GPUs with OpenCL, same applies to all following compute kernels.

The *sigmoid* compute kernel has *O(size)* time complexity.

sigmoid_prime

Figure 2-2 presents the function signature of the *sigmoid_prime* compute kernel.

```
void sigmoid_prime(data_t* ret_val, const data_t* z, int size,
                   OclWorkEngine* engine = NULL)
```

[2] In modern machine learning, the rectified linear unit (ReLU) is more preferable in order to avoid the vanishing gradient problem. In this project, however, the focus is on the execution time performance. Because ReLU and sigmoid both have *O(n)* time complexity, this would not affect the timing analyses.

Figure 2-2. Function signature of sigmoid_prime

The *sigmoid_prime* compute kernel takes in the activations and computes the derivatives of the sigmoid squashing operation

$$S'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = S(x) \cdot \big(1 - S(x)\big)$$

The *sigmoid* compute kernel has *O(size)* time complexity.

vec_mult

Figure 2-3 presents the function signature of the *vec_mult* compute kernel.

```
void vec_mult(data_t* ret_vec, const data_t* vec, int size,
              OclWorkEngine* engine = NULL)
```

Figure 2-3. Function signature of vec_mult

The *vec_mult* compute kernel takes in two vectors *ret_vec* and *vec*, performs an element-wise multiplication, and saves the products back to *ret_vec*.

The *vec_mult* compute kernel has *O(size)* time complexity.

mat_vec_add

Figure 2-4 presents the function signature of the *mat_vec_add* compute kernel.

```
void mat_vec_add(data_t* ret_mat, const data_t* vec, int row, int col,
                 OclWorkEngine* engine = NULL)
```

Figure 2-4. Function signature of mat_vec_add

The *mat_vec_add* compute kernel takes in a *row* by *col* matrix *ret_mat* and a vector *vec*, performs a row-major, element-wise addition, and saves the results to *ret_mat*. In other words, each row of *ret_mat* is added element-wise by *vec*. The size of *vec* must be equal to *col*.

The *mat_vec_add* compute kernel has *O(row * col)* time complexity.

mat_trans

Figure 2-5 presents the function signature of the *mat_trans* compute kernel.

```
void mat_trans(data_t* ret_mat, const data_t* src_mat, int row, int col,
               OclWorkEngine* engine = NULL)
```

Figure 2-5. Function signature of mat_trans

The *mat_trans* compute kernel transposes a *row* by *col* matrix *src_mat* into a *col* by *row* matrix *ret_mat*. In mathematical terms,

$$ret\_mat = src\_mat^T$$

The *mat_trans* compute kernel has *O(row * col)* time complexity.

mat_mult

Figure 2-6 presents the function signature of the *mat_mult* compute kernel.

```
void mat_mult(data_t* ret_val, const data_t* a, const data_t* b,
              int row_a, int row_b, int col_b,
              OclWorkEngine* engine = NULL)
```

Figure 2-6. Function signature of mat_mult

The *mat_mult* compute kernel performs the canonical form of matrix multiplication of a *row_a* by *row_b* matrix *a* mutiplied by a *row_b* by *col_b* matrix *b*. The result is saved as a *row_a* by *col_b* matrix *ret_val*. In mathematical terms,

$$ret\_val = a \cdot b$$

The *mat_mult* compute kernel has *O(row_a * col_b * row_b)* time complexity.

mat_mult_col

Figure 2-7 presents the function signature of the *mat_mult_col* compute kernel.

```
void mat_mult_col(data_t* ret_val, const data_t* a, const data_t* b,
                  int row_a, int row_b, int col,
                  OclWorkEngine* engine = NULL)
```

Figure 2-7. Function signature of mat_mult_col

The *mat_mult_col* compute kernel is a "shorthand" implementation of matrix multiplication using the *mat_trans* compute kernel and the *mat_mult* compute kernel. It essentially performs a transpose on the *row_b* by *col* matrix *b*, and then matrix-multiplies the

row_a by *col* matrix *a* by the transposed matrix *b*. The result is saved as a *row_a* by *row_b* matrix *ret_val*. In mathematical terms,

$$ret\_val = a \cdot b^T$$

The *mat_mult_col* compute kernel has *O(row_a * row_b * col)* time complexity.

mat_mult_row

Figure 2-8 presents the function signature of the *mat_mult_row* compute kernel.

```
void mat_mult_row(data_t* ret_val, const data_t* a, const data_t* b,
                  int col_a, int col_b, int row,
                  OclWorkEngine* engine = NULL)
```

Figure 2-8. Function signature of mat_mult_row

The *mat_mult_row* compute kernel is another "shorthand" implementation of matrix multiplication using the *mat_trans* compute kernel and the *mat_mult* compute kernel. It essentially performs a transpose on the *row* by *col_a* matrix *a*, and then matrix-multiplies the transposed matrix *a* by the *row* by *col_b* matrix *b*. The result is saved as a *col_a* by *col_b* matrix *ret_val*. In mathematical terms,

$$ret\_val = a^T \cdot b$$

The *mat_mult_row* compute kernel has *O(col_a * col_b * row)* time complexity.

2.2.4 Stochastic gradient decent (SGD) training cycle

Figure 2-9 illustrates the workflow of a training epoch on a single machine.

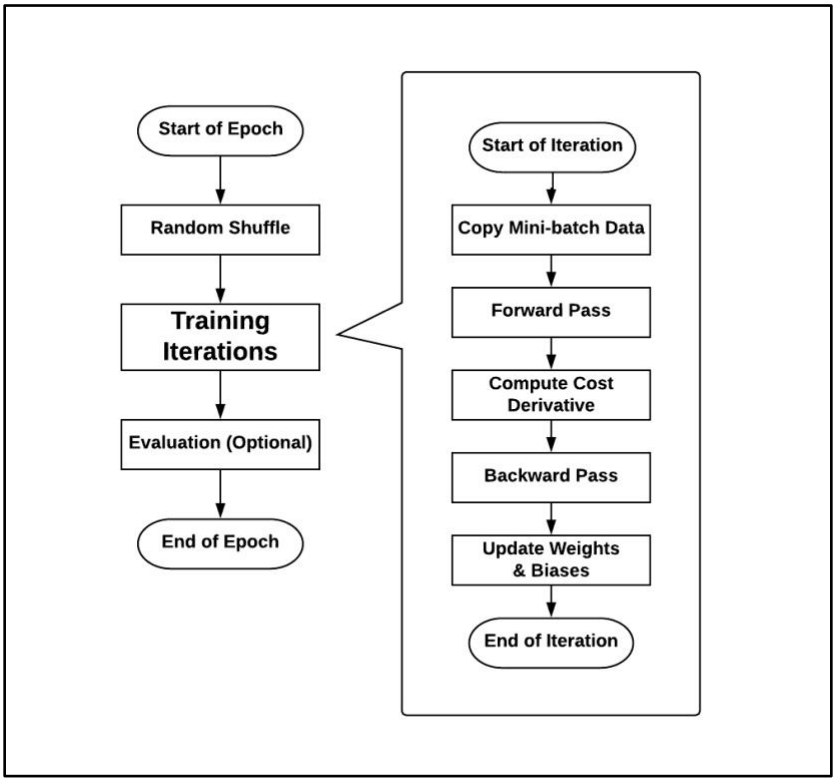


Figure 2-9. Training epoch workflow on a single machine

In each training epoch, a number of steps/iterations will be performed depending on the mini-batch size. To maintain consistency with Nielsen's implementation in [6], the forward pass, the compute cost derivative process and the backward pass together are referred to as *backpropagation*.

Before explaining the implementation for the sub-procedures in detail, the notations of variables and functions are listed in Table 2-2 for better mathematical representation.

Notation	Corresponding Variable Name (If Applicable)	Variable or Function	Definition
L	*num_layers*	variable	total number of layers
n	*train_data_size*	variable	training data size
m	*mini_batch_size*	variable	mini-batch size
x	*activations[0]*	variable	copied training data inputs (mini-batch)
y	*y[]*	variable	copied training data outputs (mini-batch)
w	*weights[]*	variable	weights (by layer)
b	*biases[]*	variable	biases (by layer)
C'		variable	cost derivatives (by layer)
a	*activations[]*	variable	activations of neurons (by layer)
S		function	sigmoid function
S'		function	sigmoid prime function
σ'	*primes[]*	variable	sigmoid function derivatives (by layer)
∇w	*nabla_w[]*	variable	partial derivatives of weights (by layer)
∇b	*nabla_b[]*	variable	partial derivatives of biases (by layer)
η	*eta*	variable	learning rate
λ	*lambda*	variable	normalization factor

Table 2-2. Function signature of sigmoid

The detailed mathematical derivation and proof of each sub-process is beyond the scope of this project, [6] is recommended for further reading.

Random shuffle and copy mini-batch data

Shuffling the dataset memory directly can cause unnecessary overhead when the data width is large. When considering extending the implementation to a distributed setting, having to synchronize on the entire shuffled dataset is an efficient design choice. Instead, simply create a vector of indices, shuffle the indices and then copy the data entry corresponding to the indices by the mini-batch size for each training iteration. In the process of copying mini-batch data, the current mini-batch is copied to the first activation layer *activations[0]*.

Forward pass

The forward pass propagates values including the activations and sigmoid derivatives from the input layer all the way up to the final output layer. In mathematical terms, the process is expressed as

$$a_{i+1} = S(a_i \cdot w_i + b_i) \qquad \text{for layer } i \in [0, L-2]$$

$$\sigma_i' = S'(a_i \cdot w_i + b_i) \qquad \text{for layer } i \in [0, L-2]$$

The corresponding implementation is listed in Figure 2-10.

```
for (int i = 0; i < num_layers - 1; i++) {

    mat_mult_col(activations[i+1], activations[i], weights[i],
                 worker_batch_size, sizes[i+1], sizes[i], engine);

    mat_vec_add(activations[i+1], biases[i],
                worker_batch_size, sizes[i+1], engine);

    sigmoid_prime(primes[i], activations[i+1],
                  worker_batch_size * sizes[i+1], engine);

    sigmoid(activations[i+1], worker_batch_size * sizes[i+1], engine);

}
```

Figure 2-10. Forward pass implementation

Compute cost derivative

As the name implies, computing cost derivative is the process of generating the derivatives of the cost function for the propagated output against the ground truth. For the cross-entropy cost function, the mathematical representation of this operation is simply

$$\nabla b_{L-2} = a_{L-1} - y$$

The corresponding implementation is listed in Figure 2-11.

```
for (int i = 0; i < worker_batch_size * sizes[num_layers-1]; i++) {

    nabla_b[num_layers-2][i] = activations[num_layers-1][i] - y[i];

}
```

Figure 2-11. Cost derivative implementation

Backward pass

The backward pass of the training iteration computes the gradients of each activation layer based on the cost derivatives. In practice, the backward pass should cost the largest portion of execution time in the sequential training process because of the matrix multiplications involved.

In mathematical terms,

$$\nabla b_i = (\nabla b_{i+1} \cdot w_{i+1}) \odot S_i' \qquad \text{for layer } i \in [0, L-3], \text{in reverse order}$$

$$\nabla w_i = \nabla b_i^T \cdot a_i \qquad \text{for layer } i \in [0, L-2], \text{in reverse order}$$

Figure 2-12 shows the implementation for the backward pass.

```
mat_mult_row(nabla_w[num_layers-2], nabla_b[num_layers-2],
    activations[num_layers-2], sizes[num_layers-1], sizes[num_layers-2],
    worker_batch_size, engine);

for (int i = num_layers - 3; i >= 0; i--) {

    mat_mult(nabla_b[i], nabla_b[i+1], weights[i+1], worker_batch_size,
             sizes[i+2], sizes[i+1], engine);

    vec_mult(nabla_b[i], primes[i], worker_batch_size * sizes[i+1],
             engine);

    mat_mult_row(nabla_w[i], nabla_b[i], activations[i], sizes[i+1],
                 sizes[i], worker_batch_size, engine);

}
```

Figure 2-12. Backward pass implementation

Update mini-batch

The mini-batch update is the last step in a training iteration where the computed gradients are applied to the parameters. The mathematical representation for this procedure is

$$b_i \leftarrow b_i - \frac{\eta}{m} \cdot \nabla b_i \qquad \text{for layer } i \in [0, L-2]$$

$$w_i \leftarrow \left(1 - \frac{\eta\lambda}{n}\right) w_i - \frac{\eta}{m} \cdot \nabla w_i \qquad \text{for layer } i \in [0, L-2]$$

The implementation for mini-batch update is included in Figure 2-13.

```
for (int j = 0; j < worker_batch_size; j++) {

    for (int i = 0; i < num_layers - 1; i++) {

        for (int k = 0; k < sizes[i+1]; k++) {

            biases[i][k] = biases[i][k] - (eta / worker_batch_size) *
                           nabla_b_sum[i][j * sizes[i+1] + k];

        }

    }

}

data_t reg = 1.0 - eta * lmbda / train_data_size;

for (int i = 0; i < num_layers - 1; i++) {

    for (int j = 0; j < sizes[i+1] * sizes[i]; j++) {

        weights[i][j] = reg * weights[i][j] - (eta / worker_batch_size) *
                        nabla_w_sum[i][j];

    }

}
```

Figure 2-13. Update mini-batch implementation

Evaluation (optional)

The network is evaluated against the test dataset after each training epoch. The evaluation process is marked as optional because it is does not necessarily affect the training procedure. For the practical usage of a learning model, of course, measurements are required on how well the learner has achieved after a certain number of training epochs. However, one could also argue about the number of epochs between each evaluation. In this project, the model is evaluated against the entire test set after each epoch for the sake of both model integrity and canonicalization.

The evaluation procedure performs the forward pass on the test inputs and comes up with the predicted output label from the neuron in the output layer with the highest activation value.

The total number of correct predictions are counted by comparing the predicted output and the ground truth. Figure 2-14 covers the implementation of the evaluation process.

```
int num_iterations = worker_test_size / worker_batch_size;
int arg_max;
data_t* cell;
int sum = 0;
for (int i = 0; i < num_iterations; i++) {
    for (int j = 0; j < worker_batch_size * sizes[0]; j++) {
        activations[0][j] = test_data_x[i*worker_batch_size*sizes[0] + j];
    }
    feedforward();
    for (int j = 0; j < worker_batch_size; j++) {
        arg_max = 0;
        cell = &activations[num_layers-1][j * sizes[num_layers-1]];
        for (int k = 1; k < sizes[num_layers-1]; k++) {
            if (cell[k] > cell[arg_max]) {
                    arg_max = k;
            }
        }
        if (arg_max == (int)test_data_y[i * worker_batch_size + j]) {
            sum++;
        }
    }
```

Figure 2-14. Evaluation implementation

2.3 Trial run

Till this point, the majority of the structures needed to train a neural network model sequentially have been covered[3]. Before exploiting the parallelization possibilities, the implementation is tested using the MNIST dataset in order to check whether the sequential model is learning properly. Figure 2-15 is an excerpt output log for a 30-epoch training session.

```
Epoch 0
Random shuffle (ms):        1
Copy shuffled data (ms):    16
Back propagation (ms):      4988
Update w & b (ms):          73
Evaluation (ms):            699
Total (ms):                 5850
Test accuracy: 9552 / 10000

Epoch 1
Random shuffle (ms):        1
Copy shuffled data (ms):    15
Back propagation (ms):      4911
Update w & b (ms):          69
Evaluation (ms):            698
Total (ms):                 5761
Test accuracy: 9646 / 10000

Epoch 2
Random shuffle (ms):        1
Copy shuffled data (ms):    15
Back propagation (ms):      4868
Update w & b (ms):          68
Evaluation (ms):            694
Total (ms):                 5710
Test accuracy: 9703 / 10000

...
```

```
...

Epoch 9
Random shuffle (ms):        1
Copy shuffled data (ms):    15
Back propagation (ms):      4849
Update w & b (ms):          67
Evaluation (ms):            697
Total (ms):                 5692
Test accuracy: 9732 / 10000

...

Epoch 29
Random shuffle (ms):        1
Copy shuffled data (ms):    15
Back propagation (ms):      4875
Update w & b (ms):          67
Evaluation (ms):            694
Total (ms):                 5716
Test accuracy: 9781 / 10000
```

Figure 2-15. Excerpt output log of 30 epochs using sequential implementation

In this trial run, a neural network model with 1 hidden layer of 100 neurons is trained, η set to 0.5 and λ set to 5.0. After 30 training epochs, the model achieves a test accuracy of 97.81 percent, which is on par with the 97.92 percent claimed by [6] using the same setup. The fact that

[3] The sequential implementation of the compute kernels is very intuitive and thus not discussed in this chapter.

the test accuracy started as high as 95.52 and gradually increased after each epoch is a sign that the model is sound.

2.4 Parallelization using heterogeneous compute devices

From timing results in the trial run output (Figure 2-15), it is obvious that the backpropagation process takes the majority of the execution time. The second most time-consuming portion is evaluation. Both backpropagation and evaluation consist mainly invocations of the compute kernels covered previously in Chapter 2.2.3.

In the trial runs, compiling the sequential implementation using the *-O3* flag will provide up to 4 times speed up. This is a joint result of loop unfolding, AVX support and other compiler optimizations. The sequential implementation of compute kernels mainly involves repeating operations that do not have dependencies in between. As an example, the sigmoid compute kernel as listed in Figure 2-16, simply computes the sigmoid function for each independent element in the *ret_val* array. This presents an opportunity to execute the same operation for each element in parallel.

```
for (int i = 0; i < size; i++) {
    ret_val[i] = 1.0 / (1.0 + exp(-ret_val[i]));
}
```

Figure 2-16. Sequential implementation of the sigmoid compute kernel

At the time this thesis is written, a consumer-grade multi-core CPU would typically have core counts no more than 8. On the other hand, an entry level dedicated graphics cards or even an integrated graphics card on the CPU can have more than 20 cores, making the cost per core much lower on the side of GPUs. Although the term “core” is defined differently on the CPU vs the GPU which poses a somewhat unfair comparison, it is still considered reasonable using the core count as a measurement of the ability to parallel-compute small and simple tasks. Since the

operations for the compute kernels are simple and mostly free of branches, using GPUs would pay off in the long run in terms of compute power over cost, especially when the workload is heavy. Apart from the additional core count, GPU also schedules a number of tasks simultaneously and utilizes fast context switches to hide memory latencies.

The OpenCL 1.2 standard is used in order to interface with a GPU, because it is not tied to a specific vendor and not restricted to GPUs alone. In order to dispatch the computationally intensive tasks onto the GPU, on pieces of implementation are needed: for the *host* and for the *compute device*. The host in this project is the CPU that executes the main framework for training the neural network model. The compute device can be any SIMD architecture such as GPU, FPGA or even CPU that supports the OpenCL standard. A compute device consists of a number of *compute units*, and inside each compute unit, there is a number of *processing elements*. The hardware counterparts of these terms vary by architecture and vendor.

A program that gets executed on the compute device consists of functions called OpenCL *kernels*. Figure 2-17 provides the source code of the OpenCL kernel program for the compute kernels used in the learner model. Note that the aim here is not to build the fastest possible implementation, because optimization often depends on the specific device and platform used. However, one cannot omit the fact that a more sophisticated implementation may experience significantly speed-up using techniques such as setting up memory barriers and balancing the workload on each processing element. The simple and straight-forward implementation uses global memory only, because the focus of the timing analysis is on scaling instead of raw performance of execution time.

```
typedef float data_t;

__kernel void sigmoid(__global data_t* s)
{
    int gid = get_global_id(0);
    s[gid] = 1.0 / (1.0 + exp(-s[gid]));
}

__kernel void sigmoid_prime(__global data_t* sp,
                            __global const data_t* z)
{
    int gid = get_global_id(0);
    data_t s = 1.0 / (1.0 + exp(-z[gid]));
    sp[gid] = s * (1.0 - s);
}

__kernel void vec_mult(__global data_t* b,
                       __global const data_t* a)
{
    int gid = get_global_id(0);
    b[gid] *= a[gid];
}

__kernel void mat_vec_add(__global data_t* mat,
                          __global const data_t* vec,
                          int col)
{
    int gid_x = get_global_id(0);
    int gid_y = get_global_id(1);
    mat[gid_y * col + gid_x] += vec[gid_x];
}

__kernel void mat_trans(__global data_t* ret_mat,
                        __global const data_t* src_mat,
                        int row, int col)
{

    int gid_x = get_global_id(0);
    int gid_y = get_global_id(1);
    ret_mat[gid_y * row + gid_x] = src_mat[gid_x * col + gid_y];
}

__kernel void mat_mult(__global data_t* c,
                       __global const data_t* a,
                       __global const data_t* b,
                       int row_b, int col_b)
{
    const int gid_x = get_global_id(0);
    const int gid_y = get_global_id(1);
    data_t value = 0;
    for (int k = 0; k < row_b; k++) {
        value += a[gid_y * row_b + k] * b[k * col_b + gid_x];
    }
    c[gid_y * col_b + gid_x] = value;
}
```

Figure 2-17. Implementation for OpenCL kernels on compute device

A running instance of an OpenCL kernel is called a work-item. [20] states that "a work-item is executed by one or more processing elements as part of a *work-group* executing on a compute unit". Each work-item is associated with a *global ID* that serves as an indexing scheme to uniquely identity itself. The indexing can be 1-dimensional, and up to 3-dimensional depending on the work dimension. Table 2-3 summarizes the work dimension of each kernel and lists whether or not it contains loops or not.

Kernel Name	Work Dimension	Contain Loops?	Corresponding Compute Kernel(s)
sigmoid	1	No	*sigmoid*
sigmoid_prime	1	No	*sigmoid_prime*
vec_mult	1	No	*vec_mult*
mat_vec_add	2	No	*mat_vec_add*
mat_trans	2	No	*mat_trans*
mat_mult	2	Yes	*mat_mult, mat_mult_col, mat_mult_row*

Table 2-3. Work dimension and complexity of each OpenCL kernel

In order to easier set up the host side and communicate with the compute devices, the *OclWorkEngine* class is implemented as a dedicated framework for using OpenCL. The detail usage along with design ideas is covered in Chapter 3.2.

2.5 Extension to distributed learning using MPI

There is always a limit to the number of compute devices available on a single host. For GPUs, this generally means the maximum number of PCI-Express lanes a single CPU socket can support. Another direction of parallelism is to coordinate with multiple instances of hosts. This is often called *distributed learning*.

The open standard message passing interface (MPI) is used for communication between nodes. Table 2-4 lists the MPI operations used in the distributed learner implementation.

Operation Name	MPI function	Definition
broadcast	*MPI_Bcast*	Broadcast a message from the root process
reduce	*MPI_Reduce*	Applies a reduction operation on all tasks and places the result in the receive buffer on the root process
scatter	*MPI_Scatter*	Each process receives a segment from the root
gather	*MPI_Gather*	Each process sends contents to the root (opposite of scatter)

Table 2-4. MPI operations used in the distributed learner [24]

Figure 2-18 and Figure 2-19 illustrates the epoch workflow and iteration workflow, respectively, of a distributed training session using MPI with 3 workers as an example.

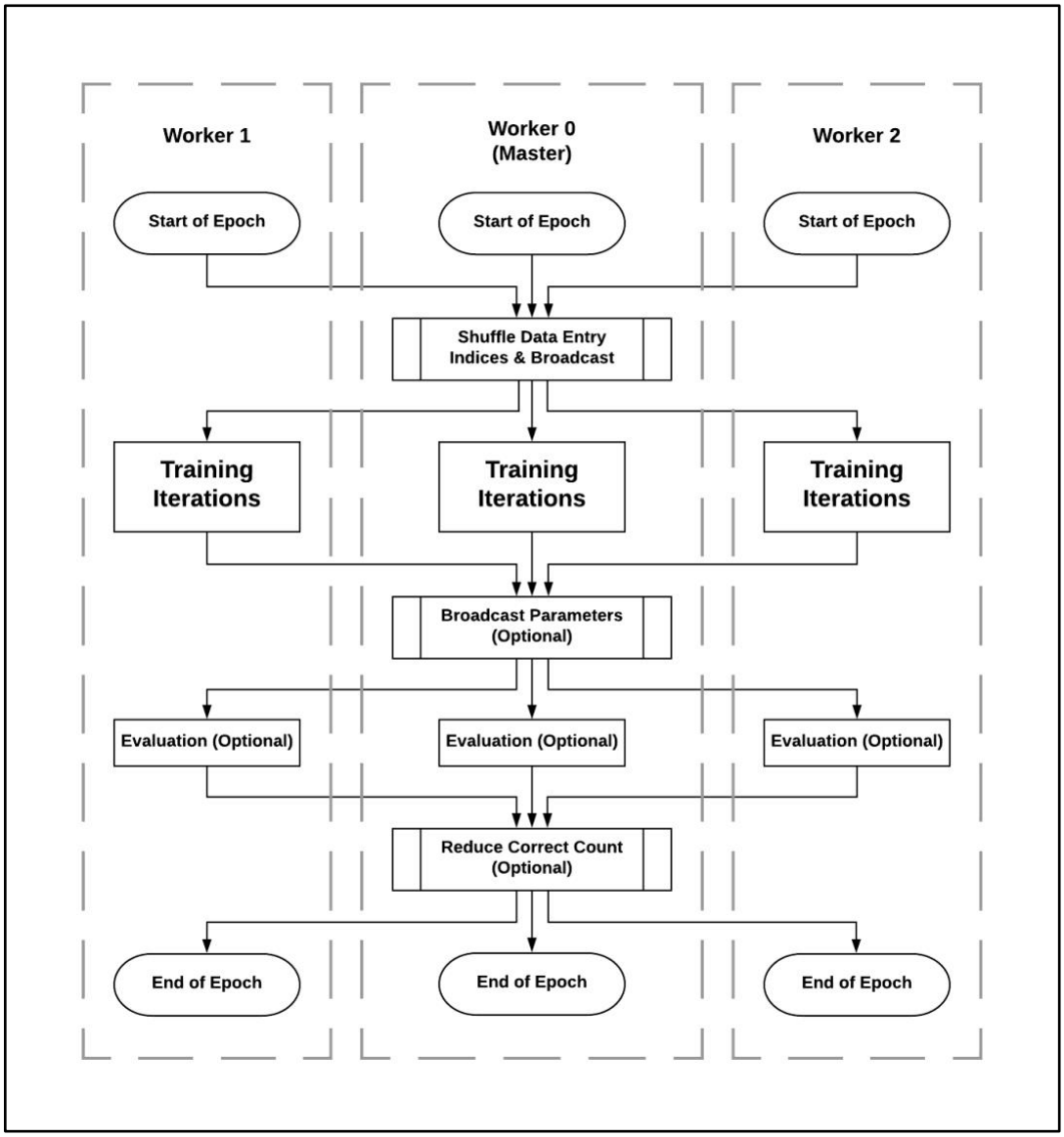


Figure 2-18. Training epoch workflow of distributed learning using MPI

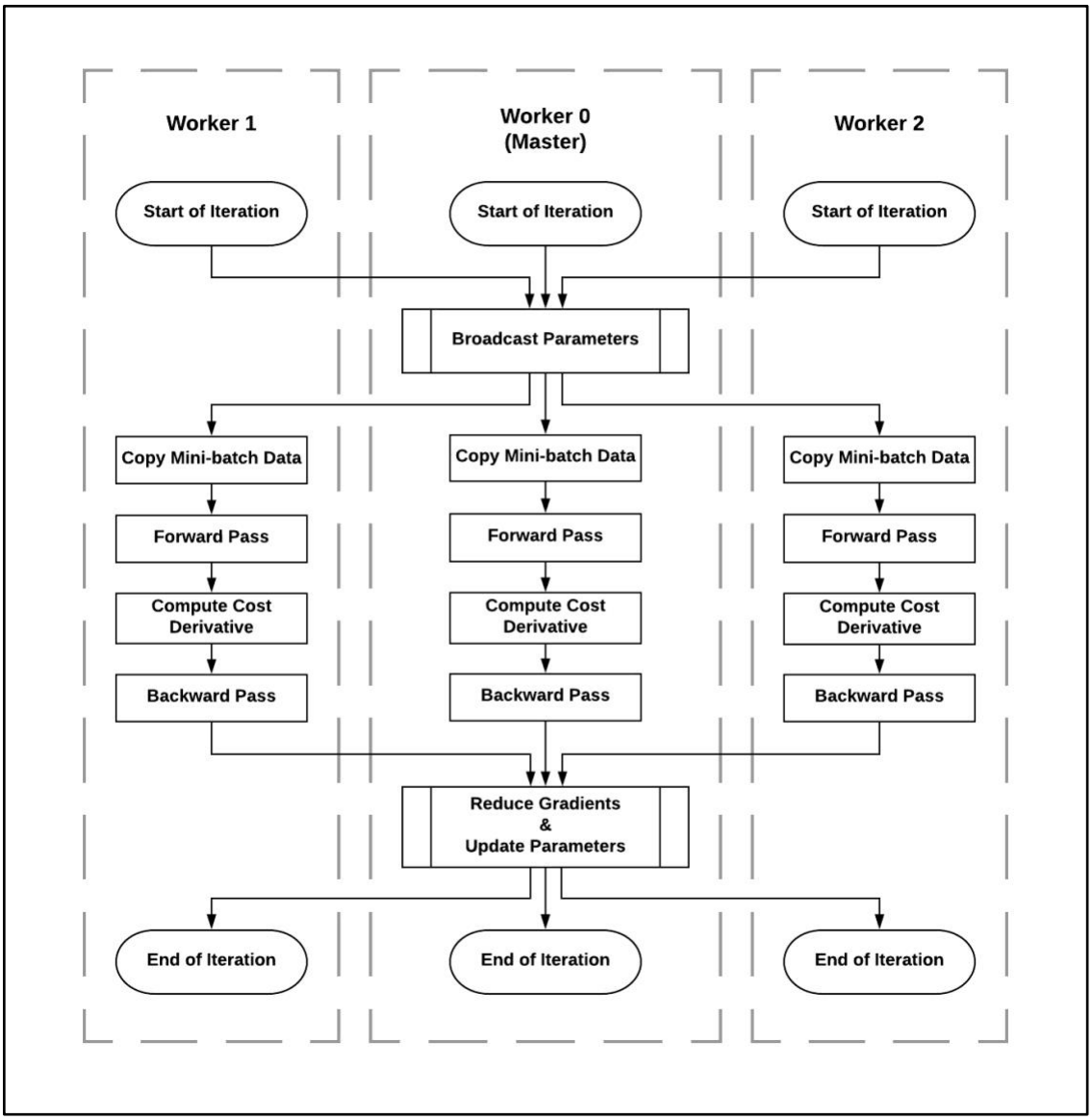


Figure 2-19. Training iteration workflow of distributed learning using MPI

The distributed implementation takes the synchronous approach. At the beginning of each epoch, the indices of the training dataset are shuffled on the master and then scattered to the other workers. At the beginning of each iteration, all parameters are broadcast to all workers other than the master. In addition, the gradients of each worker are reduced via summation and applied to the parameters at the end of each training iteration. Since the parameters used for each training iteration are identical across all workers, the distributed implementation is functionally equivalent to the sequential version.

Chapter 3 – Framework APIs

This chapter documents the APIs of the frameworks. Despite the focus being the public APIs, insights are provided on the internal data structures for using the APIs more effectively. A brief usage example is also included for each of the framework.

For implementation details and complete source code, refer to the supplement source files under "cpp/".

3.1 The Network class: framework for dense neural network

The dense neural network framework is implemented as a C++ class named as *Network*.

3.1.1 Constructor

The constructor as presented in Figure 3-1 takes in two mandatory/required parameters, namely *sizes* and *num_layers*. The former is a pointer to a 1-D array that specifies the fully connected neural network structure, including the input layer and the output layer, whereas the later specifies the number of layers (i.e. the size of the array).

```
Network(const int* sizes, int num_layers, bool useOCL = false,
        bool useMPI = false, std::ostream* out = &std::cout);
```

Figure 3-1. Network class constructor

There are three optional parameters. These optional parameters allow the user to specify if OpenCL devices are involved via the *useOCL* flag, and to specify if distributed learning is used via the *useMPI* flag, and finally to specify if an output location other than the standard output is used (e.g. storing the log information into a separate text file). Figure 3-2 provides a simple example of instantiating a network instance.

```
int sizes[3] = [784, 30, 10];

Network net(sizes, 3, useOCL, useMPI);
```

Figure 3-2. Network class instantiation example

In this example, a network of three layers is created, including an input layer of 784 neurons, a hidden layer of 30 neurons, and an output layer of 10 neurons. If distributed learning is enabled as specified by the *useMPI* flag, every work node spawned will instantiate its own network instance with an identical layer structure.

3.1.2 Load dataset

Once a network is instantiated, the dataset can be loaded using the *load_dataset* method (Figure 3-3). The user needs to provide the file location for both the training data file of *train_data_size* number of training examples and the test data file of *test_data_size* number of test examples.

```
void load_dataset(const char* train_file_name, const char* test_file_name,
                  int train_data_size, int test_data_size);
```

Figure 3-3. Network class load_dataset method

The expected data file format follows the binary version of CIFAR-10 dataset [25]. For each image example, the first byte is the output label, and the next *input_layer_size* number of bytes are the values of the pixels of the image. For the MNIST training dataset, the data file would be formatted as in Figure 3-4. The dataset files are included under "cpp/neuralnet_data/".

```
<1 x label><784 x pixel>
...
<1 x label><784 x pixel>
(for 50,000 examples)
```

Figure 3-4. MINIST data file format

If distributed learning is enabled as specified by the *useMPI* flag, an additional step of MPI *Bcast* operation will be performed in order to broadcast the datasets to all worker nodes other than the master node. And in this case, only the master node will perform the actual file read.

3.1.3 Training using stochastic gradient descent

Calling the SGD method (Figure 3-5) starts the training process for a given number of epochs.

```
void SGD(int epochs, int mini_batch_size, data_t eta, data_t lmbda);
```

Figure 3-5. Network class SGD method

The *mini_batch_size* parameter indicates the number of examples trained in a single iteration. For a mini-batch size of 10, this would mean 10 examples trained each iteration for a total of 50,000 / 10 = 5,000 iterations in one epoch. The parameter *eta* specifies the learning rate, and the parameter *lmbda* is a normalization factor used to achieve better training accuracy. The last two tuning parameters are equivalent to those used in the Python version implemented by Michael Nielson [6]. Note that the data type is parameterized as *data_t* so that the user may configure as *float* or *double*.

If distributed learning is enabled as specified by the *useMPI* flag, each node will start the training process asynchronously[4] upon calling the *SGD* method. The synchronization points will be reached when the corresponding MPI operations are called, such as when the master node broadcasts the updated weights and biases of all layers at the beginning of each training iteration.

[4] Note that this does not mean the implementation is asynchronous. It is simply the fact that there are parts in the training where each worker performs asynchronous computation locally.

3.1.4 Example usage

Figure 3-6 demonstrates a usage example from instantiation to an SGD training session.

```
bool useOCL = true;

bool useMPI = true;

int sizes[3] = [784, 30, 10];

Network net(sizes, 3, useOCL, useMPI);

net.load_dataset("mnist_train.bin", "mnist_test.bin", 50000, 10000);

net.SGD(30, 1000, 0.5, 0.0);
```

Figure 3-6. Network class usage example

In this simple example, a network with one hidden layer is instantiated for the MNIST dataset. The *useOCL* and *useMPI* flags are both enabled, implying a distributed learning scenario with multiple compute devices. After loading the dataset files, the SGD training process performs 30 epochs with a mini-batch size of 1000, a learning rate of 0.5 and a default 0.0 normalization factor.

3.2 The OclWorkEngine class: abstraction interface for OpenCL

The *OclWorkEngine* class is a utility class for mitigating the overhead when using the OpenCL C++ framework in order to perform parallel computation on OpenCL compatible devices such as GPUs. The implementation of *OclWorkEngine* is built based on the utility functions provided in OpenCL Programming Guide [26].

3.2.1 Constructor

The constructor of the *OclWorkEngine* class is presented in Figure 3-7. It takes in two optional parameters.

```
OclWorkEngine(int index = 0, bool enableProfiling = false);
```

Figure 3-7. OclWorkEngine class constructor

The former parameter *index* is used for organizing the usage of multiple GPUs on various compute nodes in the distributed learning mode. The latter *enableProfiling* is for timing purposes. Setting the value true enables more fine-grained timing measurements using the OpenCL profiling features available on the compute devices, yet it may affect the performance. Upon instantiating an *OclWorkEngine* instance, the context and command queue are allocated.

Although the target OpenCL compute devices are GPUs, there should be minimal modification to the source necessary for using other architectures that support OpenCL 1.2 standard.

3.2.2 Create OpenCL program

The *createProgram* method (Figure 3-8) compiles the OpenCL program file and internally stores a handle to the OpenCL program object.

```
void createProgram(const char* fileName);
```

Figure 3-8. OclWorkEngine createProgram method

In case an error occurs when the program fails to compile, the program build log will display as the error message.

3.2.3 Create OpenCL kernels

The createKernel method (Figure 3-9) essentially calls the *clCreateKernel* function from the OpenCL 1.2 API to generate a handle of the kernel object.

```
void createKernel(const char* kernelName);
```

Figure 3-9. OclWorkEngine class createKernel method

Internally the kernel handle is stored in the hash map *kernels* as a C++11 standard *unordered_map*, and the memory objects hash map *memObjects* adds the kernel object handle as a new key, ready to allocate buffer objects depending on the kernel arguments.

It is important to note that the *createProgram* method must be invoked before calling the *createKernel* method, otherwise an error message will display because the OpenCL program is not compiled yet.

3.2.4 Set OpenCL kernel arguments

The setKernelArg method (Figure 3-10) sets the argument for the kernel.

```
cl_ulong setKernelArg(const char* kernelName, int argIndex,
    cl_mem_flags memFlag, size_t size, const void* argValue,
    cl_bool blocking = CL_FALSE);
```

Figure 3-10. OclWorkEngine class setKernelArg method

If the argument is a single value, the *memFlag* parameter should be set to 0. Otherwise if the argument is an array, the memFlag should be CL_MEM_READ for read-only content or CL_MEM_READ_WRITE for both reading and as return values. The *size* parameter should indicate the size of memory pointed by the *argValue* parameter (e.g. *sizeof(int)* if passing a single *int* variable). For arrays, the memory objects hash map *memObjects[kernel]* attempts to add the key-value pair of the kernel argument index *argIndex* and the allocated buffer object for future reference.

For timing purposes, the *blocking* parameter specifies if the method blocks and return after the memory object (if applicable) has been created and the corresponding data has been transferred to the compute device.

3.2.5 Enqueue OpenCL kernel

The *enqueueKernel* method (Figure 3-11) enqueues the work items for execution on the compute device.

```
cl_ulong enqueueKernel(
    const char* kernelName, cl_uint workDim,
    const size_t* globalWorkSize, const size_t* localWorkSize,
    bool blocking = false);
```

Figure 3-11. OclWorkEngine class enqueueKernel method

The *kernelName* parameter specifies the name of the kernel for execution. The *workDim* parameter indicates the dimension of the index space. The *globalWorkSize* parameter sets the total number of work items (i.e. number of kernel instances) on each index space dimension, and the *localWorkSize* parameter sets the number of work items on each index space dimension in an OpenCL workgroup. The parameters *workDim*, *globalWorkSize* and *localWorkSize* are equivalent to the OpenCL 1.2 specifications [20].

For timing purposes, the *blocking* parameter specifies if the method blocks and return after all work items have completed execution on the compute device.

3.2.6 Enqueue OpenCL buffer read

The enqueueReadBuffer method (Figure 3-12) enqueues the buffer-read command.

```
cl_ulong enqueueReadBuffer(
    const char* kernelName, int argIndex, size_t size, void* result,
    cl_bool blocking = CL_TRUE);
```

Figure 3-12. OclWorkEngine class enqueueReadBuffer method

The host memory location is indicated by the *result* parameter. The *argIndex* parameter must be the same as previously specified by invoking the *setKernelArg* method.

For timing purposes, the *blocking* parameter specifies if the method blocks and return after all values have been transferred from the compute device back to the host memory.

3.2.7 Example usage

Figure 3-13 shows an example usage of the *OclWorkEngine* class that performs the sigmoid function on all values in the array *ret_val*.

```
// Dynamically instantiate engine, enable profiling
OclWorkEngine *engine = new OclWorkEngine(0, true);

// Compile the program file
engine->createProgram(OCL_PROGRAM);

// Create kernel
engine->createKernel(SIGMOID_KERNEL);

// Set kernel arguments and create OpenCL buffer object
elapsed = engine->setKernelArg(SIGMOID_KERNEL, 0, CL_MEM_READ_WRITE,
                               size * sizeof(data_t), ret_val, CL_TRUE);

// Enqueue kernel for execution
elapsed = engine->enqueueKernel(SIGMOID_KERNEL, 1, globalWorkSize, NULL,
                                true);

// Copy back results from compute device to host
elapsed = engine->enqueueReadBuffer(SIGMOID_KERNEL, 0,
                                    size * sizeof(data t), ret val);
```

Figure 3-13. OclWorkEngine class usage example

Since the sigmoid kernel is a 1-D operation, the only argument that needs to be passed to the kernel is the array. The size of the array is passed as the global work size so that each instance of the kernel is handled on one processing element. After the enqueued kernel is completed, the values are copied from the buffer object to the *ret_val* array.

3.3 The NanoTimer class: nanosecond-precision timer

The *NanoTimer* class is a utility class for measuring the elapsed time on the host machine. It is based on *timer.h* in C11, which provides the *timespec* struct holding an interval broken down into seconds and nanoseconds and the *clock_gettime* function for retrieving the current time [27]. When calling the *clock_gettime* function, *CLOCK_REALTIME* is used to get the total elapsed time, because a large portion of the OpenCL kernel execution are completed on

the compute device, which does not add up towards processor time on the CPU. Therefore, measuring the total elapsed time is a more precise reflection of the actual performance.

3.3.1 Constructor

The only constructor is the default constructor (Figure 3-14).

```
NanoTimer();
```

Figure 3-14. NanoTimer class constructor

In essence, there are three storage pieces:

1. The start time as a *timespec* struct indicating the start of the current measuring interval of interest.
2. The end time as a *timespec* struct indicating the end of the current measuring interval of interest.
3. A differential interval represented by two separate *uint64_t* values which stand for second and the remainder in nanoseconds.

Upon instantiation, the default constructor stores the current time as both the start time and the end time and sets the differential interval to zero.

3.3.2 Restart, stop and resume timer

Figure 3-15 lists the restart method, the stop method and the resume methods for the timer.

```
void restart();
void stop();
void resume();
```

Figure 3-15. NanoTimer class restart method, stop method and resume methods

The restart method reinitializes the timer and clears any previously recorded time. The stop method updates the internally-stored end time using the current time. The resume method updates the differential interval by computing the elapsed time then restarts the timer.

3.3.3 Get elapsed time

As per Figure 3-16, there are two methods for retrieving the elapsed time from a timer instance.

```
void getElapsed(uint64_t &sec, uint64_t &nano);

uint64_t getElapsedMS();
```

Figure 3-16. NanoTimer class get elapsed time methods

The *getElapsed* method provides nanosecond resolution. The user needs to pass two variables as reference in order to get the time values. The *getElapsedMS* method is a "shorthand" way of retrieving the time elapsed converted to the closest millisecond (rounded down).

3.3.4 Example usage

Figure 3-17 demonstrates an example for using the *NanoTimer* class. Both methods for retrieving the elapsed time are used in this example.

```
// Instantiate a timer
NanoTimer timer;

// Restart the timer
timer.restart();

//////////////////////////
// Task to execute      //
//////////////////////////

// Stop the timer
timer.stop();

// Get elapsed time in millisecond
std::cout << "Elapsed in ms: " << timer.getElapsedMS() << std::endl;

// Resume the timer
timer.resume();

//////////////////////////
// More task to execute //
//////////////////////////

// Stop the timer
timer.stop();

// Get elapsed time as seconds and nanoseconds
uint64_t sec, nano;
timer.getElapsed(sec, nano);

std::cout << "Elapsed in ns: " << (sec * 1.0 * 1e9 + nano) << std::endl;
```

Figure 3-17. NanoTimer class usage example

Chapter 4 – Benchmark Results

4.1 Experimental Setup

4.1.1 Hardware specifications

The benchmarks are run on the NewRiver GPU compute engine cluster of Virginia Tech Advanced Research Computing (ARC). The hardware specifications of each node in the GPU compute engine are listed as follows:

- CPU: 2 x E5-2680v4 2.4GHz (Broadwell), 28 cores
- RAM: 512 GB
- GPU: 2 x NVIDIA P100 GPU
- Local disk: 2 x 200GB SSD
- Inter-node network: 100 Gbps EDR-Infiniband providing low latency communication between compute nodes for MPI traffic

The NewRiver cluster uses the Terascale Open-source Resource and QUEue Manager (TORQUE) for managing the resource allocations [28]. It uses Portable Batch System (PBS) for job scheduling. OpenCL 1.2 is provided by NIVIDIA CUDA 8.0.61. The MPI implementation is provided by MVAPICH2 (ver 2.2a), which is based on MPI 3.1 standard [29].

4.1.2 Benchmarking the Network class

The epoch time is benchmarked for training neural networks of varying layer sizes with different hardware resources. Because the control flow of each training epoch is identical, training accuracy is irrelevant to the execution time. The timing data for the first 10 training epochs are collected as the basis for the analysis.

The benchmark test is to run the SGD training algorithm on the MNIST dataset under different configurations. There are three configurable aspects:

1. Number of workers: this essentially means the number of cores running the training instance (one instance/worker on each core).
2. Mini-batch size: number of examples trained in each iteration for all workers combined. Note that the batch size per worker (i.e. worker batch size) is calculated using integer division, which means for smaller mini-batch sizes, the total number of trained examples may be less than the configured size. (For instance, a mini-batch size of 100 on 8 workers would result in only 12 examples trained per worker.)
3. Network structure: this specifies the hidden layer sizes.

The values of the three configurable items are listed in Table 4-1 for CPU and Table 4-2 for GPU, respectively.

Configured Item	**Values**
number of workers	1, 2, 4, 8, 16, 32
mini-batch size	100, 1000, 2000, 5000, 10000
network structure (hidden-layer sizes)	1 hidden layer: [32], [64], [128], [256], [512] 2 hidden layers: [32, 16], [32, 32], [32, 64], [64, 16], [64, 32], [64, 64], [128, 16], [128, 32], [128, 64], [256, 16], [256, 32], [256, 64]

Table 4-1. Configured items and corresponding values for tests using CPU

Configured Item	**Values**
number of workers	1, 2, 3, 4, 5, 6, 7, 8
mini-batch size	10, 100, 1000, 2000, 5000, 10000
network structure (hidden-layer sizes)	1 hidden layer: [32], [64], [128], [256], [512] 2 hidden layers: [32, 16], [32, 32], [32, 64], [32, 128], [64, 16], [64, 32], [64, 64], [64, 128], [128, 16], [128, 32], [128, 64], [128, 128], [256, 16], [256, 32], [256, 64], [256, 128], [512 ,16], [512, 32], [512, 64], [512, 128]

Table 4-2. Configured items and corresponding values for tests using GPU

For CPU, the allocated resources on the GPU cluster are 4 nodes with 16 CPU cores on each node. For GPU, the allocated resources on the GPU cluster are 4 nodes with 2 CPU cores and 2 GPUs on each node. In each training epoch, the time measurements are outputted for the operations listed in Table 4-3.

Operation	Use OpenCL Kernels?	Use MPI?
random shuffle	No	No
MPI scatter indices	No	Yes
copy shuffled data	No	No
MPI bcast train	No	No
backpropagation	Yes	No
MPI reduce (gradients)	No	Yes
update weights & biases	No	No
MPI bcast (evaluation)	No	Yes
evaluation	No	No
MPI reduce (evaluation)	No	Yes

Table 4-3. Measured operations in Network class benchmark

Since there are components that use OpenCL kernels or the MPI, both the *OclWorkEngine* class and the key MPI operations for fine-grained analyses are benchmarked.

Last but not least, the source code including the Makefile for the *Network* class benchmark is under "cpp/neuralnet/".

4.1.3 Benchmarking the OclWorkEngine class

The OpenCL implementation of the compute kernels using the *OclWorkEngine* class is benchmarked. Recall in Table 2-3 there are three types of OpenCL kernels:

1. 1-D kernels: *sigmoid*, *sigmoid_prime*, *vec_mult*
2. 2-D kernels without loops: *mat_vec_mult*, *mat_trans*
3. 2-D kernels with loops: *mat_mult*

Table 4-4 lists the input sizes for each type of kernel. For each 1-D kernel, the size on the only work dimension is varied. For each 2-D kernel without loops, the sizes on both work

dimensions are varied, namely row and column. And lastly for the 2-D kernels with loops, both the row size and the column size are varied, as well as the number of loop iterations in the kernel loop.

OpenCL Kernel Type	Corresponding Compute Kernel(s)	Corresponding OpenCL Kernel(s)	Input Sizes
1-D	*sigmoid*, *sigmoid_prime*, *vec_mult*	*sigmoid*, *sigmoid_prime*, *vec_mult*	size $= 2^i$ for $i \in [6, 24]$
2-D without loops	*mat_vec_add*, *mat_trans*	*mat_vec_add*, *mat_trans*	row size $= 2^i$ for $i \in [5, 12]$, column size $= 2^j$ for $j \in [5, 12]$
2-D with loops	*mat_mult*, *mat_mult_col*, *mat_mult_row*	*mat_mult*	row size $= 2^i$ for $i \in [5, 12]$, column size $= 2^j$ for $j \in [5, 12]$, loop iterations $= 2^k$ for $k \in [5, 12]$

Table 4-4. Work dimension and complexity of each OpenCL kernel

Both the sequential (CPU) and the OpenCL (GPU) implementation of the compute kernels are tested. For isolating the test environment from potential sharing users, the entire node is allocated with all 28 cores and 1 GPU.

The source code for the *OclWorkEngine* class is included under "cpp/oclengine_test/".

4.1.4 Benchmarking MPI performances

Since the focus is on the training epoch, the only two MPI operations benchmarked are *MPI_Bcast* and *MPI_Reduce*. For the MPI benchmark, both the input size and the hardware resource allocation are varied in order to study the effects of the node structure.

Table 4-5 lists the configurations for testing the two MPI operations. Note that the allocation on the NewRiver cluster for this project is limited to a maximum of 8 nodes.

Operations	Compute Nodes	Input Sizes	Processors per Node (PPN)	Processor Counts
MPI_Bcast, MPI_Reduce	8	$2^i \ for \ i \in [8, 23]$	1	1, 2, …, 8
			2	1, 2, …, 16
			4	1, 2, …, 32
			8	1, 2, …, 64
			16	1, 2, 4, 8, 16, 24, 32, 48, 64, 80, 96, 112, 128

Table 4-5. Configuration of MPI benchmark

The source code for the MPI operations is included under "cpp/mpitest/".

4.1.5 Notes on availabilities of benchmark results

Due to the vast number of input combination tested, it is impossible to present and discuss all results in this thesis. For the purpose of explaining the key aspects, only featured examples are presented and explained. However, the complete output logs are included under "benchmark/output/". In addition, the Python implementation for plotting and modeling the various execution times is included under "python/".

4.2 Benchmark results of the Network class

The epoch time averaged over 10 epochs is converted to the number of examples trained per millisecond—the combined *throughput* of all workers:

$$\text{Throughput} = \text{Number of examples trained per millisecond} = \frac{\text{Training data size}}{\text{Average epoch time}}$$

The throughput is as a function of both the total number of workers and the mini-batch size. In addition, the speed-up over each dimension is calculated as

$$\text{Speed-up over worker count} = \frac{\text{Throughput under current worker count}}{\text{Throughput under minimum worker count}}$$

for each mini-batch size, and

$$\text{Speed-up over mini-batch size} = \frac{\text{Throughput under current mini-batch size}}{\text{Throughput under minimum mini-batch size}}$$

for each worker count.

Figure 4-1 shows the throughput of the smallest-sized neural network ([784, 32, 10]) versus the largest-sized ([784, 256, 64, 10]) tested on the CPU. Figure 4-2 shows the speed-up over worker count for each mini-batch size, and Figure 4-3 shows the speed-ups over mini-batch size for each worker count.

Figure 4-4 shows the throughput of the smallest-sized neural network ([784, 32, 10]) versus the largest-sized ([784, 512, 128, 10]) tested on the GPU. Figure 4-5 shows the speed-up over worker count for each mini-batch size, and Figure 4-6 shows the speed-up over mini-batch size for each worker count.

All plots are demonstrated in log-log scale.

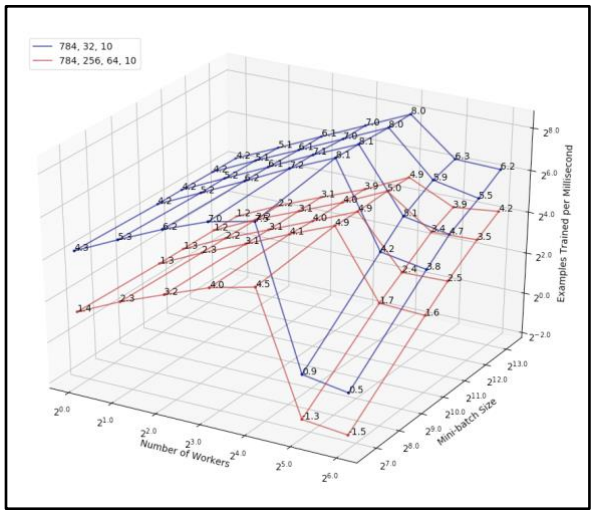


Figure 4-1. Distributed training throughput of neural networks using CPU

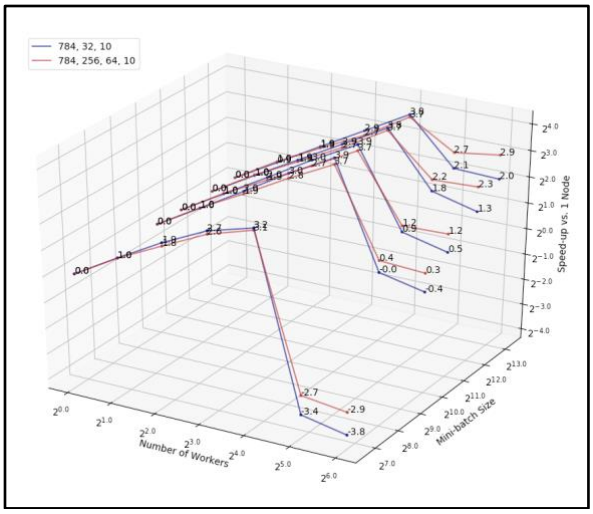


Figure 4-2. Speed-up over worker count of neural networks using CPU

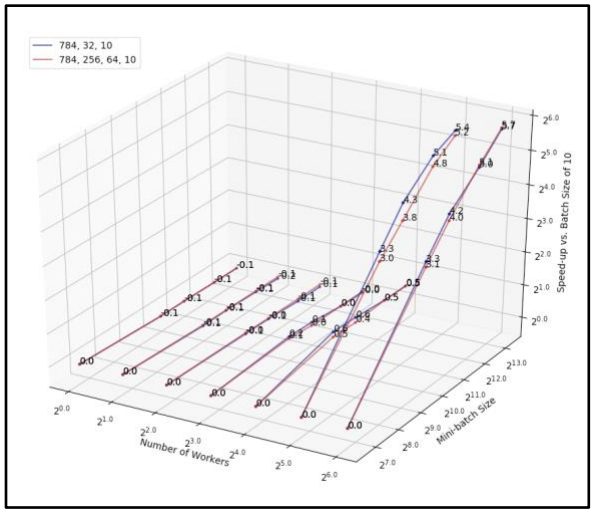


Figure 4-3. Speed-up over mini-batch size of neural networks using CPU

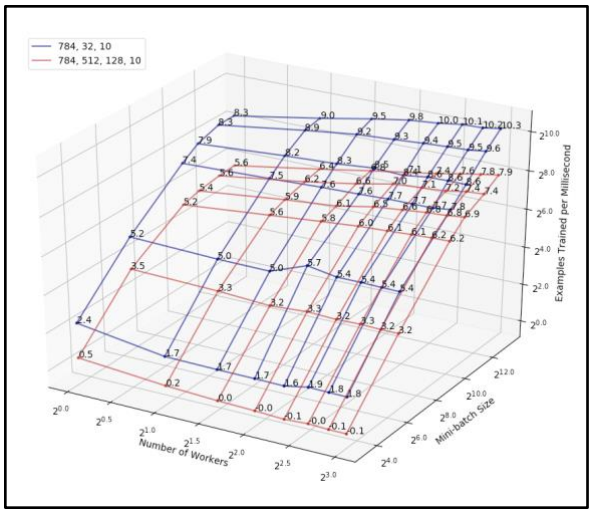


Figure 4-4. Distributed training throughput of neural networks using GPU

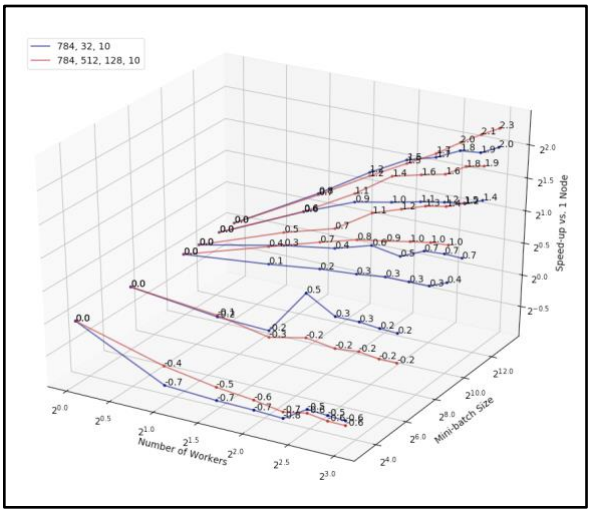


Figure 4-5. Speed-up over worker count of neural networks using GPU

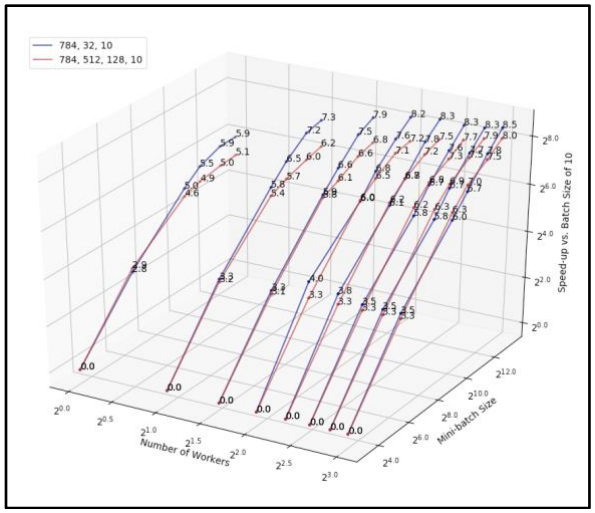


Figure 4-6. Speed-up over mini-batch size of neural networks using GPU

4.3 Benchmark results of the OclWorkEngine class

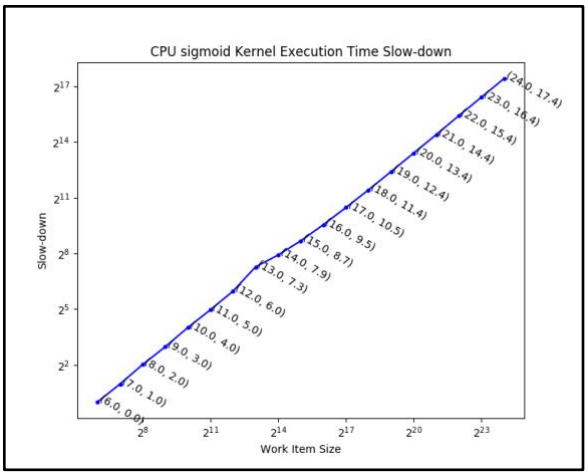
Figure 4-7 to Figure 4-11 covers the timing results for the sequential implementation of compute kernels. Figure 4-7 presents the results for 1-D kernels *sigmoid*, *sigmoid_prime* and *vec_mult* executed on the CPU. Figure 4-8 presents the results for 2-D kernels without loops. Note that for 2-D kernels without loops, the data size is the row size multiplied by the column size. The slow-down is calculated as

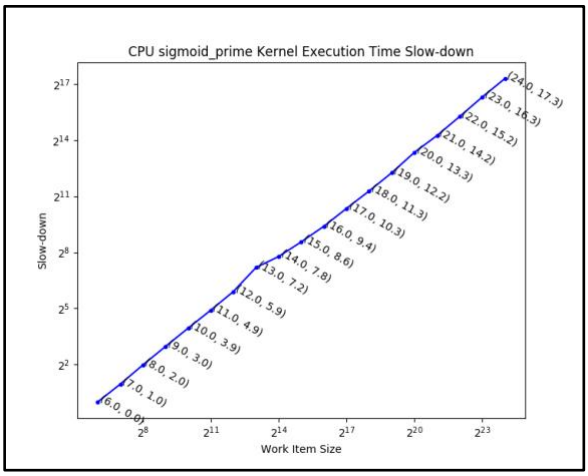
$$\text{Slow-down over data size} = \frac{\text{Execution time under current data size}}{\text{Execution time under minimum data size}}$$

Figure 4-9 to Figure 4-11 presents the results for the *mat_mult* compute kernel, the execution time is plotted as a function of both the outer loop and the inner loop. Same as for the timing results of network model training, the scaling on each dimension is included as well (Figure 4-10 and Figure 4-11). Because the *mat_mult_col* and *mat_mult_row* compute kernels have almost identical functionality and timing behavior as *mat_mult*, the duplicate results are excluded.

The results for the OpenCL implementation of compute kernels are included in Figure 4-12 through Figure 4-16 in the same way as for the sequential implementation.

All plots are demonstrated in log-log scale. Note that the dimensions for *mat_vec_add*, *mat_trans* and *mat_mult* are reduced by 1, and only the average values of data points that share the same data-size and/or internal loop size are shown. Error bars are presented as red stripes crossing each corresponding data point as the standard deviation.





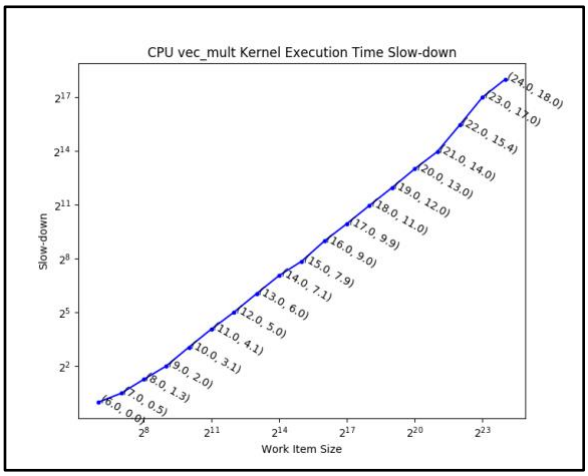
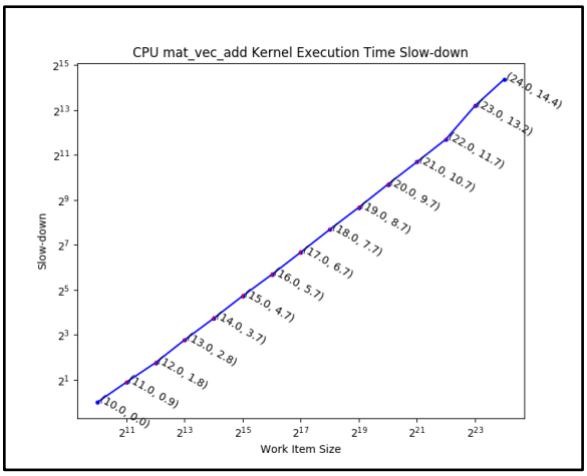


Figure 4-7. Slow-down over work-item size of 1-D compute kernels using CPU



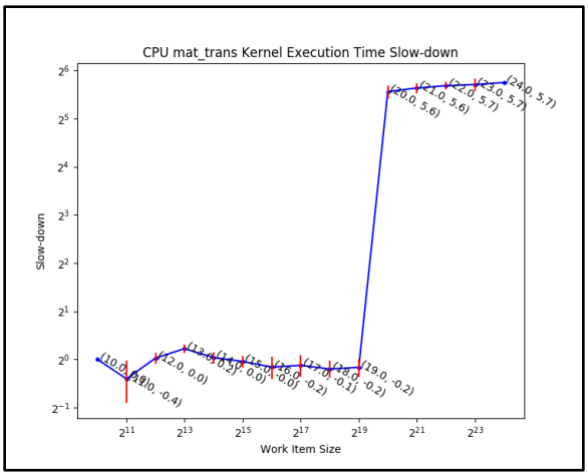


Figure 4-8. Slow-down over work-item size of 2-D compute kernels (without loops) using CPU

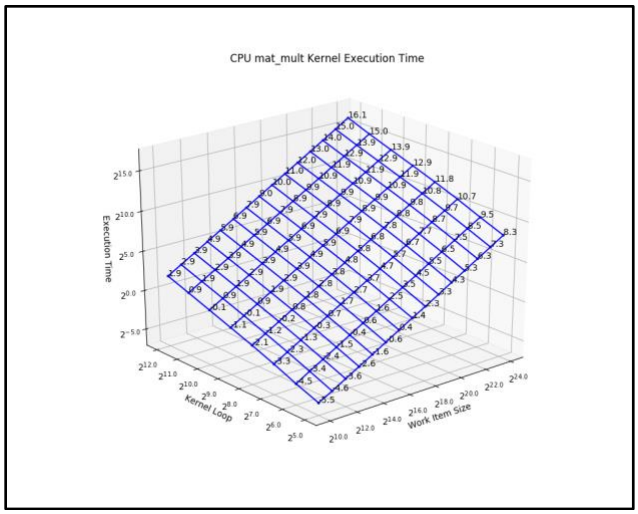


Figure 4-9. Execution time of mat_mult compute kernel using CPU

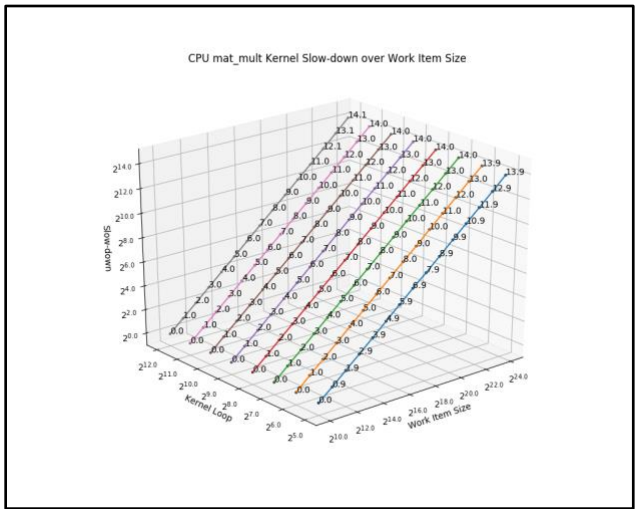


Figure 4-10. Execution time slow-down of mat_mult compute kernel over outer-loop using CPU

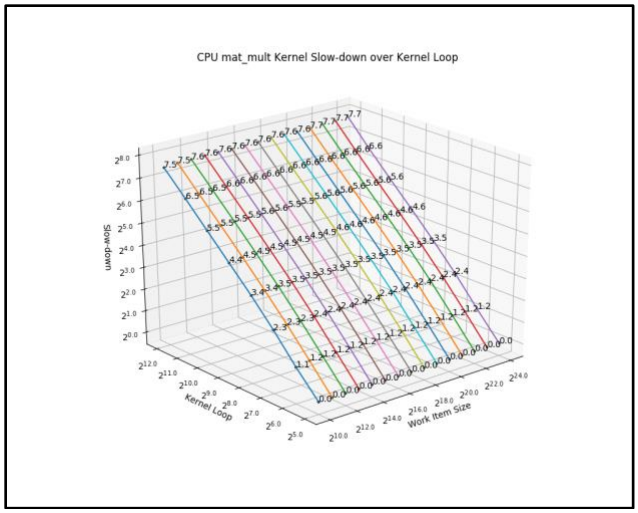
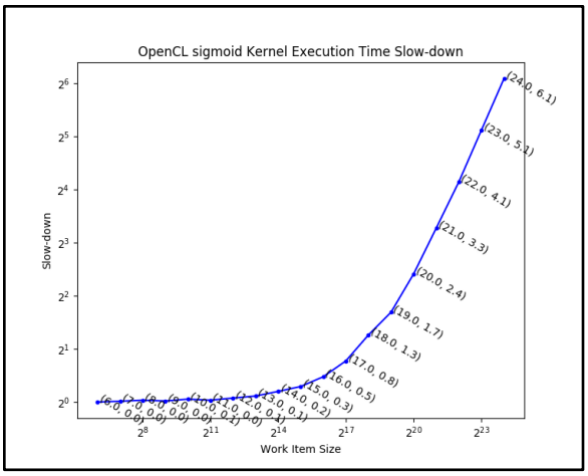
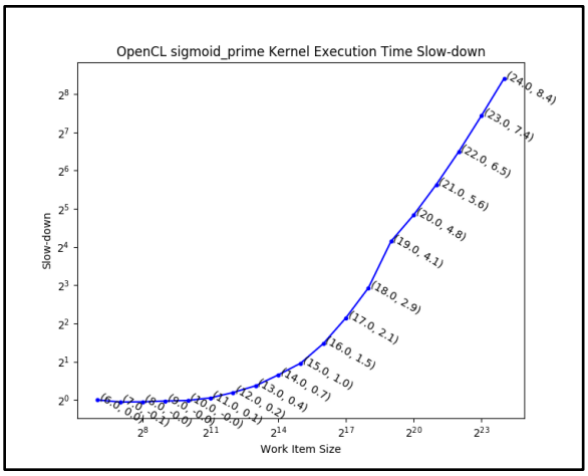


Figure 4-11. Execution time slow-down of mat_mult compute kernel over inner-loop using CPU





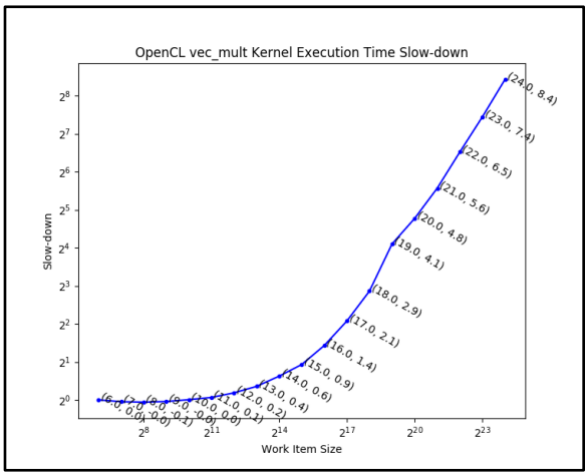
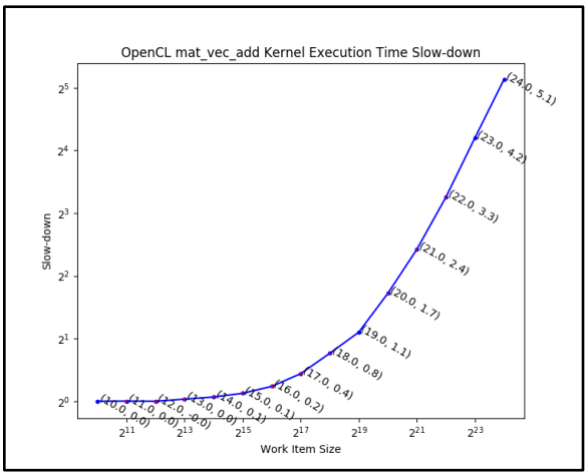


Figure 4-12. Slow-down over work-item size of 1-D compute kernels using GPU



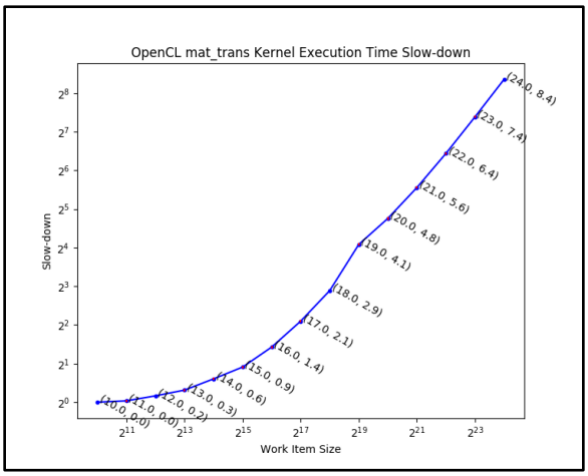


Figure 4-13. Slow-down over work-item size of 2-D compute kernels (without loops) using GPU

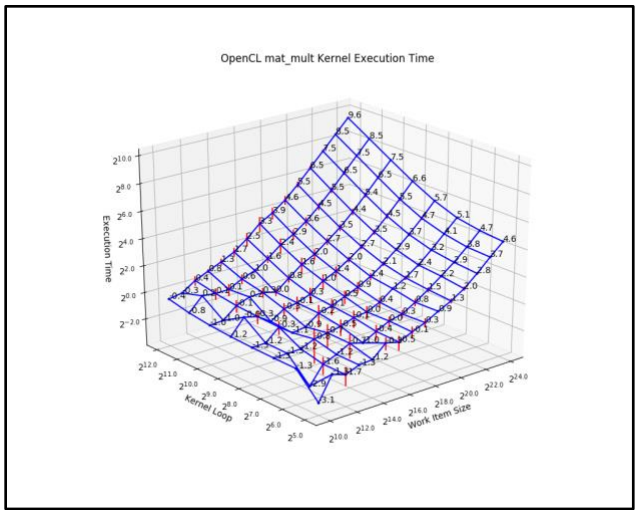


Figure 4-14. Execution time of mat_mult compute kernel using GPU

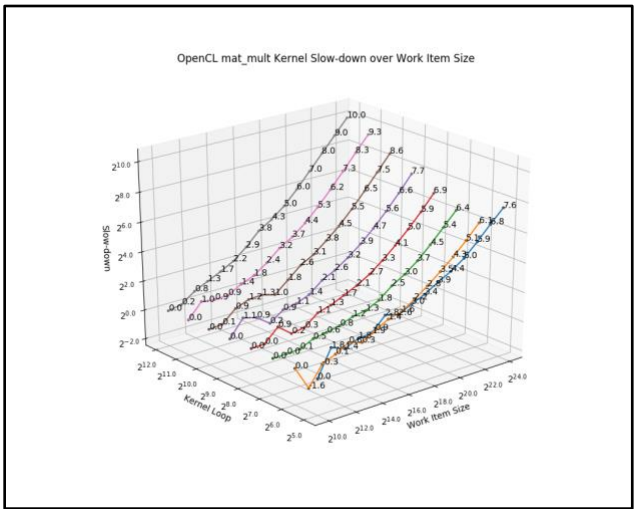


Figure 4-15. Execution time slow-down of mat_mult compute kernel over work-item using GPU

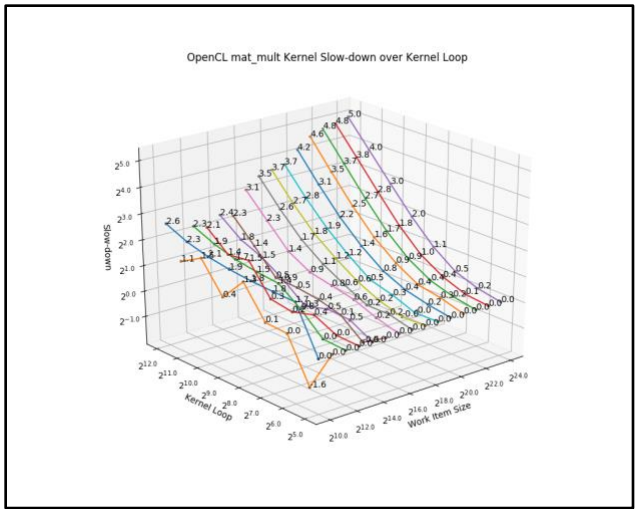


Figure 4-16. Execution time slow-down of mat_mult compute kernel over kernel loop using GPU

4.4 Benchmark results of MPI operations

The results for *MPI_Bcast* are presented from Figure 4-17 to Figure 4-31, covering all PPN configurations. For each PPN value, the execution time is a function of both the number of processes and the data size. Then the execution time slow-down over each of the two dimensions would be

$$\text{Slow-down over process count} = \frac{\text{Execution time under current process count}}{\text{Execution time under minimum process count}}$$

for each data size, and

$$\text{Slow-down over data size} = \frac{\text{Execution time under current data size}}{\text{Execution time under minimum data size}}$$

for each process count.

The results for *MPI_Reduce* are presented in the same fashion from Figure 4-32 to Figure 4-46.

All plots are demonstrated in log-log scale.

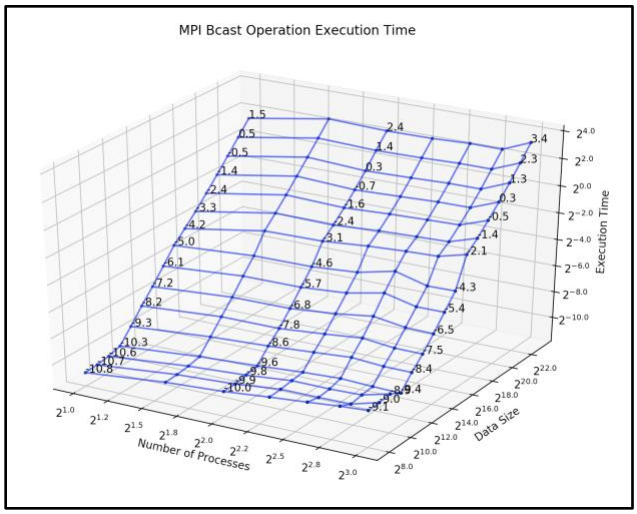


Figure 4-17. Execution time of MPI_Bcast over processes and data size (PPN=1)

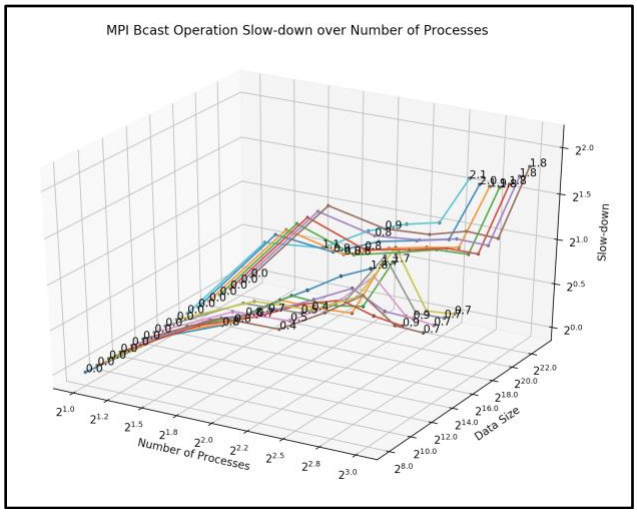


Figure 4-18. Slow-down of MPI_Bcast over processes (PPN=1)

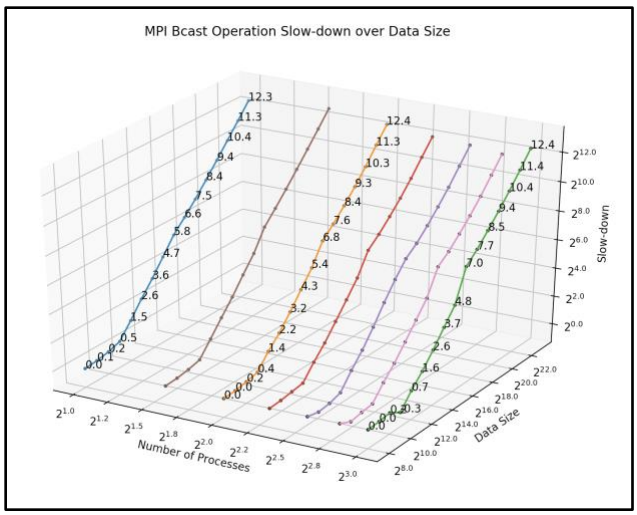


Figure 4-19. Slow-down of MPI_Bcast over data size (PPN=1)

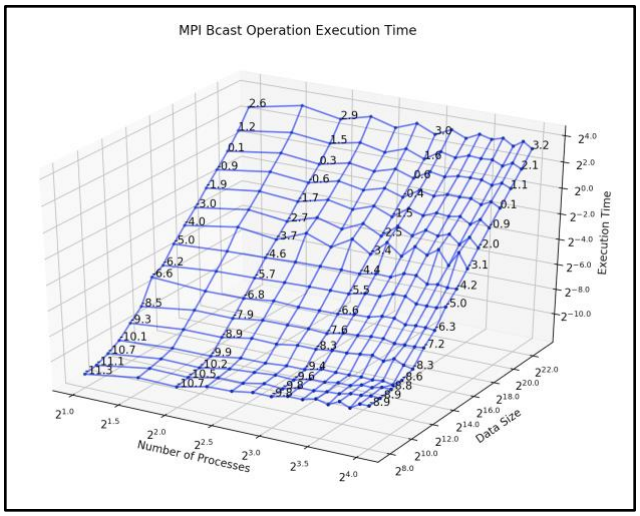


Figure 4-20. Execution time of MPI_Bcast over processes and data size (PPN=2)

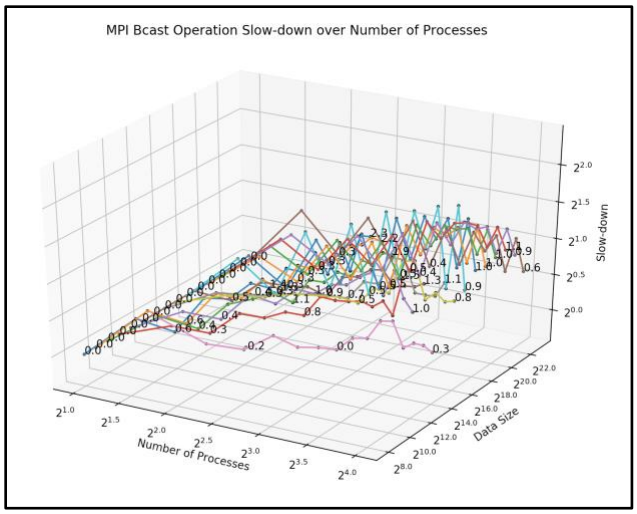


Figure 4-21. Slow-down of MPI_Bcast over processes (PPN=2)

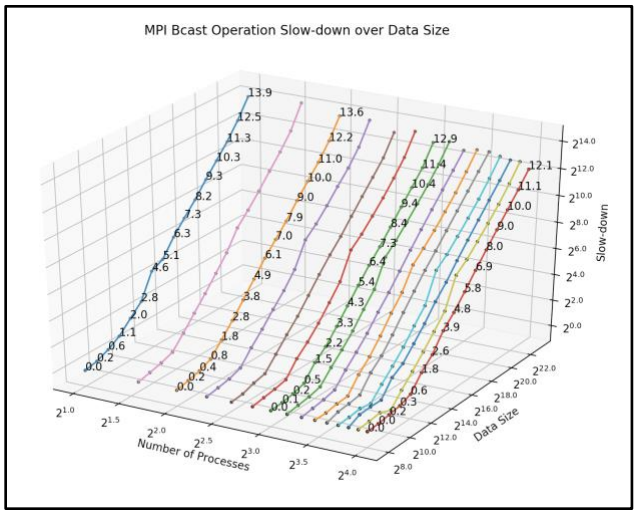


Figure 4-22. Slow-down of MPI_Bcast over data size (PPN=2)

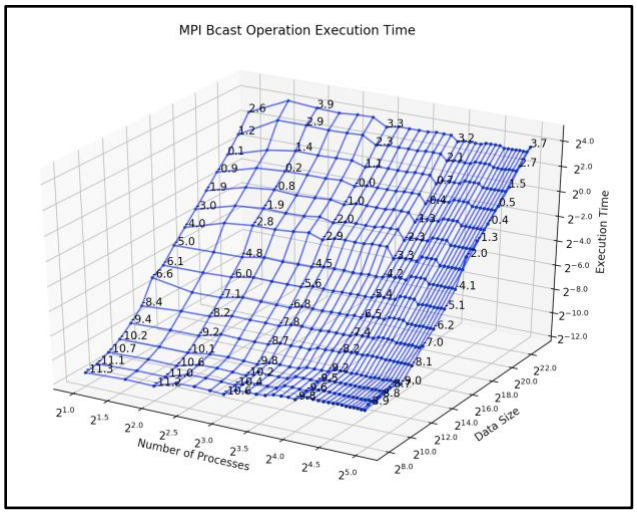


Figure 4-23. Execution time of MPI_Bcast over processes and data size (PPN=4)

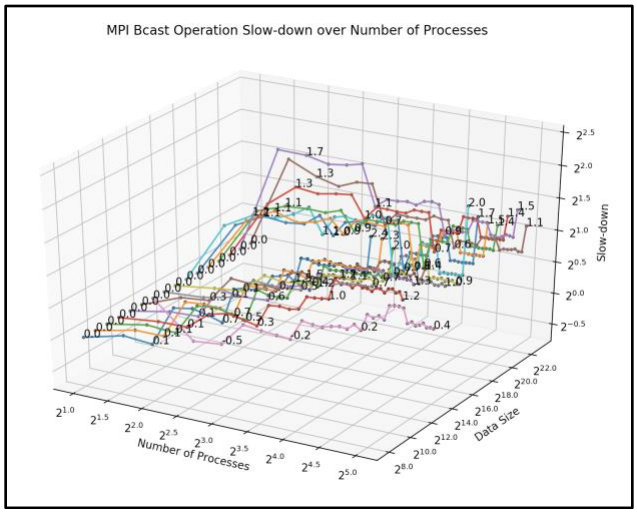


Figure 4-24. Slow-down of MPI_Bcast over processes (PPN=4)

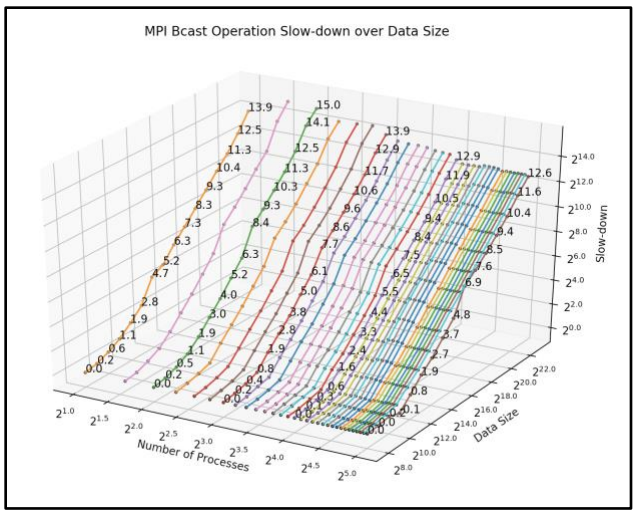


Figure 4-25. Slow-down of MPI_Bcast over data size (PPN=4)

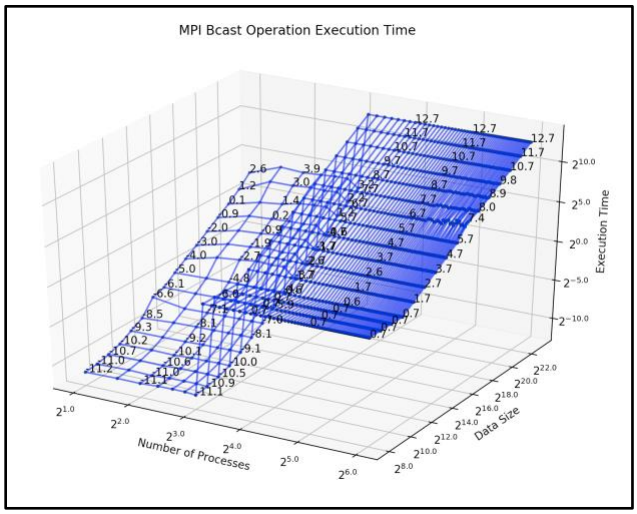


Figure 4-26. Execution time of MPI_Bcast over processes and data size (PPN=8)

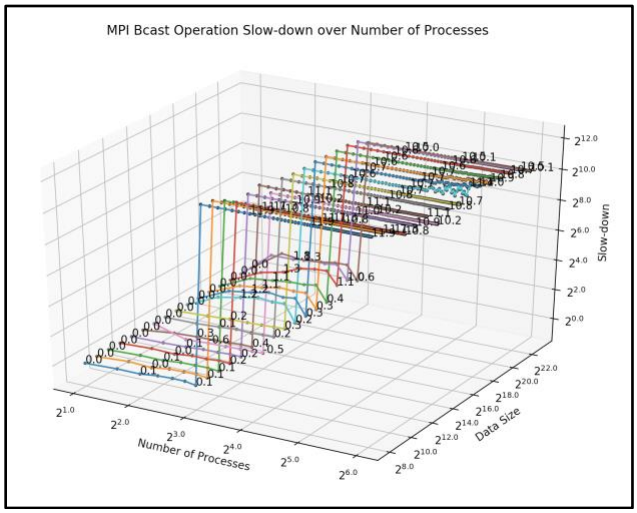


Figure 4-27. Slow-down of MPI_Bcast over processes (PPN=8)

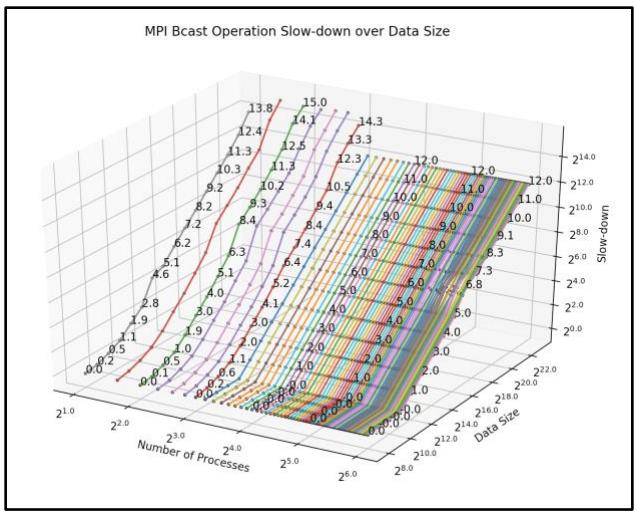


Figure 4-28. Slow-down of MPI_Bcast over data size (PPN=8)

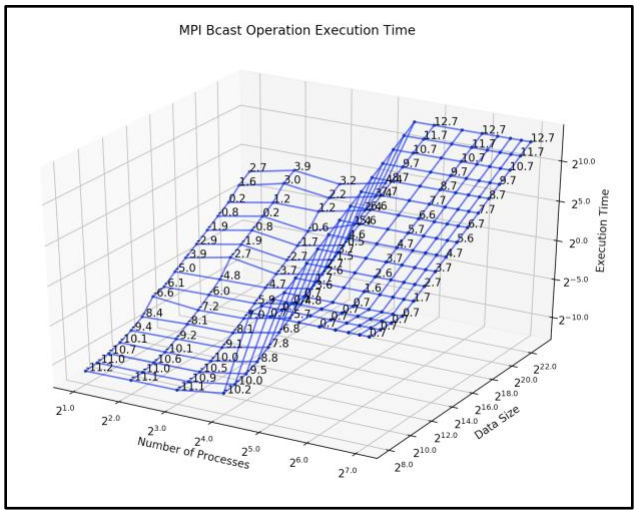


Figure 4-29. Execution time of MPI_Bcast over processes and data size (PPN=16)

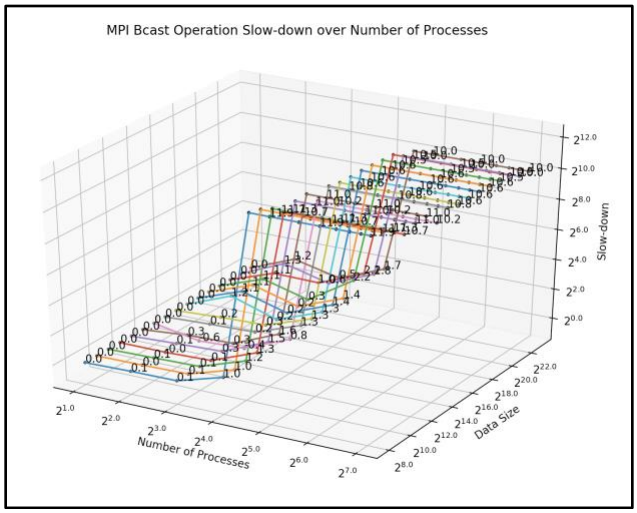


Figure 4-30. Slow-down of MPI_Bcast over processes (PPN=16)

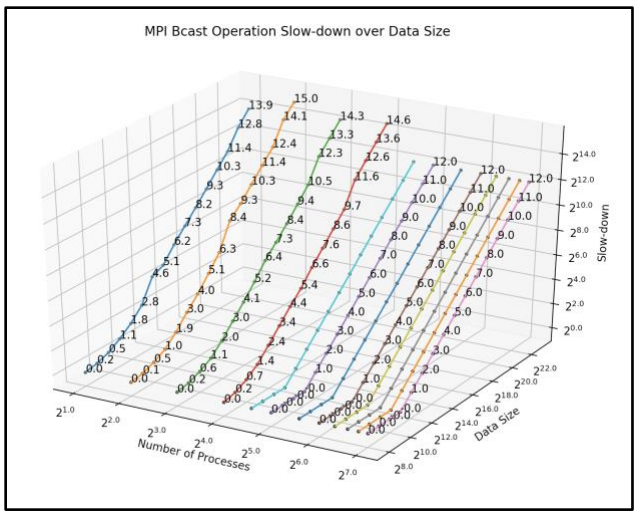


Figure 4-31. Slow-down of MPI_Bcast over data size (PPN=16)

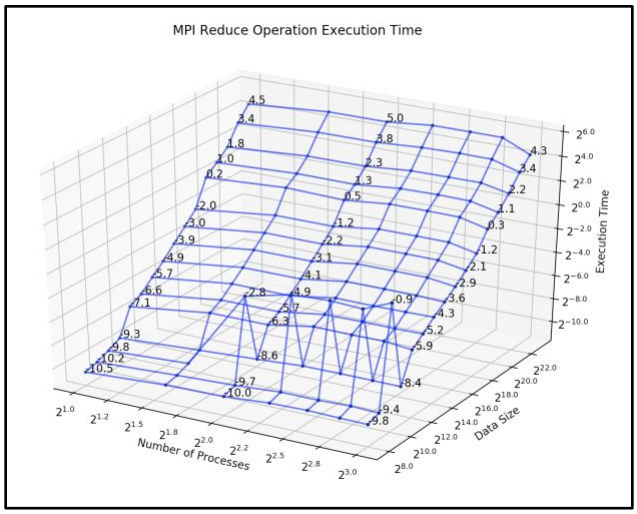


Figure 4-32. Execution time of MPI_Reduce over processes and data size (PPN=1)

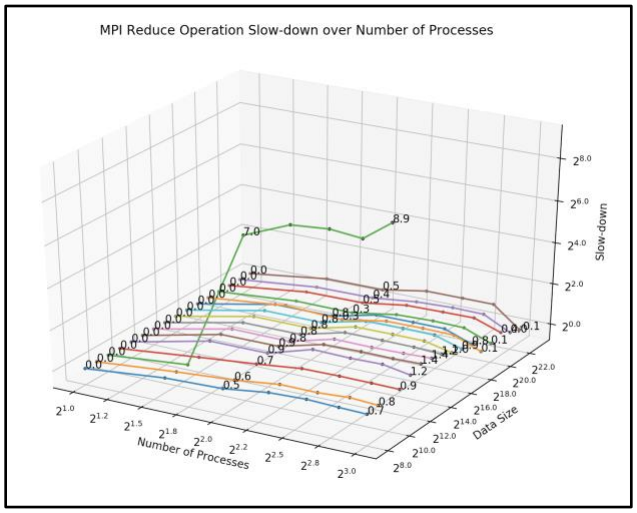


Figure 4-33. Slow-down of MPI_Reduce over processes (PPN=1)

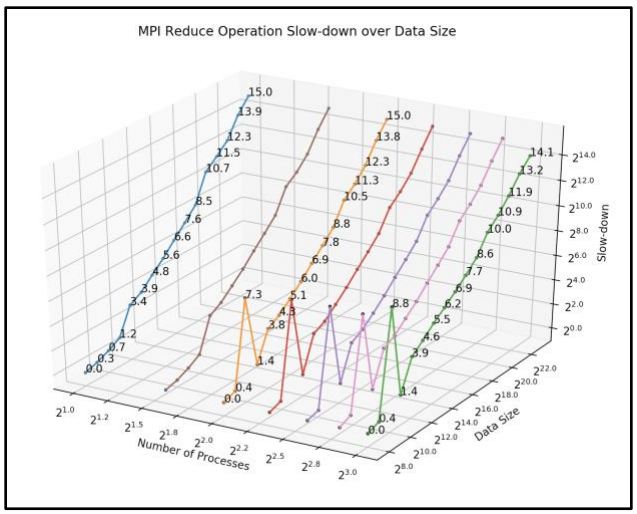


Figure 4-34. Slow-down of MPI_Reduce over data size (PPN=1)

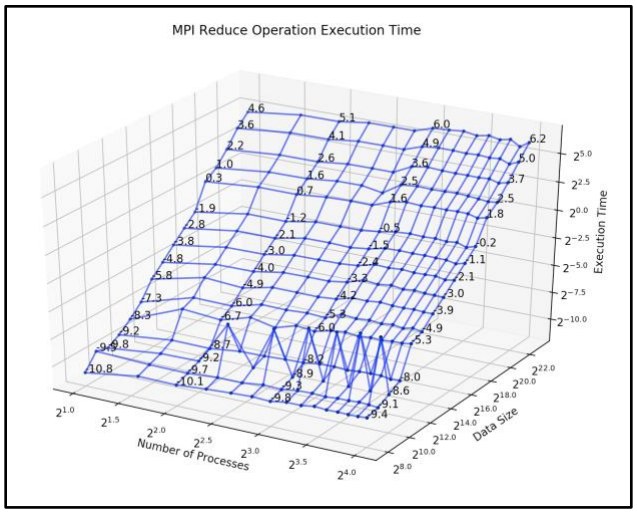


Figure 4-35. Execution time of MPI_Reduce over processes and data size (PPN=2)

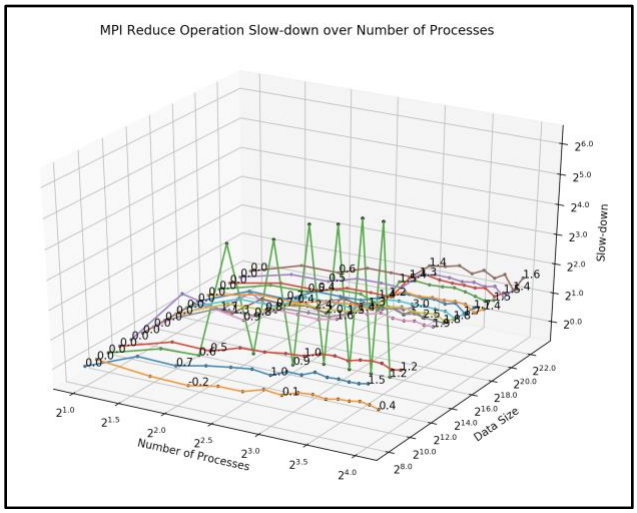


Figure 4-36. Slow-down of MPI_Reduce over processes (PPN=2)

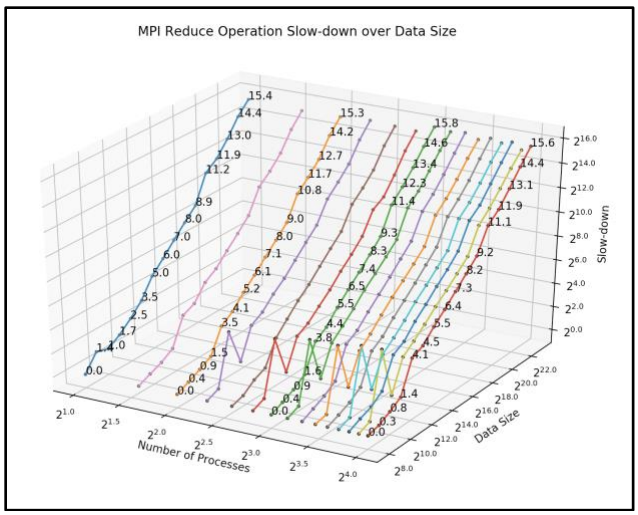


Figure 4-37. Slow-down of MPI_Reduce over data size (PPN=2)

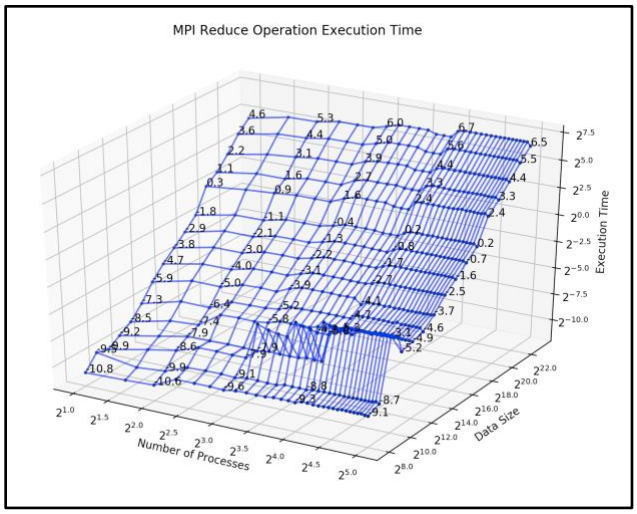


Figure 4-38. Execution time of MPI_Reduce over processes and data size (PPN=4)

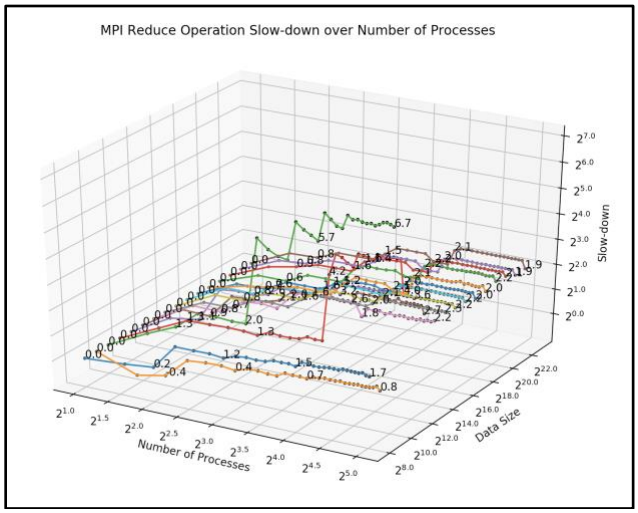


Figure 4-39. Slow-down of MPI_Reduce over processes (PPN=4)

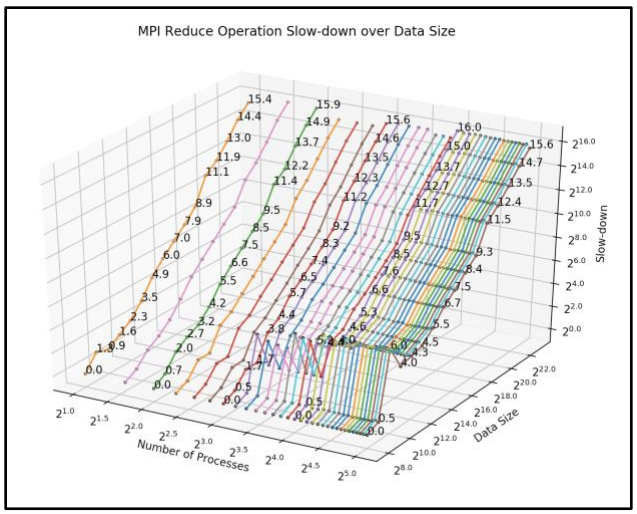


Figure 4-40. Slow-down of MPI_Reduce over data size (PPN=4)

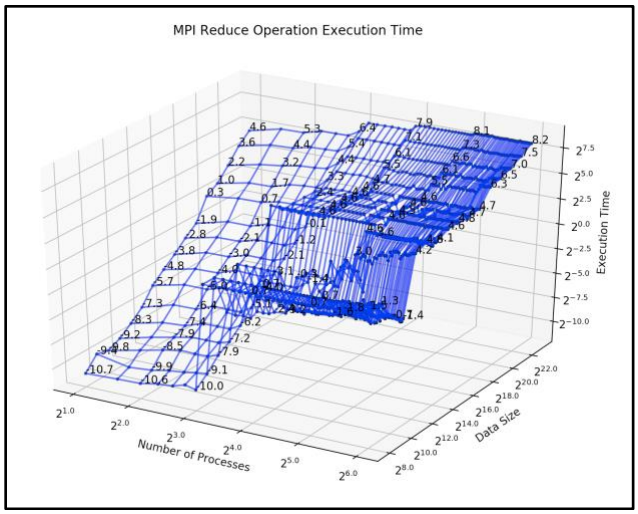


Figure 4-41. Execution time of MPI_Reduce over processes and data size (PPN=8)

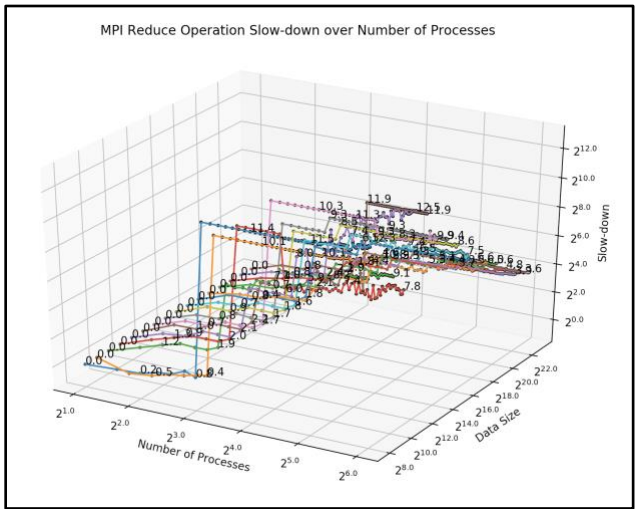


Figure 4-42. Slow-down of MPI_Reduce over processes (PPN=8)

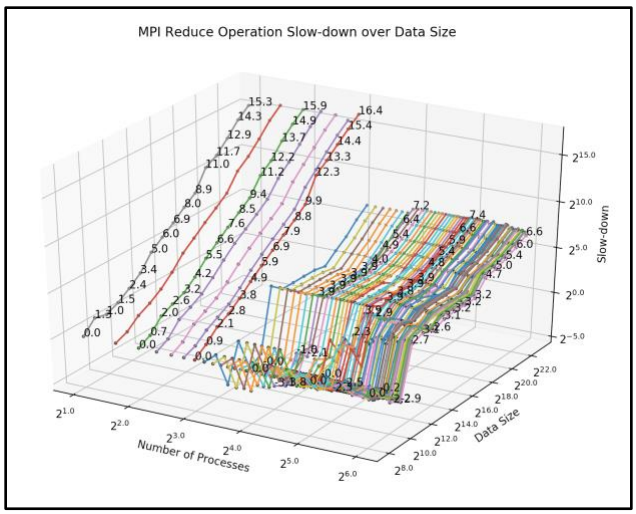


Figure 4-43. Slow-down of MPI_Reduce over data size (PPN=8)

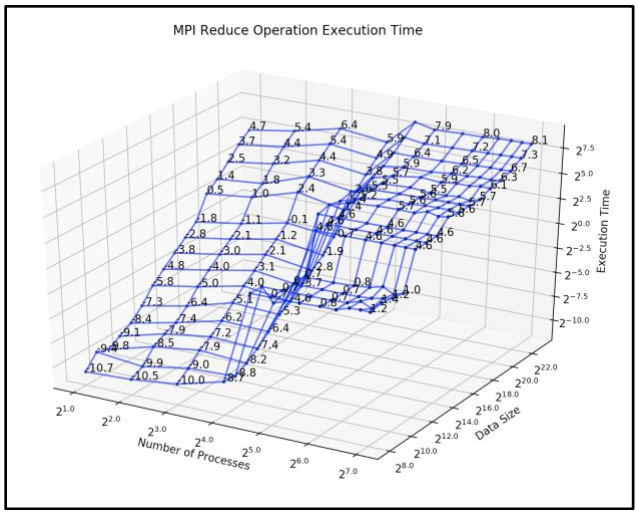


Figure 4-44. Execution time of MPI_Reduce over processes and data size (PPN=16)

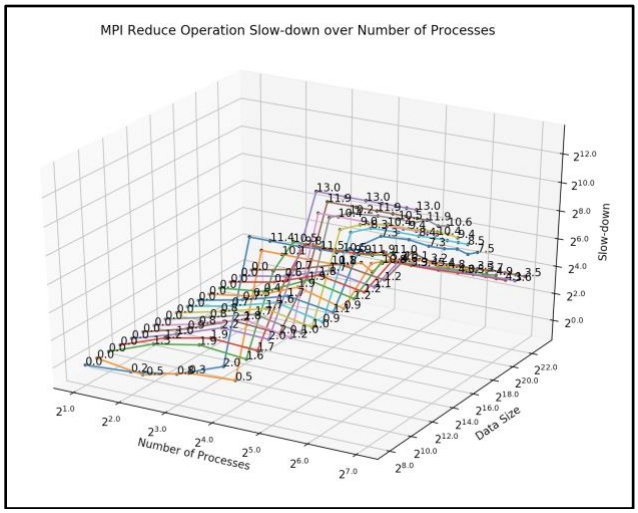


Figure 4-45. Slow-down of MPI_Reduce over processes (PPN=16)

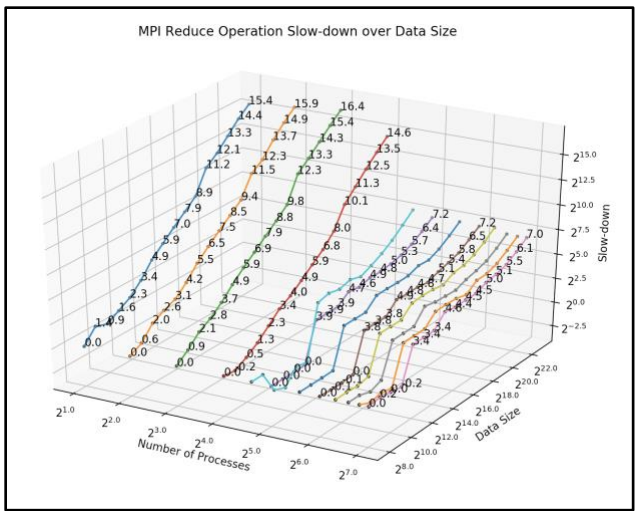


Figure 4-46. Slow-down of MPI_Reduce over data size (PPN=16)

Chapter 5 – Discussion

5.1 Modeling the training epoch throughput: a bottom-up approach

In order to understand the scaling of training neural networks, a model for the throughput of the training epoch must be generated first. This requires modeling the execution time spent on the computation-intensive portions in a single epoch, namely the compute kernels and the MPI operations. In this chapter, a bottom-up approach is presented by first fitting the benchmark timing results of compute kernels and MPI operations, and then using these timing models to derive the training epoch throughput. The final throughput model outputs prediction plots resembling Figure 4-1 to Figure 4-6. Using these timing and throughput models (i.e. compute kernels, MPI operations and neural network training epoch), the effects of various hardware aspects on scaling the training process can be predicted.

5.2 Defining fit error

Root-mean-square error (RMSE) is a common method for computing the fit error. The RMSE over a dataset of N data points $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ is:

$$RMSE = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} \big(y_i - f(x_i)\big)^2}$$

One drawback of using RMSE to calculate the error is that large-valued points would be weighted significantly more than the small-valued points. Instead, the symmetric mean absolute percentage error (SMAPE) given by

$$SMAPE = \frac{1}{N} \cdot \sum_{i=1}^{N} \frac{|f(x_i) - y_i|}{|f(x_i)| + |y_i|}$$

is used in order to get a fair representation across all data points [30].

5.3 Execution time of compute kernels: an analytical model

Now that the benchmark results are available, it is necessary to update the definition of 1-D compute kernels versus 2-D compute kernels. Besides the *sigmoid*, *sigmoid_prime* and *vec_mult* compute kernels, the *mat_vec_add* kernel has the same timing characteristics as a 1-D compute kernels for both the sequential implementation and the OpenCL implementation, if the data size (for sequential) or work-item size (for OpenCL) is defined as the product of row size and column size. The reason this transformation works for the sequential implementation is rather obvious—essentially, the 2-level nested loop is unrolled as a simple 1-level loop. On the other hand, the reason this also works for the OpenCL implementation is that regardless of the work dimension, the scheduling policy treats work-items equally for 1-D and 2-D indexing for simple kernels.

For the rest of the analysis, details on the *mat_trans* kernel are excluded because in terms of time complexity, it is an order of magnitude less expensive than the compute kernels that invoke it. Nonetheless, a note on the odd shape demonstrated in Figure 4-8 where the execution time experiences a harsh step-up: this is because of the data size reaches the cache size limit and eventually gets penalized by higher memory latencies.

The Python implementation for modeling the compute kernels is included in "python/plot_oclengine_test.py".

5.3.1 Sequential implementation

The benchmark results from Figure 4-7 to Figure 4-11 for the sequential implementation presents obvious linear relationships in loglog scale, indicating a polynomial growth of execution time over data size. Ideally, the time complexity of any sequential implementation should be *O(n)* where n denotes the total number of iterations, yet various overheads such as memory

accesses resulting cache miss cause the actual complexity being not strictly linear for some compute kernels. When fitting the growth function for the sequential implementation, the order of the polynomial growth function is computed under the loglog scale, and then transform back to normal scale. Hence, the growth function is computed as

$$F(x) = y_{\min} \cdot \left(\frac{x}{x_{\min}}\right)^{\frac{\log_2 y_{\max} - \log_2 y_{\min}}{\log_2 x_{\max} - \log_2 x_{\min}}}$$

based on the smallest data point $(x_{\min}, y_{\min})$ and the largest data point $(x_{\max}, y_{\max})$. Figure 5-1 shows the model fitting 1-D sequential compute kernels. The growth function model is marked in solid red line, and the fit error is included on the top left corner in each sub-figure. Figure 5-2 shows the model fitting the *mat_mult* sequential compute kernel, including the fit error.

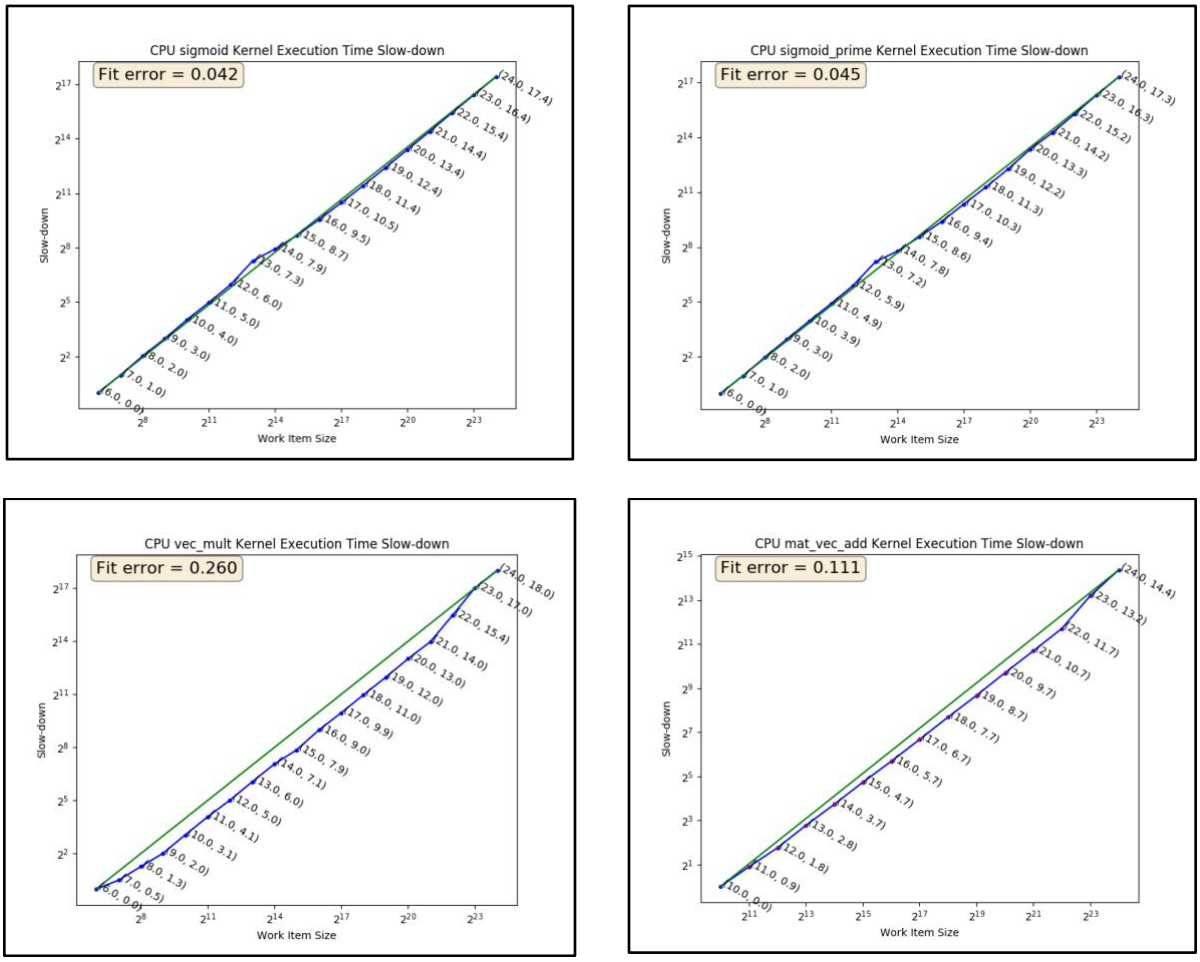


Figure 5-1. Fitting sequential 1-D compute kernels (CPU)

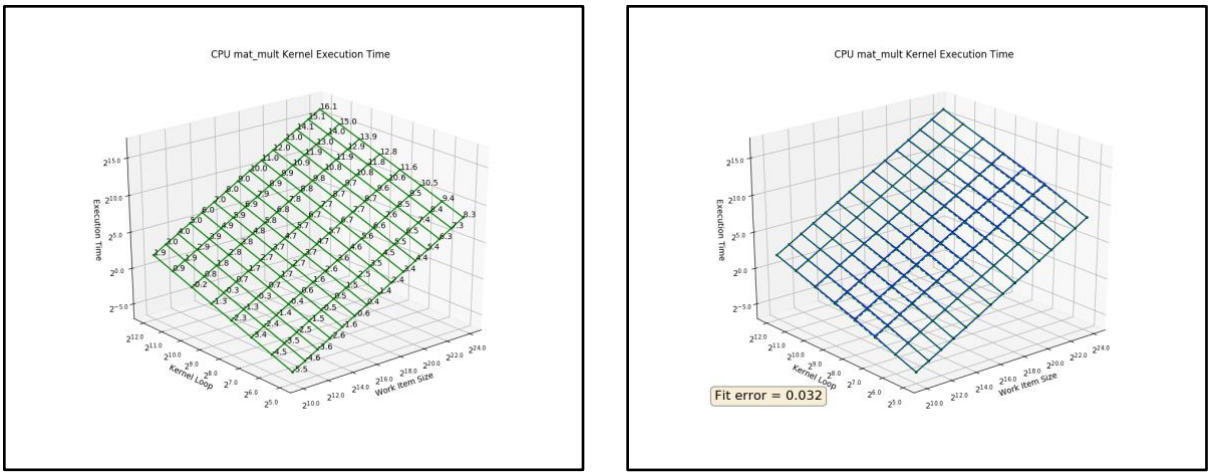


Figure 5-2. Fitting sequential mat_mult kernel (CPU)

5.3.2 OpenCL implementation

There are 3 steps in order for an OpenCL kernel to execute on the GPU or any other compute device:

1. *Memory creation*: creates the memory objects associated with the variables for the input and output, and passing/transferring the data from the host to the compute device. In modern computers, data transfer is typically done through the PCI-Express bus lane.
2. *Enqueue for execution*: executes the enqueued kernel on the compute device.
3. *Copy-back*: transfers the result data from the compute device back to the host, which usually also utilizes the PCI-Express bus lane.

For the memory transfer steps 1 and 3, there are basically two portions of time cost: a fixed set-up time cost and a variable time cost depending on the transfer data size. The function to fit the execution time for the memory transfer steps is

$$F(x) = \frac{y_{\max} - y_{\min}}{x_{\max} - x_{\min}} \cdot (x - x_{\min}) + y_{\min}$$

based on the smallest data point $(x_{\min}, y_{\min})$ and the largest data point $(x_{\max}, y_{\max})$.

Memory creation

Both the OpenCL event profiling results and the time recorded by the *NanoTimer* class on the host are included. Figure 5-3 and Figure 5-4 present the memory transfer timing model fitting the memory creation step, against the profiling results and the host timer results, respectively. In general, *Host Time = OpenCL Profiling Time + Overhead*.

The host timer results are of the most interest because they provide a total representation of the time needed to run the task. However, using the host results may potentially include noises and overhead from various sources. Therefore, the fitting errors from both are covered so as to check if the overhead has a significant distortion to the overall timing.

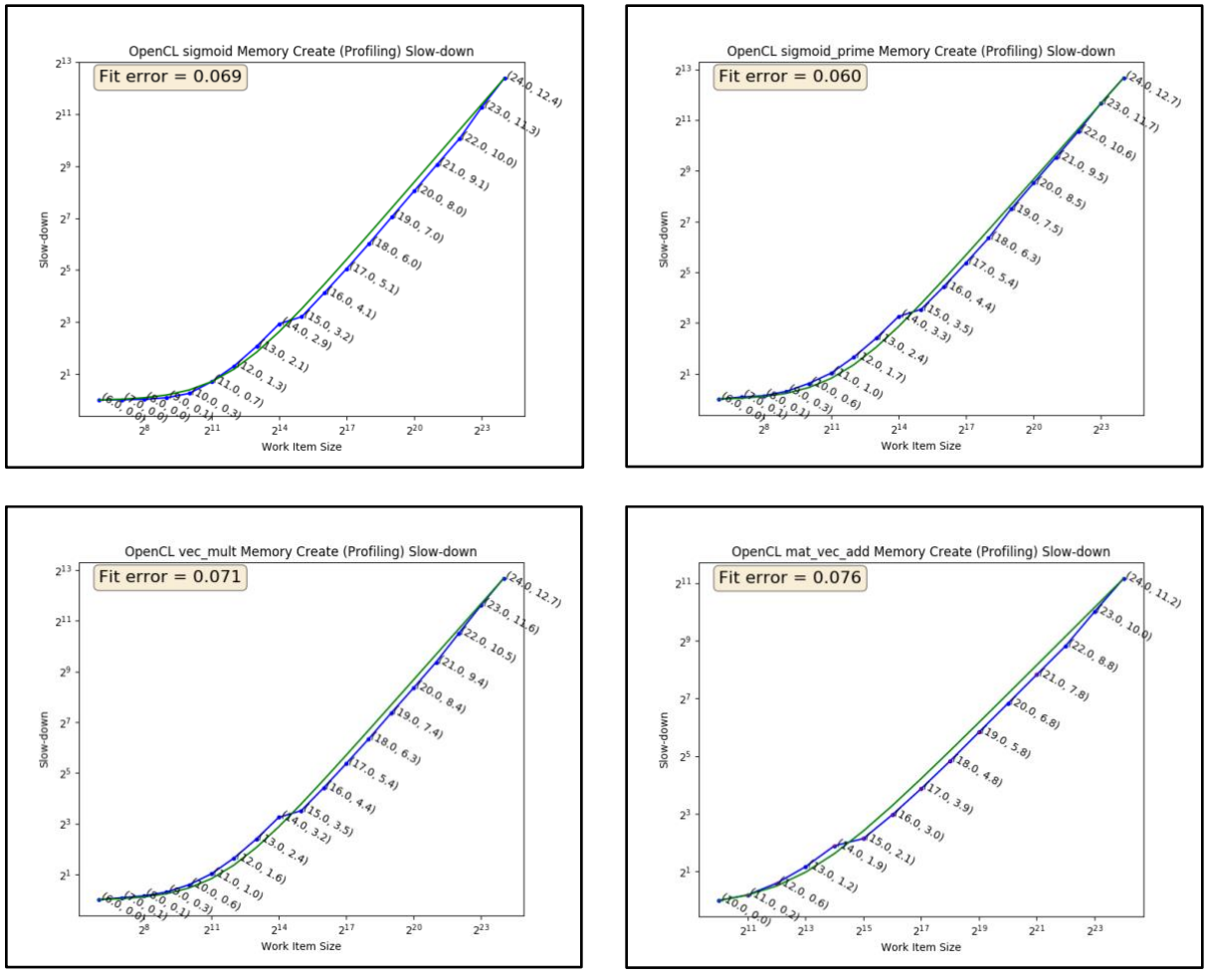


Figure 5-3. Fitting 1-D OpenCL kernels (create memory objects, OpenCL profiling)

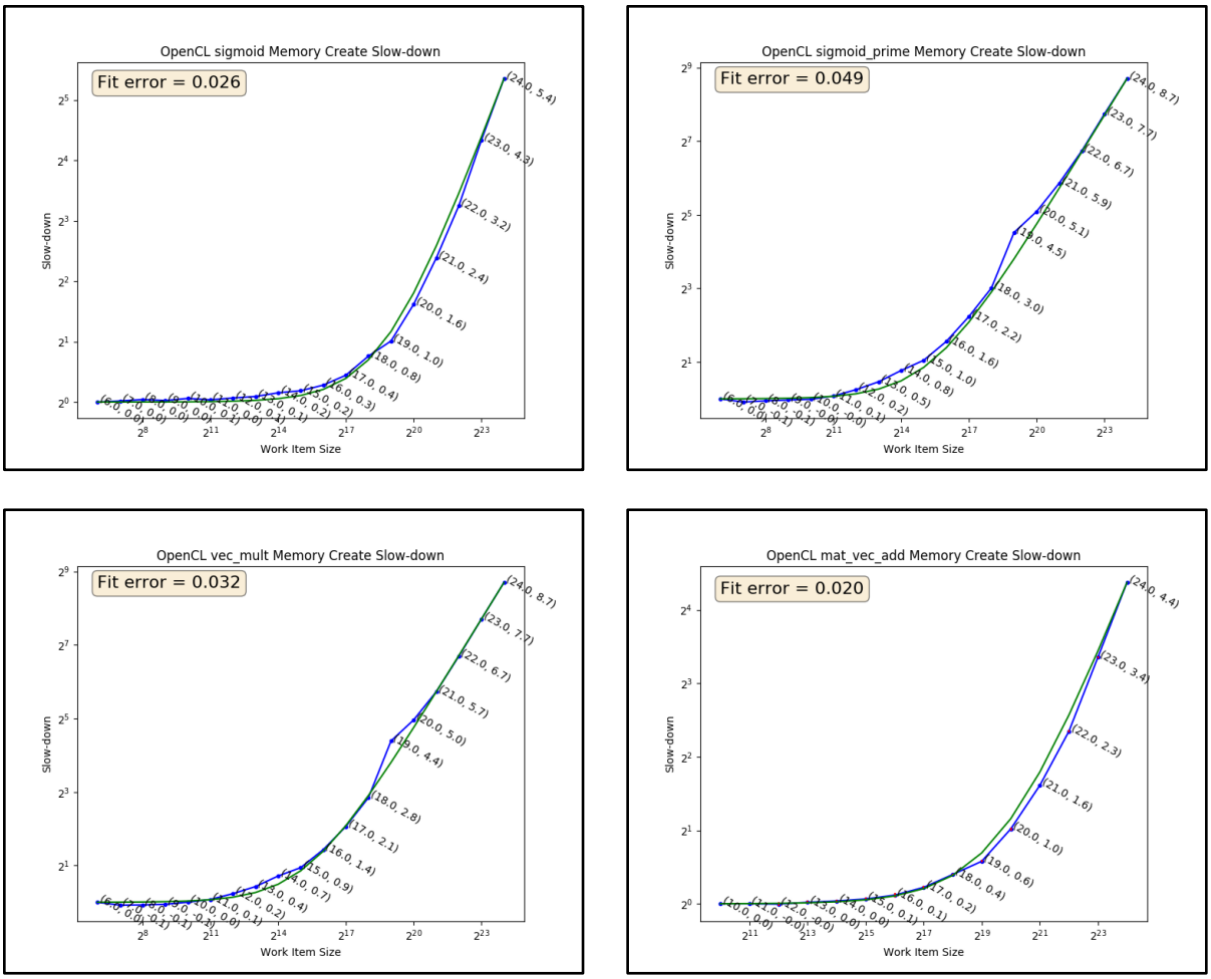


Figure 5-4. Fitting 1-D OpenCL kernels (create memory objects, host timer)

The fit errors from both the OpenCL profiling results and the host timer results are relatively low, suggesting a good fit and non-disturbing overhead (e.g. constant cost or proportional to the theoretical cost).

Copy-back

Figure 5-5 and Figure 5-6 present the memory transfer timing model fitting the copy-back step, against the profiling results and the host timer results, respectively.

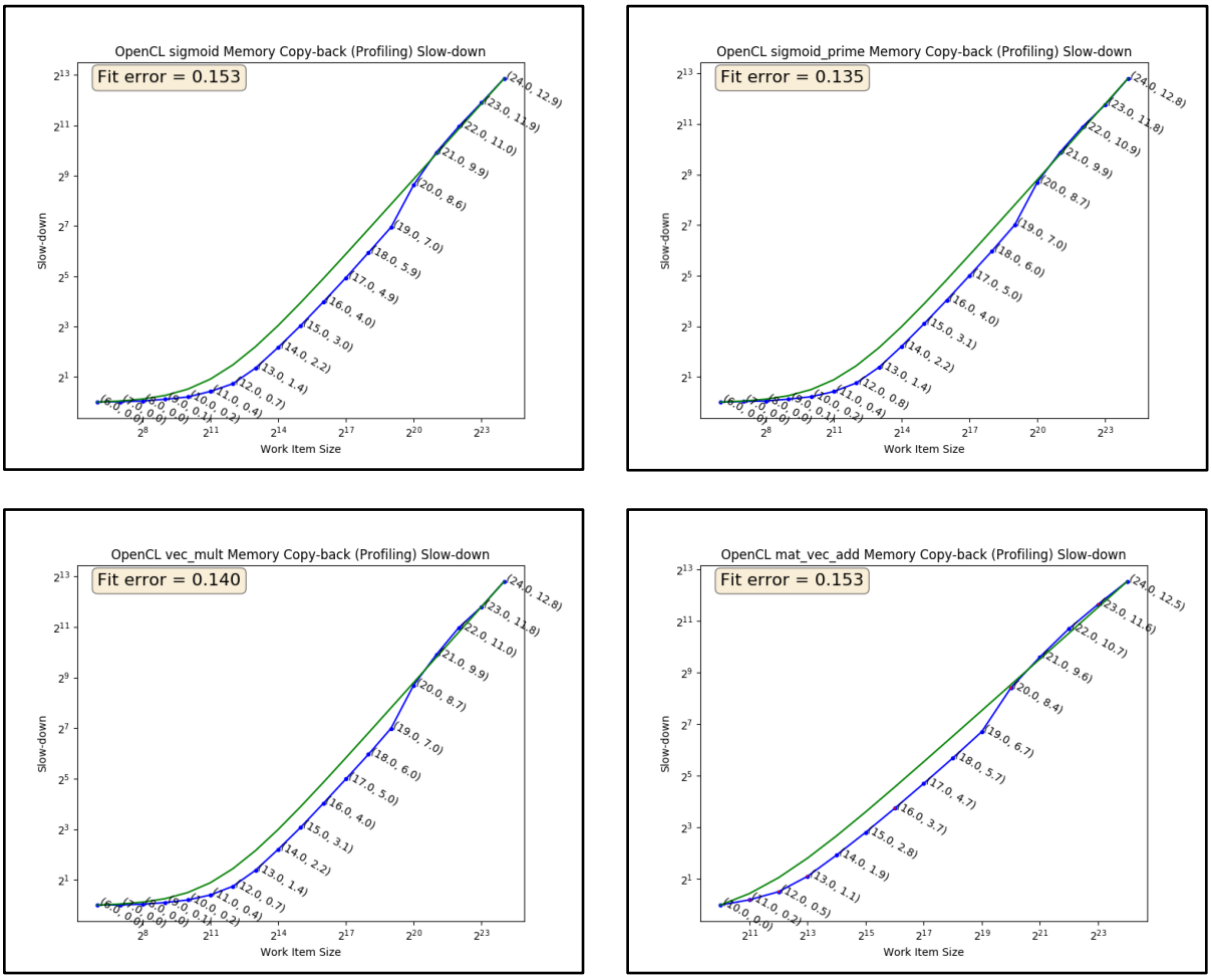


Figure 5-5. Fitting 1-D OpenCL kernels (copy back memory, OpenCL profiling)

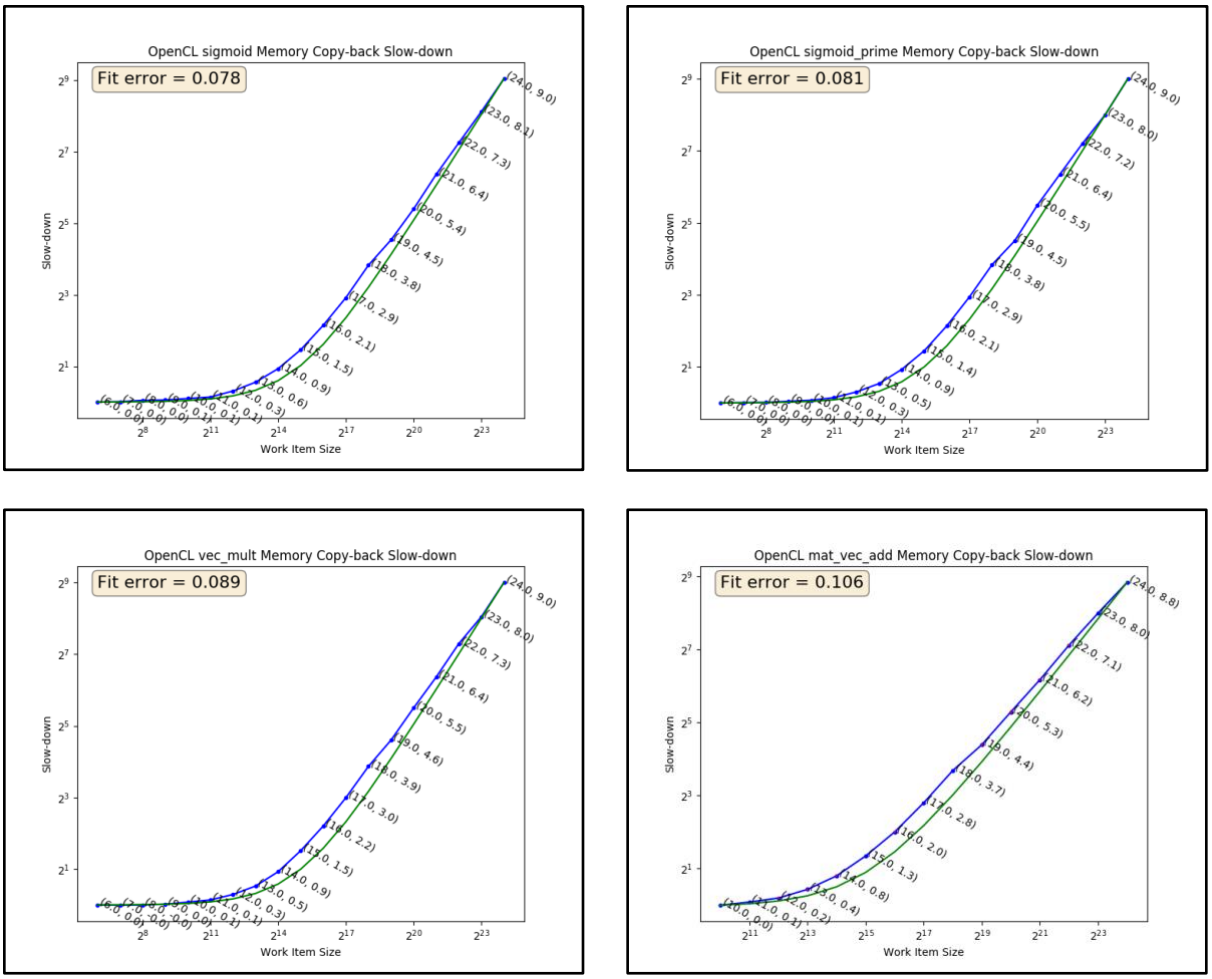


Figure 5-6. Fitting 1-D OpenCL kernels (copy back memory, host timer)

Enqueue (kernel) for execution

OpenCL kernels are executed as work-items. Each work-item is associated with a workgroup, and the work-items in a workgroup execute concurrently. An OpenCL compute device supports a number of compute units stacked with processing elements (e.g. vector units in a GPU core), and each workgroup is associated with one compute unit. Meanwhile, each compute unit may run only one workgroup at a time. As a result, if the number of work-items is less than the maximum number of processing elements available, the theoretical execution time of the enqueued kernel should be identical regardless of the work-item size, because it only takes each occupied processing element to finish one instance of a kernel. Table 5-1 lists the compute device specification details retrieved from OpenCL profiling.

OpenCL device	NVIDIA Corporation, Tesla P100-PCIE-12GB
Max workgroup size	1024
Max clock frequency (MHz)	1328
Global memory size (bytes)	12786073600
Local memory size (bytes)	49152
Max memory allocation size (bytes)	3196518400
Max compute units	56
Max work item dimensions	3
Max work item sizes	1024, 1024, 64

Table 5-1. OpenCL profile information of GPU used in benchmarks

Normally, the max workgroup size of a kernel may be smaller than that of the device if the kernel is complex and uses significant amounts of local or private memory. However, since the kernels implemented are simple enough, the max workgroup size (i.e. maximum number of work-items in a workgroup) of both the device and the kernels is 1024. Hence, for work-item size of less than $56 \times 1024 = 57344 \approx 2^{15.8}$, the execution time should be almost constant. However, this is not always true because the fast context-switching capability of the GPU allows workgroups to inter-leave the compute unit which hides the memory latency. Therefore, it is likely that the GPU can handle more work-items than the processing element count before

significantly increasing the execution time. After the point where the constant region ends, the GPU essentially functions as a CPU with a massive core count, having a close to linear time complexity when the data size increases exponentially. Eventually, the growth of the execution time over very large data size becomes same as a sequential implementation, with a constant speed-up. The growth function should thus be a linear function in the form of

$$F(x) = \begin{cases} \text{Base Time Cost}, & x \leq x_0 \\ \text{Variable Time Cost} \cdot (x - x_0) + \text{Base Time Cost}, & x > x_0 \end{cases}$$

where x_0 denotes the end of the constant region. Solving x_0 directly is counter-intuitive since the obscure behavior has been mentioned as a side-effect of optimization on the GPU. However, note that if 2 data points are picked, with $(x_{\max}, y_{\max})$ in the near-sequential region where the data size is significantly larger than x_0, and $(x_{\min}, y_{\min})$ in the constant region yet close to x_0, then

$$\text{Variable Time Cost} \approx \frac{y_{\max} - y_{\min}}{x_{\max} - x_{\min}}$$

Recall that the function to fit the execution time for the memory transfer steps is

$$F(x) = \frac{y_{\max} - y_{\min}}{x_{\max} - x_{\min}} \cdot (x - x_{\min}) + y_{\min}$$

Even better, the same function for the enqueue for execution step can be applied again! For example, to apply this approximation technique, estimate

$$x_0 \approx \text{Max Workgroup Size} \times \text{Compute Unit Count}$$

then set

$$x_{\min} = \frac{1}{8} \cdot x_0$$

and

$$x_{\max} = 128 \cdot x_0$$

Figure 5-7 and Figure 5-8 present the model fitting the enqueue execution step, against the OpenCL profiling results and the host timer results, respectively. Relatively small fit errors gained from both results indicate a successful model.

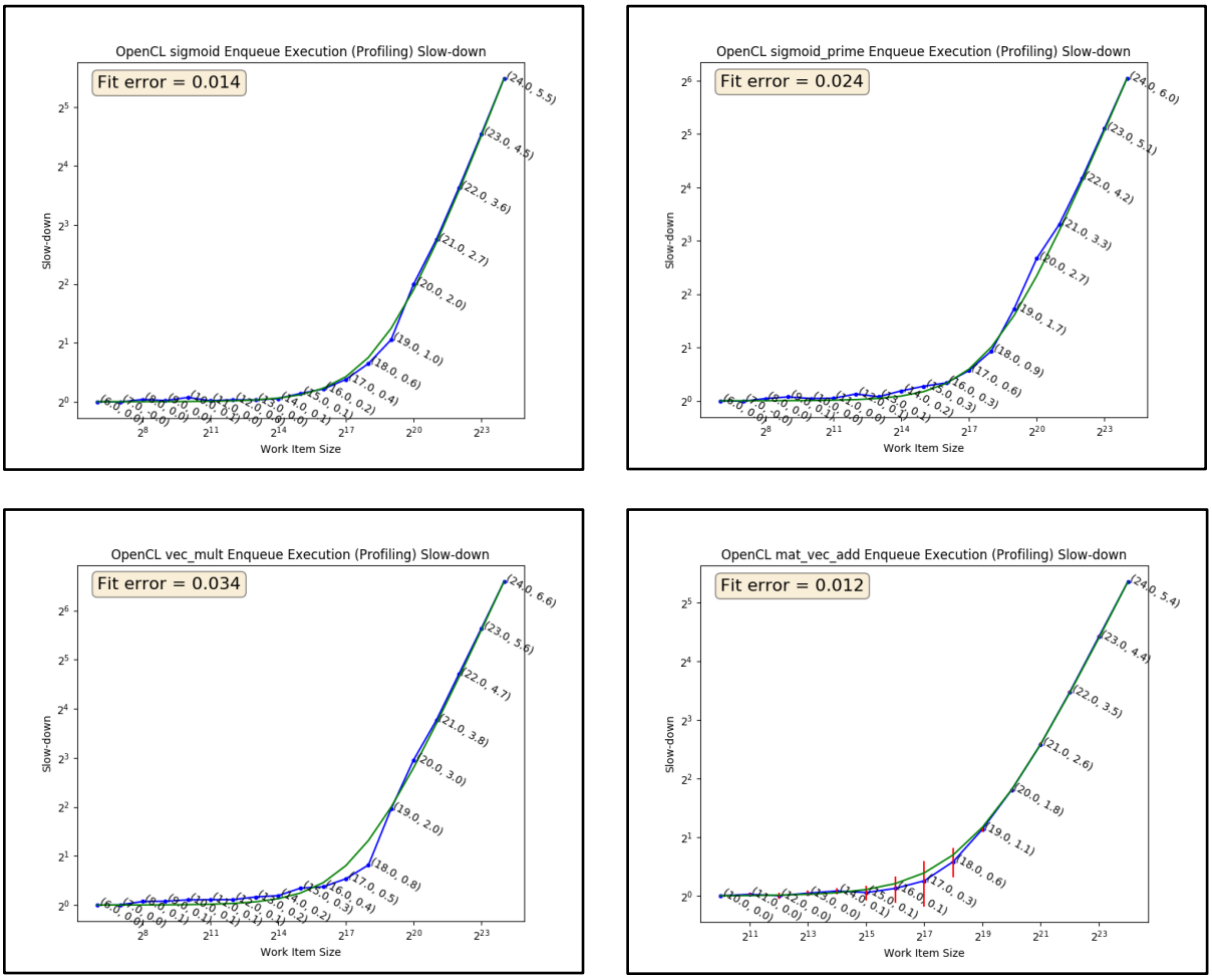


Figure 5-7. Fitting 1-D OpenCL kernels (kernel enqueue execution, OpenCL profiling)

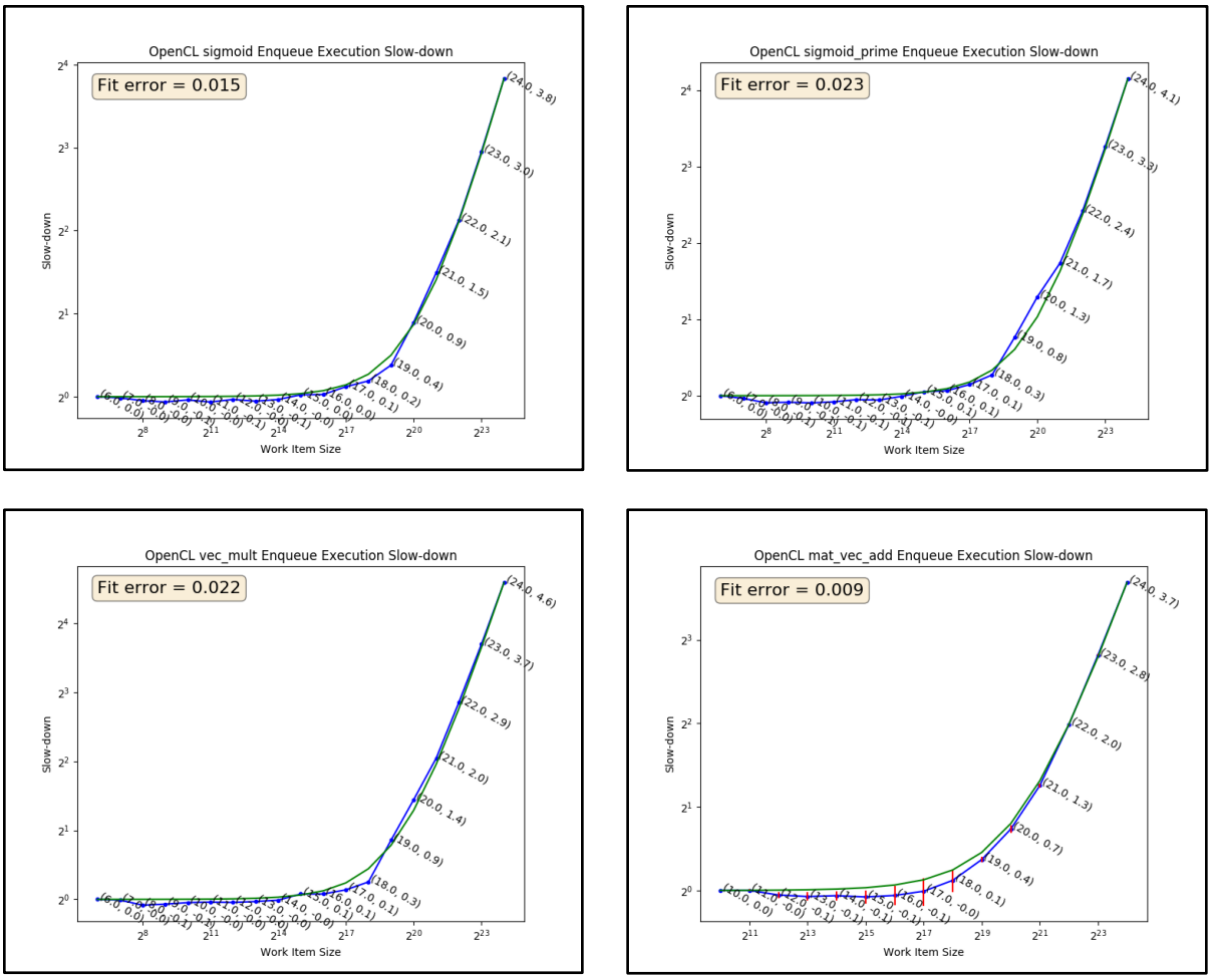


Figure 5-8. Fitting 1-D OpenCL kernels (kernel enqueue execution, host timer)

Combined

At this point, all 3 steps of running an OpenCL kernel have been covered. Since the models of all steps use a same underlying linear function, simply apply the same model to the combined overall execution time results. Figure 5-9 demonstrates excellent fitting results with reasonably low fit errors, mostly under 5%.

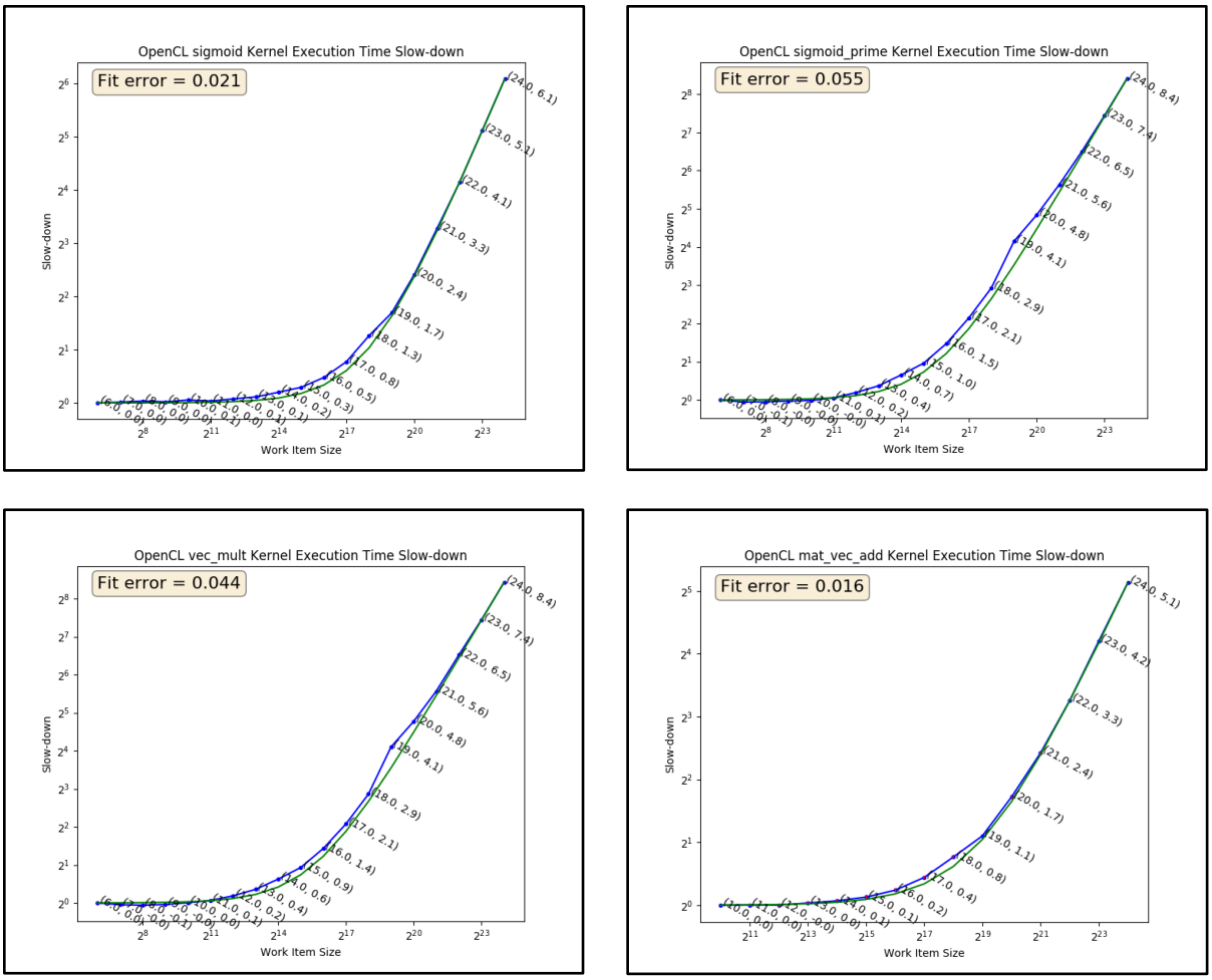


Figure 5-9. Fitting 1-D OpenCL kernels (combined, host timer)

For future references, the growth function is categorized into three characteristic regions:

1. *The perfect-parallel region*, where the number of work items is less than the maximum parallelizable size. In the perfect-parallel region, it costs nearly constant time to complete all work items, and the effect of work item size is minimal.

2. *The transition region*, where the number of work items exceeds the maximum parallelizable work item size. In the transition region, the extra work items cause a non-linear and rapid increase in execution time, yet not enough to fully occupy the parallel computing device into a linear slow-down over number of work items.
3. *The linear region*, where the excessive work items result in a linear slow-down over number of work items. The parallel computing device behaves similar to a sequential device except with a constant speed-up.

The low fit errors for 1-D OpenCL kernels provides a good theory base towards the 2-D kernel *mat_mult*, hence using the linear model to fit over both the work-item size and the kernel loop size. Figure 5-10 shows the results of fitting the OpenCL implementation of the *mat_mult* compute kernel. The fit model is solely presented on the left side, and the fit error with fit model in dotted line is on the right side.

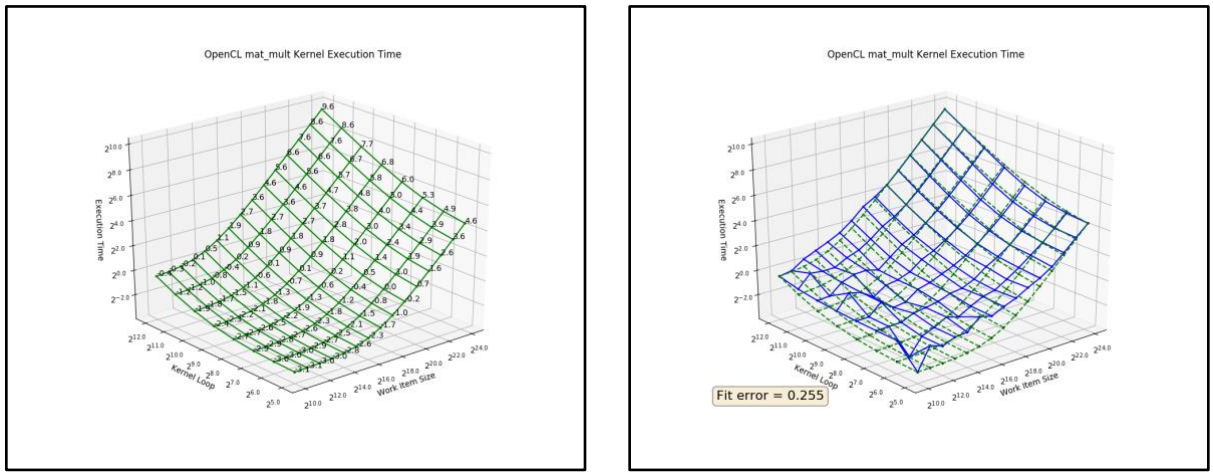


Figure 5-10. Fitting mat_mult OpenCL kernel (combined, fit error included)

An error of 25.5% is observed, which is relatively high despite all the low errors achieved so far on the 1-D kernels. What went wrong then? First of all, the outliers mostly concentrate around smaller kernel loops and work-item sizes with considerable averaging errors (see Figure

4-14). Secondly, break down the combined execution time into the 3 steps and analyze the timing results. As in Figure 5-11 to Figure 5-13, the memory creation step is causing the outliers.

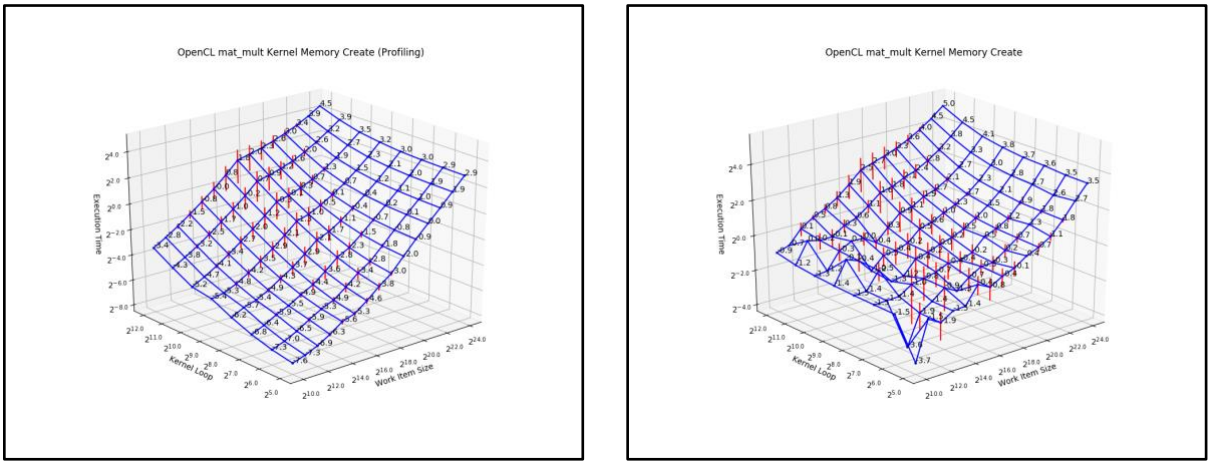


Figure 5-11. Memory creation of mat_mult OpenCL kernel, OpenCL profiling (left) and host timer (right)

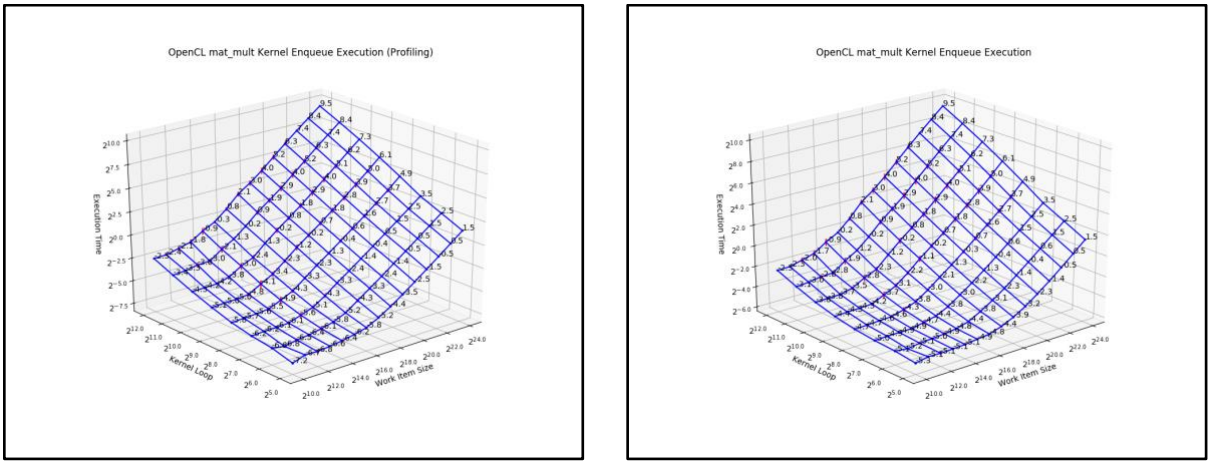


Figure 5-12. Enqueue execution of mat_mult OpenCL kernel, OpenCL profiling (left) and host timer (right)

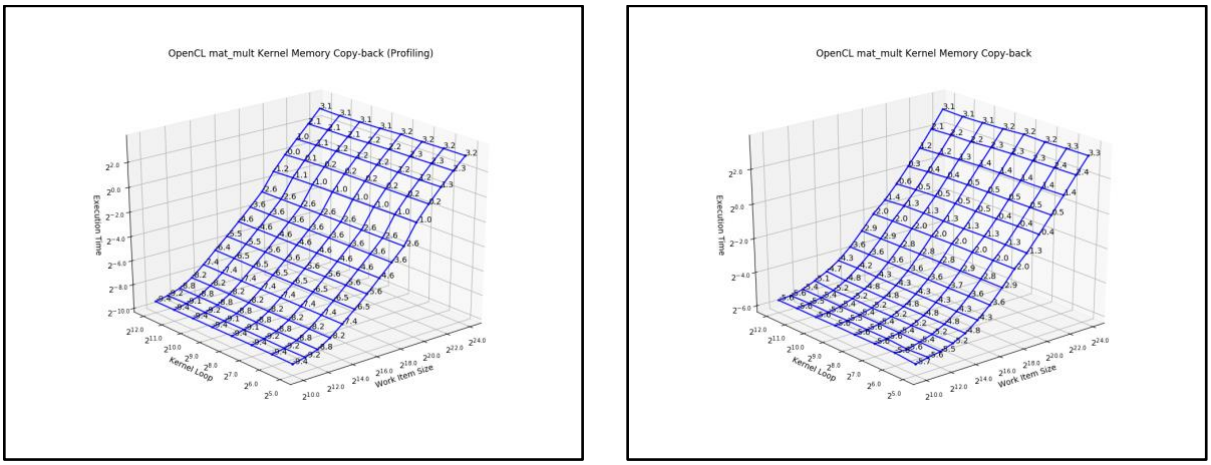


Figure 5-13. Copy-back of mat_mult OpenCL kernel, OpenCL profiling (left) and host timer (right)

In terms of memory creation, both OpenCL profiling and host timer results show significant averaging errors. This is because the memory size space complexity is *O((row_a + col_b) * row_b)* instead of *O(row_a * col_b * row_b)*. However, it is unclear why the execution time of memory creation is not independent on the kernel loop size. On the other hand, the copy-back memory transfer is independent of the kernel loop size, and does not violate the linear model. For simplicity, the linear fit model can still be applied as an acceptable approximation.

5.4 Execution time of MPI operations: an empirical model

There is not enough reference on MVAPICH's implementation of MPI to formulate the growth function using an analytical model. In this project, an empirical model is generalized using both the available documentation from the MVAPICH user manual and the MPI benchmark results as an observation [31].

Firstly, notice that the shape of execution time scaling over data size closely resembles the shape of the linear growth function in OpenCL kernels. Therefore, the scaling over data size is fit with the same linear model as

$$F_x(y) = \frac{z_{\max\to x} - z_{\min\to x}}{z_{\max\to x} - z_{\min\to x}} \cdot (y - y_{\min}) + z_{\min\to x}$$

where *z* denotes the benchmarked execution time, and *y* denotes the data size. The notation $z_{\max\to x}$ means the maximum value of *z* for a fixed value of *x*.

The scaling over number of processes is closely correlated to the number of processors per node (PPN). In the benchmark results, for PPN less than 8, the growth is a slight and evenly-distributed step-up as the process count increases. According to [31], the MPI implementation supports a "2-level point-to-point tree-based 'Knomial' algorithm" for small message passing, and includes optimizations for larger messages. The tree-based algorithm should be of *O(log N)*

time complexity where *N* denotes the number of processes. This agrees with the results for PPN less than 8. For PPN greater than or equal to 8, there is one single yet much more significant step-up immediately after the number of processes exceeds the configured PPN, making the smaller step-ups subtle. We name the boundary value of PPN that distinguishes the scaling behaviors as the *cut-off PPN*.

For PPN less than the cut-off PPN, scaling over process count is

$$F_y(x) = \begin{cases} z_{\min\to y}, & x < x_{\min} \\ z_{\min\to y} + \dfrac{\left\lceil \log_2 \dfrac{x}{x_{\min}} \right\rceil}{\left\lceil \log_2 \dfrac{x_{\max}}{x_{\min}} \right\rceil} \cdot \left(z_{\max\to y} - z_{\min\to y}\right), & x \geq x_{\min} \end{cases}$$

And for PPN greater than or equal to the cut-off PPN, because the smaller step-ups are omitted,

$$F_y(x) = \begin{cases} z_{\min\to y}, & x \leq \text{PPN} \\ z_{\max\to y}, & x > \text{PPN} \end{cases}$$

where x denotes the number of processes.

Figure 5-14 to Figure 5-18 present the fitting results of the *MPI_Bcast* operation for all benchmarked PPN values. Figure 5-19 to Figure 5-23 present the fitting results of the *MPI_Reduce* operation.

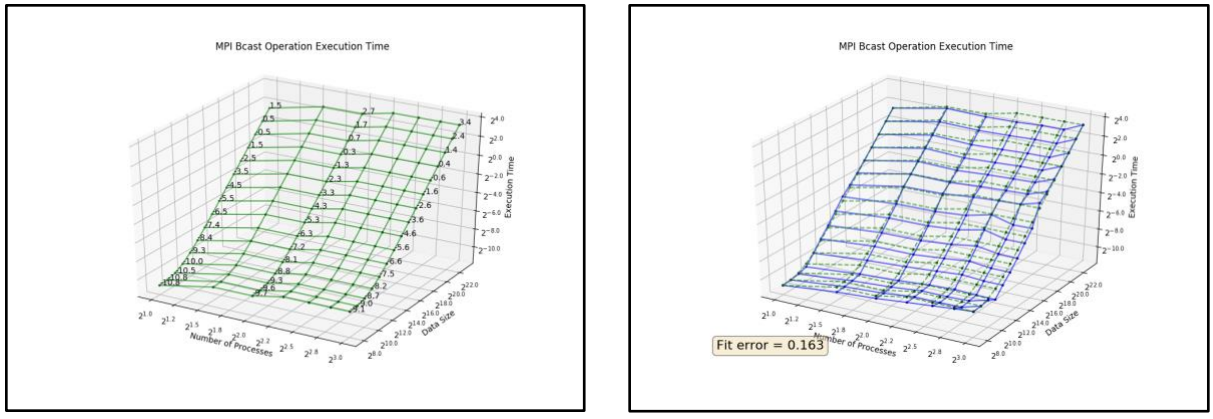


Figure 5-14. Fitting execution time of MPI_Bcast over processes and data size (PPN=1)

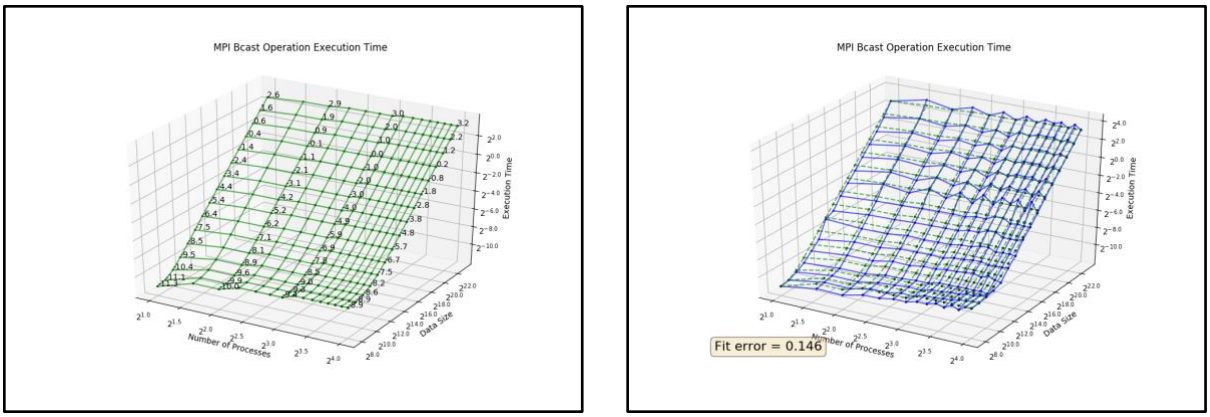


Figure 5-15. Fitting execution time of MPI_Bcast over processes and data size (PPN=2)

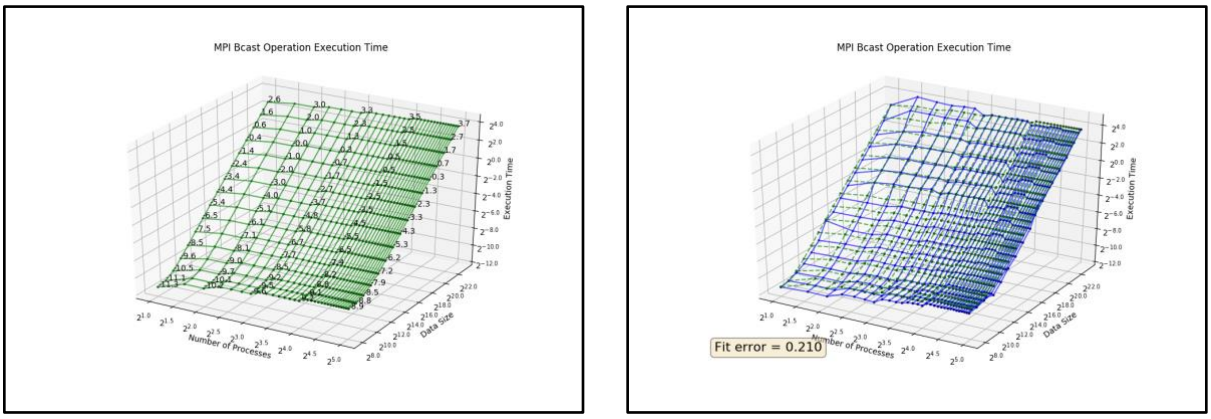


Figure 5-16. Fitting execution time of MPI_Bcast over processes and data size (PPN=4)

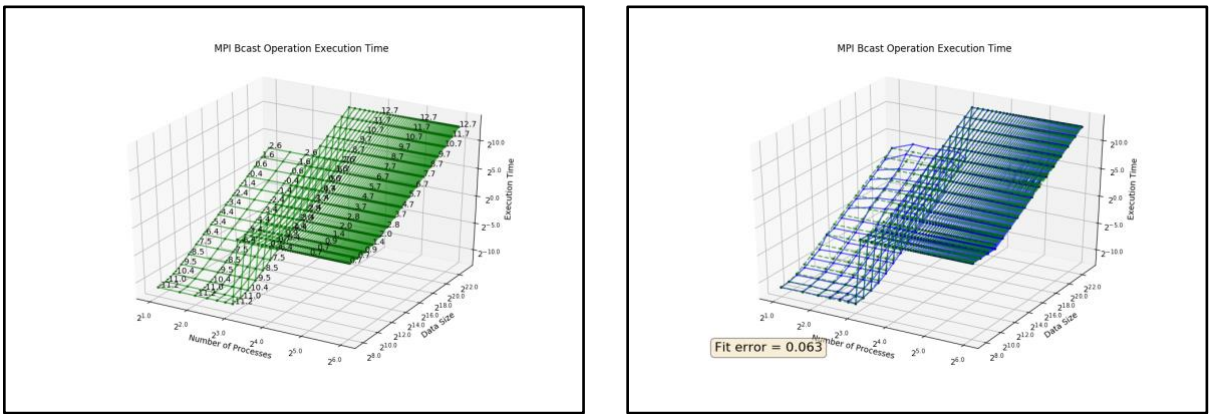


Figure 5-17. Fitting execution time of MPI_Bcast over processes and data size (PPN=8)

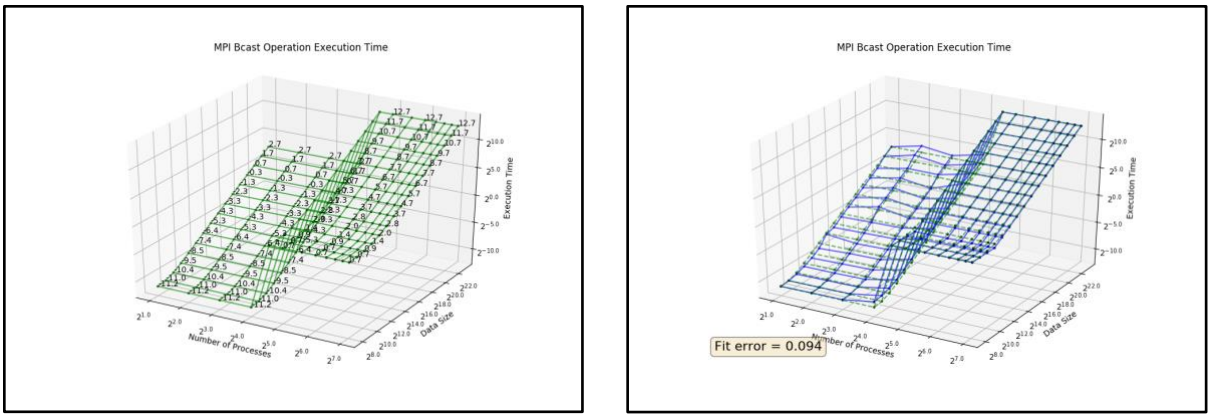


Figure 5-18. Fitting execution time of MPI_Bcast over processes and data size (PPN=16)

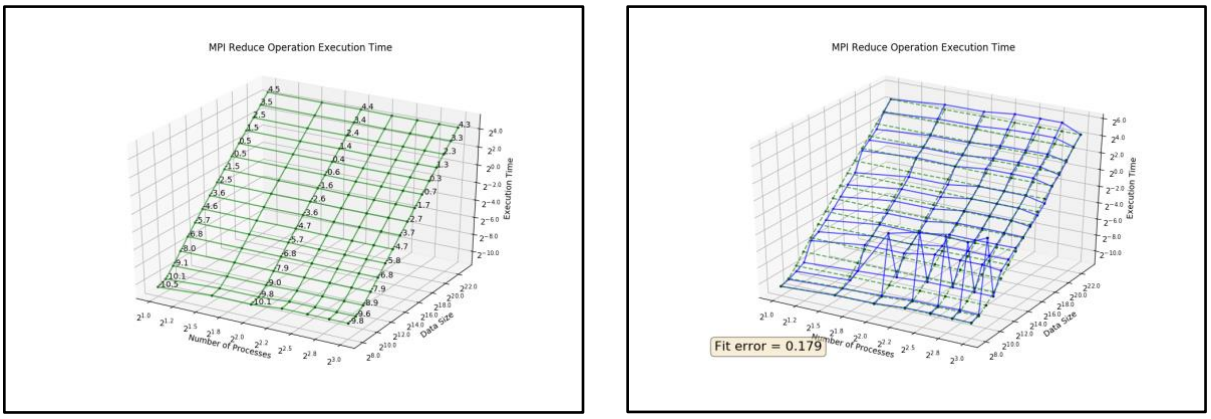


Figure 5-19. Fitting execution time of MPI_Reduce over processes and data size (PPN=1)

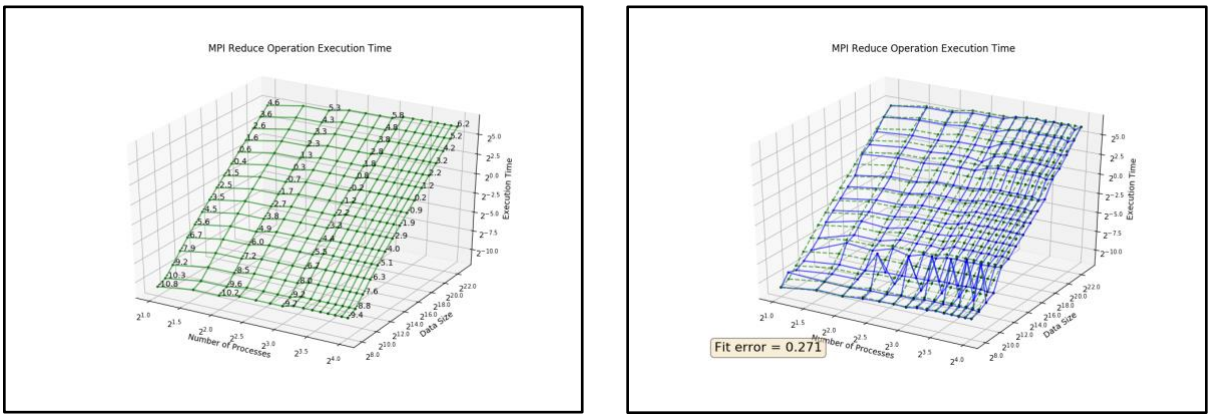


Figure 5-20. Fitting execution time of MPI_Reduce over processes and data size (PPN=2)

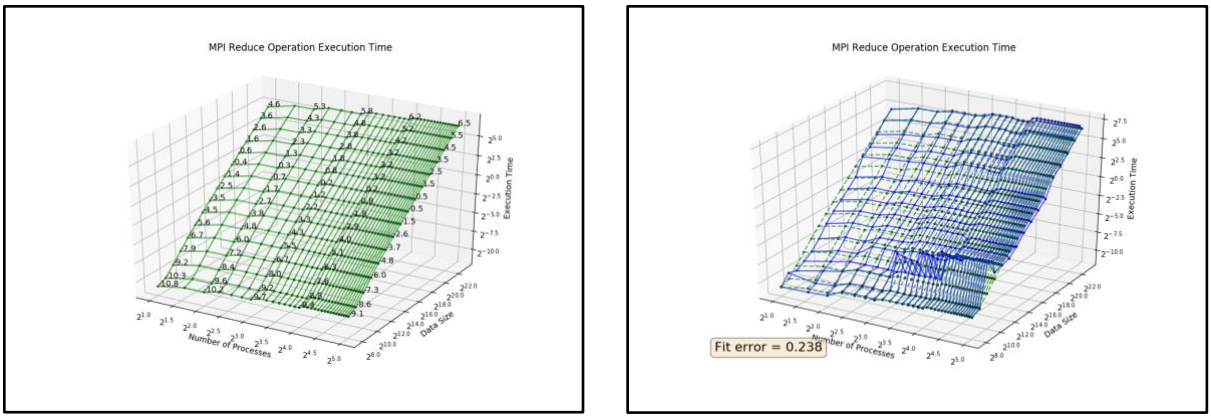


Figure 5-21. Fitting execution time of MPI_Reduce over processes and data size (PPN=4)

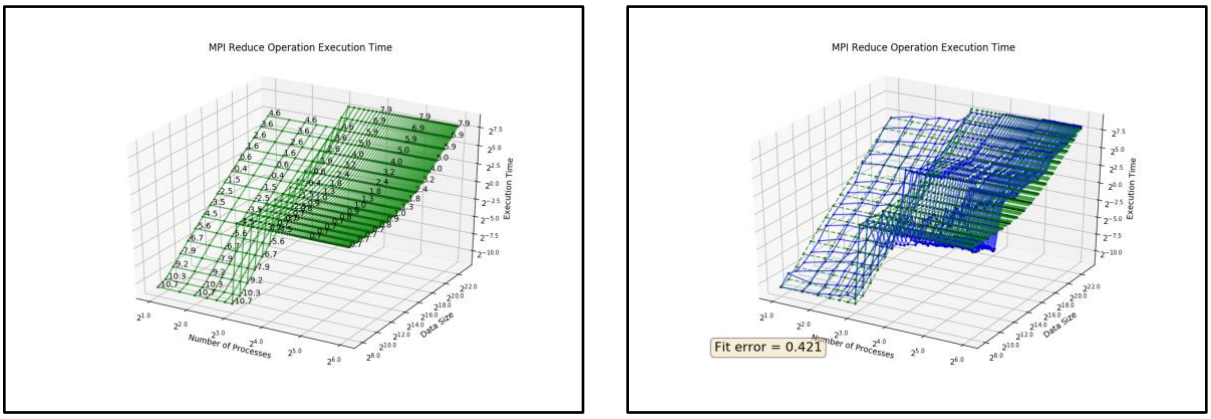


Figure 5-22. Fitting execution time of MPI_Reduce over processes and data size (PPN=8)

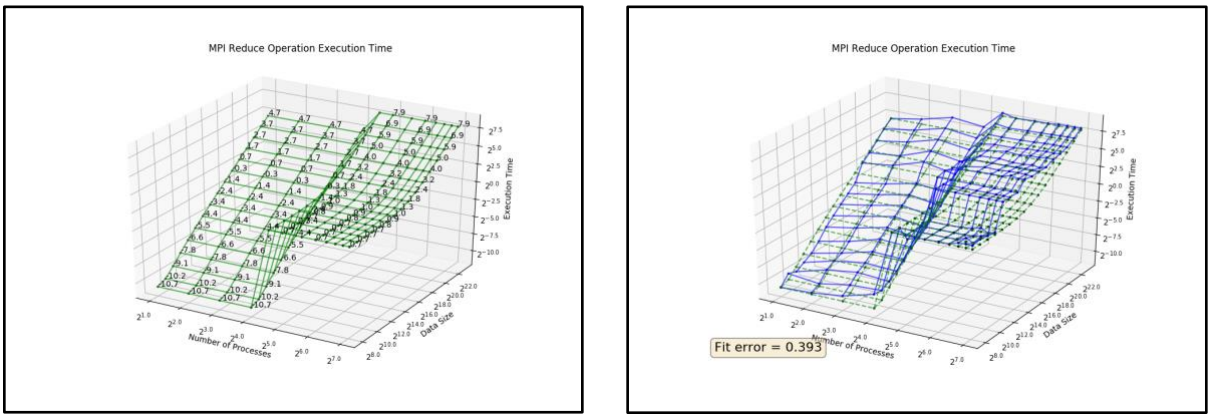


Figure 5-23. Fitting execution time of MPI_Reduce over processes and data size (PPN=16)

The fitting errors are relatively low for the *MPI_Bcast* operation, yet high for *MPI_Reduce* especially for large PPN values. The high errors for *MPI_Reduce* are mainly resulted from the evident outliers at a data size of 2^{12}. Also, the nature of *MPI_Reduce* differs from *MPI_Bcast* in that it adds an additional sum-up step. The task of developing a better model for the MPI operations is left as part of future work.

5.5 Putting back together: modeling the execution time for training epoch

Adding all steps together, the final model feeds the various workload into the corresponding models, sums up the total estimated time for a training epoch and converts to the throughput as the number of examples trained per millisecond. In the Python implementation, operations are categorized into 3 classes: ordinary sequential operations as the *Operation* base class, MPI operations as the *MpiOperation* sub-class, and compute kernel operations as the *OclOperation sub-class*. All 3 classes have the *get_cost* method for retrieving the predicted time cost given the necessary workload sizes. For details of the throughput projection model, refer to the implementation source code in "python/plot_neuralnet/plot_neuralnet_timing.py".

Figure 5-24 and Figure 5-25 are comparisons of the benchmark results versus the model prediction for CPU and GPU, accordingly. Except only a few outliers, the predicted throughputs achieve good fidelities both in terms of magnitude and scaling over worker count and mini-batch size. Therefore, the final throughput projection model is successful.

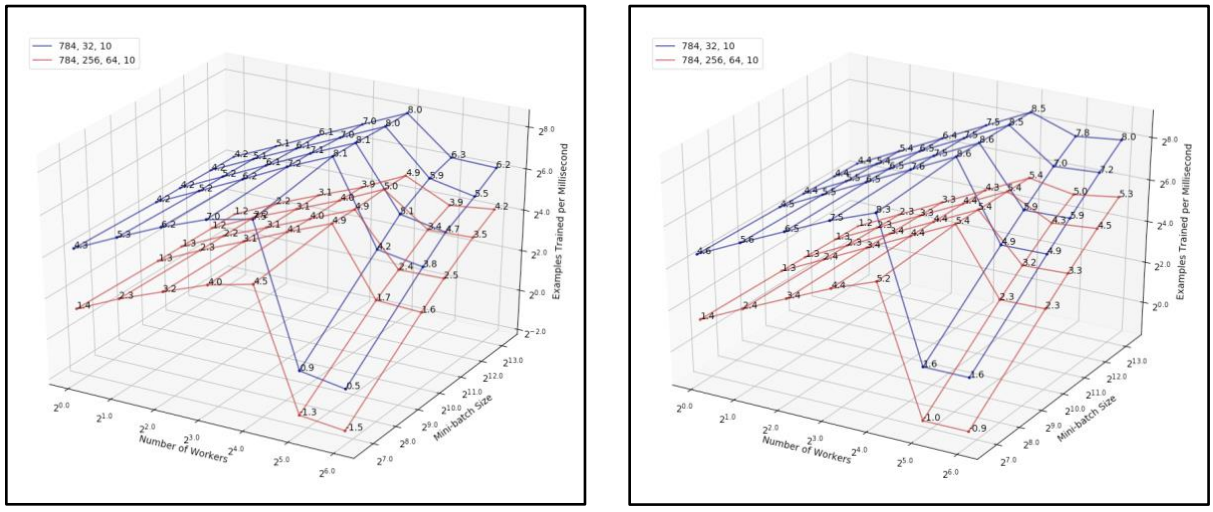


Figure 5-24. Throughputs of neural network training (CPU), original (left) vs. prediction (right)

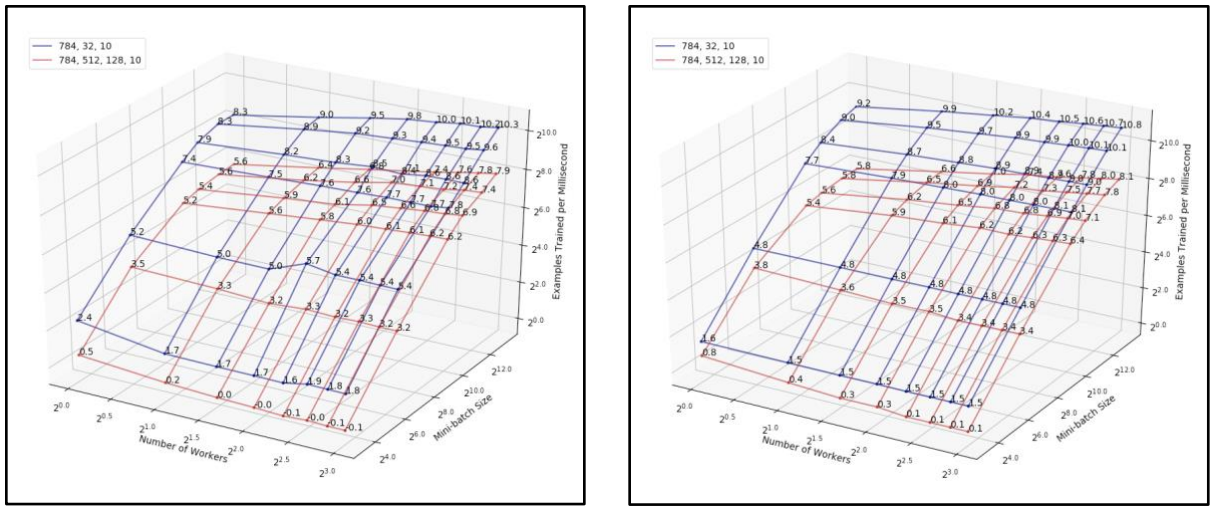


Figure 5-25. Throughputs of neural network training (GPU), original (left) vs. prediction (right)

One cause of the offset between the original and the projected throughput is that the cost of normal sequential operations such as index shuffling and parameter update is set to 0. Since these sequential operations are least significant in term of execution time, the impact is negligible. For an even more accurate representation, it is recommended to set the normal operation cost as the constant factor between CPU execution time and sequential instruction count.

5.6 CPU vs. GPU

It is a common research interest to compare the performances of GPU and the CPU in machine learning. Figure 5-26 and Figure 5-27 show the GPU speed-up over CPU for 1-D and 2-D compute kernels, respectively. All plots are in loglog scale. Apparently, it is not always better to use the GPU for data-parallel components. For the *vec_mult* and *mat_vec_add* kernels, in particular, the GPU never out-performs the CPU. In addition, the GPU speed-up eventually levels off and approaches to a constant maximum when the data size increases asymptotically, which means the GPU essentially is not much different from a processor that just have more cores.

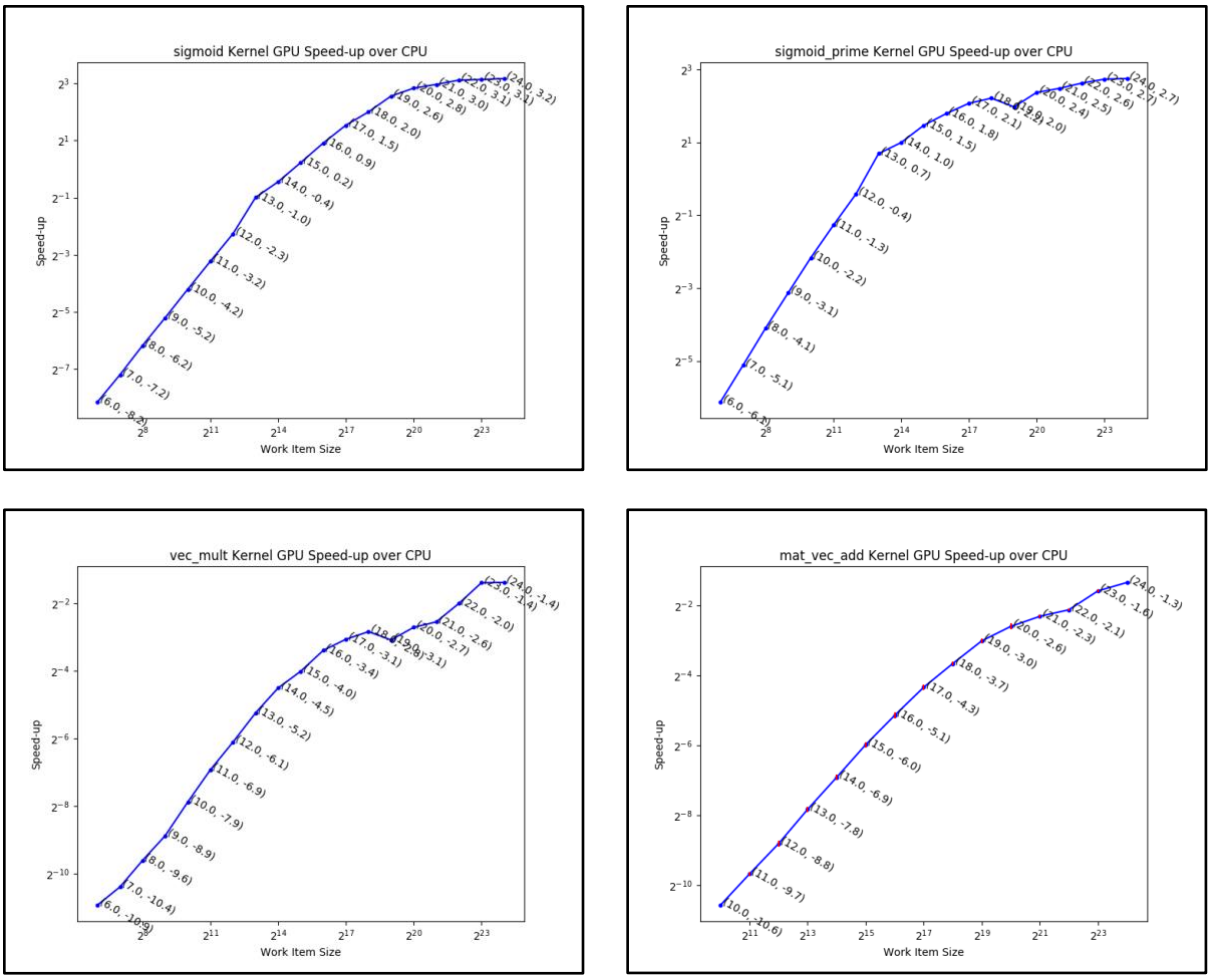


Figure 5-26. GPU speed-up over CPU on 1-D compute kernels

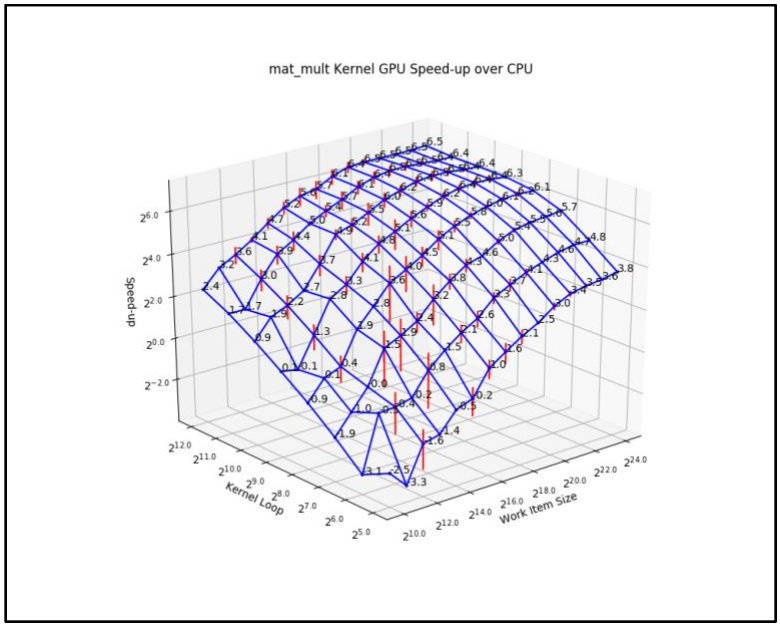


Figure 5-27. GPU speed-up over CPU on mat_mult compute kernel

5.7 Effects of various parameters

5.7.1 Neural network size

Using the complete projection model, the throughput of neural networks with larger network size can be predicted.

Figure 5-28 and Figure 5-29 present the predicted training throughputs of a smaller network with 1 hidden layer of 32 neurons and a larger network with 4 hidden layers of 1024 neurons each, using CPU and GPU, respectively.

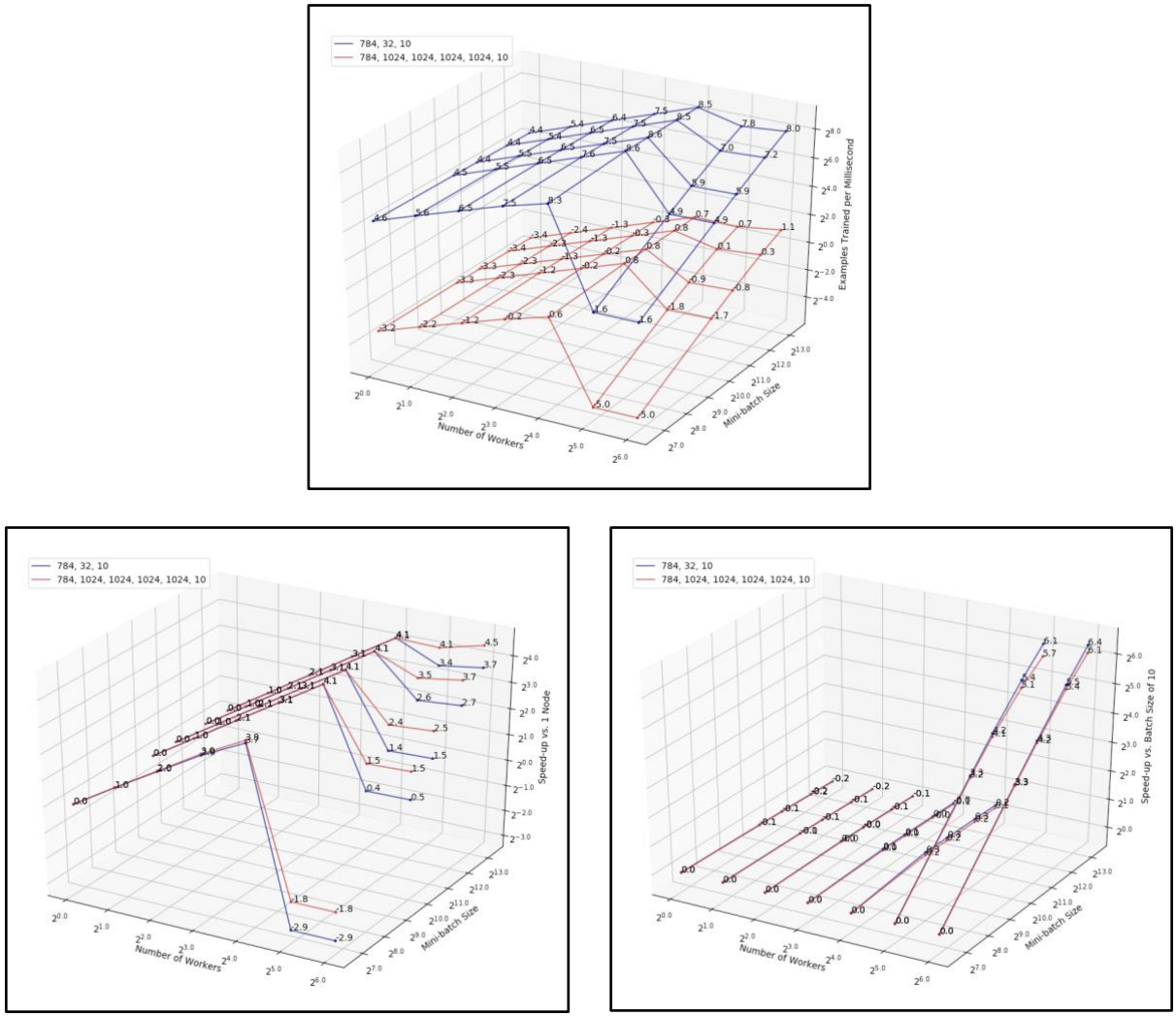


Figure 5-28. Predicted throughputs of neural network training (CPU), small vs. large network

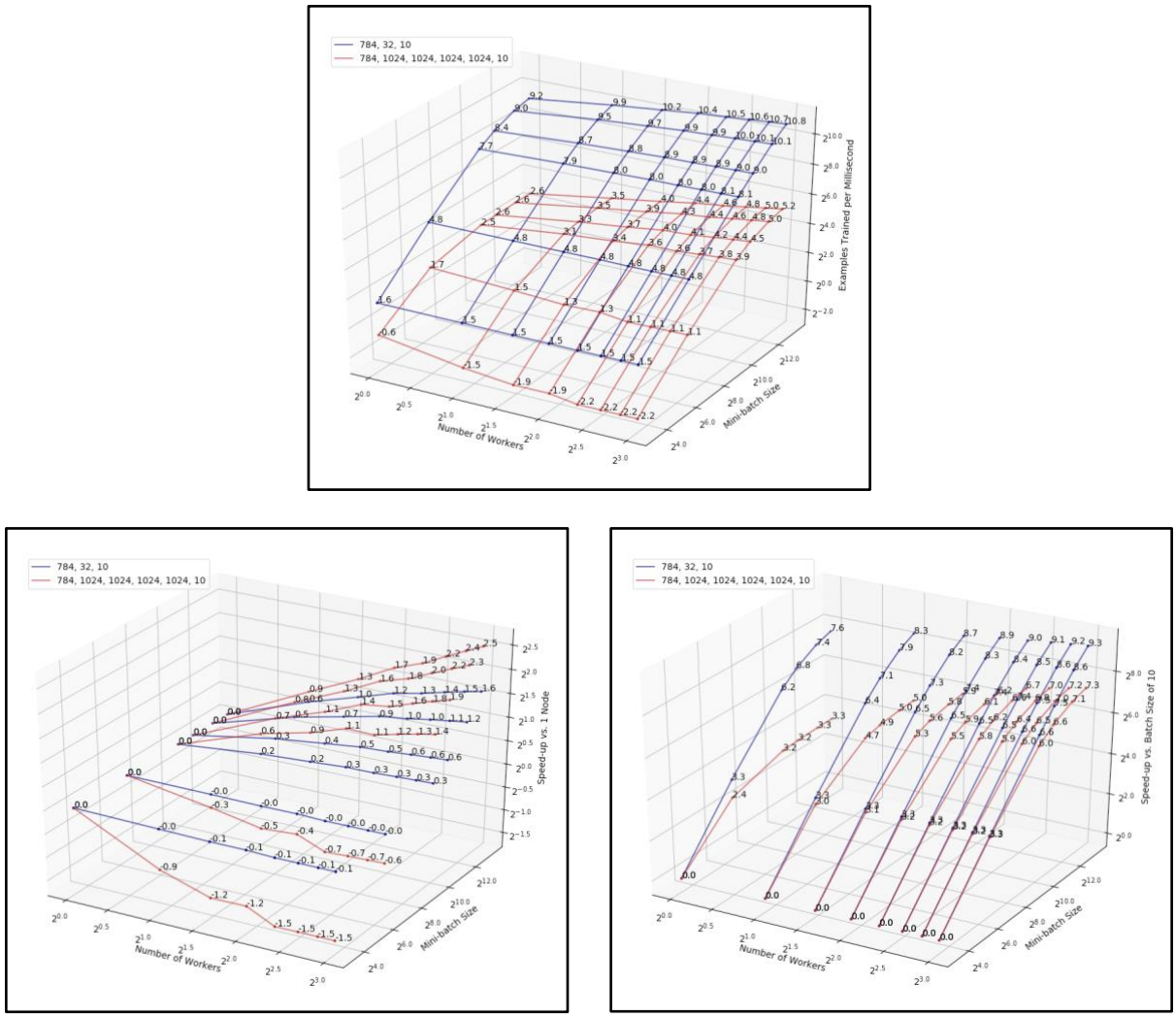


Figure 5-29. Predicted throughputs of neural network training (GPU), small vs. large network

Regardless of the network structure, the CPU training throughput speed-up over number of workers on a single node (i.e. worker count no more than the PPN) is consistent and perfectly linear. When the PPN asymptotically increases, the speed-up will eventually level off. But that is a long way to go, because the number of processors per CPU socket available is limited even for state-of-the-art computers, compared to the GPU.

Similar pattern is observed for the GPU training of the larger network with the largest mini-batch size, when increasing the worker count. The cut-off PPN is set to 8 for the GPU training scenario so the MPI overhead has not kicked in yet. In this case, the mini-batch size and

the layer sizes are large enough to fully occupy a single GPU into the linear region. Hence the multiple workers leveraged workload and regain the optimal parallel speed-up on each GPU. For any smaller mini-batch size, however, not only is there no significant speed-up when more workers are involved, but it may also slow-down as a joint result of MPI communication and GPU memory transfer overhead.

There are two other crucial resemblances between the CPU plots and the GPU plots, the first being the resemblance of CPU speed-up when worker count is greater than the cut-off PPN versus the GPU speed-up over mini-batch size for smaller network. The speed-up of the former is gained from reducing the total MPI communication time, by sending larger chunks of data less frequently. It is also a linear speed-up because the reduced communication overhead is directly proportional to the reduced training iteration count, thus also directly proportional to the increased mini-batch size. For the GPU, the speed-up is also gained from transferring larger chunks of data less frequently. The speed-up on the GPU is mostly linear for smaller sized network with larger worker counts, because this is where the GPU still gains the maximum parallelism from a relatively small work-item size (i.e. before it hits the linear region).

The other resemblance is between the constant throughput for CPU on a single node—regardless of worker count or mini-batch size, and the single worker GPU scaling level-off when mini-batch size increases for the larger network. The case for CPU is that the total number of sequential loop iterations is invariant under different mini-batch size, hence the execution time is constant. The single GPU, on the other hand, enters the linear region when the network size and mini-batch size are both considerably large and thus scales as a linear function. This reveals the similarity between a GPU and a multi-core CPU in terms of data-parallelism.

5.7.2 CPU and GPU: clock frequency and core count

The higher CPU and GPU clock frequency evidently brings a speed-up directly proportional to the clock frequency. Although there has long been a warning on the , chip manufacturers are still improving the single core clock frequency, yet at a much slower pace than several decades ago [32].

Some also claimed the end game of the multi-core scaling and a "dark silicon" era [33]. However, CPU and GPU core counts are still constantly increasing by the time of this research. An increased number of vector units per CPU will increase the throughput in general, but will not scale up over increasing mini-batch size until the core-count is comparable to the GPU (e.g. by thousands), which is very hard to achieve in the next decade. On the other hand, an increased CPU core count per socket will not affect the scaling over mini-batch size, yet it will extend the multi-worker scaling on a single node for a larger worker count. The scaling over the worker count will stay linear until leveling-off at the break-even point where the inter-core message passing and synchronization overhead balances the parallel workload reduction speed-up, which only happens for a very high core count per socket.

Increasing the GPU "core" count typically means either increasing the number of compute units (e.g. stream processors) or the number of processing elements (e.g. vector units) in each compute unit. For a simple OpenCL kernel design (i.e. without exploiting local memory), the same effect is to increase the maximum number of concurrently running work-items. In terms of 1-D compute kernel execution time slow-down plot, this means shifting the curve to the right, making a wider perfect-parallel region. In turn, this helps regain the scale-up over mini-batch size for larger network sizes. However, the scaling as a function of worker count should still

remain completely level (or even scale down due to inter-core MPI overhead), and only scale up for network size or mini-batch size large enough to fully occupy the GPU into the linear region.

5.7.3 Communication overhead

There are two major types of communication overhead: the memory transfer overhead between CPU and GPU (or any other compute device), and the inter-node MPI operation overhead. The MPI overhead between cores, though exists, does not contribute as much as the above two sources.

There are a number of ways where the GPU communication overhead can be reduced in terms of hardware advancements:

1. Increase the PCI-Express bus lane bandwidth, which has experienced major improvements over the past few decades.
2. Lower the set-up time caused by I/O command scheduling overhead. This is highly unlikely due to the operation system constraints, unless a next-generation virtualization is achieved [34].
3. Change the topology of connection between CPU and GPU. For example, NVIDIA has come up with a high-speed connectivity alternative to PCI-Express named NVLink [35]. In addition, integrated GPUs and laptop dedicated GPUs may sometimes outperform standard PCI-Express connection in terms of memory transfer alone. Performance-wise, however, laptop graphics are usually much less powerful.

Mitigating memory transfer overhead of CPU-GPU communication will mainly improve the performances of 1-D compute kernel and 2-D *mat_mult* kernel with small matrix dimensions on the GPU. Although an increase in throughput is guaranteed, the scaling over mini-batch size

will not change significantly and will also depend on whether the improvement is on the base time cost or the variable time cost related to data size, because the majority of the execution time is spent on matrix computation.

MPI operation overhead can be lowered via higher inter-node network bandwidth and optimizing the software implementation of MPI. CPU training will benefit significantly as per the benchmark set-up. With deduced MPI overhead, the throughput of CPU training is expected to scale up much better across multiple nodes. As in Figure 5-28, the scaling over worker count for the larger network is better than the smaller network. In particular, the throughput finally starts to increase across multiple nodes for the maximum mini-batch size tested.

Chapter 6 – Conclusions and Future Work

In this thesis, an open-source implementation of feedforward artificial neural network (ANN) model is presented and explained in detail. Starting from a simple sequential implementation, the model is extended to GPU support using OpenCL and distributed training using MPI, following a data-parallel, synchronous approach. The ANN model is then benchmarked using state-of-the-art GPU clusters, and the timing results are recorded. The final throughput model not only predicts the training throughput for any particular feedforward neural network structure, but also reveals the effects of various hardware aspects as scaling factors and forecasts the possibilities to speed up training.

As a base model, the feedforward ANN implementation can be further extended to support various network structures with more sophisticated layers such as convolutional layers. Apart from the efforts in building a more versatile learner, improvements are necessary for the timing model. A better timing model with consistently lower fit errors is needed for the *MPI_Reduce* operation. A more fine-grained timing and throughput model would offer an even more realistic representation of the actual performance results.

In the speech for the NIPS 2017 Test of Time Award, Ali Rahimi expressed his concern about the lack of strict mathematical proofs in machine learning modeling techniques, where he claimed that “machine learning has become alchemy” [36]. Ali claimed that machine learning has gradually lost its roots in science and that the vast majority of the research is over-reliant on manual parameter tuning without a solid proof. This particular research takes a different angle and focuses on the fundamental performance aspects in terms of execution time. Hopefully the efforts towards canonicalization and open source programming in this research project can contribute to the overall machine learning community.

References

[1] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[2] M. Arbib, "Review of 'Perceptrons: An Introduction to Computational Geometry' (Minsky, M., and Papert, S.; 1969)," *IEEE Transactions on Information Theory*, vol. 15, no. 6, pp. 738-739, 1969.

[3] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, journal article vol. 2, no. 4, pp. 303-314, December 01 1989.

[4] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359-366, 1989/01/01/ 1989.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, p. 533, 10/09/online 1986.

[6] M. A. Nielsen, Neural Networks and Deep Learning: Determination Press, 2015. [Online]. Available.

[7] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948-960, 1972.

[8] (July 8). *Adaptive Machine Learning Acceleration*. Available: https://www.xilinx.com/applications/megatrends/machine-learning.html

[9] (July 8). *Cloud TPUs - ML accelerators for TensorFlow | Google Cloud*. Available: https://cloud.google.com/tpu/

[10] D. Steinkraus, I. Buck, and P. Y. Simard, "Using GPUs for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, 2005, pp. 1115-1120 Vol. 2.

[11] Y. Le. (2015, July 8). *Schema Validation with Intel® Streaming SIMD Extensions 4 (Intel® SSE4) | Intel® Software*. Available: https://software.intel.com/en-us/articles/schema-validation-with-intel-streaming-simd-extensions-4-intel-sse4

[12] J. R. (2017, July 8). *Intel® AVX-512 Instructions | Intel® Software*. Available: https://software.intel.com/en-us/blogs/2013/avx-512-instructions

[13] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules," in *Advances in Neural Information Processing Systems 30*, I. Guyon *et al.*, Eds.: Curran Associates, Inc., 2017, pp. 3856-3866.

[14] (2018, June 29). *1.5. Stochastic Gradient Descent — scikit-learn 0.19.1 documentation*. Available: http://scikit-learn.org/stable/modules/sgd.html

[15] (July 8). *Distributed TensorFlow | TensorFlow*. Available: https://www.tensorflow.org/deploy/distributed

[16] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in neural information processing systems*, 2014, pp. 2834-2842.

[17] D. DeVault. (2018, July 09). *Simple, correct, fast: in that order*. Available: https://drewdevault.com/2018/07/09/Simple-correct-fast.html

[18] (June 29). *Using the GNU Compiler Collection (GCC)/ Optimize Options*. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[19] (June 29). *GCC optimization - Gentoo Wiki*. Available: https://wiki.gentoo.org/wiki/GCC_optimization

[20] K. O. W. Group, "The OpenCL Specification version 1.2, Nov. 2012," ed.

[21] (July 8). *Matplotlib/ Python plotting — Matplotlib 2.2.2 documentation*. Available: https://matplotlib.org

[22] K. GROUP, "OpenCL 2.2 API Specification (Provisional)," ed, 2016.

[23] S. Sur, M. J. Koop, and D. K. Panda, "High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis," presented at the Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florida, 2006.

[24] (February 27). *MPI | Advanced Research Computing at Virginia Tech*. Available: https://www.arc.vt.edu/userguide/mpi/

[25] A. Krizhevsky, V. Nair, and G. Hinton. (2014). *The CIFAR-10 dataset*. Available: http://www.cs.toronto.edu/kriz/cifar.html

[26] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley Professional, 2011, p. 648.

[27] (June 29). *timespec - cppreference.com*. Available: https://en.cppreference.com/w/c/chrono/timespec

[28] (February 27). *NewRiver | Advanced Research Computing at Virginia Tech*. Available: https://www.arc.vt.edu/computing/newriver/

[29] (July 20). *MVAPICH / Home*. Available: http://mvapich.cse.ohio-state.edu

[30] J. S. Armstrong, *Long-range forecasting*. Wiley New York ETC., 1985.

[31] M. TEAM, "MVAPICH2-X 2.2 rc2 User Guide," 2001.

[32] B. Crothers, "End of Moore's Law: it's not just about physics," *CNet. August*, vol. 28, 2013.

[33] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365-376.

[34] G. Bernhardt, "The Birth & Death of JavaScript," ed, 2014.

[35] D. Foley, "Nvlink, pascal and stacked memory: Feeding the appetite for big data," *Nvidia. com*, 2014.

[36] B. R. Ali Rahimi. (2017). *Reflections on Random Kitchen Sinks – arg min blog*. Available: http://www.argmin.net/2017/12/05/kitchen-sinks/