PERFORMANCE ANALYSIS OF ARTIFICIAL INTELLIGENCE WORKLOADS

A Thesis

by

POORNIMA BEVAKAL GURUSHANTHAPPA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2019

Major Subject: Computer Engineering

ABSTRACT

Machine Learning involves analysing large sets of training data to make predictions and decisions to achieve a specific objective. This data-intensive computation places enormous demand on the underlying hardware. To improve overall performance and make predictions quickly, extremely powerful specialized hardware has been built to make the data processing faster. Although this has yielded improved results, they are not economical. They often require significant investment in additional infrastructure and collaboration among the various hardware components. On the other hand, CPUs are cost-effective, and easily accessible for a fraction of the cost. Unfortunately, very little work has been done to identify the configuration bottlenecks in CPUs and improve their overall performance to meet the demands of Machine Learning.

This thesis aims to identify the system parameters which can be tweaked to achieve a boost in CPU performance for Machine Learning algorithms. Leveraging the Gem5 system simulator, a series of experiments were conducted varying the hardware configurations to observe the overall system performance. Analysis of the simulation results showed that CPU and system operating frequency, the L2 cache size and Indirect branch predictor can significantly affect the system performance. We strongly believe our system simulation results can further help in optimizing the performance of CPUs for machine learning workloads.

To Dad, Mom and my dear sister, Prathima.

To my beloved husband, Abhishek.

ACKNOWLEDGEMENTS

This day wouldn’t have been a reality without the support and encouragement of lot of people. I am forever grateful to them, and thank everyone from the bottom of my heart.

First and foremost, I would like to thank my committee chair, Dr. Kalafatis for providing the opportunity to work with him on this wonderful endeavour. His understanding, empathy and belief in me kept me going during times of uncertainty and helped me reach the light at the end of the tunnel. I will always be indebted to him.

I would further like to thank to my esteemed committee members, Dr. Gratz, and Dr. Tyagi, for their ideas, sage counsel and direction throughout the course of this research. They have been the compass that has steadied the ship and kept it from getting lost.

I was lucky to have Rejath George, Evagoras Stylianou and Antonis Akrytov as fellow researchers who helped immensely at various stages of the journey. Their fresh ideas, infectious energy and humour kept us sane.

Also, a heartfelt thank you goes to my friends, colleagues and the entire department faculty and staff for making my time at Texas A&M University a great experience. I will cherish all these memories for a lifetime.

Finally, big shout out to the constants in my life. Thank you, mom and dad, for your trusting your little daughter to take on this journey, far from home. Thank you for always being there for me.

And, last but not the least, thank you to my husband for his patience, love and believing in me when I myself seemed lost. Without him, I am pretty sure I would have completed my thesis much quicker.

CONTRIBUTORS AND FUNDING SOURCES

This work was supervised by a thesis committee consisting of Dr. Stavros Kalafatis [advisor] and Dr. Paul Gratz of the Department of Electrical and Computer Engineering and Dr. Aakash Tyagi of the Department of Computer Science and Engineering.

The data used for analysis for Chapter 4 was partly provided by Rejath George, Evagoras Stylianou and Antonis Akrytov.

All other work conducted for the thesis was completed by the student independently.

NOMENCLATURE

CPU	Central Processing Unit
GPU	Graphical Processing Unit
ISA	Instruction Set Architecture
IPC	Instructions per cycle
SE	System Emulation
FS	Full System
CNN	Convolutional Neural Network
ReLu	Rectified Liner Unit
MNIST	Modified National Institute of Standards and Technology
BAIR	Berkeley Artificial Intelligence Research
BSD	Berkeley Software Distribution
AI	Artificial Intelligence
ML	Machine Learning
BTB	Branch Target Buffer
SPEC	Standard Performance Evaluation Corporation

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Machine Learning (ML) and Artificial Intelligence (AI) have become an integral part of human life. It is used in day-to-day chores without ever realizing it. From simplistic applications such as Virtual Personal Assistants, to more futuristic autonomous driving vehicles and far more complex Bioinformatics and Medical Diagnosis, AI has wormed its way into human lives, helping in achieve things which could not have been possible otherwise. As AI continues to enlarge its foot print, its ability to make accurate predictions and take real-time decisions becomes all the more critical. Machine learning algorithms serve as a means to achieve this. The precision and correctness of any ML algorithm depends on its extensive training, which involves massive amounts of training data. While having large sets of training data greatly enhances an ML algorithm in accurately predicting the outcome, it in turn imposes immense pressure on the underlying hardware. As the number of parameters in the algorithm's model increase, ML frameworks could place even more computational demands on the hardware.

To speed up ML applications, dedicated hardware have been built. These dedicated accelerators have been heavily optimized for faster and parallel computations. Recent studies [19] have shown that Graphical Processing Units (GPUs) exhibit excellent performance on speeding up matrix multiplication operations, which are at the core of ML techniques. Young Jong Mo et. al. reported a speed up of 3x in computation speed for TensorFlow implementation of MNIST data classification using the multilayer perceptron on GPU-enabled platform as compared to CPU-only platform [20]. Shaoli Liu et. al. developed a novel Instruction Set Architecture (ISA) for neural networks called Cambricon, which allows neural network accelerators to flexibly support a broad range of different neural network techniques [15]. Dongjoo Shin et. al. implemented an energy efficient deep learning processor for convolution neural networks and recurrent neural networks [16]. This has resulted in dedicated hardware being widely used to accelerate ML training phase.

Although dedicated hardware is focused on computing and memory optimizations for ML algorithms, it comes at an extra cost. Additionally, it places increased infrastructure demands for high and efficient collaboration among the underlying hardware components such as CPU, memory, networking and storage devices. CPUs on the other hand, are easily available, well-supported and are traditionally cheaper. Although much work has been focused on improving hardware architectures for computing and memory optimizations, CPUs are often overlooked. Very little work has been done to study and identify the configuration bottlenecks in CPUs and improve their overall performance to meet the growing demands of Machine Learning.

There are several studies focusing on the impact of hardware configurations on the execution time for CNN. Jingjun Li et. al. [18] studies the impact of CPU frequency, GPU number, storage devices, and memory size on the execution time. Hwan Lim et al. [17] studied the impact of storage systems. In comparison, we have looked one level deeper and also focussed on micro architectural details such cache misses, branch mispredictions, memory size, clock frequency (system and CPU frequency).

Hand-written digits classification algorithm uses MNIST data set as input, which consists of 70,000 images each 28*28 pixels. Along with this, each neural network layer consists of learning parameters and activation elements which further require additional memory. It is expected that this would hamper the performance of caches and memory. However, in neural networks, input data is usually divided into mini-batches and fed to the model. If the memory size is sufficient to hold this mini-batch data and the model’s parameters, its influence on the performance could be minimal. L2 cache on the other hand is not expected to be as big as the memory, which could have significant role in performance. The algorithm consists of training and testing loops, and also many library function calls, which results in branching instructions both conditional and indirect. If the branch predictor used does not handle these branch instructions efficiently, it could hamper the performance. Like in any applications, as the operating frequency increases, significant gain in performance can be achieved, AI application performance is expected to increase with frequency increase.

Thesis Statement: AI workloads possess unique characteristics which exercises pressure on system memory, attributable to massive amounts of data involved to train the model, and branch prediction, attributable to large number of branch instructions generated by training and testing loops.

This thesis focuses on exploring and understanding the impact of these hardware configurations in general CPUs. The aim of the thesis is to analyse the performance of the workloads and identify the bottlenecks with respect to both hardware and software. In short, the key contributions are as follows:

1. Conduct comprehensive simulations of different implementations of a hand-written digit recognition algorithm on various system configurations.
2. Experimentally and systematically evaluate the results of simulations in understanding the overall effectiveness of these configurations on system performance. We believe this work paves a way for a deeper understanding of the effects the various configurations have on system performance, and provides means to improve the same.

The rest of the thesis is organised as follows. Chapter III introduces the background of CNN, specifically the CNN used for hand-written digit recognition. Chapter III presents our experimental methodology. Chapter IV demonstrates the experimental results and analyses the results to identify the bottlenecks. Chapter V provides conclusion and explores future work.

CHAPTER II

BACKGROUND

Artificial Intelligence is a rapidly developing field. A great deal of ML algorithms exits as of today, while new ones are being constantly being invented. To ease the effective development of the deep learning algorithms, multiple deep learning platforms like TensorFlow [9], Caffe, Theano, Scikit-learn, Keras, CNTK, etc., have been developed. A great deal of efforts also goes into making these platforms more efficient. Various AI platforms have been compared and contrast [10], performance analysis of different frameworks have been conducted [11] [12][13], new algorithms have been developed to reduce the data storage requirements [14], enhancements have been made in existing software platforms to increase efficiency [8].

2.1 Convolutional Neural Networks (CNN): CNN is widely used deep neural network in the domain of Computer Vision as they excel in image recognition and image classification categories. The connectivity pattern in CNNs derives its inspiration from the organization of human visual cortex. CNNs perform better than Feed-Forward Neural Networks in image processing, as they successfully capture the spatial and temporal dependencies in an image.

A convolutional neural network consists of an input layer, an output layer, and multiple hidden layers in-between. Hidden layers can be modelled as Convolution Layer, Activation Layer, Pooling Layer and Fully-connected Layer. The convolution layer performs convolution operation, which extracts the high-level features from the input image. The first convolution layer typically captures the low-level features, while the other added layers covers the high-level features. Kernel/Filter is used for accomplishing the convolution operation. Activation layer implements activation functions which decide the final value of a neuron. Rectified Linear Unit (ReLu) is the typically used activation function.

The pooling layer is used for dimensionality reduction. It reduces a n x m patch into a single value to make the network less sensitive to the spatial location. Pooling layer can be of two types – Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. Average Pooling returns the average of all the values from the portion of the image covered by the Kernel. In a fully connected layer, each neuron is connected to all the neurons from the previous layer. It is used for learning non-linear combinations of the output from previous layer.

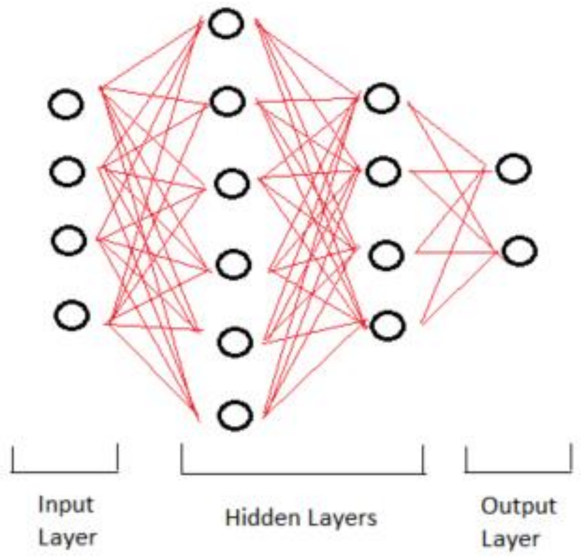

Figure 1: Convolutional Neural Network

2.2 Handwritten Digits Recognition using CNNs: Handwritten digits recognition is considered as the "Hello World" equivalent of the deep learning domain. Images of hand written digits are processed through CNN layers, and they are classified as digits between 0 and 9. The MNIST data set is used as input data for digits classification. It consists of 70,000 28*28-pixel grayscale images of hand-written single digits. The images are further broken down to 60,000 training images and 10,000 test images.

The input layer consists of neurons which maps to a single pixel in the input image. As the input images are 28*28 pixels, the input layer for digits recognition CNN has 784 neurons (28*28). There are 10 different classes each image can represent i.e., 0-9, hence the output layer has 10 neurons. The number of hidden layers and also number of neurons in each layer varies from one implementation to another. 4 different open-source deep learning platforms, TensorFlow, Caffe, Apache MXNet, and Scikit-Learn, are used to collect the micro architecture data.

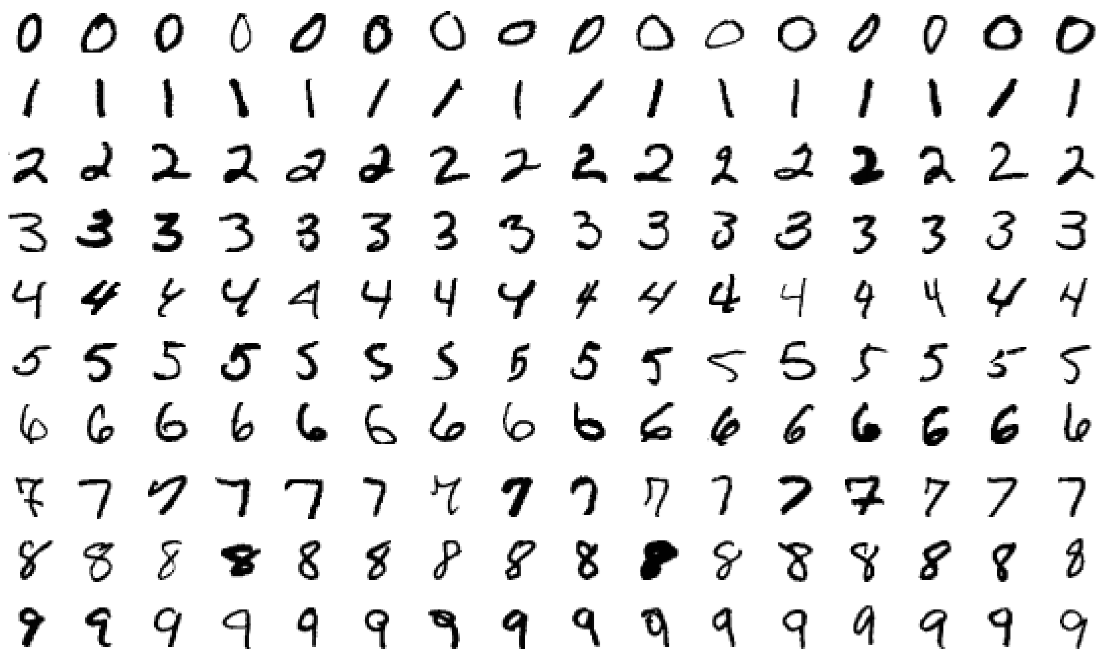

Figure 2: Sample MNIST Data Set

(Figure reused from Online source [23])

2.2.1 TensorFlow: TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is also used for machine learning applications. TensorFlow was developed by Google Brain team, released initially in 2015. It is programmed using Python, C++ and CUDA. The entire source code is hosted at GitHub [2].

TensorFlow CNN network has one input layer with 784 neurons, one output layer with 10 neurons. There are two hidden layers. Each layer has one convolution layer and one ReLu activation layer. Hidden layer 1 has 128 outputs while hidden layer 2 has 32 outputs.

2.2.2 Caffe: Caffe is a deep learning framework developed by Berkeley AI Research (BAIR), released initially in 2013. It is one of the top open source frameworks, licensed under BSD 2-Clause license. It is developed using C++, with a python, MATLAB and C++ interface. The entire source code is hosted at GitHub [3].

Caffe MNIST network used for digits recognition has one input layer with 784 neurons, one output layer with 10 neurons. The input layer is connected to a convolutional layer 1 with a 5*5 kernel and 10 outputs. The weights and bias of the convolution network neurons can be initialized while defining the layer. Xavier algorithm is used for initialization of the weights value based on the number of input and output neurons. The convolution layer is connected to a pooling layer 1, which performs max pooling with a pool kernel size 2 and a stride of 2. The pooling layer 1 is connected to a convolutional layer 2 with a 5*5 kernel and 10 outputs. Xavier algorithm is used for initialization of the weights value. The convolution layer 2 is connected to a pooling layer2 which performs max pooling with a pool kernel size 2 and a stride of 2. The pooling layer 2 is connected to a fully connected layer with 500 outputs. This is followed by an ReLu activation layer. Activation layer is connected to the output layer with 10 outputs.

2.2.3 Apache MXNet: Apache MXNet is an open-source deep learning framework developed by Apache Software Foundation. It is written in C++, Python, R, Julia, JavaScript, Scala, Go, Perl. The source code is hosted at GitHub [4].

MXNet CNN used for digits recognition has one input layer with 784 neurons, one output layer with 10 neurons. There are 3 hidden layers, the first layer consists of first convolution layer, tanh activation layer, a pooling layer, the second layer comprises of second convolution layer, tanh activation layer, a pooling layer, the third layer is made up of fully connected layer and tanh activation layer.

The first convolution layer has 20 5*5 kernels/filters which are used for carrying out the convolution operation. The type of the pooling layer is Max pooling. The second convolution layer has 50 5*5 kernels.

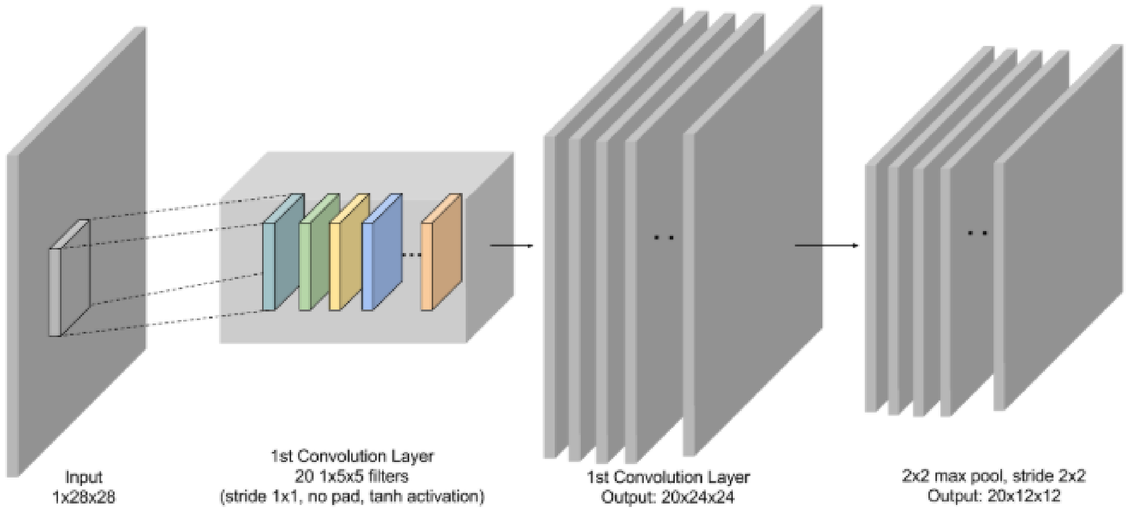

Figure 3: First Convolution Layer with tanh activation function, and pooling layer in MXNet CNN

(Figure reused from Online source [7])

2.2.4 Scikit-Learn: Scikit-learn is an open-source machine learning library designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. It was initially developed as a Google Summer of Code project by David Cournapeau in 2007. The source code is hosted at GitHub [5].

CHAPTER III

EXPERIMENTAL SET UP

In the interest of identifying the bottlenecks in the CPU architecture, performance data of lower level components of system needs to be collected. For this purpose, we use an architecture simulator. Gem5 [1][6] was chosen as the architecture simulator used to simulate a general-purpose CPU. The gem5 simulator is an open source software used for computer architecture research. It is a merge between M5 (University of Michigan) and GEMS (University of Wisconsin-Madison) simulators. It uses the CPU models, Instruction Set Architectures (ISA), I/O devices from M5 simulator, and cache coherence protocols, interconnect models from GEMS simulator, thus exploiting the goodness of both the simulators.

The simulation framework provided by gem5 is highly configurable, supporting different execution modes, multiple interchangeable CPU models, memory models, and also GPU. It supports most commercial ISAs like ARM, X86. ALPHA, MIPS, Power, and SPARC. Unix-like operating systems can also be booted on gem5. Gem5 also provides flexibility of defining the build of the system under observation. This feature of gem5 is exploited to vary the build of the simulated system to study the impact of various hardware configurations on the execution of Deep learning workloads. Gem5's full system simulation mode is used to simulate the microprocessor hardware along with Linux operating system. This requires a disk image with the operating system and the deep learning framework installed on it. The details of creating a new disk image and installing the framework are available in appendix A.

ARM and X86 are two most widely-used ISA in a CPU. While ARM dominates the mobile market, X86 dominates the Personal Computer field. The leading platforms for AI workloads are ARM-based systems. Lot of work is going on in integrating AI co-processors with ARM-based CPU. Naturally, ARM ISA becomes the first choice for our experiments. However, it does come with challenges when integrating it with gem5. The ubuntu core distribution for ARM, which is required by the full system image, does not come with a packet manager. Hence, all deep learning frameworks needs to be installed from source or requires cross compilation. This approach did not work for all frameworks. X86 ISA was chosen instead, as it is widely used in data centres and well supported by both gem5 and ubuntu.

Gem5 has its own set of limitations which makes choosing deep learning frameworks trickier. Currently gem5 doesn’t support python versions greater than 2.7, whereas deep learning frameworks have moved on to python 3+ versions. Also, 64-bit floating point values are not supported by gem5, while some deep learning frameworks extensively use 64-bit floating point values. Keeping in mind all these limitations Caffe, TensorFlow, Apache MxNet and Scitkit-learn platforms were chosen.

As shown in figure 4, the system simulated has a CPU core, L1 data and instruction cache, L2 cache and a main memory. The configurations of these system components are varied to study their effect on performance.

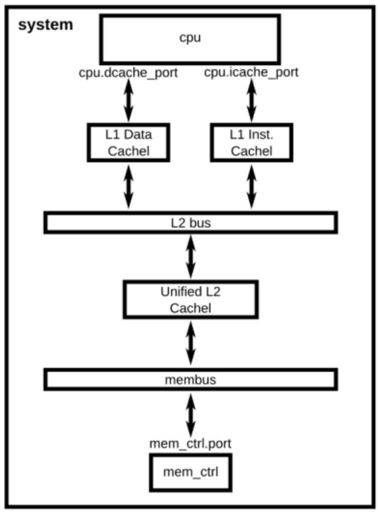

Figure 4: Simulated System Overview

Table 1 shows an overview of the configuration values chosen for experimentation. Gem5 supports three branch predictors: 2-bit local predictor, bi-mode predictor and tournament predictor. Bi-mode and Tournament predictor as chosen for observation. The bi-mode predictor is a two-level branch predictor that has three separate history arrays: a taken array, a not-taken array, and a choice array [21]. The taken/not-taken arrays are indexed by a hash of the PC and the global history. The choice array is indexed by the PC only. Because the taken/not-taken arrays use the same index, they must be the same size. The bi-mode branch predictor aims to eliminate the destructive aliasing that occurs when two branches of opposite biases share the same global history pattern. By separating the predictors into taken/not-taken arrays, and using the branch's PC to choose between the two, destructive aliasing is reduced.

Memory Size	L2 Cache Size	Associativity	System Frequency	CPU Frequency	Branch Predictor
4 GB	512kB	4	3GHz	3GHz	Bi-Mode
2 GB	1024kB	8	1GHz	1GHz	Tournament
				500MHz	

Table 1: Overview of System parameters selected for study

Tournament predictor has a local predictor, which uses a local history table to index into a table of counters, and a global predictor, which uses a global history to index into a table of counters [22]. A choice predictor chooses between the two. Both the global history register and the selected local history are speculatively updated.

Hand written digits recognition algorithm is chosen to be executed on the system simulated. Four different deep learning platform (TensorFlow, Caffe, MXNet, Scikit-learn) implementation of this algorithm is used. Each of the implementations are executed on 14 different hardware configurations. Table 2 summarizes the combinations of the system parameter configurations used for simulations.

Memory size	L2 cache size	Associativity	System frequency	CPU frequency	Branch Predictor
4GB	512KB	4	3GHz	3GHz	Bi-Mode
4GB	512KB	8	3GHz	3GHz	Bi-Mode
2GB	512KB	4	3GHz	3GHz	Bi-Mode
2GB	512KB	8	3GHz	3GHz	Bi-Mode
4GB	512KB	4	3GHz	500MHz	Bi-Mode
4GB	1024KB	4	3GHz	3GHz	Bi-Mode
4GB	512KB	4	1GHz	1GHz	Bi-Mode
4GB	512KB	4	3GHz	3GHz	Tournament
4GB	512KB	8	3GHz	3GHz	Tournament
2GB	512KB	4	3GHz	3GHz	Tournament
2GB	512KB	8	3GHz	3GHz	Tournament
4GB	512KB	4	3GHz	500MHz	Tournament
4GB	1024KB	4	3GHz	3GHz	Tournament
4GB	512KB	4	1GHz	1GHz	Tournament

Table 2: Various System Parameters combination simulated

Standard Performance Evaluation Corporation (SPEC) CPU2006 [24] benchmarks are used as baseline to compare the data generated by AI workloads. SPEC CPU2006 is industrystandardized, CPU-intensive benchmark suite, which is used widely both in industry and academics. SPEC CPU2006 workloads are executed on the gem5 simulator in system emulation mode and statistics data are collected. Some of the CPU2006 benchmark tests chosen are SPECint benchmarks - bzip2, gombk, sjeng, and SPECfp benchmarks - bwaves.

Bzip2 benchmark test performs compression and decompression of both highly compressible and not very compressible files. Gombk program plays Go game and executes a set of commands to analyse Go positions. Sjeng is a program that plays chess. Bwaves program numerically simulates blast waves in three-dimensional transonic transient laminar viscous flow.

CHAPTER IV

ANALYSIS OF RESULTS

4.1 SPEC Benchmarks: Selected SPEC benchmarks were executed with system configurations – Memory size 4GB, L2 Cache size 512KB, L2 cache associativity 4, System frequency 3GHz, CPU frequency 3GHz, and Bi-mode, and Tournament branch predictors. Performance data varied from one test to another as the application and implementation differ. Average of these values are considered to set the baseline. The execution time of the selected test suites were on an average of 2 seconds whereas CNN algorithm required an average of 400 seconds. Miss rates at L2 cache were on average of 90%. Tournament branch predictor achieved better results that Bi-mode predictor on all the test suites, having an average misprediction rate of less than 1%. Indirect branch predictor exhibited an average miss rate of 2%.

4.2 Operating Frequency: System and CPU frequencies were varied to study the influence of frequency on the performance of training neural network models. As shown in figure 5, as the frequency of operation was reduced, execution time on all the deep learning platforms increased. On Caffe, execution time increased approximately 3 times when CPU and system frequencies were reduced by 3 times. TensorFlow and MxNet showed an increase of 2.5 times, and Scikit-learn 1.4 times. On further decreasing CPU frequency to 500MHz and keeping system frequency at 3GHz, the execution time further decreased by 4.7 times for TensorFlow and MxNet, and 2 times for Scikit-learn.

It is common knowledge that when the CPU frequency is decreased, the CPU clock cycle is increased, reducing the number of instructions executed per second. As a result, the performance of all CNN models is hampered. Also, we observe difference in the performance reduction caused by reducing CPU frequency due to the fact that batch sizes and implementation differs from one deep learning platform to another. Decreasing the system frequency affects the memory access times. When the system frequency is decreased, the access time of images from the memory increases, resulting in performance loss.

All the four deep learning platforms performed best when the CPU and System were clocked at the maximum frequency possible in the simulator. It is safe to assume that the operating frequency has a huge impact on the performance of CNN networks models, higher the frequency better the performance.

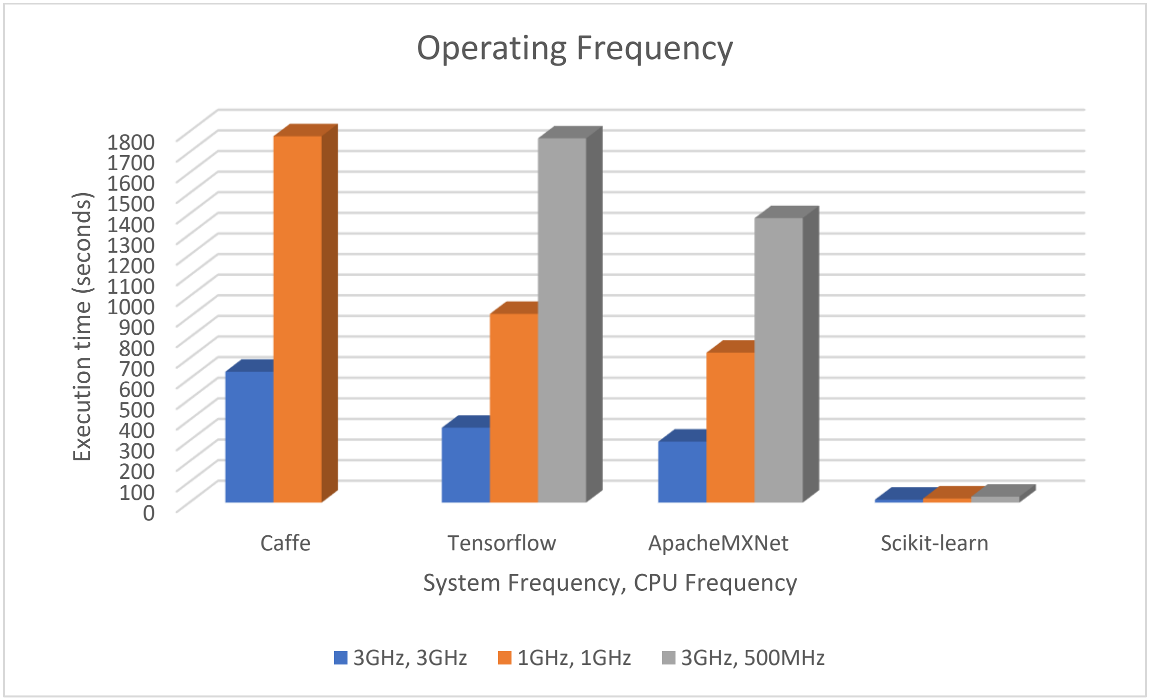

Figure 5: Operating Frequency vs Execution time

4.3 Memory Size: To study the impact of memory size on the performance of the CNN models, two memory sizes were chosen for experimentation – 2GB and 4GB. Figure 6 shows that varying the memory size had very less impact on the execution time. For Caffe, the execution time increased by 1.5% when memory size was reduced to 2GB. TensorFlow, Apache MxNet, and Scikit-learn displayed a reduction in execution time of 1%, 0.78% and 1.45% respectively. This is because of the way images are fed in to the neural networks for training. The entire data set, in our case 70,000 images, is divided into mini batches and then fed to the CNN model. The time spent by CPU on computing is more due to the vast number of parameters and complex algorithm. This part of computing time can be utilized by storage devices simultaneously to prefetch the next batch of data. As long as the memory size is sufficient to hold all the images in the min batch, the effect of memory size on performance is minimal.

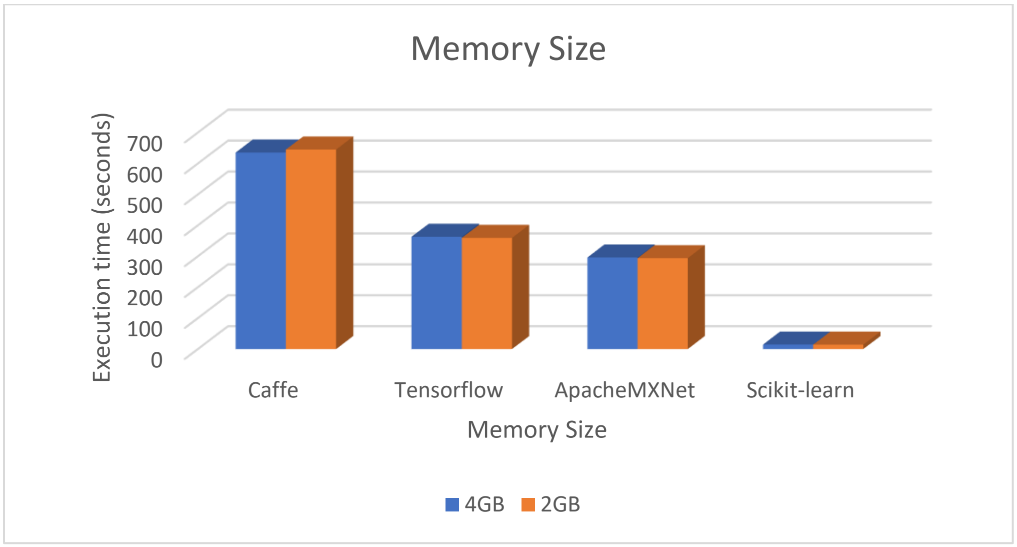

Figure 6: Memory size vs Execution time

4.4 L2 Cache: To explore the impact of L2 Cache size on the performance, cache sizes of 512KB and 1024KB were chosen. Although the miss rate in L2 cache was much lesser than the SPEC benchmarks, the number of L2 cache accesses in CNN algorithm was almost 1000 times more than that of SPEC test suites. As shown in figure 7, increasing the L2 cache size reduced the miss rate. TensorFlow showed a reduction in miss rate of approximately 10%, and MxNet approximately 5%. There was very little change in Scikit-learn as the data sets involved is very less.

The training loop in the CNN network model involves the input data image to be processed through all the layers in the model. Each layer holds a value called weights which aids the learning process. As these weights are frequently used in the loops, they would occupy the L1 cache. L2 cache would then be used for the input data images. Typically used L2 cache is not sufficient to hold these the data images. The result is we have a higher miss rate at this level. Thus, increasing the cache size helps improve the performance with increase in hit rate. To further support this conclusion, an experiment was conducted by increasing the L2 cache size to 16MB. Figure 8 shows that, for Apache MXNet, the cache miss rate decreased to 6.33% from 36.38%, while for TensorFlow, the cache miss rate decreased to 9.43% from 44.78%.

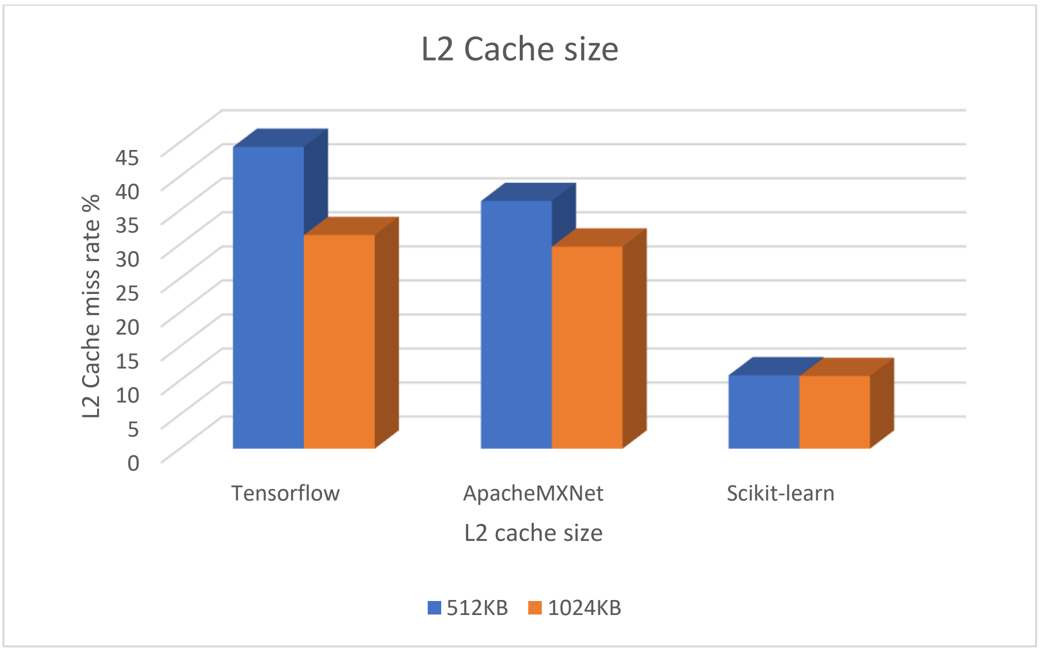

Figure 7: Cache size vs Miss rate

L2 Cache size was further varied and Apache MxNet algorithm was executed to understand the working set size of the algorithm. As shown in figure 9, 512KB cache had a miss rate of 36.38%, 1MB cache had a miss rate of 29.67%, 3MB cache had a miss rate of 21.09%, 6MB cache had a miss rate of 11.06%, 12MB cache had a miss rate of 7.11, and 16MB cache had a miss rate of 6.33. As cache size was increased the miss rate decreased and approached saturation around 12MB, indicating the working set for Apache MxNet implementation to be around 10MB-12MB.

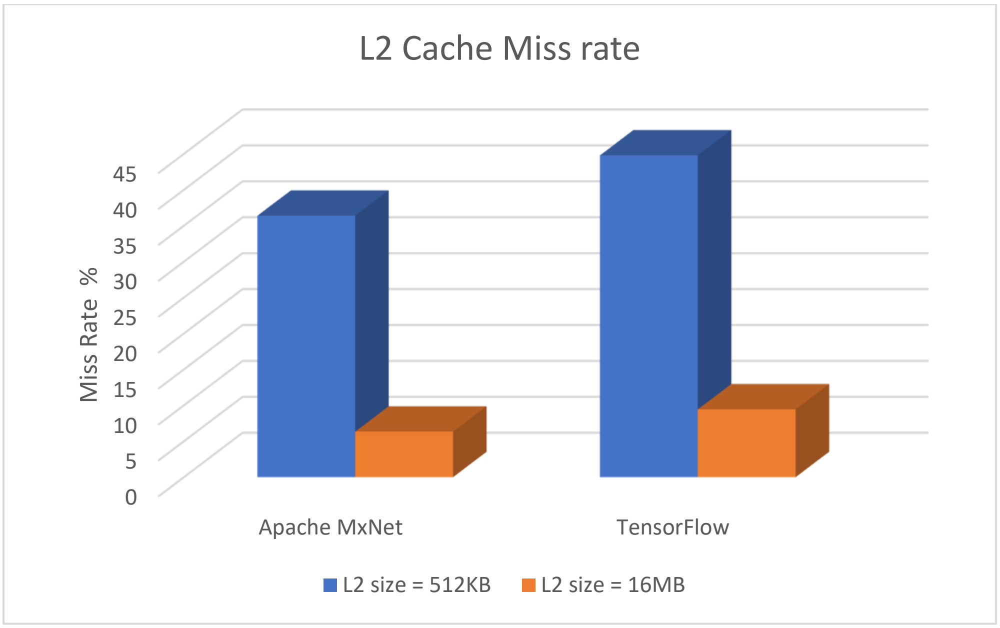

Figure 8: L2 cache size vs Cache miss rate

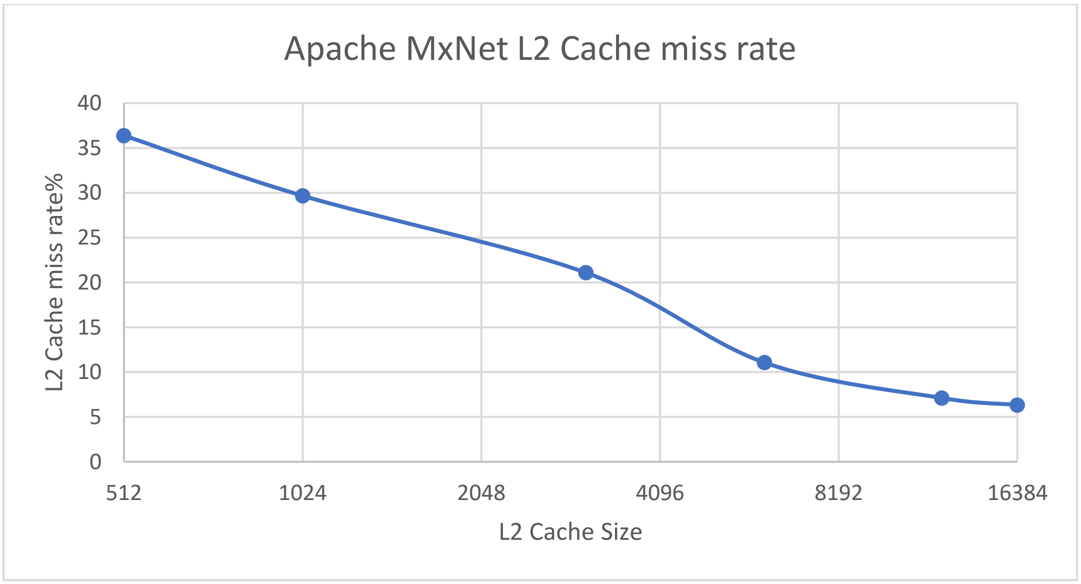

Figure 9: Apache MxNet L2 cache size vs Cache miss rate

Figure 10 shows that increasing the L2 cache associativity had very less effect on the miss rate. Given the fact that each data image is unique, and the chance that the same data image would be present in cache when it is referenced again is very thin owing to large data sets and small batch-size, associativity needs to be a very large number to gain any significant improvement in hit rate.

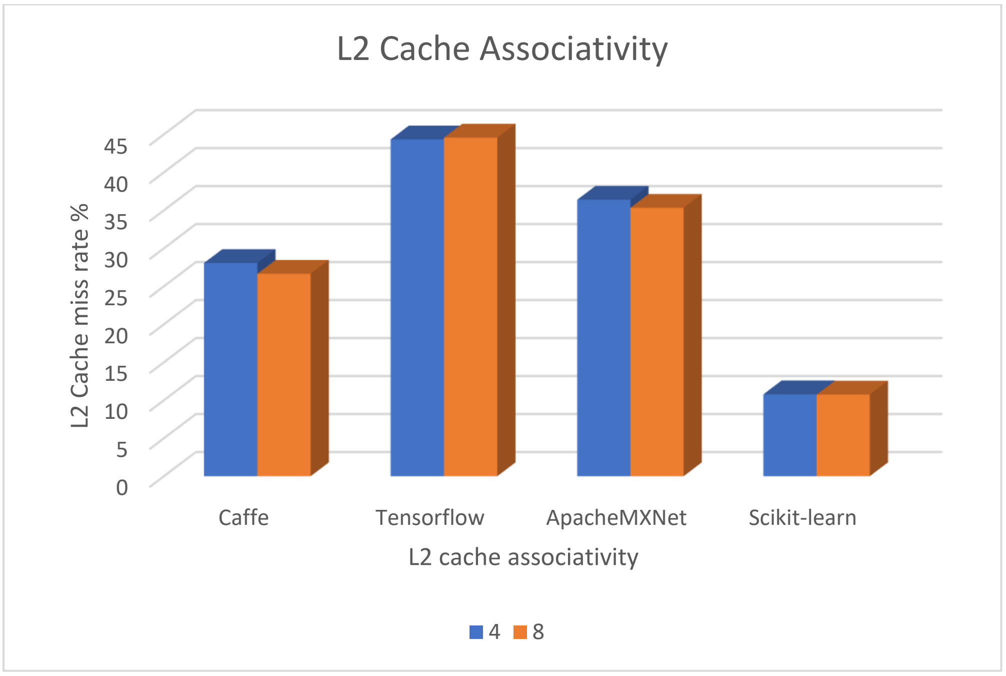

Figure 10: Cache Associativity vs Miss rate

4.5 Branch Predictors: Bi-Mode and Tournament branch predictors miss prediction rates were very similar for all 4 frameworks, as show in figure 11. This is due to the fact that majority of the condition branches produced are result of the for loop used for training and testing, where both the branch predictors are proficient. On the other hand, indirect branch predictor shows a very high miss rate (target address entry not found in the branch target cache). Whereas, SPEC benchmarks had an average miss rate of around 2%. Caffe shows a miss rate of 23.78%, TensorFlow 44.78%, MxNet 50.94%, and Scikit-learn 53.14%.

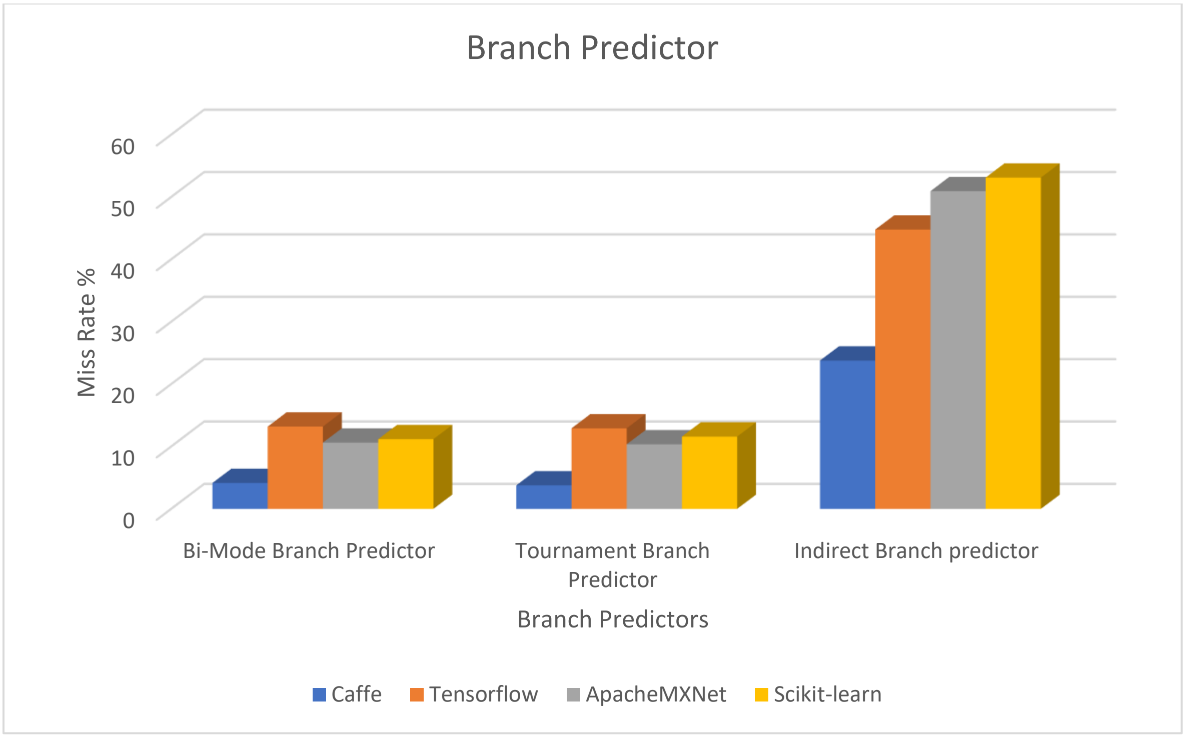

Figure 11: Branch Predictors vs Miss rate

The implementation of CNN layers in any deep learning platform uses library functions. As each layer is a library function, and each image needs to be processed by each layer, this generates a massive number of indirect branches. In the four platforms that were considered, around 45% of the total branches generated were indirect branches. Due to this huge number, correct handling of the indirect branches plays an important role in performance.

The indirect branch predictor implemented in the gem5 system had 256 sets of entries in the branch target address cache, each set being a 2-way set (each hashed index into branch target address cache can store target address for two different tag values). It was observed that more than 40% of indirect branches fetched didn't have an entry of target address in the cache. While the indirect branches which did have an entry of target address had a correct prediction rate of approximately 83%.

To inspect the effect indirect branch target address cache on the miss rate, the set entries were increased to 512, and each set was configured to be 8-way set. As shown in figure 12, the branch target address cache miss rate for TensorFlow decreased to 38.9% from 43.8%. The branch target address buffer used for conditional branch predictor has a size of 4096 entries. As the number of indirect and conditional branches are comparable, increasing the number of unique entries similar to branch target address buffer size would yield a much better hit rate.

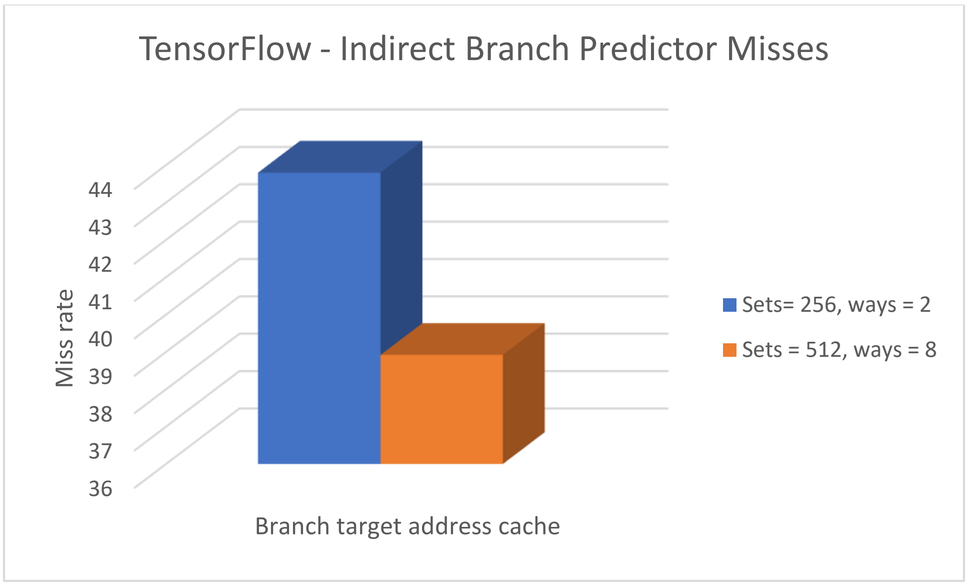

Figure 12: Indirect Branch Target Address cache size vs Miss rate

CHAPTER V

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions: As Machine Learning algorithms become more ubiquitous, specialized hardware have been constructed to meet their ever-growing computing demands. Even though dedicated hardware have been widely successful in meeting these demands, they come at steep costs. On the other hand, general purpose CPU are often neglected in this aspect. Very little work is done to understand the configuration bottle-necks which hold the key to unlocking CPU performance to match the computing standards imposed by Machine Learning.

In this thesis, we have assessed the impact of various CPU configurations on the performance of CNN algorithm. In particular, we have focused on understanding the effect operating frequency, L2 Cache size, branch predictors and size of memory have on CPU performance.

Pushing the frequency to the highest possible limit extracted maximum efficiency from the framework. Increasing the L2 cache size resulted in an increase in the hit rate of input data images. While increasing the L2 cache size resulted in a positive impact, increasing the L2 cache associativity had minimal effect on the miss rate and did not provide any significant improvement. Memory size was shown to have minimal impact on the performance. Indirect branch predictors had huge miss rates. Increasing the branch target address cache size showed a positive effect by decreasing the miss rate.

Frequency scaling showed the highest impact on the performance of CNN algorithm. The CPI was reduced by an average of 0.6 when frequency was scaled 3 times. Memory references constituted of an average of 45% of the total number of instructions simulated, whereas the branch instructions constituted an average of 20% of instructions executed. Even though the memory references constitute a higher fraction of the simulated instructions than branch instructions, around 98% of them are serviced by L1 data and instruction caches. L2 cache misses per kilo instruction was found to be around 5. The impact of the indirect branch predictor was found to be greater than that of L2 cache size.

Despite the fact that the results presented in this work are based on a very basic CNN algorithm, the same can to extrapolated to more complex CNN algorithms. Other CNN implementations would constitute of similar layers as present in hand written digit recognition algorithm, but would be more complex, consisting of much larger hidden layers both in size and numbers. Higher hidden layers imply higher learning parameters which leads to higher memory requirements and much larger working sets, and also higher number of branch instructions. Hence memory and branch predictors play a vital role in CNN algorithm performance.

5.2 Future work: All the experiments presented here were conducted on a single core CPU. We do not explore the effect of multi-core CPU and multi-threaded ML algorithm implementation, on the performance. Our work concentrated on CNN, particularly hand written digit classification algorithm. Other deep learning networks like RNN can also be explored to gain better understanding of the bottlenecks.

REFERENCES

1. Gem5. *http://gem5.org/* *https://gem5.googlesource.com/public/gem5.*

2. TensorFlow. *https://www.tensorflow.org/* *https://github.com/tensorflow/tensorflow.*

3. Caffe. *http://caffe.berkeleyvision.org/.* *https://github.com/BVLC/caffe.*

4. MxNet. *https://mxnet.apache.org/* *https://github.com/apache/incubator-mxnet.*

5. Scikit-learn. *https://scikit-learn.org/stable/* *https://github.com/scikit-learn/scikit-learn.*

6. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. *"The gem5 Simulator"*, International Symposium on Computer Architecture (ISCA) 2011.

7. *https://mxnet.incubator.apache.org/versions/master/tutorials/python/mnist.html, Online.*

8. Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, Mengxian Chi: *"Improving the Performance of Distributed TensorFlow with RDMA",* Int J Parallel Prog (2018) 46: 674. https://doi.org/10.1007/s10766-017-0520-3

9. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., *"Tensorflow: A system for large-scale machine learning,"* in Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)., 2016.

10. Kuo Zhang Salem, Alqahtani, and Murat Demirbas: *"A Comparison of Distributed Machine Learning Platforms"*, 26th IEEE International Conference on Computer Communication and Networks (ICCCN), 2017.

11. Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee: *"Performance Analysis of CNN Frameworks for GPUs",* IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2017.

12. Shaohuai Shi and Xiaowen Chu: *"Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs",* IEEE 16th Int. Conf. on Dependable, Autonomic & Secure Comp, 2018.

13. Rubén D. Fonnegra, Bryan Blair, Gloria M. Díaz: *"Performance comparison of deep learning frameworks in image classificationproblems using convolutional and recurrent networks"*, IEEE Colombian Conference on Communications and Computing (COLCOM), 2017.

14. Saritha. Kinkiri and Wim J.C.Melis: *“Reducing Data Storage Requirements for Machine Learning Algothims using Principle Component Analysis”,* International Conference on Applied System Innovation (ICASI), 2016.

15. Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, Tianshi Chen: *“Cambricon: An instruction set architecture for neural networks,”* in Proc. International Symposium on Computer Architecture (ISCA), 2016.

16. Dongjoo Shin, Jinmook Lee, Jinsu Lee, Juhyoung Lee, Hoi-Jun Yoo*: “An energy-efficient deep learning processor with heterogeneous multi-core architecture for convolutional neural networks and recurrent neural networks”,* IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), 2017.

17. Seung-Hwan Lim, Steven Robert Young, Robert Patton: *“An analysis of image storage systems for scalable training of deep neural networks”*, Workshop on Big Data Benchmarks, 2016.

18. J. Li, C. Zhang, Q. Cao, C. Qi, J. Huang and C. Xie: *"An Experimental Study on Deep Learning Based on Different Hardware Configurations"*, International Conference on Networking, Architecture, and Storage (NAS), 2017.

19. Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N. E., And Moshovos, A.: *“Cnvlutin, ineffectual-neuron-free deep neural network computing”,* International Symposium on Computer Architecture (ISCA) 2016.

20. Young Jong Mo, Joongheon Kim, Jong-Kook Kim, Aziz Mohaisen, and Woojoo Lee: *“Performance of Deep Learning Computation with TensorFlow Software Library in GPU-Capable Multi-Core Computing Platforms”,* IEEE International Workshop on Machine Intelligence and Learning (IWMIL), Milan, Italy, 4 July 2017.

21. Gem5 tutorials *http://www.gem5.org/docs/html/classBiModeBP.html, Online.*

22. Gem5 tutorials *http://www.gem5.org/docs/html/classTournamentBP.html, Online.*

23. *https://en.wikipedia.org/wiki/MNIST_database, Online.*

24. *https://www.spec.org/cpu2006/, Online.*

APPENDIX A

GEM5 FULL SYSTEM SIMULATION SET-UP

This appendix covers all the steps required to set-up a simulation environment to run CNN workloads.

A.1 Installing gem5

a) Install required dependency packages

   sudo apt install build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev python-dev python

b) Get the source code

   git clone https://gem5.googlesource.com/public/gem5

c) Modify source code to include the required branch predictor

   Edit file gem5/src/cpu/simple/BaseSimpleCPU.py and add the appropriate branch predictor.

d) Build gem5 binary (2 binaries need to be built – one for BiMode branch predictor and another for Tournament branch predictor)

   scons build/<CONFIG>/gem5.opt -j<N>

   where <CONFIG> is the gem5 build configuration available in build_opts folder specifying the ISA and the coherence protocol, and <N> is the number of threads used for compilation.

A.2 Building Linux kernel

a) Download Linux kernel version 4.8.13 from https://www.kernel.org/

b) Download the config file from http://www.lowepower.com/jason/files/config

c) Build Linux kernel using make.

A.3 Creating disk image

a) Create an empty disk image

b) Mount the created disk image

sudo mount -o loop,offset=<VALUE> <PATH_TO_DISK_IMAGE> <PATH_TO_FOLDER_WHERE_IMAGE_WILL_BE_MOUNTED>

where VALUE is the product of start value of partition and sector size of the disk image.

c) Download the Ubuntu core release 16.04 'ubuntu-base-16.04-core-amd64.tar.gz' from

http://cdimage.ubuntu.com/ubuntu-base/releases/16.04/release/

d) Copy Ubuntu core files onto the disk image

sudo tar xzvf ubuntu-base-16.04-core-amd64.tar.gz -C <PATH_TO_FOLDER_WHERE_IMAGE_WAS_MOUNTED>

A.4 Install required frameworks on the disk image

a) copy /etc/resolv.conf onto the new disk

b) Update init script - Using precompiled binaries

   a. wget http://cs.wisc.edu/~powerjg/files/gem5-guest-tools-x86.tgz
   b. tar xzvf gem5-guest-tools-x86.tgz
   c. cd gem5-guest-tools/
   d. sudo ./install

c) Change the root directory to directory where image was mounted (say mnt)

   a. sudo /bin/mount -o bind /sys mnt/sys
   b. sudo /bin/mount -o bind /dev mnt/dev
   c. 5sudo /bin/mount -o bind /proc mnt/proc
   d. sudo /usr/sbin/chroot mnt /bin/bash

d) Install the required frameworks using apt package manager.

e) After all installations unmount all of the directories, we used bind on.

   a. sudo /bin/umount tmp/sys
   b. sudo /bin/umount tmp/dev
   c. sudo /bin/umount tmp/proc
   d. sudo umount tmp

A.5 Running gem5 full system simulation

a) Create directory structure as below (required by gem5)

```
<FULL_SYSTEM_IMAGE_PATH>/
+ binaries/
        + vmlinux
+ disks/
        + linux-x86.img
```

b) Set environment variable M5_PATH

```
export M5_PATH=<FULL_SYSTEM_IMAGE_PATH>
```

c) Create a script file <name>.rcS with commands needed to start execution of CNN workload on the simulated system.

d) Launch gem5 full system simulation

```
./build/X86/gem5.opt –outdir=<OUTDIR_PATH> configs/example/fs.py --disk-
image=<path_to_disk_image>
--kernel=<path_to_kernel_binary> --script=<name.rcS>
```

Other system configuration parameters can also be added as arguments.