THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# SCALABILITY STUDY OF MACHINE LEARNING ALGORITHM TEACHING AND INFERENCE ON CENTRAL PROCESSING UNITS

SWASTI MEHRA
SPRING 2020

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree in
Computer Engineering with honors in
Computer Engineering.

Reviewed and approved* by the following:

Anand Sivasubramaniam
Distinguished Professor of Computer Science and Engineering
Thesis Supervisor

Chitaranjan Das
Distinguished Professor of Computer Science and Engineering
Honors Advisor

* Electronic approvals are on file.

# ABSTRACT

Machine Learning is widely used in academics and industry to study patterns, give recommendations, and build statistical models. We need scalable infrastructure to train models quicker as well as fit into the working memory of the training device. Vertical scaling is expensive, so even with expensive and big machines with lots of memory, it would be more efficient by cost to use many small machines. In our research we characterize the resource utilization of machine learning workloads during the teaching and inference phases. This will further help us understand how to improve a systems capability to run such workloads more efficiently and realize how to optimize the system for these workloads. In addition, we will be able to determine the limiting factor in the performance of the machine learning teaching and inference models by running them on a standard system while using profiling tools to see the load on memory, CPU, and power. We can then analyze how different methods of handling system processes (such as page replacement, threading, etc.) affect the performance of the workload. Afterwards, we will determine whether core scaling and memory binding play an important role in determining the scalability of machine learning model training. Finally, we will investigate if hyperthreading improves performance for model training, and whether the same applies to inter-operator parallelism.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGEMENTS

I want to thank Dr. Anand Sivasubramaniam for introducing me to the topic of Machine Learning and characterizing process information of these workloads on a system, while understanding system architecture. I also want to thank Dr. Chitaranjan Das for his valuable inputs on the scope of the research and for always pointing me in the right direction as an advisor. In addition, I would like to thank Mr. Adithya Kumar, graduate student at Penn State University, for mentoring me throughout my research. He shared his expertise by talking through difficult concepts with me and guiding me through the problem-solving process. Kumar gave active feedback on my research, which proved to be vital throughout the process.

# Chapter 1
# Introduction

Machine Learning is a method of data analysis that automates the process of model building and inference with minimal human interaction. Machine Learning models are iterative and are exposed to large amounts of data to independently learn and recognize patterns and learn from previous computations to produce reliable, repeatable decisions and results. Machine Learning is not a new science but has gained a new momentum due to our ability now to iteratively apply complex mathematical computations on big data. [1]

It is essential for the success of machine learning models to be scalable. To make a machine learning algorithm useful in real life (like playing a game of chess, generate real faces, recognize images) requires training data in massive amounts (hundreds of GB) and very high processing power on specialized hardware. With the spread of the internet, the amount of data an average internet user creates is exponentially rising, as well as the amount of data that can be stored on a piece of hardware is getting cheaper and compact, following Moore's law [2]. This increase in the amount of information available provides data to be leveraged in order to train models further.

We need scalable models to train models quicker as well as be able to fit into the working memory of the training device. Vertical scaling is expensive, so instead of having one expensive and big machine that contains lots of memory, it would be more efficient by cost to use many smaller machines to accomplish the same task. [3]

Data is iteratively fed into the training algorithms, the data access and transport speed for this is effected by memory management, data is transport mechanisms from storage to registers for the algorithm to access, as well as the von Numann bottleneck [4] can impact performance. The highly iterative training process can be parallelized, however allocating more computing resources to reduce training time is not efficient. To scale computations in machine learning the system must run fast matrix multiplication with less power consumption. There is also specialized hardware like ASICs (Applied Specific Integrated Chips), for increased performance. CPUs are scaler processors, GPUs are vector processors and ASICs are matrix processors.

We aim to characterize the resource utilization of machine learning workloads during the teaching and inference phase. This will further help us understand how to improve a systems' capability to run such workloads more efficiently and realize how to optimize for scalable model training. We use TensorFlow as the machine learning library to run the tests, characterize and understand the load on the different components of the system. We will further attempt to find parameters that would help optimize the system for a machine learning workload. We intend to find the limiting factor in the performance of the Machine Learning teaching and inference models by running them on a standard system and using profiling tools to see the load on memory, CPU, and power. We also analyze how different methods of handling system processes (like page replacement, threading etc.) affect the performance of the workload.

# Chapter 2

# Literature Review

## Model Training

Machine Learning aims to leverage data, algorithms and statistics to perform specific tasks without using explicit instructions and minimal human interference in the long run. In the initial stages this requires data collection and model building. A model in machine learning is simply the question answering system, this is created in the training process. Training a model simply means learning from the determining features and reducing the loss to have an accurate answer (model output) every time. In supervised learning, for example a model learns from training data to adjust weights to minimize loss. Loss is the penalty of a bad prediction, the goal of training a model is to find a set of weights that have low loss, on average, across most samples, this process is iterative [5] [6].

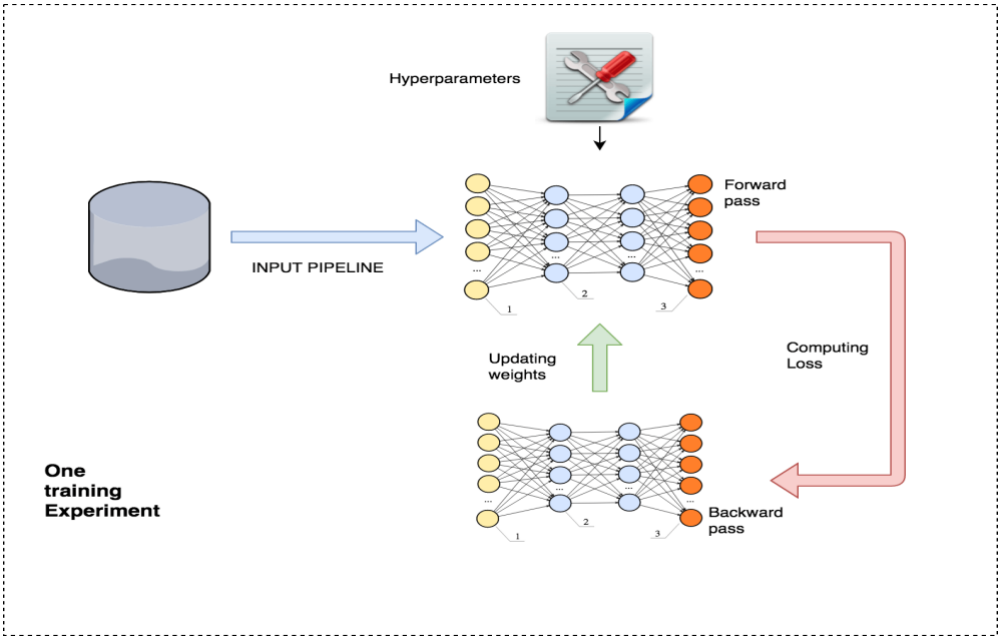



**Figure 1. Model Training Workflow**

The model training loop is also the time when model hyperparameters are introduced into the predictions, these are used to estimate model parameters, they are specified by the practitioners and use heuristics as a starting point in determining the weights of the neural network connections in order to reconcile the differences between the actual and predicted outcomes for subsequent forward passes, predictions and loss calculations [6] [7].

## Neural Networks

Neural Networks are a set of algorithms that are modeled after the human brain, designed to recognize patterns, to cluster, label and classify input data to produce predictions as output. The neural network layers are made up of nodes, a node is a basic unit of computation, it combines the input data with weights to assign weightage/significance to an input source or characteristic. These weighted inputs are summed and passed through activation functions that determine the importance and accuracy of the input signal [9].

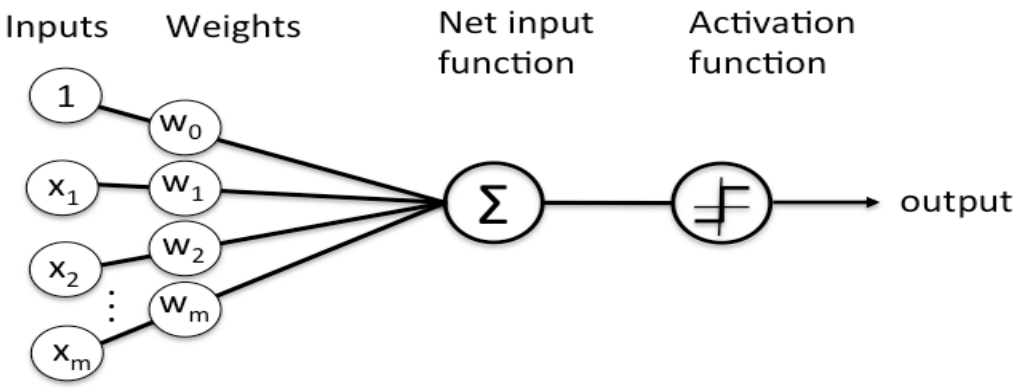



**Figure 2. A Single node in a Neural Network**

## Non-Uniform Memory Access

Memory from various points in the address space have different performance characteristics. A systems memory is hierarchical and systems require modified operating system kernels with NUMA support to understand the topological properties of system memory. Fast systems require memory at each socket, and memory access across socket memories is inefficient as it adds additional latency since it requires the traversal of the memory interconnect first. Proper placement of data improves bandwidth and latency to memory. Future generations of computers will have increasing differences in performance depending on the position of the core on the die relative to the controller, making new functionality in operating systems to access these different kinds of memory very important.

While training a Machine Learning model, a lot of data is ingested and iteratively accessed; having efficient memory access can make a significant difference in overall performance, as will be observed in our experiments in this paper. The NUMA system classifies memory into NUMA nodes, and each node has an affinity to processors and devices that can use memory on the NUMA node with the best performance. Below is an example of a NUMA system with 2 NUMA nodes, 8 processors and an interconnect between each node [10].

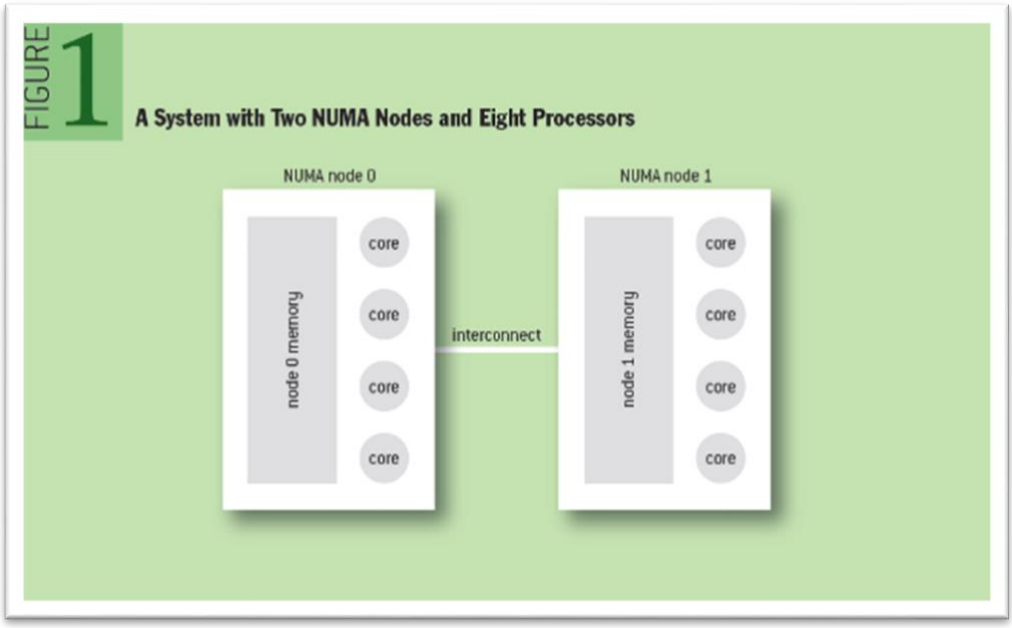



**Figure 3. NUMA System with 2 nodes and 8 cores**

## Wide and Deep Learning

A wide and deep model jointly trains a wide linear model and a deep neural network for memorization and generalization, to bring us one step closer to how a human brain learns to generalize memorized information and remember exception rules. Memorization is learning frequent co-occurrence of features and exploiting the correlation on the basis of historical data. Generalization on the other hand is based on transitivity of data, and explores new feature combinations that are not in the history. Wide and deep models have been found to be very effective for recommender systems, this recommender system is productionized on Google Play. The Wide and Deep models are trained every time a new set of training data arrives, there are optimizations in place such as a warm starting that initializes the new model with the embeddings and linear model weights of the old model, however a model training still requires

the input layer to take the new training data, concatenating the embeddings with dense features and passing this through the ReLU and logistic loss calculation unit [11] [12].

## Inter and Intra Operation Parallelism

TensorFlow provides configurations to select operation parallelism settings using inter-operation and intra-operation parallelism. The intra-op-parallelism runtime setting controls parallelism inside an operation, it is for operations that can be parallelized internally, for example matrix multiplication. TensorFlow will schedule tasks in a thread pool with the specified number of threads. Intel recommends setting this environment variable to the number of physical cores in the system.

The inter-operation parallelism controls the parallelism among independent operations. According to Intel this variable is recommended to be set to the number of parallel paths in the model, empirical testing is the best way to adjust the inter-operation thread pool size for the specific model [15].

## Scale Speedup

The parallel run time is defined as the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on $p$ processors ($S = T_S/T_P$). The efficiency is defined as the ratio of speedup to the number of processors. Efficiency

measures the fraction of time for which a processor is usefully utilized ($E = \frac{S}{P} = T_S/pT_P$). The cost of solving a problem on a parallel system is defined as the product of run time and the number of processors. A cost-optimal parallel system solves a problem with a cost proportional to the execution time of the fastest known sequential algorithm on a single processor. Scalability is a measure of a parallel system's capacity to increase speedup in proportion to the number of processors [14].

For our experiments we aim to compare the change in accuracy to the change in performance to determine the speedup as we scale the model. We aim to determine an optimal scale speedup which produces above acceptable accuracy with reduced processing power and time as simply increasing processing power to improve performance is not a viable approach since there is a hard upper limit to increasing number of processors and it is also cost heavy.

## Memory Interleaving

Memory interleaving is a technique used to compensate for the relatively slower access time to DRAM. The main memory is divided into banks where memory is stored in a round robin fashion to be able to access memory individually without any dependency on other memory calls. In general, the CPU is more likely to need to access the memory for a set of consecutive words (either a segment of consecutive instructions in a program or the components of a data structure such as an array, the interleaved (low-order) arrangement is preferable as consecutive words are in different modules and can be fetched simultaneously. In case of high-order arrangement, the consecutive words are usually in one module, having multiple modules is not helpful if consecutive words are needed [16].

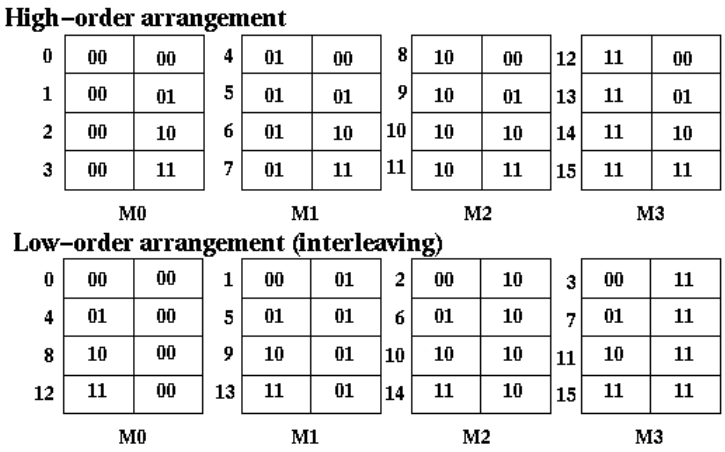



**Figure 4. Memory Interleaving Example**

# Chapter 3

# Tools Overview

## Experiment System's NUMA configuration

Our system for the experiments described in the founding's of this thesis is the Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz. It is a 2-socket machine with 6 cores per socket and 2 threads per core with hyperthreading enabled. A core is the smallest independent unit that implements a general-purpose processor and a processor is an assembly of cores. Our machine has 24 processors numbered 0 to 23. Hyperthreading allows for concurrent scheduling of 2 processes per core due to which we get effectively 12 cores, the operating system sees 12 cores and assigns processes accordingly, but there are only 6 hardware cores.

Each socket has a NUMA node associated with it in our system, this is not necessary for all systems, processor configurations and memory configurations together impact the overall performance of the system. For us, since each physical socket has a NUMA node associated with it, it is most efficient for the processes running on a physical socket to access memory from the same NUMA node, accessing data from the other node adds compute time due to the limited memory bandwidth and added latency of accessing memory not on the chip.

The command 'lscpu -e' gives a quick view of the processors on the physical chip and the hyperthreaded processors. CPU 0 and CPU 12 are on node 0, socket 0 and core 0 – this shows that they are on the physical core on the chip and have the same closest physical memory that is fastest to access. These CPUs use the same hardware resources, so it is capable of concurrently scheduling processing on CPU0 and CPU12 but they use the same physical resources.

```
wattson09.cse.psu.edu 435% lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE MAXMHZ    MINMHZ
0   0    0      0    0:0:0:0       yes    2000.0000 1200.0000
1   1    1      1    1:1:1:1       yes    2000.0000 1200.0000
2   0    0      2    2:2:2:0       yes    2000.0000 1200.0000
3   1    1      3    3:3:3:1       yes    2000.0000 1200.0000
4   0    0      4    4:4:4:0       yes    2000.0000 1200.0000
5   1    1      5    5:5:5:1       yes    2000.0000 1200.0000
6   0    0      6    6:6:6:0       yes    2000.0000 1200.0000
7   1    1      7    7:7:7:1       yes    2000.0000 1200.0000
8   0    0      8    8:8:8:0       yes    2000.0000 1200.0000
9   1    1      9    9:9:9:1       yes    2000.0000 1200.0000
10  0    0      10   10:10:10:0    yes    2000.0000 1200.0000
11  1    1      11   11:11:11:1    yes    2000.0000 1200.0000
12  0    0      0    0:0:0:0       yes    2000.0000 1200.0000
13  1    1      1    1:1:1:1       yes    2000.0000 1200.0000
14  0    0      2    2:2:2:0       yes    2000.0000 1200.0000
15  1    1      3    3:3:3:1       yes    2000.0000 1200.0000
16  0    0      4    4:4:4:0       yes    2000.0000 1200.0000
17  1    1      5    5:5:5:1       yes    2000.0000 1200.0000
18  0    0      6    6:6:6:0       yes    2000.0000 1200.0000
19  1    1      7    7:7:7:1       yes    2000.0000 1200.0000
20  0    0      8    8:8:8:0       yes    2000.0000 1200.0000
21  1    1      9    9:9:9:1       yes    2000.0000 1200.0000
22  0    0      10   10:10:10:0    yes    2000.0000 1200.0000
23  1    1      11   11:11:11:1    yes    2000.0000 1200.0000
wattson09.cse.psu.edu 436%
```

**Figure 5. CPU and core configurations on the physical sockets**

Reading the /proc/cpuinfo file we can understand the processor placements on the physical chip and the NUMA node associated with each socket. There are 6 cores per socket, with 12 threads in a socket. CPU 0,2,4,6,8,10,12,14,16,18,20 and 22 are on socket 0 and use NUMA node 0 while CPU 1,3,5,7,9,11,13,14,15,17,19,21 and 23 are on socket 1 and use NUMA node 1. A processor or CPU is a logical CPU, this depicts the view of the operating system showing how many processes can be scheduled simultaneously.

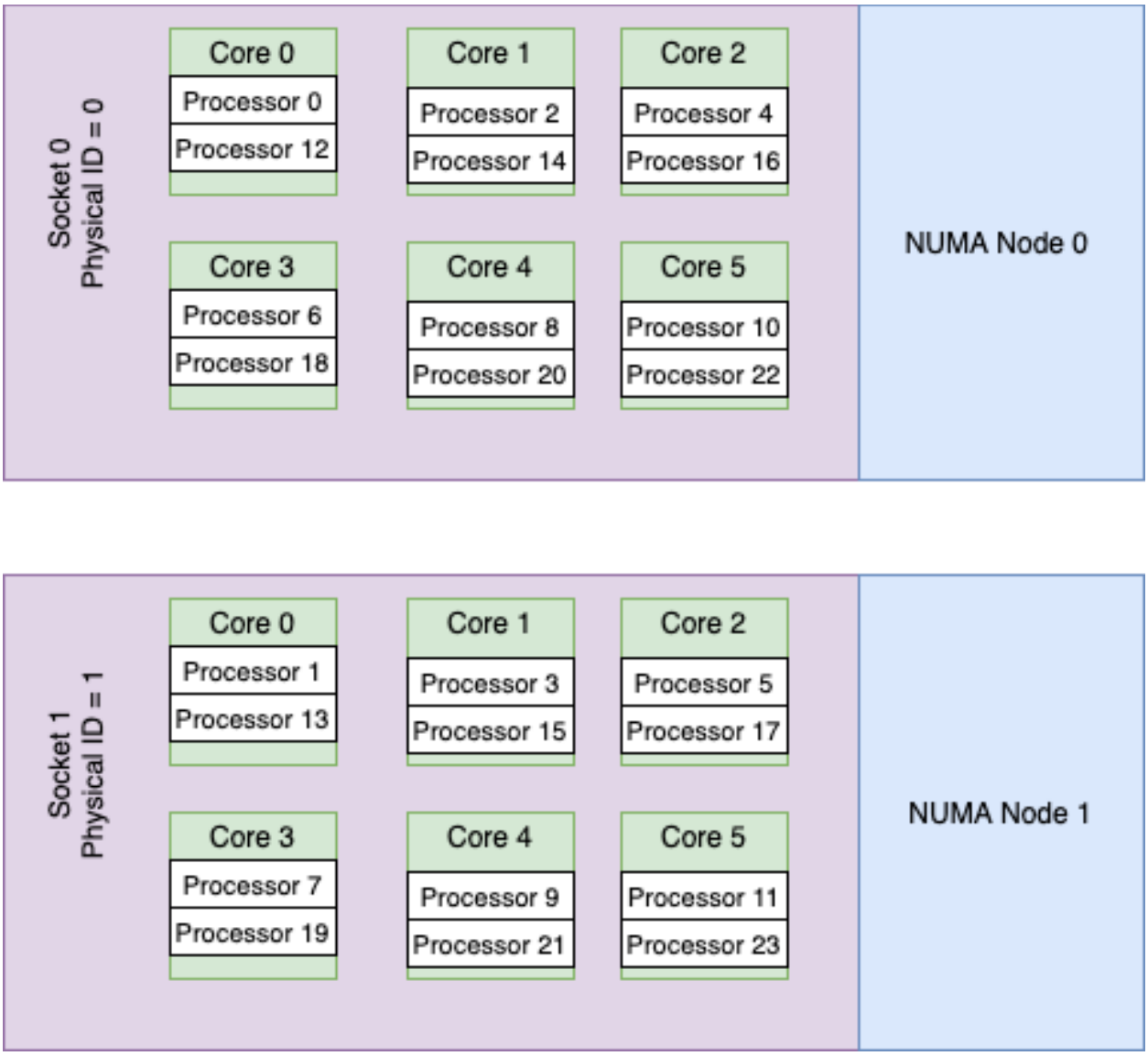



**Figure 6. Logical CPU placement on the physical chip on sockets**

**Datasets**

For our experiment we used two different datasets to get a basic understanding of the topic and build motivation towards the exact parameters we used in the experiments presented in this thesis. The first dataset used is the Fashion-MNIST that contains 60,000 training samples and 10,000 testing samples of pieces of clothing. Each example of a greyscale image and has a

label identifying the type of clothing item. The Fashion-MNIST dataset is a commonly used dataset for image recognition models, so we wanted to start our scalability studies with this dataset. This is an example of supervised learning since the training data provides the image as well as the associated label that classifies the item in the image. Using the TensorFlow library in Python to build the model to run these predictions the accuracy results are quite favorable, so it is a good Machine Learning dataset to study, since the model returns the correct class with a high-confidence accurately.

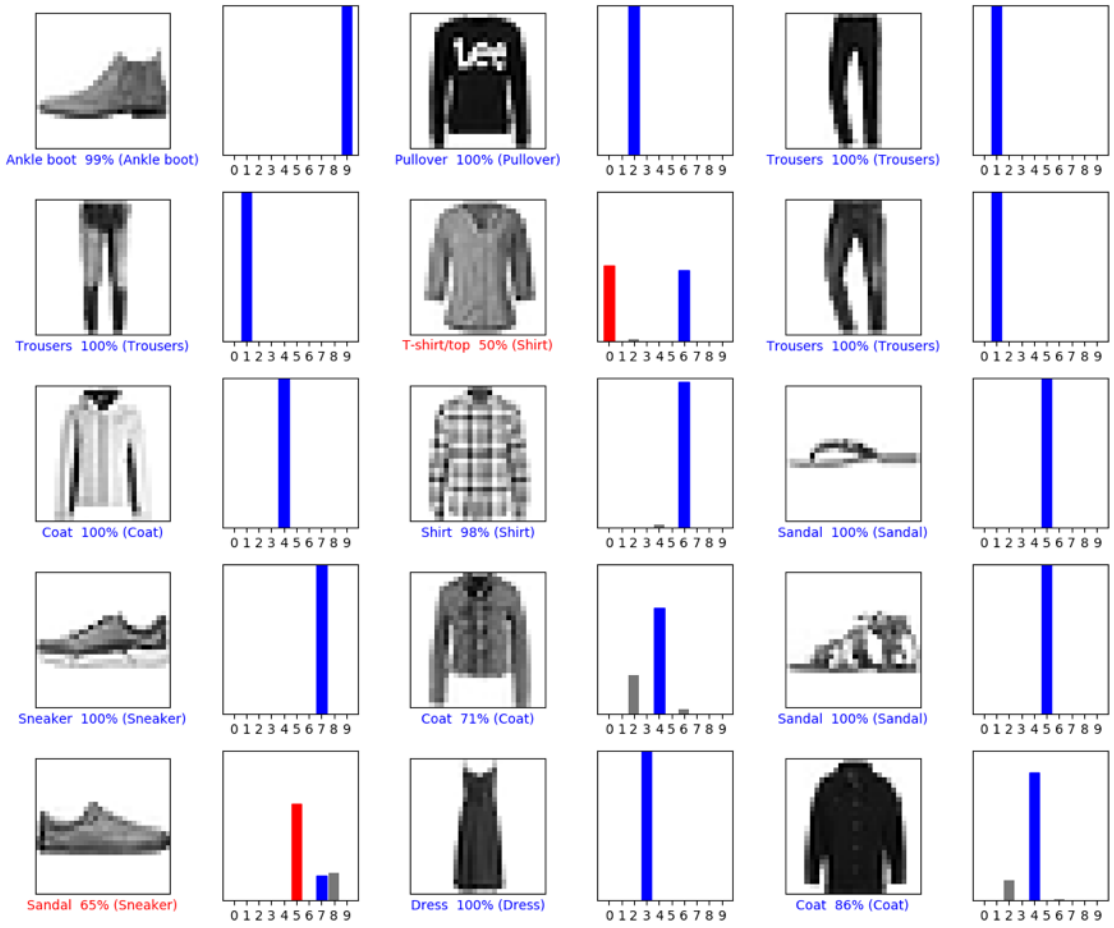



**Figure 7. Fashion-MNIST Model Predictions**

The Wide and Deep model uses the Census Income Dataset [15] containing over 32,000 training and 16,000 testing samples with attributes including age, occupation, education, and

income bracket. We use the wide and deep model that predicts the income label, the wide model can memorize interactions with data with a large number of features like these, and the deep model can generalize the data, the wide and deep combines the two and can also learn exceptions. For the purposes of this example, the model builders chose the Census Income Data Set to allow the model to train in a reasonable amount of time. They claim that the deep model performs almost as well as the wide and deep model on this dataset. The wide and deep model truly shines on larger data sets with high-cardinality features, where each feature has millions/billions of unique possible values (which is the specialty of the wide model) [16].

## Optimizers

Optimizers help minimize (or maximize) an Objective Function (Error function) that are used to set and update the weights and biases in the direction of the optimal solution. It is the optimizer that makes changes to the weights of the model using the direction from the loss function calculations. First, The Optimizer class is initialized with given parameters, but no Tensor is created. In a second step, invoking get_tensor method will actually build the TensorFlow Optimizer Tensor, and return it [18]. TensorFlow provides a variety of optimizers in its libraries to use out of the box and also provides functionality to write custom optimizers for the models. In our research we made use of the Adam and Stochastic Gradient Descent optimizers.

# Chapter 4

## Experiment: Performance across CPUs

We expect that increasing processing power and assigning more CPUs to train our model should improve the performance. The model training process involves a lot of computation and calculations in the ReLU, loss calculations, and weight calculation stages, leveraging parallel processing to speed up each computation should positively impact the overall performance. This is the basic instinct behind parallel processing when a single program is broken up into independent parts and the results of these parallelly computed individual parts are merged for an overall faster solution. The cost versus performance is an important measurement to consider as system resources are limited, and processing power cannot be arbitrarily increased in real life.

Another important consideration for parallelization is how efficiently can the task be done concurrently, as dependencies add serial requirements and are inefficient to compute parallelly as they depend on conditions upstream that must be fulfilled first. Also, data dependencies across parallel processes on different nodes in the CPU requires communication across nodes that can be very expensive on computation cycles so they must be minimized and overlapped with computation [15]. The reason that compute capacities have increased much more than memory bandwidth following Moore's law is that program memory is usually much bigger in terms of hardware required than the processor. Faster components are more expensive to produce, consume more power and require newer technology, that means that the CPU can be made faster than the memory while keeping within a budget [16].

For effective parallelization, programmers need to consciously build the program to take advantage of parallelization and write custom code to be parallelized to avoid race conditions and upstream dependencies that could essentially make the parallelization counter intuitive by actually adding processing overheads while also using more resources, TensorFlow can take uninformed Python code written using the TensorFlow library and turn it into appropriately parallelized code that can take advantage of a GPU or TPU(Tensor Processing Units). This can be automated in large part because of machine learning's heavy use of matrix operations, which are easily parallelized [18].

## Fashion MNIST

Keeping the model constant, we change the number of logical CPUs allotted to the process to run or the memory configurations allotted to see the impact on performance. The performance is measured as wall clock time taken to reach required accuracy, this is when the process completes running. Our expectation is that more parallel processing will improve efficiency and reduce time taken, however we want to also observe load on a single hardware core and memory latency between NUMA nodes.

Using the 'numactl' command on Linux we can control the number of processors assigned to the task. Assigning one thread per core in one socket till each core is utilized (6 threads). Then, utilizing hyperthreading to assign 2 threads on the same while staying on the same socket gives us up to 12 threads. Expanding from here to two sockets allows up to 24 CPUs to parallelly schedule the processes of model training.

Table 1. Execution time across CPUs assigned without NUMA considerations

| CPUs used | Number of CPUs | Average Time (seconds) |
|---|---|---|
| 0 | 1 | 55.70848107 |
| 0,2 | 2 | 52.5390594 |
| 0,2,4 | 3 | 47.76953363 |
| 0,2,4,6 | 4 | 44.62922502 |
| 0,2,4,6,8 | 5 | 42.9579463 |
| 0,2,4,6,8,10 | 6 | 40.50696492 |
| 0,2,4,6,8,10,12 | 7 | 41.41678119 |
| 0,2,4,6,8,10,12,14 | 8 | 40.67656898 |
| 0,2,4,6,8,10,12,14,16 | 9 | 41.29061198 |
| 0,2,4,6,8,10,12,14,16,18 | 10 | 41.83172035 |
| 0,2,4,6,8,10,12,14,16,18,20 | 11 | 41.94293666 |
| 0,2,4,6,8,10,12,14,16,18,20,22 | 12 | 41.95351219 |
| 0-2,4,6,8,10,12,14,16,18,20,22 | 13 | 42.42555094 |
| 0-4,6,8,10,12,14,16,18,20,22 | 14 | 41.99365187 |
| 0-6,8,10,12,14,16,18,20,22 | 15 | 42.63701653 |
| 0-8,10,12,14,16,18,20,22 | 16 | 44.13888478 |
| 0-10,12,14,16,18,20,22 | 17 | 43.61157393 |
| 0-12,14,16,18,20,22 | 18 | 43.61488366 |
| 0-14,16,18,20,22 | 19 | 45.28833652 |
| 0-16,18,20,22 | 20 | 45.24622822 |
| 0-18,20,22 | 21 | 45.14145947 |
| 0-20,22 | 22 | 44.76190948 |
| 0-22 | 23 | 45.73434758 |
| 0-23 | 24 | 45.05802155 |

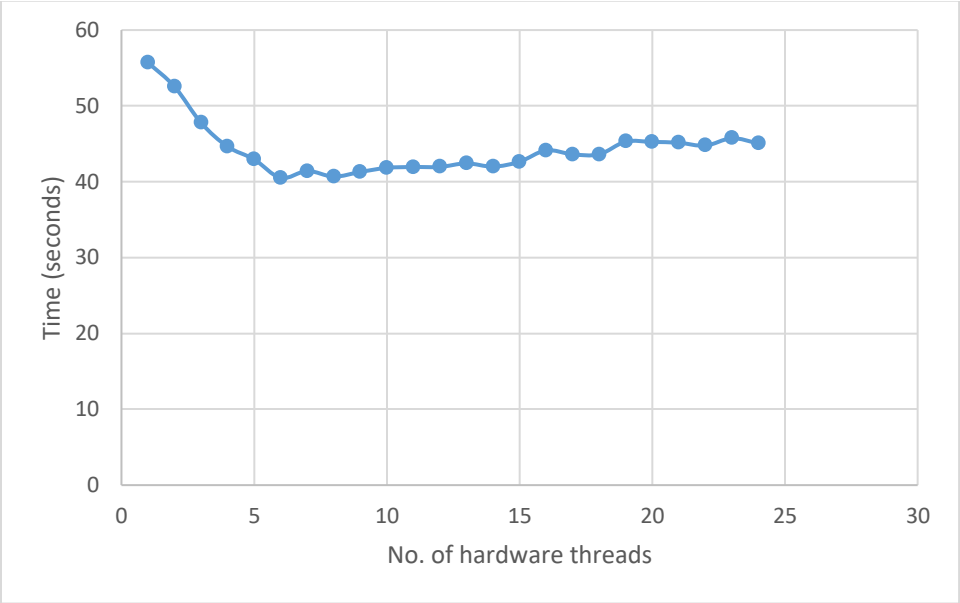



**Figure 8. Execution Time vs. number of threads without NUMA considerations**

We observe a 37% improvement in the time required to train the model from using only one thread on one core to using 6 threads on 6 cores on the same socket, which is also the most time efficient running configuration for CPU assignments. Assigning further after the one process per core configuration increases training time as the threads are now sharing the hardware core for physical resources, and moving to a new socket increase latency due to limited memory bandwidth to communicate between the nodes.

A point of interest on the graph is the peak at 7 hardware threads being used, the logical CPUs in use are: 0,2,4,6,8,10 and 12. CPUs 0 and 12 are on the same core and therefore share hardware, assigning the process to both of these CPUs increases training time. In all NUMA systems, when the memory is located with processor X and the code is running on processor Y (where X & Y aren't the same processor), every memory access will be bad for performance. So, allocating memory on the right NUMA node will certainly help. Therefore, the best training time is achieved at one thread per core on the socket with all cores in the socket being utilized.

This finding motivated us to further investigate the result of assigning threads in the order of cores, therefore only utilizing a new core when all threads in the current core are utilized. Again, using the numactl command we train the same Fashion-MNIST model using cores incrementally to see the impact of cores on training time.

**Table 2. Execution time across CPUs**

| CPUs used | Number of CPUs | Average Time (seconds) |
|---|---|---|
| 0 | 1 | 55.51468379 |
| 0,12 | 2 | 65.93169141 |
| 0,12,2 | 3 | 52.94081931 |
| 0,12,2,14 | 4 | 51.02283266 |
| 0,12,2,14,4 | 5 | 46.65497568 |
| 0,12,2,14,4,16 | 6 | 45.72486472 |
| 0,12,2,14,4,16,6 | 7 | 43.81104424 |
| 0,12,2,14,4,16,6,18 | 8 | 43.69985437 |
| 0,12,2,14,4,16,6,18,8 | 9 | 42.69280622 |
| 0,12,2,14,4,16,6,18,8,20 | 10 | 42.47134326 |
| 0,12,2,14,4,16,6,18,8,20,10 | 11 | 41.69253724 |
| 0,12,2,14,4,16,6,18,8,20,10,22 | 12 | 41.66431026 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1 | 13 | 42.21683812 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13 | 14 | 42.86210787 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3 | 15 | 42.97455266 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15 | 16 | 43.33942018 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5 | 17 | 43.26101291 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17 | 18 | 43.61898901 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,133,15,5,17,7 | 19 | 43.78118851 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19 | 20 | 43.89095798 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9 | 21 | 43.76108971 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21 | 22 | 43.55454891 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21,11 | 23 | 43.90372281 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21,11,23 | 24 | 43.90343845 |

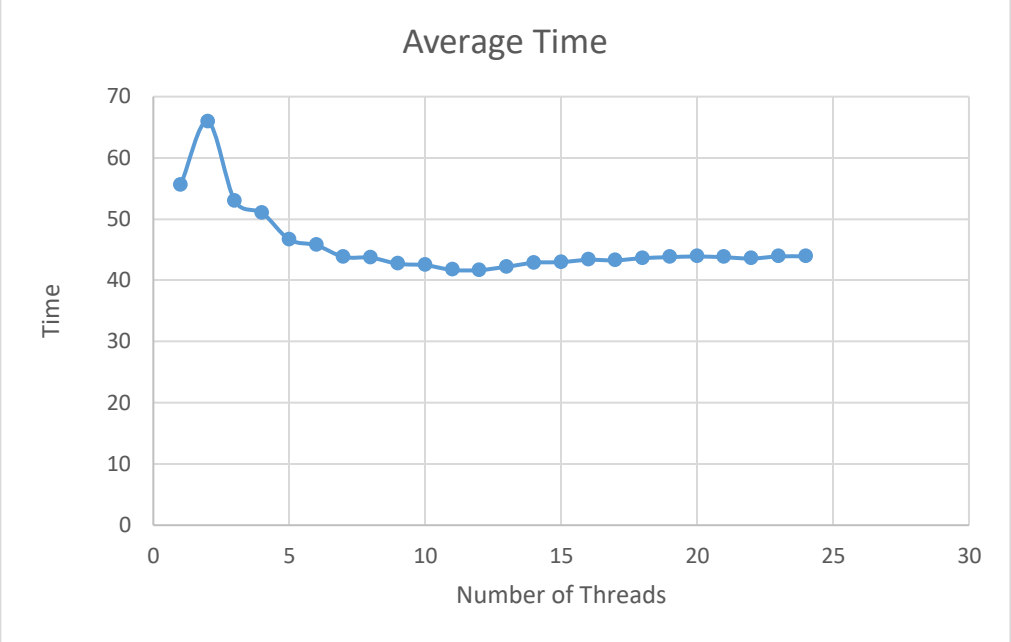



**Figure 9. Execution time vs. number of threads**

A very important point of interest in this experiment is 2 threads getting assigned, there is a distinct peak in training time when 2 threads are assigned on the same core – this is the data point for allocating CPU 0 and 12 – both of these are at physical address 0 core 0. This confirms our understanding from the previous experiment where we saw an increase in training time when using the multiple threads on the same core.

We also observe that the tail end of both experiments doesn't show any significant improvement in performance even though they use the most resources as all cores and threads are being utilized for the task, these tasks are high cost but relatively low efficiency as they are not comparably improved.

To ensure that we are not measuring the training time for an accurate model, we performed the experiments again while measuring the final accuracy and loss along with training time and observed the same trends with a minimum accuracy of 87.77% and a standard deviation of 0.15 in accuracy.

Table 3. Accuracy Loss and Time across CPUs

| CPUs used | Average Accuracy | Average Loss | Average Time |
|---|---|---|---|
| 0 | 88.274 | 33.8277044 | 52.6109275 |
| 0,12 | 88.144 | 34.0828999 | 62.7671298 |
| 0,12,2 | 87.771 | 34.8694614 | 50.1742908 |
| 0,12,2,14 | 87.97 | 34.2152158 | 48.1330197 |
| 0,12,2,14,4 | 88.253 | 33.752413 | 43.5679296 |
| 0,12,2,14,4,16 | 88.009 | 34.0259658 | 42.5894236 |
| 0,12,2,14,4,16,6 | 88.009 | 33.9668157 | 40.7443727 |
| 0,12,2,14,4,16,6,18 | 88.114 | 34.1414777 | 40.6137685 |
| 0,12,2,14,4,16,6,18,8 | 88.287 | 33.4276155 | 39.3068003 |
| 0,12,2,14,4,16,6,18,8,20 | 88.017 | 34.0909901 | 39.3169499 |
| 0,12,2,14,4,16,6,18,8,20,10 | 88.178 | 34.1238776 | 38.6085581 |
| 0,12,2,14,4,16,6,18,8,20,10,22 | 88.291 | 33.4246089 | 38.5505942 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1 | 88.122 | 34.1129593 | 39.0003901 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13 | 87.857 | 34.7241274 | 39.8941121 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3 | 88.277 | 33.6253057 | 40.0978346 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15 | 88.25 | 33.9791908 | 40.4143277 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5 | 88.151 | 33.8184579 | 40.3571455 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17 | 88.042 | 34.1390842 | 40.7782115 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,133,15,5,17,7 | 88.462 | 33.2744858 | 40.6113888 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19 | 88.125 | 34.1160684 | 41.0054312 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9 | 88.246 | 33.6965795 | 41.315187 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21 | 88.07 | 34.079284 | 41.0557889 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21,11 | 88.22 | 34.0417455 | 40.8359526 |
| 0,12,2,14,4,16,6,18,8,20,10,22,1,13,3,15,5,17,7,19,9,21,11,23 | 88.251 | 33.6958681 | 41.7189501 |

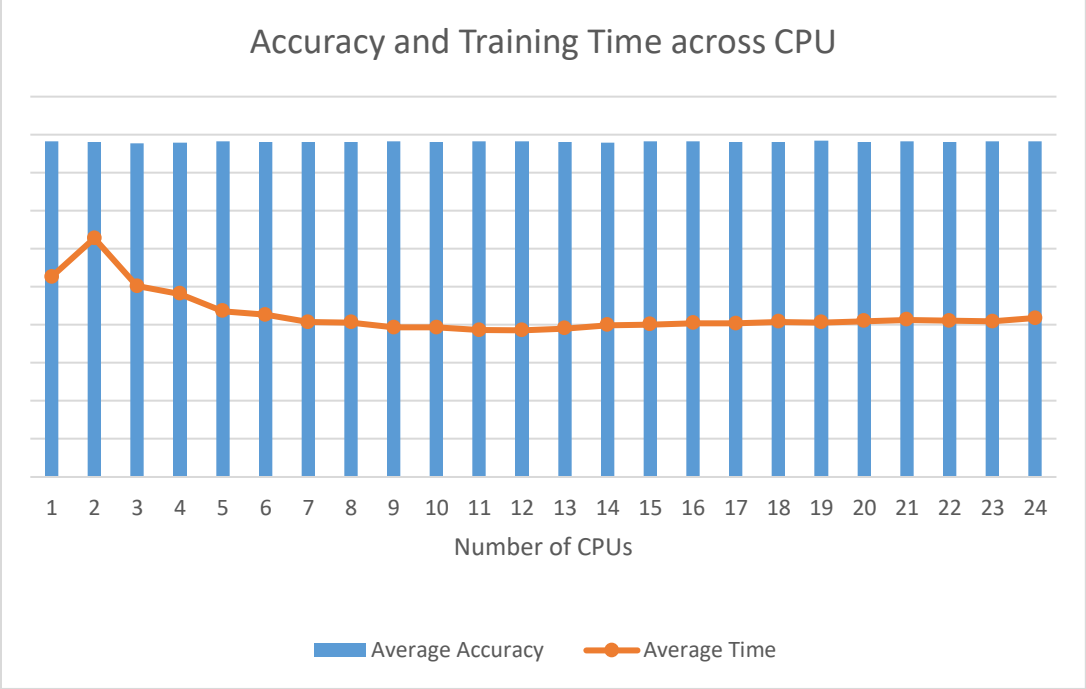



**Figure 10. Accuracy and Time across CPUs for Fashion-MNIST**

## Wide and Deep

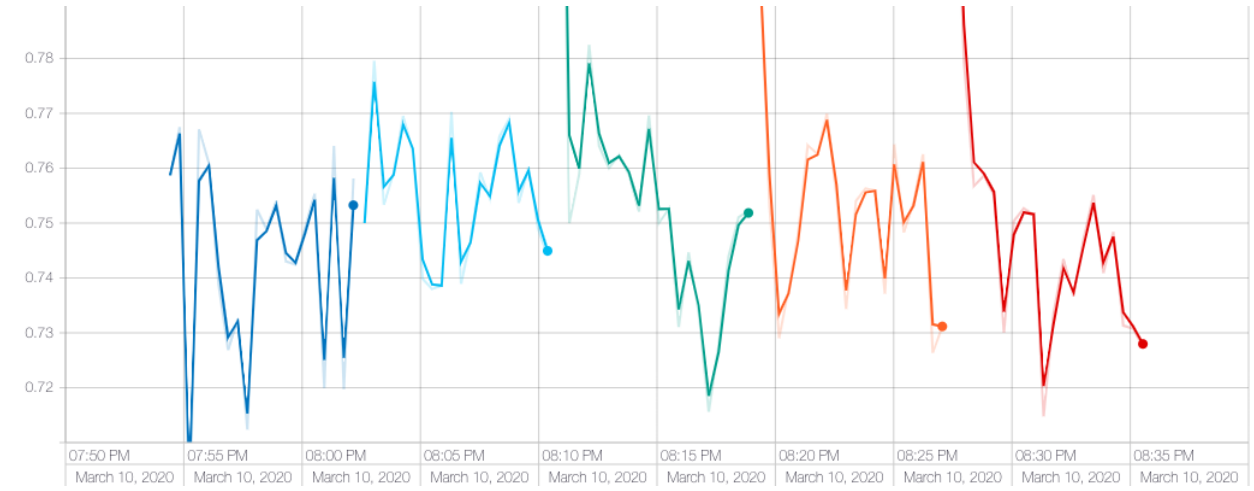
Operators manipulate tensors, i.e. n-dimensional arrays. The parallelism within an operator can be exploited with single instruction multiple data (SIMD), multi-threading and data parallelism, made possible with the intra flag in the TensorFlow library. Further, the parallelism across operators (inter operator parallelism) can be exploited by asynchronous scheduling to place independent operators on different hardware units. Our machine has 2 sockets, therefore 2 distinct hardware units with their own local memory nodes attached to them, each hardware unit has 6 cores and 12 available threads due to hyperthreading. We want to compare the performance of allotting operator parallelism to each core versus each available thread as seen by the Operating System due to hyperthreading. Synchronous scheduling is beneficial in both single-

socket and multi-socket systems, the best performance is achieved by balancing intra- and inter-operator parallelism.

After observing the impact of multithreading and multiple CPU usage on model training with the Fashion MNIST dataset we are motivated to test the impact of controlling CPU usage on the wide and deep model which is different in terms of data type, training set size as well as model training algorithm to see if similar trends exist in different models. Measuring and visualizing the accuracy scalar with TensorFlow logger and Tensor Board to see the accuracy response across inter and intra operation parallelism. Running the wide and deep algorithm with no inter and intra flags (the system picks the appropriate number of intra and inter threads from the thread pool), and testing for 6 intra threads and 12 intra threads to parallelize each operation into the 6 cores or the 12 hyperthreads available per socket and its performance impact. Further parallelizing independent operations using 2 inter operation threads since our test system has 2 sockets we test 6 intra and 2 inter threads as well as 12 intra and 2 inter threads.


**Figure 11. Accuracy for Wide and Deep model**

The intra12inter2 has the steepest accuracy increase proving the fastest accuracy gain over time by parallelizing operations on the 12 CPUs and 2 sockets. The second-best accuracy increase is of intra6inter2 when independent operations are parallelized on the two sockets and each operation is parallelized on hyperthreads. The width of the computation graph quantifies the inter-operation parallelism. Intra parallelism across all hyperthreads has the shortest width quantifying the model's inter-operator parallelism, our model is not highly inter-operator parallelizable as it does not have many parallel dataflow circuits in the graph.



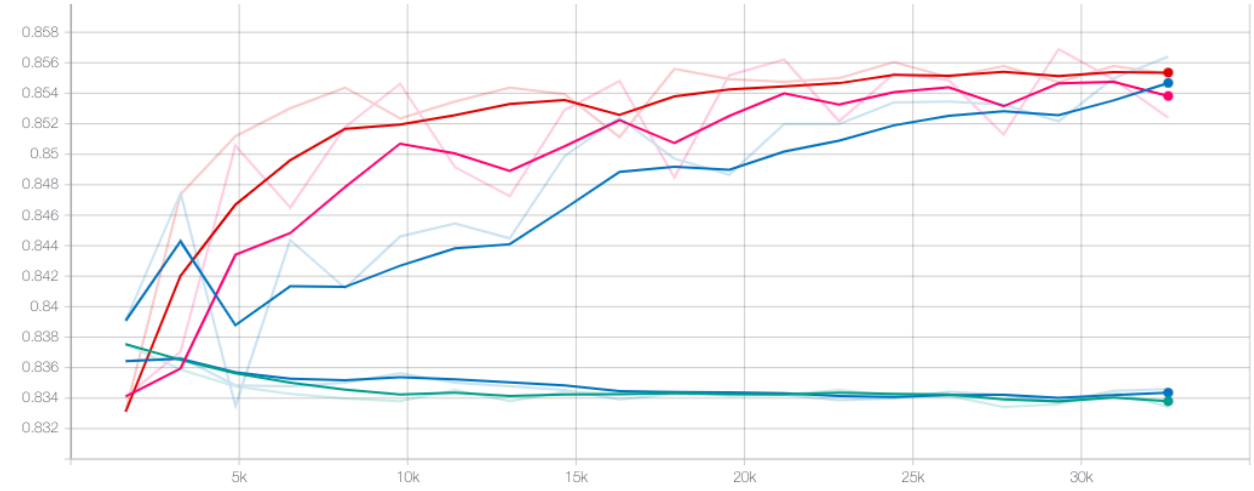

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| intra12/eval | 0.7312 | 0.7311 | 32.58k | Tue Mar 10, 20:27:03 | 7m 42s |
| intra12inter2/eval | 0.728 | 0.7274 | 32.58k | Tue Mar 10, 20:35:32 | 7m 59s |
| intra6/eval | 0.7449 | 0.744 | 32.58k | Tue Mar 10, 20:10:22 | 7m 44s |
| intra6inter2/eval | 0.7519 | 0.7522 | 32.58k | Tue Mar 10, 20:18:51 | 7m 59s |
| wideDeep/noFlags/eval | 0.7533 | 0.7581 | 32.58k | Tue Mar 10, 20:02:09 | 7m 44s |

**Figure 12. Wide and Deep Model Precision and Width Metrics**

Comparing the wide model and deep model with the wide and deep model we aim to correlate accuracy and training time of the best performing CPU configurations of each model

training type, i.e. the intra and inter operator allocation. We see characteristics of the wide model and the deep model combine into the wide and deep model in the precision, loss and accuracy scalars. All training algorithms maintain a minimum accuracy of 83%, but allocating thread pools and hardware resources positively impacts accuracy and the time to accuracy.

There is a tradeoff between training time and accuracy as the wide and deep algorithm takes significantly longer to train but ultimately consistently provides better accuracy even with no thread pool optimization. This particular census data model is biased towards deep learning accuracy and the wide and deep model is able to utilize the efficiency of the wide model that quickly trains to be accurate enough (above 83% accuracy) and the deep model trains for generalization and finding patterns not present in history data.




| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| deep/intra12inter2/eval | 0.8544 | 0.8564 | 32.58k | Tue Mar 10, 21:30:59 | 4m 58s |
| deep/intra6inter2/eval | 0.8539 | 0.8524 | 32.58k | Tue Mar 10, 21:20:26 | 4m 58s |
| wide/intra12inter2/eval | 0.8338 | 0.8334 | 32.58k | Tue Mar 10, 21:04:37 | 5m 40s |
| wide/intra6inter2/eval | 0.8343 | 0.8346 | 32.58k | Tue Mar 10, 20:52:52 | 5m 34s |
| wideDeep/intra12inter2/eval | 0.8553 | 0.8553 | 32.58k | Tue Mar 10, 20:35:32 | 7m 59s |
| wideDeep/wideDeep/noFlags/eval | 0.8538 | 0.8555 | 32.58k | Tue Mar 10, 20:02:09 | 7m 44s |

**Figure 13. Wide and Deep Training Accuracy vs. Time**

Now, seeing that intra12inter2 is giving the best performance on the 2 socket 12 core machine we have, we experiment further with $n$ number of cores allocated across 2 sockets. We set the inter flag to inter=2 allowing for inter-operator parallelization and observe the scaling of the wide and deep model across increasing intra-operator parallelization to essentially understand the parallelizability of the wide and deep model. Further, we observe the scalability across a single hardware chip using all cores for scalability across n-cores in a single socket (restricted hardware resources) and try to find the optimal configuration and compare the differences in the experiments.

The first, sixth and twelfth run are the most interesting and characteristic. They signify the following:

**Two Socket Experiment: operators parallelized on 2 sockets**

One: Core 1 used only (threads 1 and 12).

Six: All six cores on Socket 0 used.

Twelve: All twelve cores on both sockets used.

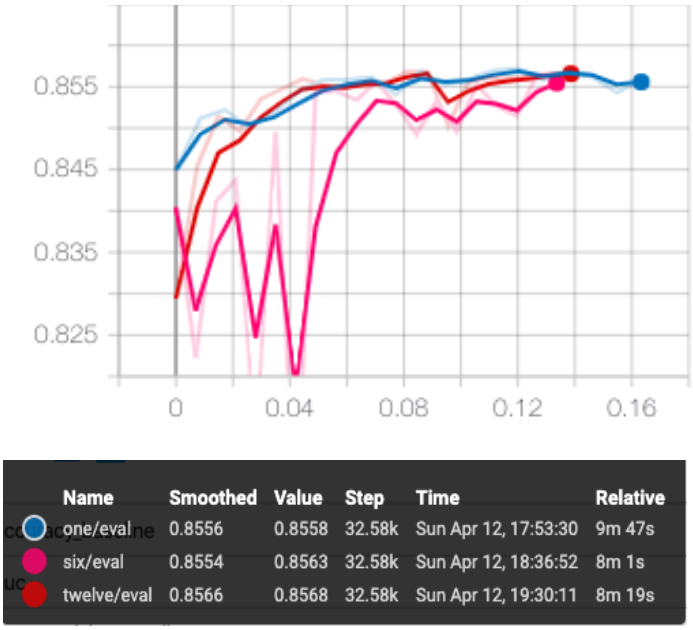



**Figure 14. Two Socket intra-operator parallelism (wide and deep)**

One core takes the longest training time to reach the final accuracy value but it has the smoothest curve to accuracy, this is because all the data is readily available in the node but takes long to process all the data as there is very less parallelization of operations. Six cores use all the cores on the same socket and have a lot of variation in accuracy over time. However, six cores overall take less time to reach the best possible accuracy than one core, while its overall time to accuracy is slow. Given arbitrary time constraints, parallelization across six cores does not perform well as the accuracy dips below the threshold (83% guaranteed by the developers) before it reaches a final accuracy of 85% which is better than the promised accuracy. Utilizing all the cores on both sockets shows a consistent upward accuracy, starting below the threshold, it steeply increases over a short time, thus this parallelization performs well for a short time limit, but uses a lot of hardware resources. Therefore, the next parallelization we observe is restricting to one socket to reduce the hardware usage but maintain performance.

**One Socket Experiment: no inter-operator parallelization**

One: Core 1 thread 1 used.

Six: One thread per core on Socket 0 used.

Twelve: All threads on all cores on Socket 0 used.

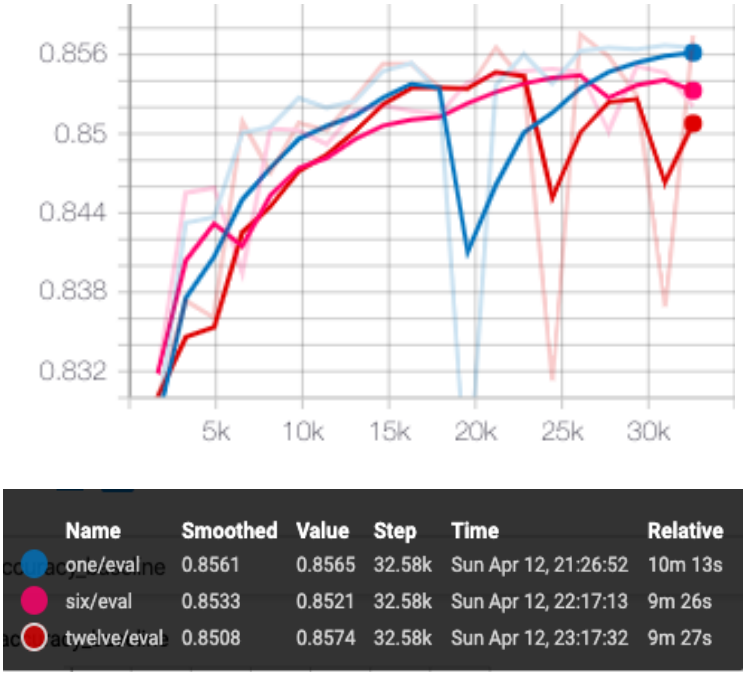



**Figure 15. One Socket intra-operator parallelism (wide and deep)**

Maintaining one socket at a time and using no inter-operator parallelism by setting inter flag to 1, we see that the total training time for all experiments increased compared to when operations were parallelized. An interesting trend in this data is that one thread per core (Six) has a steeper slope of increase in accuracy than 2 threads per core (Twelve), there is also a steadier increase for one thread than for hyperthreading which causes dips in accuracy.

Thus, given an experiment with $t$ time and limited resources of $n$ cores to reach the best possible accuracy having one thread per core (no hyperthreading) across all cores and using all sockets available

will give the best and fastest results. Therefore, hyperthreading impedes performance as a result of the sharing of physical hardware resources.

## Chapter 5

## Experiment: Performance across memory configurations

Two important factors that impact performance due to memory storage is throughput and latency. Throughput is the quantity of memory operations going through a processor and latency is the time needed to perform an action or fetch data from memory[23]. Non-uniform memory access (NUMA) systems, like our test system are server platforms with more than one system bus. Our system has 2 sockets and 1 NUMA node associated with each socket. These platforms can utilize multiple processors on a single motherboard, and all processors can access all the memory on the board, however, when a processor accesses memory that does not lie within its own node, data must be transferred over the NUMA connection at a rate that is slower than it would be when accessing local memory. Thus, memory access times are not uniform and depend on the location (proximity) of the memory and the node from which it is accessed.

The x86 CPU architecture has supported NUMA for a number of years. Modern operating systems such as Linux support NUMA-aware scheduling, where the OS attempts to schedule a process to the CPU directly attached to the majority of its RAM. In Linux, it is possible to further manually tune the NUMA subsystem using the ‘numactl’ utility. We utilize this utility to examine machine learning workload performance across memory configurations by binding the process and the memory to the same or different nodes [24].

Our expectation from this experiment is that being on the same NUMA node as the current processing socket would improve performance due to the decreased memory latency as

local memory access is the fastest. Large memory areas shared across CPUs are best placed using interleaving so the objects are distributed over all available nodes, machine learning workloads take up large memory areas and repeatedly recalculate weights which cyclically utilize memory, having a low memory access time will reduce the overhead of getting the data for each comparison and thus improve the overall performance of the workload run. Further, we observe the voluntary and involuntary context switches as well as minor page faults to analyze the correlation between memory and thread overhead and performance declines.

When a page is allocated it is not placed on a NUMA node until it is first touched. A hardware fault will be generated when a process touches or writes to an address (page fault) that has not been used yet. The physical page is allocated during page-fault handling. The default allocation policy is for the OS to place the page on the node where the CPU is running. It is at the page allocation time that the allocation policy occurs [26].

## Fashion MNIST

Keeping the model constant, we now change the memory configurations to study the impact of NUMA nodes on the Fashion MNIST workload. From the first experiment we studied performance impacts of CPU allocation configurations and gained motivation to further study the impact of memory latency and analyze the optimal memory allocation as well as the correlation of page faults and context switches to system performance. Soft page faults occur when data being sought is actually in memory, but can’t be found by address, this is quickly corrected to retrieve the data.

In this experiment we test a few configurations of memory interleaving and binding to gain an understanding of performance impacts which would indicate towards whether is consecutive memory access if interleaving is significantly more efficient than memory binding. We are expecting memory binding to give positive results as we maintain memory locally reduce NUMA access across nodes.

Our first test is to check the response with no controlling of memory configurations and observe the default case. Further, from the first experiment we determined that using all nodes on one core gave the best performance as the memory access time is reduced, going further we try to utilize all the memory banks within a socket or interleave memory to compare the performance due to only memory impacts. Memory binding to nodes works to use a node preferentially to store data, keeping the same node as the process to see the impact of memory binding and therefore memory access time on performance, to observe if it is a significant change in execution time.

**Table 4. Training Time across Memory Configurations**

| Memory Configurations | Average Time |
|---|---|
| all | 44.3771529 |
| all, cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 41.6843095 |
| interleave all | 43.4357842 |
| interleave all, cpubind = 0,12,2,14,4,16,6,18,8,20,10,22 | 42.0083139 |
| interleave = 0, cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 41.7073533 |
| cpu node bind =0, membind = 0,1 | 41.7534961 |
| cpu node bind =0, membind = 0 | 41.8946379 |
| cpu node bind =0, membind = 0,1, cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 41.5584751 |

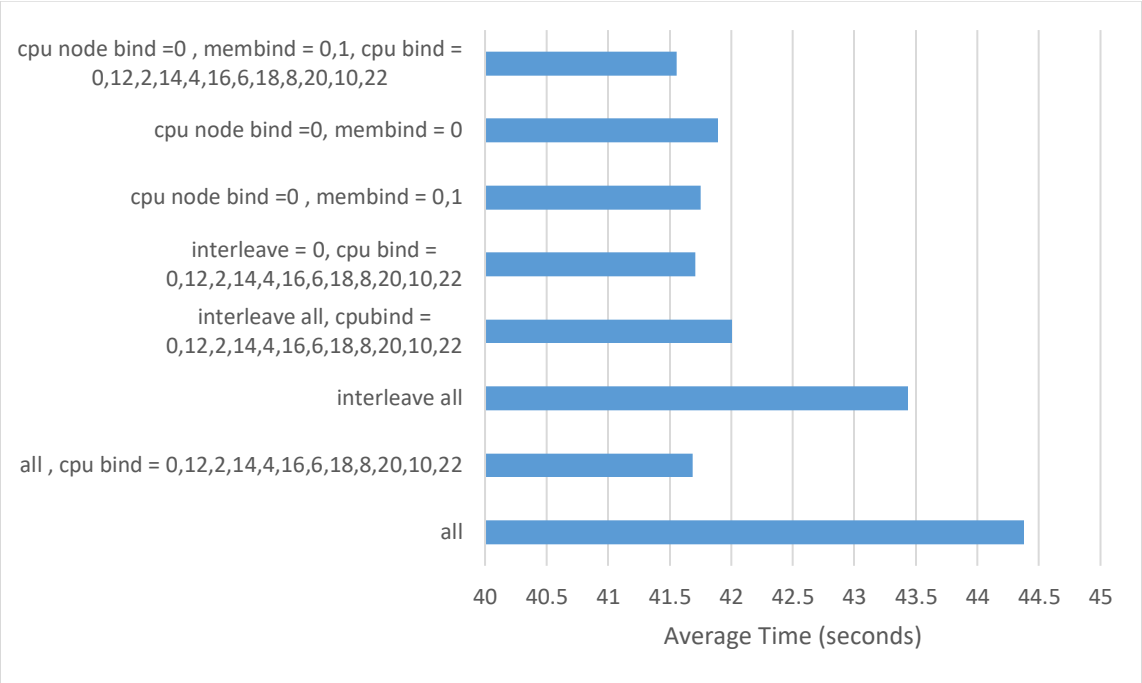



**Figure 16. Memory configurations vs. Average Training Time**

The most efficient run is without memory interleaving while utilizing all cores on the first socket, from the previous experiment we know that processing all threads on one socket is the most efficient thread configuration for our test, further this comparison between interleaved and non-interleaved memory proves that non-interleaved memory configuration is faster, this is the opposite of the expectation and memory interleaving is meant to make memory access for contiguous memory faster. The interleave policy distributes memory allocations equally on all nodes regardless of which threads access it. Interleaving ensures that memory allocations are balanced but not necessarily that memory accesses will be balanced. Interleaving works on the page granularity level, therefore this motivates us to check for page faults and context switches.

From this experiment we also observe that the average times for the same dataset and model is much lower across all runs as compared to the previous experiment using varying number of CPUs. The standard deviation is also much greater by changing CPU allocation than by changing memory allocation. Therefore, CPU allocation plays a heavier role in determining

system performance while running machine learning workloads and memory configurations, if used per the use case can further help improve the performance.

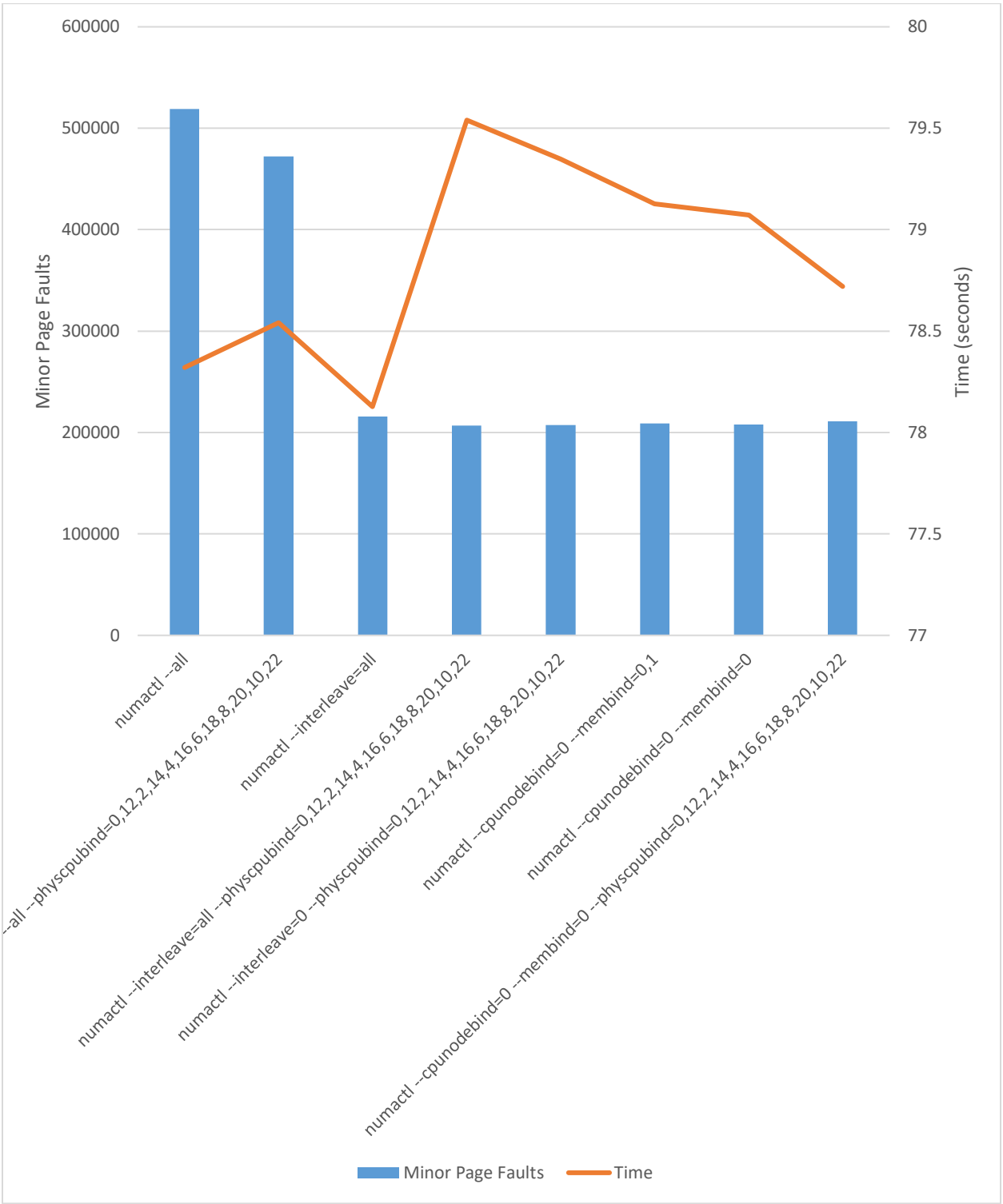



**Figure 17. Minor Page Faults and Training Time across Memory Configurations**

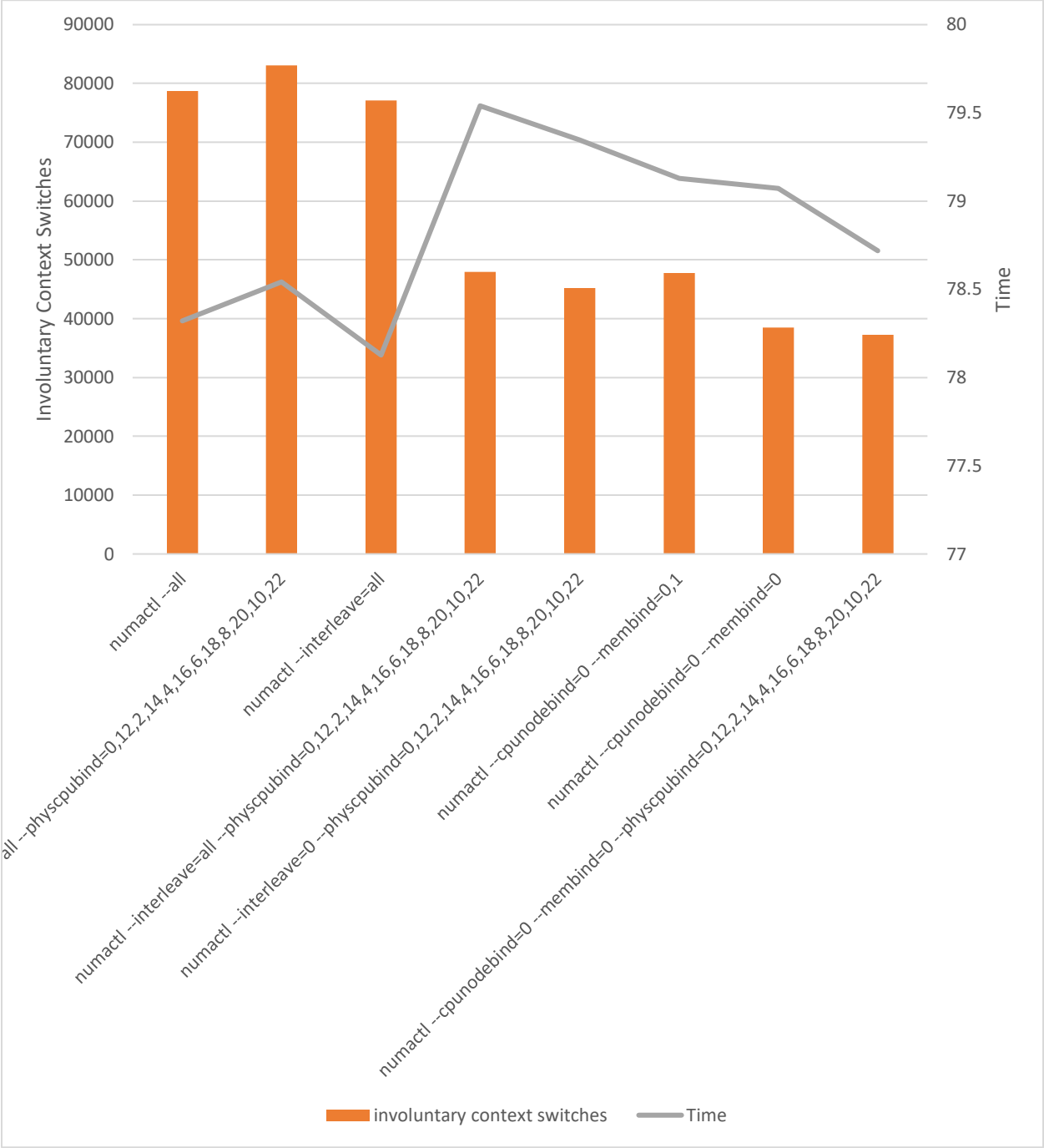



**Figure 18. Involuntary Context Switches and Time across Memory**

Minor page faults are caused by a page miss in the memory bank at the local node, this adds the miss found time and the remote memory access time, this increases training time,

therefore reducing performance. An involuntary context switch occurs when a thread has been running too long without making a system call that blocks (voluntary context switch) and there are processes waiting for the CPU, more involuntary context switches occur when a program is very CPU intensive. A highly multi-threaded application also increases the probability of involuntary context switches occurring. Having less threads than CPU cores help reduce number of involuntary context switches.

The added time due to inter-node memory access across NUMA nodes is due to the increased distance from one node to the other. There is an added distance which is double the distance in our test system to access memory in the opposite node to the current processing node.

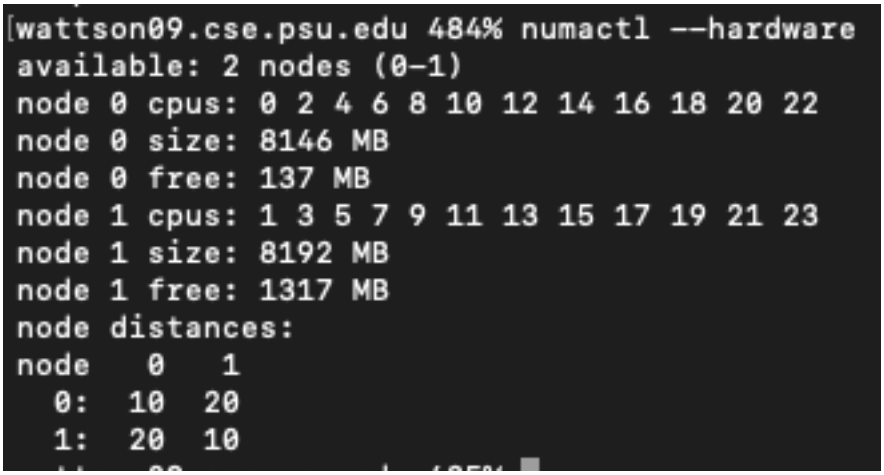



**Figure 19. NUMA hardware**

## Cache Statistics

We aim to study cache statistics across memory configurations and the impact of cache misses and branch misses on training time. We understand from a computer architecture theory standpoint that cache misses have a penalty time of cache access time + access time to main memory, however, we aim to study if these cache miss penalties are large enough to impact the

overall performance of the machine learning model and if it should be considered for a significant contributor in training time, learning this will help us realize the importance of memory configuration and maximum impact of memory usage optimizations on the training time of models on a large scale.

**Table 5. Cache statistics with perf command**

| memory configuration | task-clock (CPUs) | context-switches (M/sec) | page-faults (M/sec) | branch-misses (% of all branches) | L1-dcache-loads (M/sec) | L1-dcache-load-misses(% of all L1-dcache hits) | LLC-loads (M/sec) | LLC-load-misses(% of all LL-cache hits ) | Time (seconds) |
|---|---|---|---|---|---|---|---|---|---|
| all | 2.072 | 0.009 | 0.009 | 2.86% | 487.011 | 10.07% | 16.642 | 24.46% | 43.35486184 |
| all , cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 2.105 | 0.009 | 0.005 | 3.01% | 548.191 | 10.05% | 17.6 | 5.39% | 41.86561167 |
| interleave all | 2.106 | 0.009 | 0.002 | 2.44% | 534.918 | 8.51% | 18.154 | 16.33% | 41.82015163 |
| interleave all, cpubind = 0,12,2,14,4,16,6,18,8,20,10,22 | 2.134 | 0.009 | 0.002 | 3.11% | 536.465 | 9.88% | 17.103 | 5.37% | 41.75787444 |
| interleave = 0, cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 2.131 | 0.009 | 0.002 | 3.12% | 537.193 | 9.63% | 17.65 | 5.27% | 41.88808544 |
| cpu node bind =0 , membind = 0,1 | 2.14 | 0.009 | 0.002 | 3.14% | 539.955 | 9.78% | 17.111 | 5.31% | 41.48588839 |
| cpu node bind =0, membind = 0 | 2.129 | 0.009 | 0.002 | 3.01% | 546.879 | 9.77% | 17.535 | 5.28% | 41.63952407 |
| cpu node bind =0 , membind = 0,1, cpu bind = 0,12,2,14,4,16,6,18,8,20,10,22 | 2.138 | 0.01 | 0.002 | 3.05% | 543.304 | 9.53% | 16.961 | 5.50% | 41.50273856 |

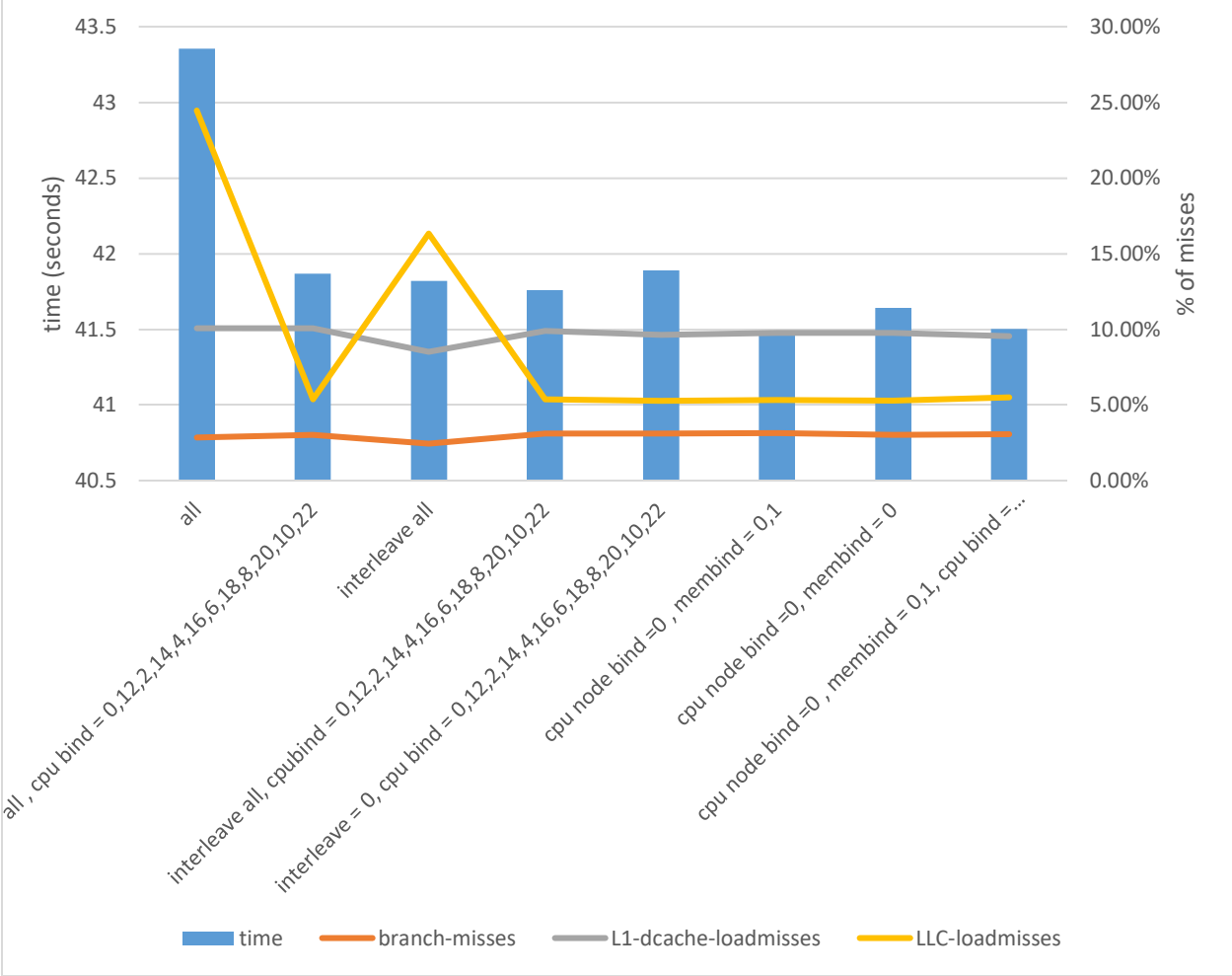



**Figure 20. Cache Statistics across memory configurations**

We use the perf tool to get cache hit and miss statistics for the program run using different NUMA configurations and observe a proportional relation between number of LLC-load misses and the total execution time of the training model. The LLC is the Last Level Cache and it refers to the highest-level cache that is shared by all the CPUs, a miss in the LLC cache would mean that the data has to be retrieved from main memory and is not in any cache. We get a large number of LLC cache misses in the run “all” and “interleave all”, using all cores in any configuration increases the number of cache misses and impacts overall time due to the increased time to fetch from main memory. It is important to note here that we measure the percentage of

misses out of all calls to that cache, this can be read as the percentage being the probability of having a cache miss for the given setup.

The L1-d cache misses do not vary much by the memory configuration, the L1-d cache is the lowest level cache and the first cache that is accessed for the data, from the L1-d cache the data fetching proceeds hierarchal downwards to higher level caches and main memory. The minimal variability in L1-d cache miss rates except for interleave all shows that the L1-d cache which is associated to a NUMA node provides the same performance across configuration. However, interleaving memory lowers the cache miss rate and is also meant to reduce the access time to main memory, we see this take effect as despite higher number of LLC-cache misses the overall training time did not drastically increase, in fact it improved marginally compared to other cases.

# Chapter 6

## Results

In this research, we used TensorFlow libraries and popularly used, machine learning models to do a scalability study of accuracy, training time and machine resources to study the impact on performance of limiting hardware resources and time.

Through the preliminary studies we determined that core scaling and memory binding play an important role in determining the scalability of machine learning model training. Allocating hardware resources using custom configurations for optimization give the best results. Hyper-threading does not aid in performance as it requires resource sharing between threads. Utilizing parallel processing across operators significantly improves time to accuracy and total training time of the model.

Allocating cores on the same socket aid in performance by reducing the memory access time of inter-NUMA node memory accesses since distances to memory nodes are greater for other NUMA nodes as compared to local access.

Wide and Deep models specifically are interesting to study as they clearly depict the tradeoff between performance and accuracy. Studies of machine learning models show that wide and deep models combine to predict generic history data as well as new sample points for recommendations not present in training. While wide models are faster to train as well as model from a programming perspective, they have a limit to accuracy due to the traditional dependence of models on history and training data. On the other hand, deep models are more time intensive

to train but reach a higher final accuracy as well as have a better time to minimum threshold accuracy.

# BIBLIOGRAPHY

[1] SAS, [Online]. Available: https://www.sas.com/en_us/insights/analytics/machine-learning.html.

[2] Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Moore%27s_law.

[3] S. Kansal, "Codementor," 7 May 2019. [Online]. Available: https://www.codementor.io/blog/scaling-ml-6ruo1wykxf.

[4] K. Pingali, "The von Neumann Bottleneck Revisited," 26 July 2018. [Online]. Available: https://www.sigarch.org/the-von-neumann-bottleneck-revisited/.

[5] "Descending into ML: Training and Loss," developers.google.com, [Online]. Available: https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss.

[6] Amazon, "Training ML Models," [Online]. Available: https://docs.aws.amazon.com/machine-learning/latest/dg/training-ml-models.html.

[7] J. Brownlee, "What is the Difference Between a Parameter and a Hyperparameter?," 26 July 2017. [Online]. Available: https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/.

[8] M. Mayo, "Neural Network Foundations, Explained: Updating Weights with Gradient Descent & Backpropagation," [Online]. Available:

https://www.kdnuggets.com/2017/10/neural-network-foundations-explained-gradient-descent.html.

[9] C. Nicholson, "A Beginner's Guide to Neural Networks and Deep Learning," [Online]. Available: https://pathmind.com/wiki/neural-network.

[10] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *ACM Queue*, vol. 11, no. 7, 2013.

[11] H.-T. Cheng, L. H. Koc, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu and H. Shah, "Wide & Deep Learning for Recommender Systems," *arXiv*, 2016.

[12] Heng-Tze Cheng, "Wide & Deep Learning: Better Together with TensorFlow," Google, 29 June 2016. [Online]. Available: https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html.

[13] N. Greeneltch and J. X., "Maximize TensorFlow* Performance on CPU: Considerations and Recommendations for Inference Workloads," Intel, 25 01 2019. [Online]. Available: https://software.intel.com/en-us/articles/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference.

[14] J. Zhang, "Parallel Computing Chapter 7 Performance and Scalability," [Online]. Available: https://www.cs.uky.edu/~jzhang/CS621/chapter7.pdf.

[15] R. Wang, "Memory Interleaving," 29 11 2005. [Online]. Available: http://fourier.eng.hmc.edu/e85_old/lectures/memory/node2.html.

[16] R. Kohavi and B. Becker, "Census Income Data Set," Silicon Graphics, 01 05 1996. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Census+Income.

[17] "Predicting Income with the Census Income Dataset," tensorflow, 19 8 2019. [Online]. Available: https://github.com/tensorflow/models/tree/master/official/r1/wide_deep.

[18] tflearn, [Online]. Available: http://tflearn.org/optimizers/.

[19] S. Cook, "Optimizing Your Application," in *CUDA Programming*, 2013, pp. 305-440.

[20] J. Philip J. Koopman, "The Limits of Memory Bandwidth," in *Stack Computers: the new wave*, Ellis Horwood, 1989.

[21] T. E. Bettilyon, "High Performance Computing is More Parallel Than Ever," 20 12 2018. [Online]. Available: https://medium.com/tebs-lab/the-age-of-parallel-computing-b3f4319c97b0.

[22] F. Denneman, "MEMORY DEEP DIVE: OPTIMIZING FOR PERFORMANCE," 20 February 2015. [Online]. Available: https://frankdenneman.nl/2015/02/20/memory-deep-dive/.

[23] C. Hollowell, C. Caramarcu, W. Strecker-Kellogg, A. Wong and A. Zaytsev, "The Effect of NUMA Tunings on CPU Performance," *Journal of Physics*, vol. 664, 2015.

[24] C. CATES, "PERFORMANCE IMPLICATIONS OF NUMA WHAT YOU DON'T KNOW COULD HURT YOU!," [Online]. Available: https://www.cmg.org/wp-content/uploads/2015/10/numa.pdf.

[25] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Communications of ACM*, vol. 61, no. 8, pp. 65-72, 2018.

[26] J. Li, N. K. Sharma, D. R. K. Ports and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," *Department of Computer Science & Engineering, University of Washington.*

[27] "TensorFlow on CPUs," New Zealand eScience Infrastructure , 20 02 2020. [Online]. Available: https://support.nesi.org.nz/hc/en-gb/articles/360000997675-TensorFlow-on-CPUs.

**ACADEMIC VITA**


## Swasti Mehra

**swastime9@gmail.com**


### Education:

- Pennsylvania State University, Computer Engineering B.Sc., 2020
- Schreyer Honors College, Dean's List (all semesters)
- Academic Excellence Scholarship, President's Freshman award and Spark's Award
- Skills**:** C, C++, Java, Python, C#, UNIX, MIPS architecture, Verilog, Object Oriented Design, MySQL

### Experience:

- **Software Development Intern,** Susquehanna International Group
  - Designed alongside the Fixed Income team a backcasting system to simulate the trading environment with different user set configurations.
  - Developed a Windows form application and a backcasting service to asynchronously run services and simulate the day's trading activities. Further implemented backcasting in an in-production service keeping the control flow unchanged while running the service autonomously with history data and new configurations.
  - Developed and implemented features on the fixed income electronic trading system to increase trader productivity by supporting bulk updates of user configurations and allow symbol subscription.
- **Summer Engineering Intern,** Bharti Airtel LTD.
  - Designed and implemented API in Java to periodically search several servers for relevant files and built a File Monitoring System for the Information Management Team.
  - Conducted Proof of Concept for an Application Performance Monitoring system by using open source tools from Elastic for data monitoring. Real time data collection (Logstash), custom visualizations (Kibana) and searching and filtering (Elasticsearch).
  - Automated the existing processes for the operational reporting of financial and fraud management data from core systems to business users.
- **Undergraduate Researcher**, Engineering REU
  - Used Computer Vision concepts to build a MATLAB script to identify malaria affected blood cells on a video capture of a blood sample on a chip.
  - Built an amplifying circuit using Arduino PWM for a preloaded microfluidic centrifugal disk.
  - Test the motor and circuit for capability to perform Nucleic Acid testing to detect diseases by taking air samples.
  - Presented Final Project at the Engineering Design Showcase Fall 2018.

### Leadership:

- **President,** Association for Computing Machinery
  - Lead a team of 13 officers and 670 general members
  - Launched DevPSU- a learning initiative that grew to 200 members aiming to make AI and collaborative software development approachable.
  - Created a learning module and project on Visual Recognition using Microsoft Azure API and an Image classifier using TensorFlow.
  - Wrote problems for CodePSU (Penn State's annual competitive programming competition) in 2017, 2018 and 2019.
- **Technology Captain**, Penn State Dance Marathon
  - Created a point of sales system for the Merchandise committee to track sales and inventory (Django).
  - Created a quizzing platform for the THON eLearning Management System (Django) in a team of 3 captains.
- **Learning Assistant**, Penn State Computer Science Engineering
  - Mentor students by hosting office hours and recitations to help students understand in class concepts by applying them to assignments and projects in CMPSC200(MATLAB), CMPSC465 (Data structures and algorithms) and CMPEN454 (Computer Vision).