Performance Analysis for Machine Learning Applications

Performance Analysis for Machine Learning Applications

A DISSERTATION PRESENTED
BY
YU WANG
TO
THE DEPARTMENT OF SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
SEP 2019

Thesis advisor: Professor David Brooks and Gu-Yeon Wei Yu Wang

Performance Analysis for Machine Learning Applications

Abstract

Performance analysis has been driving the advancements of software and hardware systems for decades. Proper analysis can reveal system and architectural bottlenecks, provide essential information for choosing frameworks and platforms, and further lead to performance optimizations. Systematic performance analysis is difficult since it involves various problem domains, algorithms, software stacks, systems, and hardware. Recently, the surge of machine learning applications and domain-specific architectures stimulates rapid evolution of all those dimensions, thus making performance analysis increasingly challenging.

To tackle this problem, this dissertation conducts deep and systematic performance analysis for a variety of workloads, software frameworks, and hardware systems, and demonstrates proper ways to apply several performance analysis methodologies. First, we study the performance analysis methodology for general-purpose processors, a traditional and mature industry, based on CPU benchmarks and demonstrate the importance of using *representative* benchmarks in drawing insights about hardware systems. Next, with the lessons learned from traditional methods, we rigorously study deep learning frameworks by applying proper analysis methods at corresponding scenarios. We extract the performance implications of key design features from those frameworks, and the insights are distilled into a set of simple guidelines to tune framework features. The proposed guidelines nearly close the performance gap between the state of the art and the global optimum. Further, we propose a systematic methodology to facilitate performance analysis for rapidly evolving deep learning models and platforms. The proposed methodology can reveal deeper insights that are difficult to discover for traditional approaches. We demonstrate its utility with deep learning by comparing two generations of specialized hardware (Google's Tensor Processing Unit v2 and v3), three heterogeneous platforms (TPU, GPU, and CPU), and different versions of specialized software (TensorFlow and CUDA). Finally, since machine learning techniques advance rapidly and architects need to be aware of emerging applications, we take the first step towards analyzing Bayesian inference, an important branch of machine learning. Optimization mechanisms are proposed based on the analysis.

With the methodologies and analysis presented in this dissertation, we hope to encourage researchers and engineers to apply our methodologies to new platforms, software, and applications for systematic performance analysis. We envision this to help resolve existing performance bottlenecks and design better software and hardware for the current and future applications.

Contents

Listing of figures

List of Tables

Dedicated to my mentors and peers who have been working relentlessly to advance science one step at a time.

The unexamined life is not worth living.

Socrates

1

Introduction and Background

Performance analysis quantifies performance, profiles workloads, and reveals system bottlenecks, for usually two non-exclusive purposes, cross-comparing and improving existing platforms. Platform cross-comparison provides essential information on choosing proper platforms, when the marketplace of computing hardware offers numerous choices for similar functionalities, such as general microprocessors and specialized accelerators. Understanding architectural and system bottlenecks through analysis is necessary to improve current system or design new ones.

1.1 The Role of Performance Analysis

Performance analysis is usually the first step towards performance optimizations. Given a plethora of workloads running on a platform, performance analysis needs to be conducted to extract important computation kernels, analyze their computational characteristics, find system or architectural bottlenecks, after which software or hardware optimizations for that platform can be developed.

Performance analysis has been driving general-purpose compiler and architecture advancements. For example, through data dependence analysis of non-numerical workloads, researchers find that most communication is to update shared memory locations, and proactive communication reduces execution latency by decoupling communication from computation. Helix-RC[37] is therefore designed to decouple communication from thread execution, by adding a lightweight architectural enhancement co-designed with a parallelizing compiler. It speeds up SPEC CINT2000 benchmarks by 6.85× on average. Performance analysis sometimes leads to software enhancements. Engineers in Google analyze the temporal locality of memory page references, and find large fractions of cold memory pages, i.e., pages not used for a given time interval. A software-defined far memory[126] based on zswap [*] is then proposed and it leads to over 67% memory cost reduction since deployed in Google's datacenters in 2016.

As the end of Dennard scaling and Moore's law has slowed the performance improvement of general-purpose microprocessors[?] [91], the design of domain-specific hardware is becoming more and more relevant. The Google's Tensor Processing Unit (TPU) is a prominent example of domain-specific hardware[111,54,200]. Its development was motivated by the observation that, with conventional CPUs, Google would have had to double their datacenter footprint to meet the internal demand for machine learning workloads. Since matrix multiplication is one of the most important kernels of such workloads, the core of TPU, matrix units (MXUs), is built for high-throughput matrix mul-

[*] Linux zswap: https://www.kernel.org/doc/Documentation/vm/zswap.txt

tiplication with systolic arrays[124]. Google has been using TPUs for their large-scale production systems, including Search, Translate, and Gmail. Compared to acceleration of killer applications like machine learning, datacenter workloads are usually too diverse and need broad acceleration for low-level routines including remote procedure calls, data serialization and memory allocation[113]. With such observations, Mallacc[114], an in-core hardware accelerator, is designed for high-performance memory allocators, and it reduces malloc latency by up to 50%.

As the key part of performance analysis, benchmarks have been the driving force for compiler and architecture design for decades. For example, it has been over thirty years since the first version of SPEC CPU benchmark suite released in 1988 and it is still widely used for measuring CPU performance. It is hard to make any progress without benchmarks or performance analysis. Benchmark suites have been developed for various platforms. Examples include SPEC[92], Geekbench[125], and PARSEC[33] for CPUs; Rodinia[42], SHOC[53], and Parboil[179] for GPUs; MachSuite[161] for accelerators; Rosetta[78] for FPGAs. Some benchmark suites are designed for certain applications, instead of for platforms. Domain-specific benchmark suites are especially popular for deep learning. Examples are CortexSuite[186], TonicSuite[84], Sirius[85], Fathom[15], DAWNBench[49], and MLPerf[151]. Categorizing benchmark suites involves many dimensions, and some suites mentioned above include microbenchmarks, and some include selected workloads that represent real-world applications.

Performance analysis is challenging; it is a multi-disciplinary field that involves various problem domains, algorithms, software stacks, systems, and hardware. Performance analysis using different types of benchmarks provides different advantages, while showing different constraints. As an important type of workloads, this thesis dives deep into deep learning, performance analysis of which is especially difficult and should be conduct with caution.

1.2 Challenges of Analysis for Deep Learning

Performance analysis studies the interactions of workload characteristics, software implementations, and architecture of hardware platforms, and the problem becomes more challenging when the three dimensions evolve rapidly. That is the case for deep learning.

The state of the art models for deep learning has been updated every a few months. Taking vision models as an example, ever since AlexNet[123] was published in 2012, more models started to appear, including CaffeNet[106] (2014), VGG[176] (2015), ResNet[87] (2015), GoogleNet[181] (2015), Inception[182] (2016), DenseNet[100] (2016), and ResNext[208] (2016). Model evolution is fast for other model types as well, such as sequence-to-sequence translation[193] and recommendation models[79,65,145].

The training and inference of such models is extremely intensive, and its performance is crucial especially in production. As a result, a plethora of specialized platforms for deep learning have been developed, including Google's TPU[111,54], NVIDIA's GPU with Tensor Cores, Facebook's Zion platform[130], Intel's Nervana NNP-T[211], Cerabras' Wafer-Scale Engine[40], to name just a few. There is an increasing trend in terms of the AI hardware platforms announced at Hot Chips conference over the past four years. The number increases from four (2016), seven (2017), seven (2018) to nine (2019). More and more AI hardware startups and companies initially not known for hardware start to present their latest AI hardware. It is predicted that deep learning hardware market will reach at least 60 billion dollars by 2025[187,141].

Given the variety of workloads and hardware platforms, software stacks need to be flexible and efficient to take advantage of the performant hardware. Software and hardware co-design is the way to further scale up the performance, which leads to specialized software. Such software include frameworks and their components for specialized platforms, such as TensorFlow, Caffe2, Pytorch, and libraries including MKL, MKL-DNN, CUDNN, and CUDA. Software stacks are updated frequently, enhanced with specialized optimizations for certain hardware platforms and models.

Facing with numerous choices of software and hardware for deep learning, the first and foremost question from users is which one to choose. Performance is one of the most important factors. On the other hand, software and hardware engineers would like to further optimize existing systems for current and emerging deep learning workloads. Taking a step back, architects need to decide how to design a better software-hardware system, given lessons learnt from existing designs. Proper performance analysis can help to find answers to those questions.

Despite the expert knowledge required in this multi-disciplinary field of performance analysis, deep learning exhibits extra challenges.

- Deep learning models are diverse, including various problems domains, models, and datasets.
- The pace of this field is fast and the state-of-the-art models evolve every few months.
- Deep learning software and hardware systems are diverse and rapidly evolving as well.
- There are varying performance metrics, such as accuracy, time to train, and latency of inference. Metrics need be chosen properly.

Performance analysis needs to be conducted carefully to avoid bias and extract deep insights. Proper performance analysis methodology should have a comprehensive set of benchmarks that representing state-of-the-art and emerging workloads and datasets. Researchers need solid understanding of cross-stack workload resource requirements and bottlenecks. Meanwhile, researchers need to be aware of emerging workloads.

1.3 Bayesian Inference

The recent surge of machine learning has motivated computer architects to focus intently on accelerating related workloads, especially in deep learning. However, for unsupervised learning, Bayesian

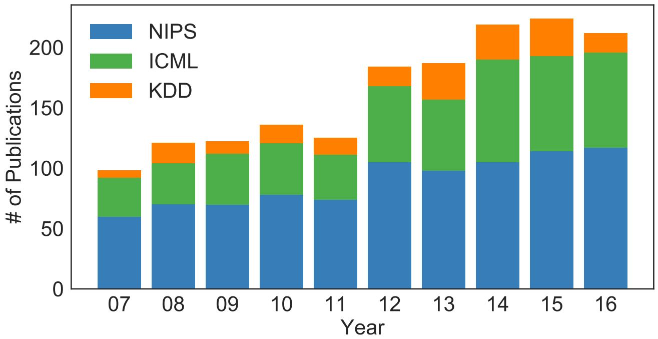


Figure 1.1: Increasing Bayesian inference publications in top machine learning conferences from 2007 to 2016.

methods often work better than deep learning with unlabeled or limited data, and can leverage informative priors, and has interpretable models. Bayesian inference is becoming increasingly popular over the years since it is a great complement to deep learning. Some recent revolutionizing deep learning research adopts Bayesian techniques, and notable examples include generative adversarial networks (GAN), Bayesian neural networks (BNN), and variational autoencoder. Figure 1.1 shows the number of publications focusing on Bayesian inference in top machine learning conferences, including conference on Knowledge Discovery and Data Mining (KDD), International Conference on Machine Learning (ICML), and conference on Neural Information Processing Systems (NeurIPS). The publications about Bayesian inference have been increasing for the ten years between 2007 to 2016. While many architects focus on optimizing deep learning, we also need to be prepared for the upcoming workloads, and Bayesian inference is a promising example.

1.4 Thesis Overview

The goal of this thesis is to systematically understand popular and emerging applications so that their performance can be optimized. The progress of this thesis is driven by two major factors, workloads and analysis methods.

Table 1.1: Summary of workloads, software and hardware studied in this thesis.

Stack	Chapter 2	Chapter 3	Chapter 4	Chapter 5
WORKLOADS	Traditional	Deep Learning	Deep Learning	Bayesian
SOFTWARE	C/C++	TensorFlow, Caffe2	TensorFlow, CUDA	Stan
HARDWARE	CPU	CPU	TPU, GPU, CPU	CPU

1.4.1 Workloads Overview

From workload perspective, this thesis starts with general-purpose workloads, then thoroughly analyzes popular deep learning models, and takes the first step towards understanding emerging workloads of Bayesian inference. The workloads, software (frameworks) and hardware platforms of this thesis are summarized in Table 1.1.

We start from learning the past experience of traditional workloads on traditional platforms (Chapter 2), by analyzing general microprocessors' performance for CPU benchmark suites, SPEC[92] and Geekbench[125]. Their performance repositories have existed for decades and provide abundant data to study. The data repositories represent the maximum information available from such benchmark suites after the industry has been mature for years. Being developed with the same purposes, after a few years MLPerf is likely to serve the community in a similar way as SPEC.

We then dive into currently popular deep learning workloads and their specialized software and hardware systems (Chapters 3 and 4). Deep learning has been widely adopted in products and changing our lives by improving our daily commute, communication, shopping, and traveling experience. We explore a variety of analysis methods, software and hardware, and propose optimization techniques to improve current deep learning systems.

Besides making current deep learning workloads into better products, architects need to be prepared for the next upcoming workloads, since machine learning techniques have been evolving so rapidly. And the industrialization experience of deep learning provides perfect lessons for emerging workloads. In Chapter 5, we take a step further to conduct seminal analysis on Bayesian inference,

Table 1.2: Summary of advantages and constraints of performance analysis methods.

Method	Chapter	Advantage	Constraint
Using Real Workloads	2,3,4,5	representative (if designed properly), decades of experience	long time to develop and update, may not represent some users' interests, only represent current workloads
μbenchmarking	2,3	interpretable, stress limit	not representative of real workloads
Parameterized Benchmarking	4	interpretable, stress limit, explore design space	not design to be representative of users' interests

one of the emerging applications.

1.4.2 Performance Analysis Overview

Table 1.2 summarizes analysis methods. Benchmarking with real workloads and microbenchmarking are two common methods. This thesis (Chapter 4) proposes a generic methodology, parameterized benchmarking, to complement traditional approaches. The three methods are complementary and should be used in different scenarios.

Performance analysis with realistic workloads has been the most widely-used for decades, and is presented in all chapters of this thesis. Notable examples are SPEC[92], PARSEC[33], MLPerf[151], and BayesSuite[201] proposed in this thesis. Our community has decades of experience benchmarking with such suites, and the procedures have been largely standardized. However the usability of such benchmark suites largely depends on whether the workloads are representative. Two scenarios can make such suites not representative, being obsolete or not covering certain users' interests. Thus new realistic workloads need to be added to those suites regularly, which is time-consuming. It is especially challenging for rapidly-evolving machine learning workloads. For example, the development of MLPerf requires regular meetings between tens of companies and universities, and it took over six month to upgrade MLPerf from version 0.5 to 0.6.

Microbenchmarks are used to stress test system bottlenecks. Geekbench[125] is a widely-used microbenchmark suite. Each microbenchmark is designed to test one or several system components, such as floating-point capability, memory latency, or memory bandwidth. Microbenchmarks are usually computation kernels, such as matrix multiplication, convolution, image decoding, or memory streaming, thus microbenchmarking results are usually easier to interpret, compared to using real workloads. Microbenchmarks are also relatively more stable, so updates of which can be much less frequently. The major limit is it is hard to extrapolate or simulate real workloads' behaviors from microbenchmarks, which is discussed in details in Chapter 2.

The parameterized analysis method proposed in Chapter 4 has the advantages of the above approaches, with the goal of providing large "end-to-end" workloads covering current and future applications, and parameterizing the workloads to stress systems and explore a much larger design space. The performance of such benchmarks is easy to interpret because of the parameterized high-level workload attributes. Several data analysis and visualization techniques are coupled with those benchmarks to interpret their performance. Since they are designed to explore much larger design space of real workloads, including current and possible future workloads, they do not represent specific workloads of interests to certain users. It is both a feature and a constrain of this method, because one thing cannot be both general and specific at the same time.

1.5 Thesis Contributions

1.5.1 General-purpose workloads and platforms (Chapter 2)

The marketplace for general-purpose microprocessors offers hundreds of functionally similar models, differing by traits like frequency, core count, cache size, memory bandwidth, and power consumption. Many benchmark suites have been developed to measure processor performance, and their results for large collections of CPUs are often publicly available, such as SPEC CPU2006 and Geekbench 3 benchmark suites. In this chapter, we study those CPU performance repositories.

Specifically, we research on how useful those repositories are when consumers need performance data for new processors or new workloads. We have developed a deep neural network model, which can generate useful predictions for new processors and new workloads. Additionally, we observe that although benchmark suites are designed to cover a broad spectrum of workload types, they are self-similar, i.e. workloads in one suite are more similar to each other than to workloads in other suites. Due to the self-similarity, different benchmark suites do not give consistent performance rankings for processors and therefore their aggregate scores can be misleading. Users should be cautious when basing purchasing decisions exclusively on Geekbench 3, or evaluating research using SPEC CPU suite alone, if workloads of interest are very different from benchmark suites in use. This work reveals fundamental limitations of traditional benchmark suites, and future benchmark suites should avoid such pitfalls by regularly adding up-to-date workloads and removing obsolete ones.

1.5.2 The Design of Deep Learning Frameworks (Chapter 3)

This thesis then shifts its focus to deep learning, which has revolutionized many application domains and exhibits enormous opportunities for performance analysis and optimization. State-of-the-art deep learning frameworks, such as TensorFlow[14], Pytorch[118] and MXNet[44], has eased the programmability burden on machine learning developers. They support a wide variety of design features and enable a flexible machine learning programming interface, and therefore frameworks are the first choice for most developers to program deep learning models. Identifying and using a performance-optimal setting in feature-rich frameworks, however, involves a non-trivial amount of performance characterization and domain-specific knowledge. This thesis takes a deep dive into analyzing the performance impact of key design features and the role of parallelism. The observations and insights distill into a simple set of guidelines that one can use to achieve much higher training and inference speedup. The guidelines extracted from analyzing a set of microbenchmarks and real-world deep learning models are applied to a set of different and later models. The evaluation results show that our proposed performance tuning guidelines outperform both the Intel and TensorFlow recommended settings and nearly closes the gap between the state of the art and the global optimum. This work shows that performance analysis and optimizations can be conducted with carefully selected real workloads and microbenchmarks, and the insights can be extrapolated to a different set of workloads.

1.5.3 Parameterized Performance Analysis (Chapter 4)

To better facilitate performance analysis for rapidly evolving deep learning models and platforms, this thesis takes a step further to propose a systematic performance analysis methodology, and demonstrate its utility in deep learning. The methodology is comprised of a set of analysis techniques and parameterized workloads. As a special case of such methodology, we develop a tool for deep learning, ParaDnn, that generates end-to-end models for fully connected, convolutional, and recurrent neural networks. This methodology can be applied to analyze various hardware and software systems, and is a very good complement to traditional methods. This chapter demonstrates its utility by analyzing two generations of specialized platforms (Google's Cloud TPU v2/v3), three heterogeneous platforms (TPU, GPU and CPU), and specialized software stacks (TensorFlow and CUDA). This systematic methodology explores much larger design space of deep learning models, stresses system limits, covers state-of-the-art models, reveals deep insights that are impossible with traditional analysis approaches. One goal of this thesis is to encourage researchers to adopt such methodology to analyze other platforms.

1.5.4 Emerging Workloads: Bayesian inference (Chapter 5)

This thesis then takes the first step towards characterizing Bayesian inference workloads, an important branch of machine learning. We facilitate the study of Bayesian inference with the development of BayesSuite, a collection of seminal Bayesian inference workloads. We characterize the power and performance profiles of BayesSuite across a variety of current-generation processors and find significant diversity. Some workloads are bottlenecked by last-level-cache (LLC), and some workloads contain large fractions of computation that do not improve the inferencce quality. Based on the analysis, we then introduce a scheduling and optimization mechanism and a computation elision technique. Our proposed techniques are able to increase Bayesian inference performance by 5.8× on average. With this work, we would like to encourage the analysis and optimization of emerging workloads in our community, and Bayesian techniques are promising to drive the AI revolution.

2

General-Purpose Platforms and Workloads

To compare processor performance, the results of many benchmark suites for large collections of CPUs are publicly available. However, repositories of benchmark results are not always helpful when consumers need performance data for *new processors* or *new workloads*. To address these problems, this chapter presents a deep neural network (DNN) model, and uses it to learn the performance of Intel CPUs running the SPEC CPU2006 and Geekbench 3 benchmark suites. This chapter also quantifies the self-similarity of these suites for the first time in the literature.

2.1 Introduction

Computer hardware evolves rapidly. In 2015 alone, Intel released over 200 CPU SKUs.[*] The SKUs have different characteristics like frequency, cache size, memory bandwidth, and core count. Better configurations usually cost more. The performance of a microprocessor is not solely a function of its microarchitecture; it depends critically on the nature of the workloads running on it. Thus both CPU microarchitecture and workloads need to be taken into account when quantifying actual CPU performance. Consumers can then estimate whether investing in a costly configuration helps to meet their particular goals.

To help study processor performance, a variety of benchmark suites have been developed. For the more popular suites, such as Geekbench, SPEC CPU, and Passmark, sizable repositories of performance data have been collected, allowing the public to compare the behaviors of known configurations on standard workloads. However, there are some issues in using these repositories. The first is that consumers need to wait for comprehensive benchmark results of *new processors* to be contributed to the public repositories. The second issue is that for *new workloads*, consumers have to test a large number of possible configurations to find the most effective hardware. Daunted by this challenge, some consumers rely blindly on aggregate CPU scores in benchmark repositories.

To resolve these issues, we use statistical methods and machine learning techniques to analyze data from the SPEC CPU2006 and Geekbench 3 repositories.[†] SPEC CPU2006 is widely reported in academic studies. Geekbench 3 is used by many consumers to make purchasing decisions. We use deep neural networks (DNN) to predict the performance of Intel CPUs, and we compare the DNN prediction with that by linear regression (LR). The DNN predictive model learns interactions between processor specifications for different workloads, sharpening the usefulness of benchmark

[*] Stock Keeping Unit. In this chapter, a SKU is a CPU with a distinct processor number.

[†] The datasets in this chapter are available at `https://github.com/Emma926/cpu_selection.git`.

data repositories.

This chapter makes the following contributions:

- We use DNN to predict the performance of SPEC and Geekbench workloads on new CPU SKUs. (The mean error is 5% for SPEC and 11% for Geekbench.) We show that DNN is more accurate than traditional LR.
- With the same accuracy, we show that the performance of new workloads on just 10 SKUs can predict their performance on other SKUs.
- For the first time in the literature, we quantify the self-similarity of these widely used suites, by using DNN to cross-predict the results between the suites (with 25.9% and 14.9% mean error when predicting SPEC and Geekbench, respectively), and by comparing their CPU rankings (which are inconsistent, with footrule distance as high as 0.59).

2.2 Intel Processor Statistics

In this section, we present some statistical facts about Intel processors, SPEC performance for the Intel SKUs, and the SKUs' microarchitectures and types in our SPEC and Geekbench datasets. We show the large performance variations and a rich variety of SKUs in our datasets.

We construct our Intel processor specification dataset from http://ark.intel.com, where Intel publishes the specifications of all its processors. We collect specifications including microarchitecture, launch year, number of cores, base frequency, cache size, power, and memory type. Figure 2.1 shows the release years of Intel microarchitectures. The size of a bubble corresponds to the number of SKUs released in the corresponding year. The figure contains 17 microarchitectures that Intel has released since 1993.

Intel has a "Tick-Tock" model for microarchitecture code names. A tick represents a shrinking of the technology and a tock represents a new microarchitecture. In Figure 2.1, for example, Westmere and Sandy Bridge have 32nm feature size, and Sandy Bridge and Ivy Bridge have the same microarchitecture. Every year, there is expected to be a tick or tock. Starting in 2014, however, Intel decided to add a "refresh" cycle, to update the current microarchitecture. That is why for 2014 in Figure 2.1 there are only a few Broadwell SKUs but more Haswell SKUs. After 2016, Intel switched their model to a three element cycle known as "Process-Architecture-Optimization".

In this chapter, we use relative run time as a measure of performance. We take a Sandy Bridge processor (E3-1230) as the reference machine. We choose E3-1230 because it is common across datasets. The relative runtime of a workload is defined as

$$relative\ runtime = \frac{runtime - runtime_{ref}}{runtime_{ref}} \tag{2.1}$$

where $runtime_{ref}$ is the runtime of the corresponding workload on the reference machine. For

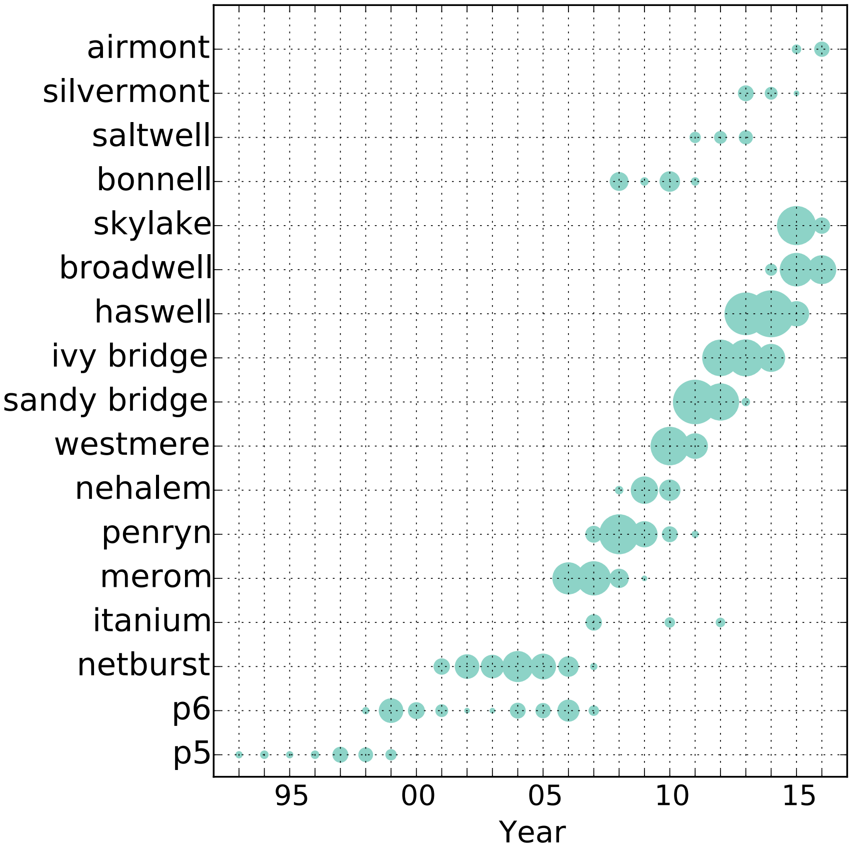


Figure 2.1: The number of SKUs released by Intel for 17 microarchitectures between 1993 and 2016. (Data from http://ark.intel.com.)

brevity, this chapter will consistently use "performance" to indicate relative run time. Thus the scaled runtime of E3-1230 is 0. In Figure 2.2, the E3-1230 performance is slightly below the average of all Sandy Bridge SKUs, so the line is centered on a value greater than 0. Since all results are presented relative to the same reference processor, the results are unaffected by the particular choice of the reference.

To show the diverse performance of different SKUs, we take the average SPEC performance on SKUs, and we gather the SKUs based on their microarchitecture code names. In Figure 2.2, the dots

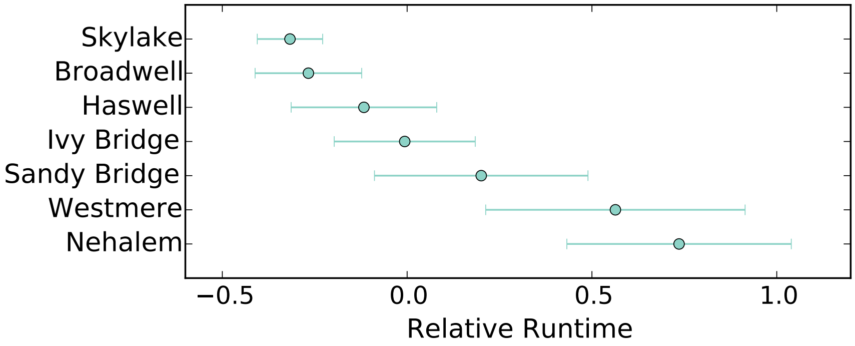


Figure 2.2: Means and standard deviations of SPEC relative run times.

show the mean relative run time of the SKUs with the same microarchitecture code name, and the lengths of the bars show the standard deviations. Longer bars indicate the performance of the SKUs is more diverse. The SKUs have the same microarchitecture but different frequency, cache size, memory size, and so on. These lead to variations in each bar, and *they contribute as much to performance variation as the microarchitectures themselves.* The bars overlap horizontally. That means when selecting SKUs to optimize performance, there can be many choices with different microarchitectures but similar performance.

The SKUs in the public repositories are random, depending on users' submissions. SPEC and Geekbench have their own setup and compilation instructions. The setup of each data point is included in the repositories. We summarize the SKUs in the SPEC and Geekbench datasets in Figure 2.3. Our work focuses on performance prediction on the SKUs currently on the market, thus we discard SKUs older than Sandy Bridge. We first do preprocessing to aggregate duplicate data. Before explaining preprocessing, we first define a CPU configuration as a SKU running at a certain frequency with a certain memory size. In this chapter, we say that a configuration (SKU, frequency, memory size) determines a workload's performance. Other factors such as compiler and OS affect performance as well, but we do not consider them in this chapter. In the repositories, multiple result submissions may have the same configuration. We average the data points that have the same workload and configuration.

Figure 2.3 shows the microarchitectures and types of the SKUs in our SPEC and Geekbench datasets. There are 352 SKUs in total for SPEC, and 119 SKUs for Geekbench. In both datasets, we have more SKUs of Sandy Bridge, Ivy Bridge, and Haswell than Broadwell and Skylake. Figure 2.3 shows that we have a fairly rich collection of SKUs.

Users can customize the memory size with a chosen SKU. We refer to a SKU with a certain memory size as a configuration. In total, we have 639 configurations of 352 SKUs for the SPEC workloads. To help visualize the performance of SPEC workloads on 639 configurations, we first show, in Figure 2.4, the means and standard deviations of the relative run times of the workloads. The blue bars are SPECfp; the red bars, SPECint. SPECint workloads are very similar to each other, except for 462.libquantum. SPECfp workloads have more diverse means and standard deviations. That indicates FP workloads are more diverse in terms of performance on the 639 configurations. The performance of 462.libquantum is more similar to SPECfp. If clustering the workloads, we would like to see that they form two clusters, with SPECint and some SPECfp workloads in one cluster and the other FP workloads, which have quite different means and standard deviations in Figure 2.4, in the other cluster.

Assuming the performance of the 639 CPU configurations describes unique characteristics of a workload, we can use a vector consisting of the performance of all configurations as the finger print of each workload. Since it is a space of 639 dimensions, we do principal component analysis (PCA) to visualize the finger prints of SPEC workloads in a two-dimensional space. We first construct a matrix of size $[28 \times 639]$ where each row is the finger print of one SPEC workload of length $=$ the number of CPU configurations. We then do PCA to reduce the number of dimensions to two and visualize the space. Figure 2.5 shows the PCA space. We only show two principal components (PCs) because the first and second PCs account for 82% of the variance.

The closer two workloads are, the more similar their performance scaling across the 639 configurations. Blue dots are SPECfp and red are SPECint. SPECfp is more spread out than SPECint. That means SPECfp workloads are more sensitive to SKU configuration differences, such as microarchitecture, frequency, and memory size. It is also an indication that it is harder for current microarchitectures to extract performance from SPECint than FP. SPECint workloads have less instruction level parallelism and more data dependencies. This is also observed by Campanoni et al.[37].

The points at the bottom of Figure 2.5 form two clusters, and the point at the top (481.wrf) is clearly an outlier. We want to find the centroids of the two clusters while leaving 481.wrf alone. Running a k-means algorithm with three clusters leaves 481.wrf in a cluster by itself. We plot the two centroids of the other two clusters using stars. 462.libquantum (the red dot at the bottom left) is clustered with SPECfp. The centroids will be used to explain our results in Section 2.5.2.

Although Figures 2.4 and 2.5 lead to consistent conclusions, they characterize the performance space differently. Figure 2.4 focuses on the means and standard deviations of the performance data, while PCA in Figure 2.5 emphasizes different performance on different SKU configurations. This can be explained with an example using two workloads (A, B) and three SKU configurations. Workload A's relative run times are 0.5, 0.7, -0.4, respectively. On the same SKU configurations, workload B's relative run times are 0.7, -0.4 and 0.5. The two workloads could look exactly the same in Figure 2.4, but PCA would show their difference.

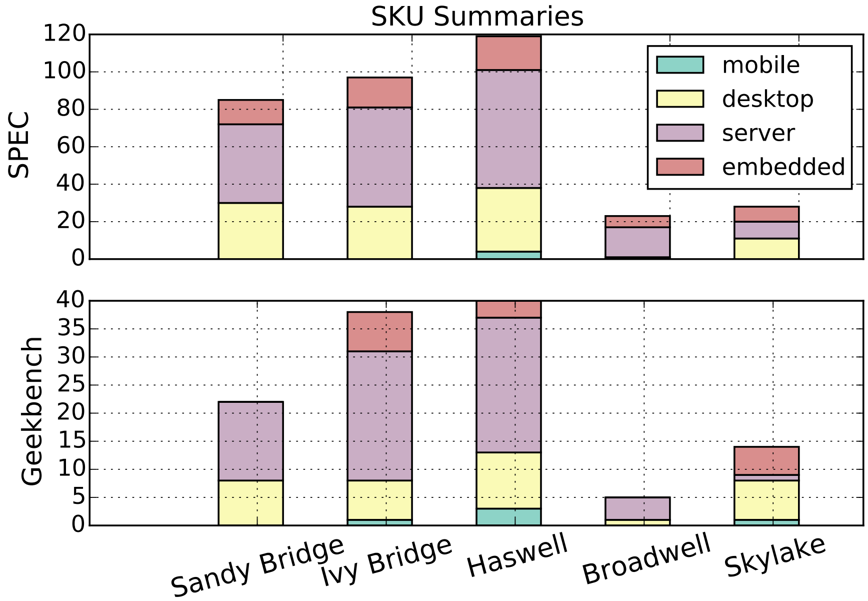


Figure 2.3: SKU breakdown in the SPEC (top) and Geekbench (bottom) datasets.

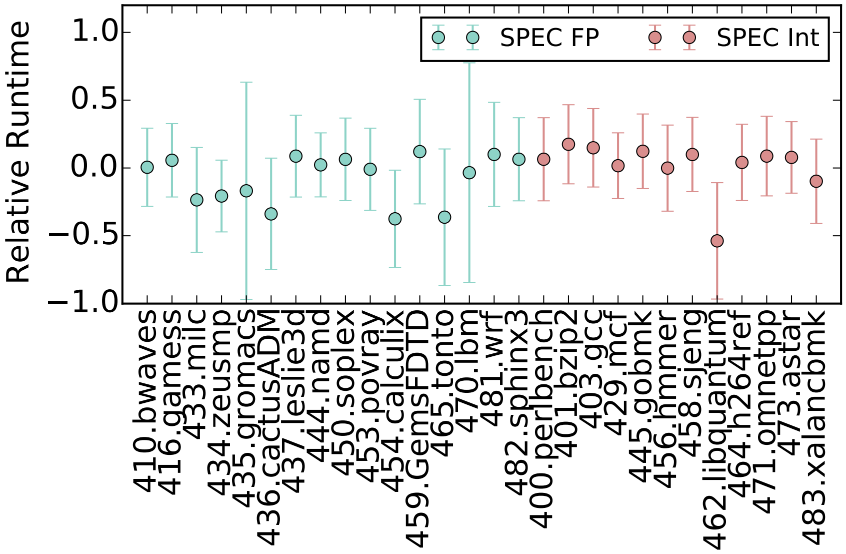


Figure 2.4: Means and standard deviations of SPEC workloads' relative run times on the SKUs in our dataset.

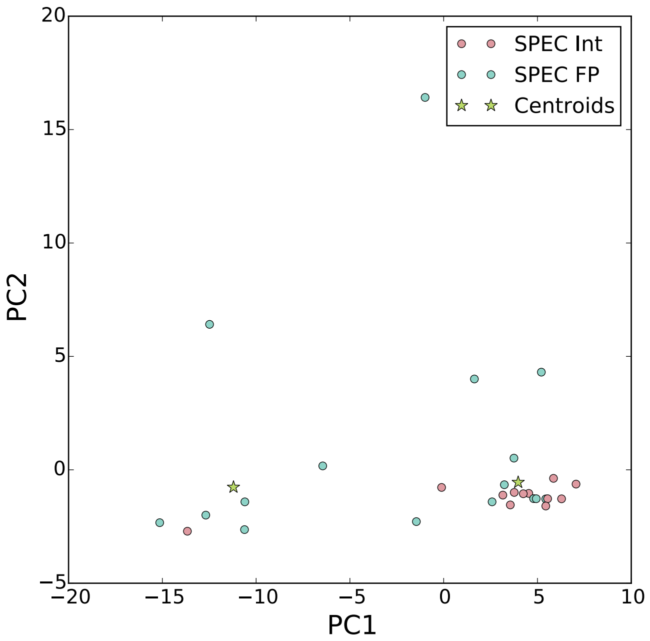


Figure 2.5: Principal component analysis of SPEC workloads' performance scaling on 639 configurations of 352 SKUs. The green stars are centroids of two clusters identified by k-means. The outlier near the top is 481.wrf.

2.3 Methodology

The previous section shows a large variation in performance. Thus a predictive model can be helpful. This section shows the method we use to build a predictive model. We present our features, data preprocessing, model selection, and metrics (baselines) for evaluating the results.

A naive way to make a prediction is to use a previous generation to predict its successor. To do this, one needs the exact configuration of interest from previous generation, because sometimes other features are more important than microarchitecture generations, shown in the bars of Figure 2.2. For the same reason, using other SKUs in one generation to predict a new SKU in the same generation is difficult. We have tried that, and our current predictive models achieve better accuracy. We eventually decide to use machine learning to learn the relations between CPU specifications and performance, to avoid the missing data problem of the naive methods.

2.3.1 Features

Table 2.1 summarizes the data in this chapter. The data include features fed to the predictive model, and the value we predict, the run time. The features include CPU specifications from the Intel CPU dataset and dynamic workload features from the SPEC and Geekbench datasets.
CPU SPECIFICATIONS: The microarchitecture code names include microarchitecture changes and technology scaling. Within one microarchitecture generation, the SKUs are different in their frequencies, last level cache sizes, TDP, and number of cores. We also add the release year as a feature. The type of the SKU is also an important feature. Intel categorizes the SKUs as mobile, desktop, server, and embedded. L2 and L1 caches per core do not change much across SKUs. Thus we only include last level cache sizes.

For discrete data, we do integer encoding and one hot encoding. Integer encoding means mapping discrete values to integers. Each feature needs only one entry. Discrete names and types usually obey the order of the assigned integers. One hot encoding uses one feature entry for one discrete value. It implies nothing more than that the discrete values are different. The encoding methods are specified in Table 2.1. We choose to map the microarchitecture code names, memory types, and instruction set extensions into consecutive integers, with their natural order. For the SKU types, it is easier to do one hot encoding. We also distinguish the workloads using one hot encoding. Before training and testing, numerical and integer data need to be standardized, but one hot encoding data do not.

DYNAMIC WORKLOAD FEATURES: : Every entry in the SPEC and Geekbench datasets contains a configuration (SKU, current frequency, memory size), the name of the workload, and performance (relative run time, which is the value to predict). We then get the CPU specifications from the Intel processor dataset by the SKU processor number. We do not use the scores provided in SPEC and Geekbench as performance metrics. For SPEC, we use the run time in seconds. For Geekbench, we assume that the total amount of work is constant for a given workload and we use the inverse of throughput to estimate run time. Details on the data preprocessing are in Section 2.3.2.

2.3.2 Data Preprocessing

The SPEC2006 repository provides run time in seconds. Geekbench 3 provides throughput, e.g., as GB/second. We first average the performance of the workloads with the same (CPU SKU, Frequency, Memory Size) configuration. Variations of system software such as OS or compiler are not considered in this chapter. Geekbench 3 run time is taken to be the inverse of throughput, assuming that one workload has a constant total amount of work. The average run time is then converted into relative run time for the chosen reference machine based on Equation 2.1. The specific choice of the reference machine does not affect the prediction. The relative performance is normalized to have mean 0 and standard deviation 1. The preprocessing steps are typical of machine learning experiments[143].

2.3.3 Model Selection

In this section, we explain our choice of deep neural networks (DNN) and linear regression (LR) for this chapter. We then illustrate our approach for tuning DNN topology and the hyperparameters of DNN and LR.

DEEP NEURAL NETWORKS: Deep neural networks (DNN) have proven successful in many domains, from regression and classification to game playing. Compared with traditional methods, DNN excels when data size is large, scales well with more data, and it does not require expertise in feature engineering. The advantages of DNN make it a very good fit for the performance prediction scenario. Thus in this chapter, we use DNN as our predictive model.

The topology and hyperparameters are tuned through model selection. To do model selection, we randomly split the dataset into a training set, a validation set, and a testing set. The testing set is the last hold out set, to avoid using all data to do model selection. We start by sweeping the hyperparameters in all experiments, including the number of layers, the nodes per layer, the learning rate, the number of training epochs, the activation function, and the optimizer. The models are trained with the training set. The model with the lowest loss on the validation set is selected.

We find that the best set of hyperparameters is always the same. Therefore in all experiments we use a DNN with 3 hidden layers and 100 nodes per layer. The learning rate is 0.001 over 300 training epochs. The activation function is tanh and the optimizer is RMSprop. Thus, for example, the number of weights for predicting SPEC is 50 million. Fortunately, training is offline and has a one-time cost.

LINEAR REGRESSION: LR has been shown to work well in the CPU performance prediction literature[129], while DNN is known to be able to explore non-linear interactions between features, and it has been shown to be more accurate than LR at performance prediction [102,103]. Unlike DNNs, LR has interpretable weight parameters. In realistic scenarios, people can choose LR or DNN based on the parameters, accuracy, and the value of interpretable weights. In order to make it a fair comparison, we reimplement LR with MAE as the loss function. That is, the model is optimized to yield the lowest MAE. Minimizing the MAE entails assuming the noise distribution to be Laplacian[48]. In our experiments, both DNN and LR have L1 regularization with a weight of 0.01. They take exactly the same training sets, testing sets, and features.

OTHER MODELS: Popular predictive models other than DNN include linear regression (LR), support vector machines (SVM), *k* nearest neighbors (kNN), principal component analysis (PCA), and genetic algorithms (GA). In Section 2.7, we list studies using some of those models. However SVM is not suitable for large scale training because of its computational complexity[199]. kNN, PCA and GA are better when there are fewer data points (< 100) and simpler feature relations. Besides, traditional methods need carefully selected features, and accuracy does not benefit from more training data. Therefore we choose to use DNN and compare it with one traditional method, LR.

2.3.4 Metrics

MEAN ABSOLUTE ERROR: We use mean absolute error (MAE) as the measure of accuracy, rather than mean squared error (MSE). The reasons are as follows.

1. There may be outliers and errors in the submitted data. With MSE, the optimizer penalizes the outliers much more than normal data. However, we do not want to overfit the model to the outliers[143].
2. If a prediction is x and the MAE of the model is 0.01, the real value of x is very likely to be $x \pm 0.01$. MSE is not as easy to interpret.

Because the MAE is in a standardized space, we use two baselines to help understand it. We assume two independent and identically distributed random variables, a and b, from Gaussian distribution N(0, 1). The expected distance between a and b is

$$\frac{2}{\sqrt{\pi}}, \tag{2.2}$$

which is approximately 1.13[185]. Therefore a prediction with MAE of 0.25 is a very good prediction. Another way to interpret the MAE is to compare it with the standard deviation of the testing set. A naive way to do prediction is to always predict the mean of the dataset. This way, by definition the MAE is the standard deviation

$$\text{MAE} = \frac{\sum_{i=0}^{n} |x_i - m|}{n} = \text{Standard Deviation} \tag{2.3}$$

where n is the number of data points, m is the mean of the testing set, and x_i is the real value of predicted performance, after normalization. Note that while the whole dataset is normalized to have mean 0 and standard deviation 1, the testing sets may be in different scales. In the captions of Figures 2.6 and 2.7, we compare the MAEs with the standard deviations of the testing sets, and show that the MAEs are much smaller. However what we just discussed are naive baselines. The next metric can show that our prediction is good enough in realistic scenarios.

TOP-*K* ACCURACY: To bring the MAE metric into realistic context, we use the predictive model to help select SKUs. We predict performance and use that prediction to rank the SKUs.

We choose top-k accuracy, which measures whether the known best SKU is among the top k predicted SKUs. It is a popular metric for image classification tasks[87,123]. Other metrics for comparing rankings include Pearson correlation, Spearman footrule distance, and Kendall rank correlation. They measure the difference between the tested ranking and the expected ranking, by considering every item in each ranking. Top-k accuracy is a better metric for this chapter, because users care more about whether the best SKU is in the top k choices, than about how similar one SKU ranking is to another. In this context, top-*k* accuracy is more interpretable than the ranking correlations.

We construct three baselines to compare with our predictive model. Without the predictive model, it is customary to compare microarchitecture code names (newer is better), frequencies (higher is better), and cache sizes (larger is better). Thus we construct the baselines taking those factors into consideration. Though the baselines are simple, this is how most customers make purchasing decisions.

1. Frequency: Rank the SKUs based on frequencies. If frequencies are the same, rank them with cache sizes.

2. Cache: Rank the SKUs based on cache sizes. If cache sizes are the same, rank them with frequencies.

3. Frequency + Cache: Rank the SKUs based on a summation of frequency and cache size with weights. The weights are discussed in Section 2.5.1. If the values are the same, rank them with microarchitecture code names.

To measure the top-*k* accuracy, we shuffle our training or testing set, find the best SKU in the testing set using real performance from the datasets, and measure how often the best SKU is one of the top *k* SKUs as ranked by our predictive model and the baselines.

Table 2.1: Data in this chapter. The data include CPU specifications (from the Intel processor dataset) and dynamic workload features (from SPEC and Geekbench datasets), and what we predict (run time).

Data	Description	Data Type
Workload	The names of the workloads	One hot encode
μarchitecture Code Name	Intel code names	Integer encode
Type	Server, desktop, mobile, or embedded	One hot encode
L3 Cache Size	Last level cache size	Numerical
Instruction Set Extensions	SSE or AVX	Integer encode
Memory Type	DDR,DDR2,DDR3,DDR4	Integer encode
Memory Channels	Number of memory channels	Numerical
Max Memory Bandwidth	Max memory bandwidth	Numerical
ECC Memory Support	Whether ECC memory is supported	Binary
Base Frequency	Nominal frequency	Numerical
Turbo Frequency	Turbo frequency	Numerical
Turbo Boost Technology	Whether Turbo Boost is supported	Binary
Cores	Number of cores	Numerical
Threads	Number of threads	Numerical
Hyper-Threading	Whether hyper-threading is supported	Binary
TDP	Thermal design power	Numerical
Year	Year of release	Numerical
Frequency	The dynamic frequency	Numerical
Memory Size	The size of off-chip memory	Numerical
Run Time	The run times of the workloads	Numerical

2.4 Case Studies Overview

In this section, we sketch the case studies, including prediction for new SKUs and for new workloads, and cross-prediction between suites. We implement our models with Keras (`https://keras.io`).

2.4.1 Case 1: Prediction for New SKUs

In this case, we show that we can predict the performance of SPEC and Geekbench workloads on new SKUs. Using our open-source model, trained with SPEC and Geekbench data for existing SKUs from online repositories, users can predict Geekbench and SPEC performance on new SKUs of interest and choose the ones likely to perform the best.

In order to test the model on new SKUs, we adopt repeated random sub-sampling validation, a widely-used cross-validation method[62]. We hold out several SKUs from the dataset, train the model with the remaining data, and test the model with the hold out set.

We randomly choose 10 percent of the SKUs from one microarchitecture at a time as our testing set. We use the rest of the data as our training set. We repeat this process 20 times for every microarchitecture. We choose 20 repetitions after trying 50 and finding that the results were statistically consistent. Comparing the MAEs of 20 DNN repetitions versus 50, the maximum absolute difference is 0.005 and the standard deviation difference is 0.009. Both are minimal.

The premise of this case study is that the predictive models are able to learn the performance function of a workload using the CPU specifications in Table 2.1. Specifically, the training set may contain SKUs with the same microarchitecture as the new SKU, and SKUs with similar features such as L3 cache, frequency, and memory size, but different microarchitecture. The predictive model then predicts the new SKU's performance as though interpolating the training set.

Note that the prediction is nontrivial, even if the same microarchitecture or the full feature set of a different microarchitecture is in the training set. As shown in Figure 2.2, both microarchitecture and other SKU features (such as frequency) introduce large variations in performance. Knowing one of them does not make the prediction easy.

2.4.2 Case 2: Prediction for New Workloads

In this study, we show that after measuring performance of a new workload on just a handful of SKUs, we can train a model to predict the workload's performance on other SKUs. Armed with such predictions, consumers can choose the best processor for the workload based on both performance and price.

To demonstrate the approach, we split our dataset, taking out one workload at a time and treating it as the new workload. Then we train the model with all the other workloads' data plus a certain amount of data from the chosen workload. We perform the experiment separately for SPEC and Geekbench. We refer to this case as self-prediction of benchmark suites.

The premise of this and the next case study is that, given the data of a new workload on several SKUs, the model can find workloads with similar performance on those SKUs, and use the performance of those workloads to predict the new workload's performance on other SKUs. Therefore the prediction accuracy should be related to the similarity of the new workload to the workloads in the training set.

For every workload A, we remove the data for workload A from the overall dataset. We call the remaining data *data_rest*. We then sample a fixed set of SKUs for workload A as a testing set, and we refer to the rest of the data for workload A as *train_pool*. We randomly pick *n* SKUs from *train_pool* (where *n* is 1, 5, 10, or 50) and combine their data with *data_rest* to form a training set. We repeat the process for 20 iterations.

2.4.3 Case 3: Cross-Prediction Between Suites

The experiments in case 2 are conducted within each benchmark suite. Case 3 is an extension of case 2 that we call cross-prediction. We predict one workload in Geekbench using all data for SPEC plus that collected using several SKUs for the chosen workload. Then we repeat with Geekbench and SPEC switched.

For every workload A in Geekbench, we split the whole dataset into the data of workload A and the rest. We then sample a fixed set of SKUs from workload A as a testing set, and we refer to the remaining data of workload A as *train_pool*. We randomly pick *n* SKUs from *train_pool* and combine their data with the SPEC data as a training set. We sweep *n* over 1, 5, 10, and 50, and we repeat the process 20 times.

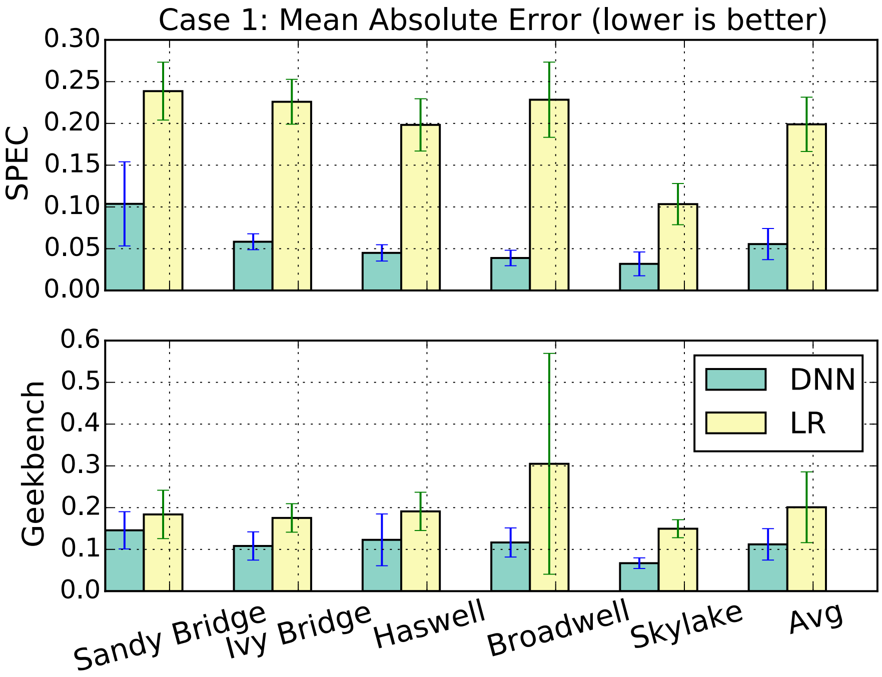


Figure 2.6: Results of predicting the performance of SPEC (top) and Geekbench (bottom) on new SKUs.

2.5 Prediction Accuracy

In this section, we show prediction accuracy measured by mean absolute error (MAE) in the case studies. We show that DNN is more accurate than LR, and case 2 (self-prediction) is more accurate than case 3 (cross-prediction).

2.5.1 Case 1: Prediction for New SKUs

Figure 2.6 compares the MAEs of LR and DNN for SPEC and Geekbench, respectively. The heights of the bars represent the average MAEs of the 20 random test sets. The error bars show one standard deviation of the MAEs. A smaller error bar indicates that different selections of testing set give large differences in MAEs, therefore the model is more robust to predict different SKUs.

For both SPEC and Geekbench, DNN always has lower MAEs and is less sensitive to the selection of the test set than LR. That means the relationships between features and performance are not simply linear. In Figure 2.6 (bottom), LR has a very large MAE and standard deviation when predicting Geekbench on Broadwell SKUs. That is because in Figure 2.3 (bottom), there are only five Broadwell SKUs, and one is desktop while the others are servers. Using servers to predict desktop performance leads to large MAEs for LR. Apparently DNN addresses this better. To compare the average MAEs (the last set of bars in Figure 2.6), LR is 19.9% and DNN is 5.5% for SPEC, and LR is 20% and DNN is 11.2% for Geekbench. DNN is a better choice when the data set is large and interactions between features are complicated.

The average MAE is 5.5% for SPEC and 11.2% for Geekbench using DNN. The MAE for Geekbench is higher than SPEC because Geekbench users are not like SPEC users, who are mostly computer architects and system engineers, benchmarking on dedicated machines. One can easily install Geekbench and produce scores on personal devices. Results may be noisy if other applications are running at the same time, causing contention in computing and memory resources. We observe larger variance in Geekbench data. We compute the average standard deviation of the performance for the same workload with the same (SKU, frequency, memory) configuration. It is 4.4% for Geekbench, and 1.1% for SPEC. The larger noise from the data makes it harder to predict Geekbench.

The accuracy of simulation-based prediction is about 4%[129], which is slightly lower than our 5.5% MAE of SPEC. Compared to simulation-based approach, the extra noise in our approach comes from the performance variation introduced by compilers and operating systems in the datasets, and no architectural features. We use no architectural features which makes the prediction harder, but it also makes the method easier to apply to any other public datasets, without the need for simulation.

2.5.2 Case 2: Prediction for New Workloads

In this section, we show the prediction results of case 2. We show that the DNN prediction accuracy is largely determined by the similarity of the predicted workload to the rest of the benchmark suite. We quantify the outlierness of a workload and show that it is positively correlated with the prediction MAE, with a Pearson correlation of 0.69.

PREDICTION RESULTS: The results of SPEC are shown in Figure 2.7 (left). The workloads are sorted based on the MAEs after adding 50 SKUs (brown line). (The line for 5 added SKUs is omitted but the average bar is shown.) The standard deviations of all the testing sets are larger than 20%, and our MAEs in all cases are lower than the standard deviation. For 5, 10, and 50 added SKUs, the MAEs are 5.7%, 5.3%, and 4.7%, respectively. MAE improvements when moving from 10 SKUs to 50 are similar to those when moving from 5 SKUs to 10. Since adding 40 SKUs is more difficult than adding 5, we conclude that using 10 SKUs for the new workload is a reasonable choice.

Similarly, the Geekbench prediction results are shown in Figure 2.7 (right). The workloads are sorted and the rightmost workloads are outliers with very high MAEs. The standard deviations of Geekbench testing sets are all above 11%. The outliers' standard deviations are at least twice their

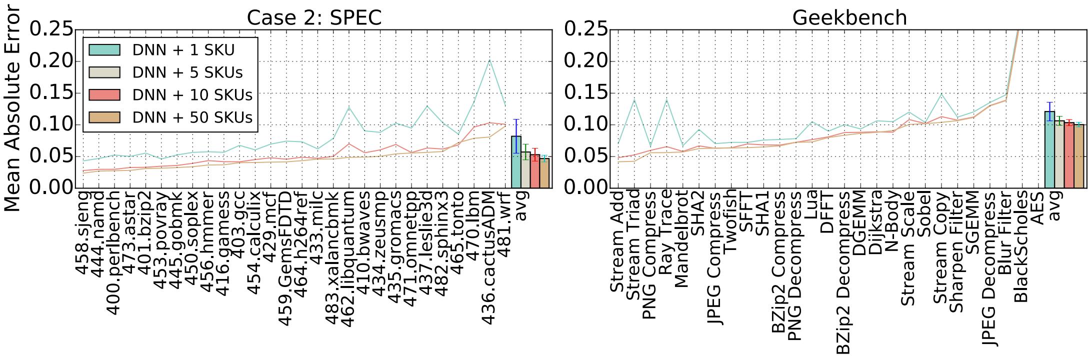


Figure 2.7: The results of case 2. The workloads are sorted by the MAE after adding 50 SKUs.

MAEs. That means our model shrinks the confidence interval by at least 50%.

Case 2 is harder than case 1. In case 1, the average MAE is 11%. In case 2, the average MAE is 14.2%, with two obvious outlier workloads (Blackscholes and AES). We conclude that case 2 works reasonably well, apart from the two outliers. Also, for the same reason as with SPEC, we choose 10 as the proper number of SKUs to use.

INSIGHTS ON BENCHMARK SIMILARITY: By studying MAEs across SPEC workloads, we find that the prediction MAE is correlated with the similarity of the predicted workload to the rest of the benchmark suite. This observation supports the statement in Section 2.4.2 that the model finds similar workloads based on the data points of the new workload, and it uses the similar workloads to predict the performance of the new workload on other SKUs.

In Figure 2.7 (left), after adding 50 SKUs it is still relatively hard to predict the workloads on the right. We call these workloads outliers. The workload with highest MAE is 481.wrf and it is also the outlier lying above all other workloads in Figure 2.5, which shows the workloads' relative performance scaling in the varied SKU configurations.

Some outliers are well-studied in prior works. Phansalkar et al. show that 436.cactusADM is an outlier in terms of memory access characteristics in Figure 9 of [154]. However, workload features collected in prior works are unable to explain why 481.wrf is such an outlier in SKU scaling space, as shown in Figures 2.7 and 2.5. Future work will try to answer that question.

To study the outlierness of the workloads in Figure 2.7 (left), we ran a k-means algorithm to split the workloads into two clusters, *cluster*1 and *cluster*2. Their centroids C_1 and C_2 are plotted as stars in Figure 2.5. We compute the distance of a workload to its cluster's centroid as

$$\begin{aligned} distance = I(x \in cluster1)||x - C_1||_2 \\ +I(x \in cluster2)||x - C_2||_2 \end{aligned} \tag{2.4}$$

where x stands for the location of the workload in Figure 2.5 and I yields 1 if its argument is true and 0 otherwise. The distance quantifies the outlierness of a workload.

Figure 2.8 plots a workload's MAE after 50 added SKUs (y-axis) against its distance (x-axis). Distance and MAE have a Pearson correlation of 0.6898. We can use distance to identify workloads with high errors. Workloads with distance higher than 5 have MAEs higher than 10%. This correlation explains why the outliers have higher MAEs than others even after adding 50 SKUs. CONCLUSION: This case shows that we are able to predict a new workload's performance by running it on 10 SKUs. The accuracy of the prediction depends on the similarity of the workload to those in the training set. Even if it is very different, our model is still able to shrink the confidence interval.

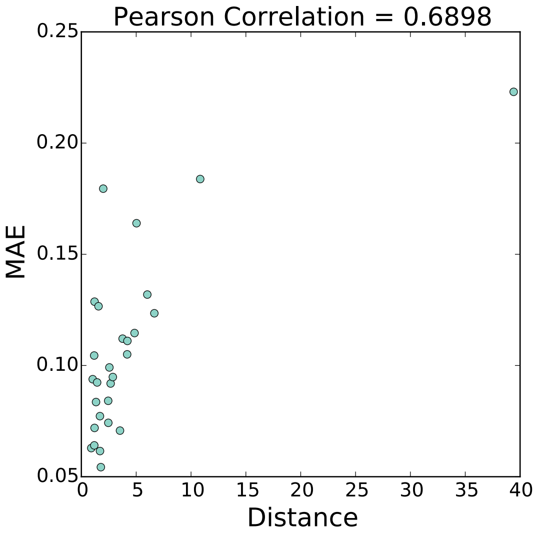


Figure 2.8: Correlation of SPEC workloads' outlierness (distance) with MAE after adding 50 SKUs. The distance of a workload is its distance from the the centroid in its cluster. Centroids are in Figure 2.5.

2.5.3 Case 3: Cross-Prediction Between Suites

Figure 2.9 shows the results of cross-prediction: using Geekbench to predict SPEC (top) and *vice versa* (bottom). Because the MAEs of the workloads and the outliers are similar to case 2 (self-prediction), we only show the average MAE bars in this case.

To compare self-prediction and cross-prediction, we first compare Figure 2.9 (top) with the bars in Figure 2.7 (left), both of which predict SPEC workloads. Cross-prediction gives higher MAEs, for any number of added SKUs. After adding 50 SKUs, the average MAE of cross-prediction is 16.9%; the average MAE of self-prediction is 4.6%. Comparing the predictions of Geekbench in Figure 2.9 (bottom) and the bars in Figure 2.7 (right), after adding 50 SKUs the average MAE for cross-prediction is 12.6% and that for self-prediction is 10%.

Cross-prediction gives consistently higher errors because the workloads in SPEC and Geekbench are very different, more different than the workloads in either benchmark suite itself. In another words, the workloads in a benchmark suite are more similar to each other than to workloads outside of the benchmark suite.

Benchmark suites like SPEC CPU2006 are expected to be diverse and representative of real-world workloads. However, a benchmark suite is developed for a purpose, and different suites usually have different purposes. The purpose itself is the reason why the workloads in the suite are similar. The SPEC CPU2006 benchmark suite is intended for computer designers and architects to use for pre-silicon design analysis. SPEC workloads are drawn from and representative of real workloads.

In contrast, Geekbench is intended to stress a certain part of the hardware. For example, memory microbenchmarks do a lot of memory streaming operations to study the bandwidth limit of the hardware system. From this point of view, it is reasonable that SPEC and Geekbench are very different and the workloads in each suite are more similar.

CONCLUSION: Cross-prediction is harder than self-prediction. To predict performance for new workloads, one needs a training set with a very diverse collection of workloads. Benchmark suites tend to be self-similar, because each serves a specific purpose. Even so, cross-prediction is useful, as shown in the next section.

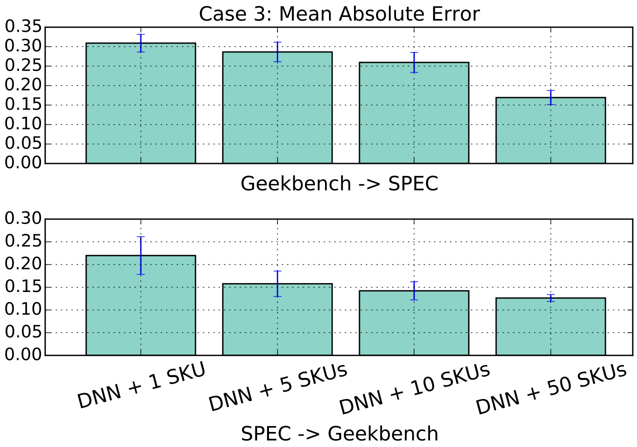


Figure 2.9: Case 3 is about cross-prediction. SPEC workloads are predicted with the Geekbench dataset (top). Geekbench workloads are predicted with the SPEC dataset (bottom). Cross-prediction has higher errors than self-prediction (case 2).

2.6 SKU Ranking Comparison

In this section, we use the performance prediction results of the DNN model to rank SKUs. We then compare the SKU rankings with the baseline metrics described in Section 2.3.4. The baselines represent the customary ways of making purchasing decisions. Our methodology makes SKU selection more accurate without requiring expertise on workload characterization or computer architecture specifications.

2.6.1 Case 1: Prediction for New SKUs

In this section, we show that our DNN model outperforms the three baselines in Section 2.3.4 when using SPEC and Geekbench to select new SKUs.

The results of SPEC and Geekbench are shown in Figure 2.10. For baseline C, a combination of frequency and cache, we swept 100 combinations of weights. We combined $frequency(GHz) \times w$ with $cache(MB) \times (1 - w)$, and swept w from 0.01 to 1, with a step size of 0.01. In Figure 2.10, we use $0.9 \times frequency + 0.1 \times cache$ so that the frequency and cache values are in about the same range. The top-3 accuracy of baseline C combining frequency and cache is no better than A (frequency) or B (cache).

Frequency is the best metric among the three baselines. The following subsections show similar results, therefore we show baseline A and drop the others in the rest of the paper.

Our predictive model outperforms all of the workload-indifferent baselines. For Geekbench, the frequency metric achieves accuracy comparable to that of our DNN model. The Geekbench suite as a whole is quite frequency-sensitive. The SPEC suite, on the other hand, is sensitive to neither frequency nor cache, which suggests that choosing SKUs for complicated workloads requires more than simply comparing frequency or cache.

CONCLUSION: When selecting a SKU that optimizes SPEC or Geekbench performance, our pre-

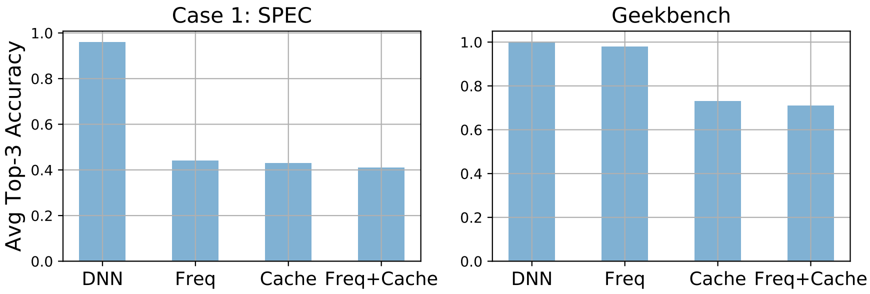


Figure 2.10: SKU ranking results of case 1. Our DNN model has the highest top-3 accuracy.

dictive model works better than simply comparing frequencies, cache sizes, or a combination of the two. Selecting a SKU by frequency is better than by cache size, and is more successful for microbenchmarks like Geekbench than for complicated workloads like SPEC.

2.6.2 Cases 2 and 3: Self- and Cross-Prediction

In this section, we compare the top-*k* accuracies of cases 2 and 3 and baseline A (frequency). Results using baselines B and C are consistent with those presented, so we omit them. We show case 2 and case 3 in the same plots to compare their results. Both cases use 10 additional SKUs for the new workload. The results appear in Figure 2.11.

It is very difficult to find the best SKU for SPEC workloads for cases other than SPEC self-prediction (case 2). The top-3 accuracy of case 3 is very small, and that of frequency is 0. We start to observe more accuracy from them when we relax the choices to top-7 SKUs. It indicates building the predictive model is the only reasonable way to find good SKUs for SPEC. The situation can get worse for the workloads in people's daily lives such as web browser, Microsoft Office suite, Adobe suite, and other customized workloads from individual companies.

Self-prediction has higher accuracies than cross-prediction for most of the workloads, suggesting self-prediction is easier. The ranking accuracies further support what has been observed from the

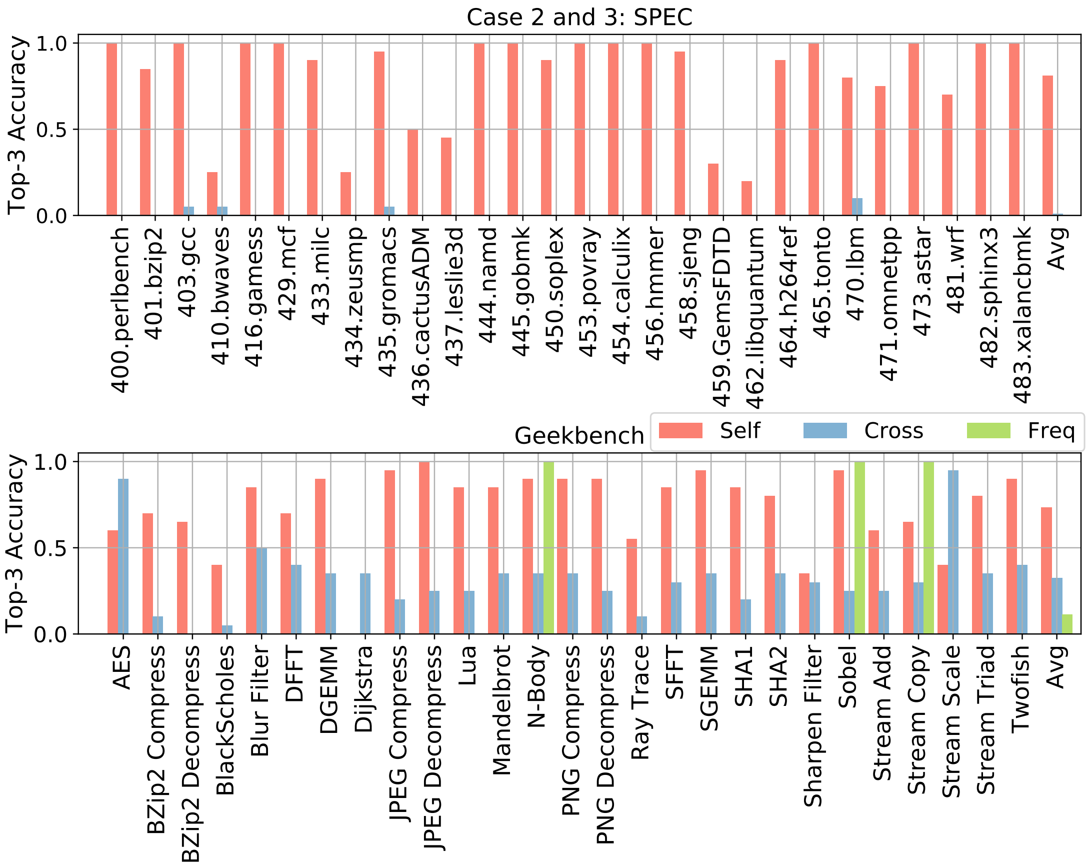


Figure 2.11: Based on SPEC (top) and Geekbench (bottom), compare the SKU rankings by self-prediction (case 2), cross-prediction (case 3) and the best baseline (Frequency).

prediction accuracy studies in Figures 2.7 and 2.9, and the self-similarity statement in Section 2.5.3. Exceptions are AES, Dijkstra, and Stream Scale from Geekbench, which have higher accuracies when trained with SPEC. Among them AES is the outlier with the highest MAE from the self-prediction accuracy results in Figure 2.7 (right). The results indicate that those three workloads are more similar to SPEC in terms of selecting the best SKUs.

Self- and cross-prediction are more accurate than frequency. Note that in cases 2 and 3, the test set is fixed, and the only variation in the 50 iterations is the random additional 10 SKUs from the new

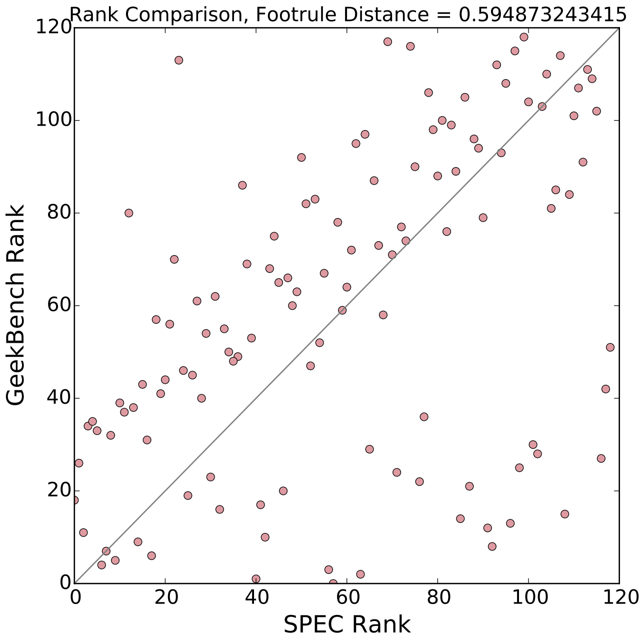


Figure 2.12: Comparison of SKU rankings based on SPEC and Geekbench averages.

workload, as described in Section 2.4. Therefore for frequency bars in Figure 2.11, there is only one ranking of SKUs to test, and the top-*k* accuracy is either 0 or 1, depending on whether the best SKU is ranked top-*k* or not. By averaging the accuracies for the whole suite, we find that frequency performs the worst. Though the previous subsection shows the accuracy of frequency to be comparable to DNN prediction for Geekbench (Figure 2.10), individual workloads may or may not be sensitive to frequency. Frequency is not a good way to select SKUs.

CONCLUSION: SPEC self-prediction is the only reasonable way to find the best SKU for SPEC workloads. Self-prediction is easier than cross-prediction in most cases. Frequency has the lowest accuracy, so frequency alone should not be the deciding factor when selecting SKUs for new workloads.

2.6.3 Benchmark Suite Comparison

In this section, we compare the benchmark suites in terms of SKU ranking. While there is no prediction involved, it is natural to compare the two benchmark suites after comparing the individual workloads in them.

Geekbench is very popular on computer enthusiast websites, with many articles claiming that "Geekbench suggests A outperforms B". Such articles affect many consumers' choices when purchasing computing products. Most people use laptops to run workloads such as web browsers, Microsoft's Office suite, Adobe software, and so on. We describe these as realistic workloads. Consumers choose CPUs based on Geekbench scores because they assume that Geekbench predicts the performance of realistic workloads. Here we view SPEC workloads as falling between realistic workloads and Geekbench in terms of complexity and real-world relevance.

To show how the two suites produce different rankings, we rank the SKUs that SPEC and Geekbench datasets have in common by averaging each benchmark suite's performance on each such SKU. Figure 2.12 shows the ranking comparison. The *x*-axis is the rank based on SPEC and the *y*-axis is the same SKU's rank based on Geekbench. The footrule distance between the two rankings is 0.59. (The range is from 0 to 1.) If the ranks were consistent on both suites, the points would lie along the gray line ($x = y$). We observe two patterns in Figure 2.12. One is the cluster of points above the gray line; Geekbench ranks those SKUs lower than SPEC (gives them higher ranking indexes). Another pattern is the cluster of points below the gray line and close to the *x*-axis. SPEC ranks those SKUs from 0 to 120, while Geekbench ranks them from 0 to around 50. In another words, the SKUs have very different SPEC performance, but are poorly differentiated by Geekbench. For example, a CPU SKU highly ranked with Geekbench, e.g., with 1-20 ranking indexes, may be ranked between 1-120 with SPEC. Users picking such SKUs may not have desired performance.

The Geekbench SKU ranking is not consistent with SPEC rankings. That means that if an enthusiast article says "Geekbench shows that A outperforms B", SPEC may suggest that B outperforms A. The fundamental cause is those benchmark suites contain very different sets of workloads. Similarly, Geekbench and SPEC workloads can be different from the realistic workloads of end users (such as web browsers, Windows Office suite, and Adobe suite), and datacenters[112] (such as ads, storage, and search). As a result, both suites may not be good indicators for end users or datacenter performance engineers.

This suggests that people use Geekbench incorrectly. The aggregate score of a benchmark suite is misleading. Geekbench is designed to stress floating point, integer, and memory operations. It makes more sense to compare the performance of individual microbenchmarks that are relevant to realistic workloads. If one runs a lot of matrix operations and image processing workloads, such as MATLAB and Adobe Photoshop, the floating point benchmarks in Geekbench are very helpful. CONCLUSION: SKU rankings by Geekbench are inconsistent with rankings by SPEC, and probably also with rankings for realistic workloads. Aggregate benchmark suite scores are misleading. Benchmarking relevant individual workloads makes more sense. For a non-expert consumer, the best way is to take all datasets available, plus the new workload's performance on 10 SKUs, and train a predictive model as in cases 2 and 3.

2.7 Related Work

2.7.1 Performance Prediction

For performance prediction, our work uses deep neural networks (DNNs), regarded as the state-of-the-art machine learning algorithm. It has been proven to be able to simulate any function[52].

Unlike simulation-based performance prediction[129,102,103,169], our approach does not use microarchitecture features such as instruction issue width, read write buffer size, or number of pipeline stages. We use only the CPU specifications that manufacturers provide. Microarchitecture features would make performance prediction easier and more accurate, but our approach is more helpful for consumers choosing CPUs. Even when manufacturers describe microarchitecture features, consumers need expert knowledge to make use of such information.

Performance prediction is a well-studied field. There are mechanistic models[115,67,46,191,190,82,68], empirical models[64,132,177], and machine learning models[217,21]. Zheng et al. used performance counters and a ”stochastic dynamic coupling” heuristic for aligning host and target execution samples to estimate performance using only binary code[218]. In contrast, we focus on CPU SKU selection for consumers, rather than performance prediction for hardware/architecture designers. Our method does not use performance counters or architecture features other than those accessible to all consumers.

Piccart et al. use data transposition to rank commercial machines[157]. Their work exploits machine similarity rather than workload similarity. Our work has different objectives and uses different metrics. It exploits workload similarity and benchmark suite self-similarity to help users pick the best CPU SKU for their needs.

2.7.2 Workload Characterization

Phansalkar et al. develop a series of SPEC workload characterizations[154,153,155,156]. However they do not focus on workload performance scaling as we do, so their classification results are not consistent with ours. Our approach provides an orthogonal way to characterize workloads.

Some researchers use static workload features to predict performance. Shelepov et al. use spatial locality, cache size, and frequency to predict speedup on a variety of processors[170,171]. Hoste et al. collect microarchitecture-independent features to classify workload performance[97,98]. Delimitrou et al. study workload characteristics for cloud scheduling[57,58]. In our work, we predict a new workload's performance on hundreds of CPU SKUs. The number of workloads is much smaller than the number of hardware platforms. Because the workload features are static, we argue that it is hard to use static features in this chapter.

2.8 Summary

To help customers select proper CPU SKUs and overcome the limitations of public benchmark datasets, we have presented statistical and predictive analysis of workload performance, with data collected from the SPEC CPU and Geekbench 3 suites and Intel processor specifications.

We demonstrated performance prediction for *new SKUs* and *new workloads* with deep neural networks (DNNs). For new SKUs, the accuracy is 5% (SPEC) and 11% (Geekbench) mean absolute error (MAE). With the same accuracy, we find that we can predict a new workload's performance by running it on 10 SKUS. We compared our predictive methods against workload-indifferent metrics for selecting processors. Our predictive model is the only approach that achieves reasonable accuracy in all three case studies. Notably, workload-indifferent methods of SKU selection do not work for new workloads.

For the first time in the literature, we showed benchmark suite self-similarity quantitatively by cross-prediction and comparison of SPEC and Geekbench SKU rankings. The accuracy of cross-prediction is lower than that for prediction within the suites because they are so different: 25.9% mean error to predict SPEC and 14.2% to predict Geekbench.

Rankings based on average Geekbench are inconsistent with those based on average SPEC. Geekbench rankings do not imply SPEC rankings. It is also hard to draw any conclusions about realistic workloads from a benchmark suite. We suggest studying the Geekbench or SPEC workloads of interest, rather than relying on aggregate scores.

3

Parallelism in Deep Learning Frameworks

This chapter takes a deep dive into analyzing the performance impact of key design features of machine learning frameworks and the role of parallelism. The observations and insights distill into a simple set of guidelines that one can use to achieve much higher training and inference speedup. The evaluation results show that our proposed performance tuning guidelines outperform both the Intel and TensorFlow recommended settings by 1.29× and 1.34×, respectively, across a diverse set of real-world deep learning models.

3.1 Introduction

The popularity of deep learning (DL) has spawned a plethora of domain-specific frameworks for machine learning (ML) including Caffe/Caffe2[106], PyTorch[118], TensorFlow[14], and MXNet[44]. These frameworks all provide high-level APIs for the building blocks of DL models, largely reducing the prototyping cycle due to substantial use of libraries. This greatly improves the productivity of developers building end-to-end DL models. In addition to programmability benefits, these frameworks also provide many DL-specific optimizations to improve performance and portability across software stacks and new hardware systems. The net result is an explosion in the development of ever more complex DL models and a concomitant increase in the computation costs of training. Recent analysis shows a 3.5 month doubling time of AI training computation for popular DL models, exceeding the traditional performance growth of Moore's Law[17].

Computing performance is especially important for deep learning. When models go into production at scale, individual performance improvements can affect datacenter resources and whether a model can be deployed to performance- and energy-constrained mobile devices[86,178,205]. Even during the prototyping phase, when human overhead is a larger bottleneck than machine overhead, model training time is still on a critical path for DL developer productivity. Since all popular DL frameworks provide high-level abstractions, one important question is how much performance overhead this improved programmability is adding. A further question is whether we can reduce this "programmability tax" by tuning the complex set of design choices available in current frameworks. Answering both questions requires a comprehensive understanding of framework performance.

Previous work compared popular DL frameworks[24], without revealing the root causes of the performance difference. We believe the performance comparison of specific frameworks is not the key issue. When we break widely-used frameworks into components, i.e., design features, we find that frameworks have significantly overlapping features, which affect performance in the same way.

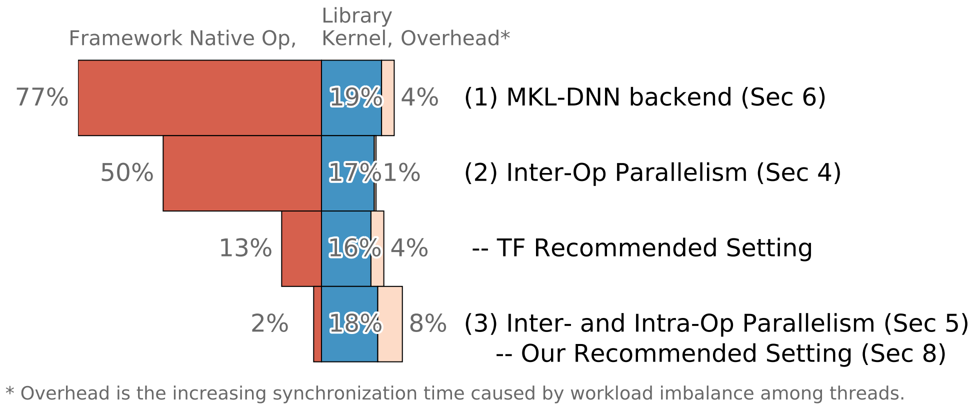


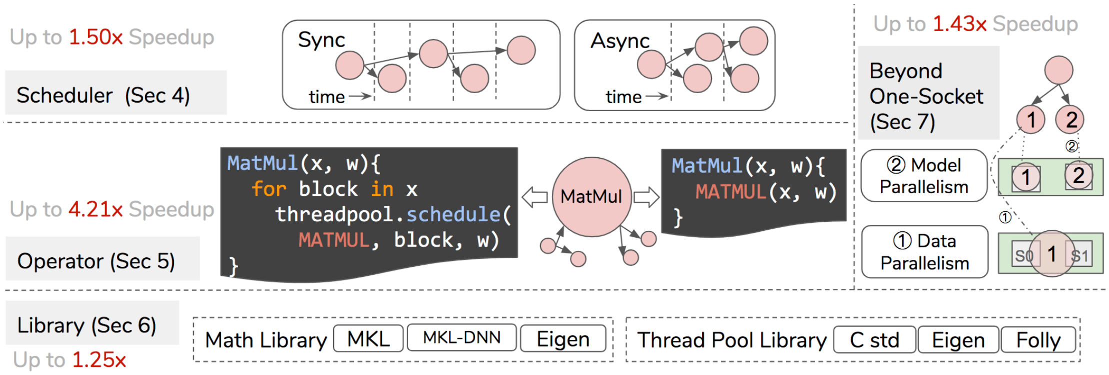
Figure 3.1: Time breakdown for Inception v3 training.

These design features include the rich set of parallelism opportunities within and between model layers and across input data (e.g., batches). Other key choices available to users include back-end math kernels, threading libraries, and scheduling policies.

Figure 3.1 shows an example of Inception v3 training on a dual-socket CPU platform, whose final performance is improved by 3.6× with properly tuned framework configurations. Fine-tuning of framework knobs requires expert knowledge of their performance impact. In this example, tuning the inter-operator parallelism speeds up the framework native operators by 1.54×. Exploiting intra-operator parallelism speeds up them by another 25×, translating to 2.4× performance improvement for the entire model. Compared to the recommended TensorFlow setting[75], our chosen setting speeds up the native operators by 6.5×, and the overall workload by 1.15×. In our evaluation results, we speedup some models by over 2×. Selecting the optimal setting is not straightforward, especially for at-scale CPU platforms that serve a large, diverse DL use cases in production datacenter fleets[86]. It thus motivates in-depth studies for better performance.

This chapter provides three major contributions.

- We provide a detailed analysis of fundamental performance implications of key framework



```
MatMul(x, w){
  for block in x
    threadpool.schedule(
      MATMUL, block, w)
}
```

```
MatMul(x, w){
  MATMUL(x, w)
}
```

Figure 3.2: An overview of the framework design features studied in this chapter.

design features on CPU platforms, including scheduling mechanisms, operator designs, library back ends, and parallelism mechanisms. We find that the programmability tax ranges from 63% to 1.3%.

- Using insights from this analysis, we propose simple guidelines for tuning framework parameters to maximize parallelism and reduce framework overheads.

- We demonstrate the usability of our approach by integrating our methodology with TensorFlow, which we will open source. Our settings achieve the same average globally optimal performance, and outperform the suggested settings from Intel[20] and TensorFlow[75] performance guides by 1.29× and 1.34×, respectively, across a set of real-world DL models, including several from the MLPerf suite[151].

3.2 Framework Design Overview

Deep learning frameworks enable high-level expression of model layers and operations, portability across platforms, and integration with convenient debugging and profiling tools, which have lowered the programming effort for DL researchers and decreased prototyping time for new models. Abstraction provided by these frameworks hides some design choices, such as run-time scheduling and actual implementation of operators, which may not be noticeable by users but are important for performance. An efficient framework design should be able to exploit the parallelism exposed by deep learning workloads. In this section, we describe how deep learning framework design choices (Section 3.2.1) exploit parallelism opportunities exposed in deep learning workloads (Section 3.2.2), and overview our framework parameter tuning methodology (Section 3.2.3). We also discuss related work.

Our performance analysis focuses on CPUs, which are the most widely-used platforms serving ML workloads in datacenters[86,79]. Performance improvement directly translates into capacity efficiency improvement. As we will see, DL frameworks offer a large set of tradeoffs to exploit parallelism that lead to significant optimization opportunities on CPUs.

3.2.1 Design Features

Figure 3.2 presents the stack of DL frameworks and design features that we study. This work focuses on frameworks with opaque (manually implemented) operators, a design adopted by popular DL frameworks. Frameworks that do not use such operators do not share the same structure and features as in Figure 3.2, including Tensor Comprehensions[192], TVM[45], and Julia[30]. Such frameworks claim to have better modularity and to extend more easily to new operators and new platforms[26], but their designs differ in important ways that put them out of the scope of this chapter.
SCHEDULER: Here "scheduler" refers to the operator scheduler, not the process scheduler of the

OS. It takes a computational graph representing the DL workload and schedules its operators based on dependencies and hardware resources. Two common approaches are synchronous and asynchronous scheduling. Synchronous scheduling schedules one operator at a time. Asynchronous scheduling schedules all operators in ready state, such that the operators can execute in parallel if hardware resources are available. In the example of Figure 3.2, asynchronous scheduling is faster than synchronous scheduling, assuming unlimited hardware units. However as we will show in Section 3.4, given limited hardware resources, the optimal mechanism usually falls between the two extremes.

OPERATOR: Frameworks include both native operators and operators based on library kernels. The way operators make use of kernels can have a surprisingly large impact on performance. For example, Figure 3.2 shows two potential implementations of a MatMul operator based on library kernel MATMUL, to implement the MatMul operator at the framework level. The right one passes arguments as is to MATMUL. The left one splits matrix x into smaller blocks and passes each block to a thread in a thread pool. We will show in Section 3.5.2 that the latter performs better because it parallelizes data preparation before entering the MATMUL kernel.

LIBRARY: Libraries are the most basic components. providing fundamental building blocks such as mathematical kernels and thread pools. Mathematical libraries provide efficient parallel implementations of common kernels. We study three widely-used libraries: MKL, MKL-DNN and Eigen. A thread pool manages a number of threads that execute tasks upon request. Math libraries use thread pools, such as that provided by OpenMP, for parallelization. Besides the thread pools used by math libraries, DL frameworks use additional thread pools to parallelize computations outside of the math kernels. We study thread pool implementations in the C standard library, Eigen, and Folly.

BEYOND ONE SOCKET: Parallelism mechanisms need to be applied based on workloads. Common mechanisms include data and model parallelism.

3.2.2 Parallelism Opportunities

A DL workload can be expressed with a computational graph, where a node represents an operator, and an edge indicates the dataflow dependencies between operators[14]. DL workloads expose the parallelism within an operator (intra-operator), between operators (inter-operator), and among requests. Efficient framework designs should exploit such opportunities.

Parallelism within an Operator

Operators manipulate tensors, i.e. *n*-dimensional arrays. The parallelism within an operator (intra-operator parallelism) can be exploited with the following techniques.

SIMD: The use of single instruction multiple data (SIMD) architectures, e.g., Intel's AVX instructions[137], is implemented in mathematics libraries. The MKL, MKL-DNN, and Eigen libraries can use AVX2 and AVX512 instructions.

MULTI-THREADING: Multi-threading is implemented at the operator and library levels. For example, MKL uses OpenMP for multi-threading. At the operator level, a framework may have a separate thread pool for further parallelism.

DATA PARALLELISM: Data parallelism splits one batch of data into multiple smaller batches. Thus it can improve performance of large-batch workloads.

Parallelism between Operators

SCHEDULING: The parallelism across operators (inter-operator parallelism) can be exploit by asynchronous scheduling, to place independent operators on different hardware units.

MODEL PARALLELISM: Model parallelism is realized by scheduling different operators (or the same operator after splitting along the model size dimension) on different hardware sockets or nodes, such as distributing large embedding tables across hardware nodes[65].

MODEL PIPELINING: Model pipelining is a special case of model parallelism[34]. One hardware node receives data from a previous node and operates on it while the previous node is computing the next training step. It allows hardware nodes to work on different training iterations at the same time. In contrast to data parallelism, model parallelism lowers the memory requirement for large models. This chapter does not study model pipelining.

Parallelism among Requests

Parallelism among requests can be exploited by batching, to transform request-level parallelism to intra-op parallelism. For example, multiple image classification requests can be combined and executed in a single session, such that the number of requests is mapped to the batch size dimension.

3.2.3 Framework Parameter Tuning

Based on our analysis of design features and parallelism opportunities, we reduce the number of design features needing to be selected from five (scheduling mechanism, operator design, math library, thread pool library, parallelism mechanism) to one, the number of asynchronous scheduling thread pools. Other features, such as operator parallelism, follow from that choice. We propose simple guidelines for tuning framework features based on a model's inter-operator parallelism, as reflected in its computational graph. To demonstrate their usability, we integrate the guidelines with TensorFlow, and achieve 1.29× and 1.34× speedup over Intel[20] and TensorFlow[75] recommended settings, respectively. We also achieve the same average performance as with globally optimal settings and 95% of globally optimal performance in the worse case. We have developed a TensorFlow plugin that sets framework parameters automatically. We will open source the plugin. Details are in Section 3.8.

Previous work proposed to tune TensorFlow parameters automatically[83], which treats the tuning process as a black box and therefore does not explain how parameters affect performance. Our work differs in three ways.

- First, our tuning method is supported by strong analysis. A deep understanding of framework designs and the root causes of performance difference makes our tuning method intuitive and lightweight.

- Second, performance with our worst-case settings differs by less than 5% from the *global optimum* obtained by exhaustive search, while previous work[83] reports large performance degradation.

- Finally, the robustness of our guidelines is validated on a different set of real-world DL workloads using a state-of-the-art two-socket system. Our evaluation highlights the importance of framework parameter tuning for state-of-the-art translation and recommendation models that exhibit immense inter-operator parallelism.

3.3 Experimental Setup

We will open-source our scripts and workloads.

CPU PLATFORMS: We use three Intel Skylake CPU platforms, *small*, *large*, and *large.2*. *large.2* contains two sockets of *large* with a peak bi-directional bandwidth of 120 GB/s. *large* and *large.2* represent widely-used datacenter servers. *small* has fewer cores; we use it to eliminate threading overhead for certain studies. We use *large* and *small* for most of the analysis where the performance tuning guidelines are summarized, and *large.2* for the evaluation of the guidelines. *small* has 32 fused multiply-add (FMA) units per core, while *large* has 64 per core. Although all of the platforms support hyperthreading, each core has only one set of FMA units, which limits the benefits of hyperthreading if both hyperthreads need FMA units. The specifications are summarized in Table 3.1. *large* and *large.2* are Amazon Web Services `m5.metal` instances. Unless otherwise specified, we use the *large* platform.

	SKU	Cores	TFLOPS	Freq	LLC
small	i7-6700k	4	0.423*	4 GHz	8 MB
large	Platinum 8175M	24	1.64*	2.5 GHz	33 MB

* Estimated with GeekBench v4[125].

Table 3.1: Hardware platforms under study. An additional platform is *large*.2 that contains two sockets of *large* with a 120GB/s bi-directional UPI bandwidth at peak.

FRAMEWORKS: Because TensorFlow supports all features, we use TensorFlow v1.13 with the MKL-DNN back end, unless otherwise specified. Conducting the same experiments with PyTorch (Caffe2 module) shows similar trends, so in this chapter we focus on TensorFlow. We set thread affinity to prioritize binding one software thread with one physical core[20].

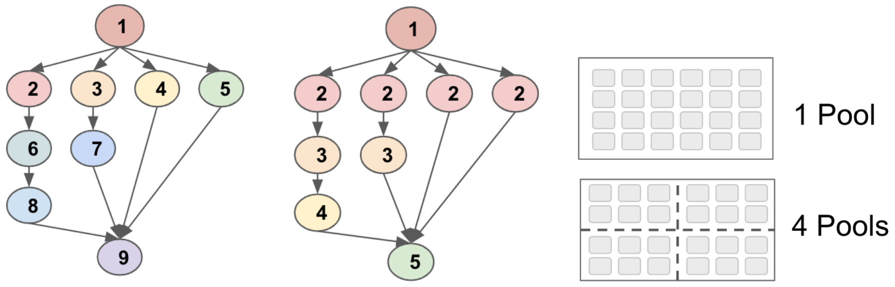
WORKLOADS: We use a set of production-size deep learning models, including three from MLPerf[151]

(ResNet-50[87], Transformer[193], neural collaborative filtering (NCF)[89]), as well as DenseNet[100], SqueezeNet[101], Inception[182], GoogLeNet[181], CaffeNet[106], ResNext[208], and Google's Wide & Deep Learning model[47]. To deeply understand the design features, we use micro-benchmarks such as matrix multiplication. We use a subset of the aforementioned models to focus our evaluation on the respective design features (Sections 3.4 and 3.5). We hold out all the non-vision models for Section 3.8 to evaluate the proposed method.

METHODOLOGY: Our profiling methodology enables thorough analysis. We produce time breakdowns (stack bars) for individual CPU cores using Linux's `perf record` and profiling one core at a time. Using floating-point performance counters, we measure performance as floating-point operations per second (FLOPS). We trace execution with performance counters by sampling instructions per cycle (IPC) every few milliseconds and ordering the samples by time stamps. The `perf stat` command with the `topdown` option produces the top-down breakdown. LLC misses, memory, and UPI traffic come from the corresponding performance counters.

TERMINOLOGY: Here are some terms used often later on.

- MKL Threads: the threads for MKL and MKL-DNN.
- Intra-Operator Threads: the threads for an operator at framework level. Abbreviated intra-op threads.
- Inter-Operator Pools: the independent thread pools in a framework, the size of each set by intra-op threads. Abbreviated inter-op pools, or pools.



(a) Synchronous **(b)** Asynchronous **(c)** Thread Pools

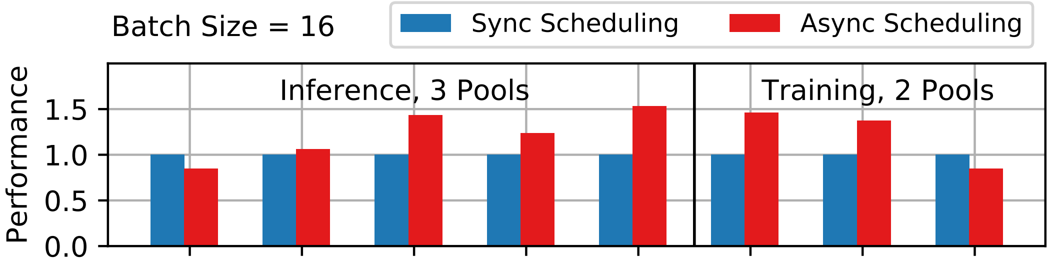
Figure 3.3: Examples of (a) synchronous scheduling, (b) asynchronous scheduling, and (c) using one and four thread pools, with the same total hardware resources.

3.4 Scheduling Mechanism

A deep learning model is expressed using a computational graph that represents the data flow between operators. At run time, operator scheduling exposes optimization opportunities, such as scheduling independent operators simultaneously. In this section we study the trade-offs of using such inter-operator parallelism. [*] We show that not all models benefit from asynchronous scheduling, and that the best setting depends on a model's inter-operator parallelism, quantified by the width of its computational graph.

Figure 3.3 shows examples of synchronous and asynchronous scheduling of an Inception module[182], and the use of one and four thread pools. Scheduling of an operator is to submit the job to a thread pool. Sometimes asynchronous scheduling, i.e., running multiple operators simultaneously, can improve performance. For the simple example shown, scheduling one operator at a time takes nine steps to finish (Figure 3.3a); scheduling four operators at a time reduces the steps to five (Figure 3.3b). One simple implementation is to create several thread pools of the same size to share the

[*] The experiments are conducted with the real-world workloads implemented in Caffe2, because the Inception architecture is important for this study, and the Caffe2 model zoo makes it more convenient to use inception.



	Caffe	Squeeze	Google	IncepV1	IncepV2	ResNext	FC-512*	FC-4k**
Max Width	1	2	4	4	4	2	2	2
Best Pools	1	1,2	2,3	2,3	2,3	1,2	1,2	1

* Similar size with the FC layers in YouTube and Facebook recommendation models.
** Similar size with the FC layers in Transformer.

Figure 3.4: (Bar Chart) The speedup of using asynchronous scheduling over synchronous. (Table) The maximum computational graph width and best numbers of thread pools. Workloads with more branches benefit from more pools.

computing hardware (Figure 3.3c), and to schedule independent operators asynchronously to the thread pools. This design is adopted by popular DL frameworks. In TensorFlow, the number of asynchronous thread pools is called the number of inter-operator threads. Caffe2 calls it the asynchronous thread pool size. In this chapter, we refer to it as the inter-op pools, as opposed to intra-op threads. We will show that synchronous scheduling is beneficial in both single-socket and multi-socket systems. The best performance is achieved by balancing intra- and inter-operator parallelism.

3.4.1 Datacenter Platform Performance

We show that the best number of thread pools is no more than the maximum number of parallel operators for a model. We use the *large* platform in Table 3.1.

Figure 3.4's bar chart shows the speedup of asynchronous scheduling on different production-size inference and training workloads. The baseline is synchronous scheduling, using one thread pool of size 24. Inference uses three thread pools, each with 8 threads; training uses two pools, each with 12 threads. Workloads benefit from asynchronous scheduling differently. Inception v1 and v2, GoogLeNet, ResNet, and FC-512 speed up more than others.

The performance difference is because of models' intrinsic inter-op parallelism, which is quantified by the width, or the number of branches, of their computational graphs. It measures the number of operators that can be scheduled in parallel. The table at the bottom of Figure 3.4 summarizes the maximum graph width and the best numbers of pools for varying batch sizes. We distinguish between inference and training workloads because the computational graphs of training workloads contain gradient and sum weight operators, which doubles the number of parallel operators. An intrinsic model limitation is that the best numbers of pools (for varying batch sizes) do not exceed the maximum graph width.

The best number of pools varies based on batch sizes. Figure 3.5 shows the performance of Inception v2 inference and ResNext training, with batch sizes of 1, 16 and 256. The use of inter-op pools benefits large-batch inference and small-batch training. This is because the extra parallelism from training operators is only beneficial for small batches. When batch size gets larger, gradient operators are much compute-intensive than sum weight operators. Thus allocating computing resources evenly for the two kinds of operators hurts performance. On the other hand, parallel operators in inference workloads stay balanced with large batch sizes, and large batch sizes make the operators more compute-intensive, thus asynchronous scheduling improves performance.

3.4.2 Inception v2 Case Study

To highlight parallelism opportunities at the intra- and inter-op levels, we use Inception v2 as an example. Its model architecture contains operator branches that can execute in parallel. In the baseline implementations that either schedule each branch naively to one CPU core or schedule one operator to all CPU cores, workload imbalance and synchronization overhead significantly reduce performance. We show that synchronization overhead can be mitigated by choosing the number/size of thread pools to better balance intra- and inter-operator parallelism. We use the *small* platform in

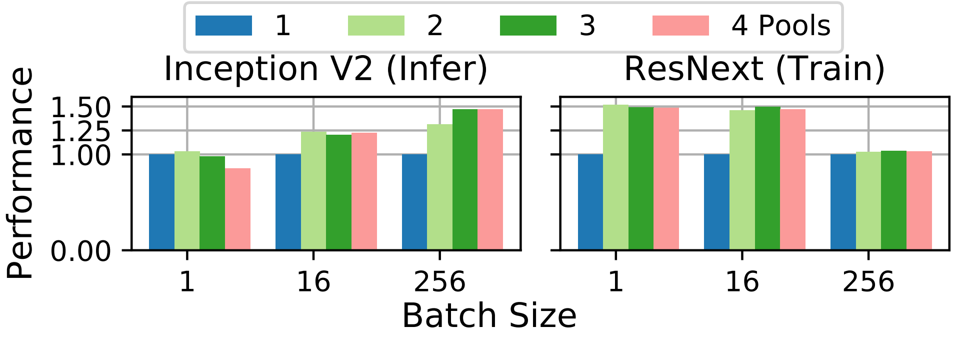


Figure 3.5: Asynchronous scheduling benefits large-batch inference and small-batch training.

Table 3.1, as it enables an exhaustive study of possible cases.

INCEPTION V2 ARCHITECTURE: To simplify the explanation of later results, we first summarize the Inception v2 architecture[182] in Figure 3.6. Figure 3.6a shows the top level architecture, color coded as areas 1 and 2. Area 1 exhibits both intra- and inter-op parallelism, while area 2 only has intra-op parallelism. Area 1 contains two inception modules. Module 4 has four branches (Figure 3.6b) and module 3 has three (Figure 3.6c). Area 1 contains eight instances of module 4 and two of module 3. Operators in area 2 are sequential, including convolution, MatMul, and mathematical operators. The convolution operators are converted to MatMul using `im2col()`, so intensive computation is mostly MatMul.

PERFORMANCE SCALING WITH POOLS AND THREADS: Figure 3.7 shows the relative performance of Inception v2 with a batch size of 16, sweeping inter-op pools and MKL threads per pool. The total number of threads on the system is the product of the two. Hyperthreads are used when more than four threads are created. Exceeding eight threads is labeled over-threading because there are more software threads than hardware threads. (Scaling is similar with batch sizes from 1 to 128.)

Hyperthreading does not improve performance significantly, such as [4,1] vs [4,2], and [1,4] vs [2,4] ([Threads, Pools]). The compute-intensive operators, Convs and MatMuls, are bottlenecked

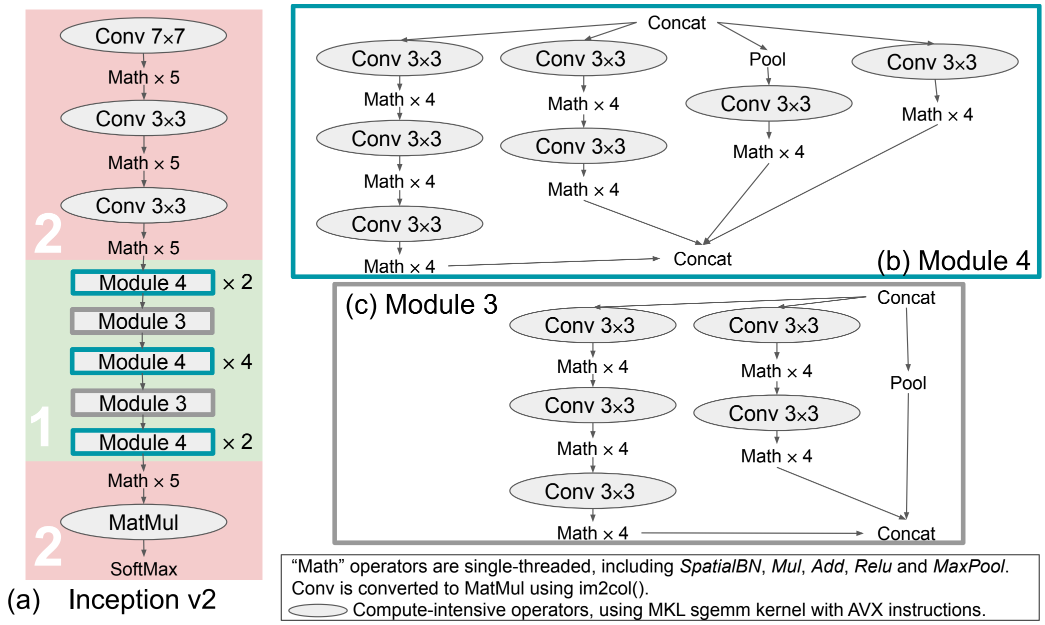


Figure 3.6: (a) Inception v2 architecture contains modules with (b) four and (c) three independent branches. Area 1 exhibits inter- and intra-op parallelism and area 2 only intra-op.

by the fused multiply-accumulate (FMA) units, which are shared between hyperthreads on the same core.

As expected, over-threading, i.e., using more software threads than hardware threads, hurts the performance, because threading overhead increases with more software threads, and computing resources are saturated. As a result, simply setting all framework knobs to the maximum does not yield the best performance.

For Inception v2, performance is best with two pools and two threads per pool. Using four total threads in other ways, such as four pools with one thread each, or one pool with four threads, cannot achieve such performance. It is because this case balances the intra- and inter-op parallelism provided by Inception v2 and reduces its synchronization overhead. Our profiling methodology reveals and visualizes the underlying causes in Figures 3.8 and 3.9.

RUN-TIME BREAKDOWN AND EXECUTION TRACES: To explain the best performed case

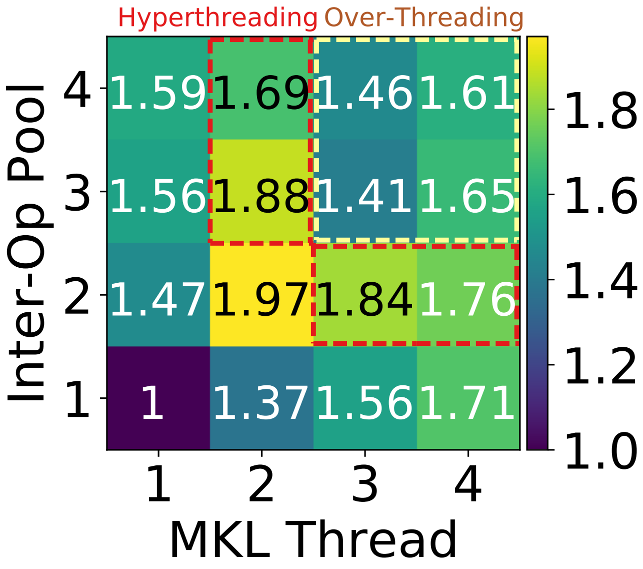


Figure 3.7: Performance of Inception v2 with different numbers of inter-op pools and MKL threads per pool. Best configuration balances intra- and inter-op parallelism.

[2,2], we select four cases, a baseline that uses only one thread, and three cases that each use four threads in total. One software thread is bound to one CPU core. Figure 3.8 shows the aggregate time breakdown, and Figure 3.9 shows the corresponding execution traces. In Figure 3.9, one iteration of Inception v2 inference is marked with red bars, and operators in execution are labeled with the corresponding color in Figure 3.6, where area 1 exhibits intra- and inter-op parallelism, and area 2 has only intra-op parallelism. The fraction of time each core spent executing (rather than synchronizing) is indicated to the right of each trace. It matches the breakdowns in Figure 3.8.

The first case, using four pools of size one, incurs high synchronization overhead in Figure 3.8 and Figure 3.9a, primarily because the operators in area 2, with only intra-op parallelism, are assigned only one core (thread), and other cores are waiting to synchronize. The trace in area 1 is slightly better, with every core executing one branch of each inception module.

The second case, using one pool with four threads, does not perform well either, because the

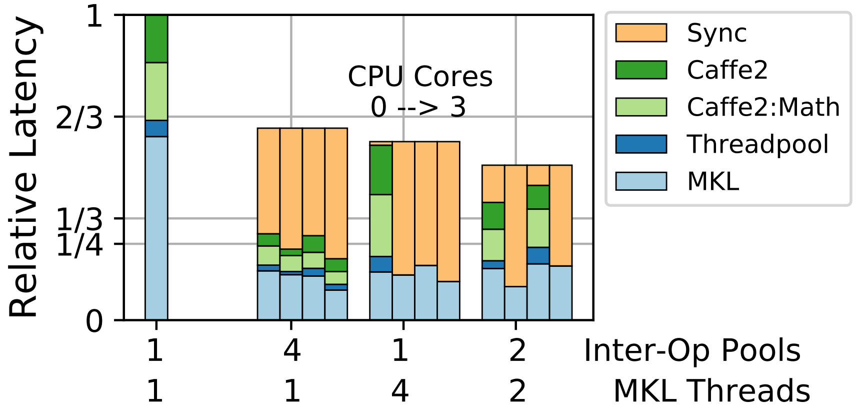
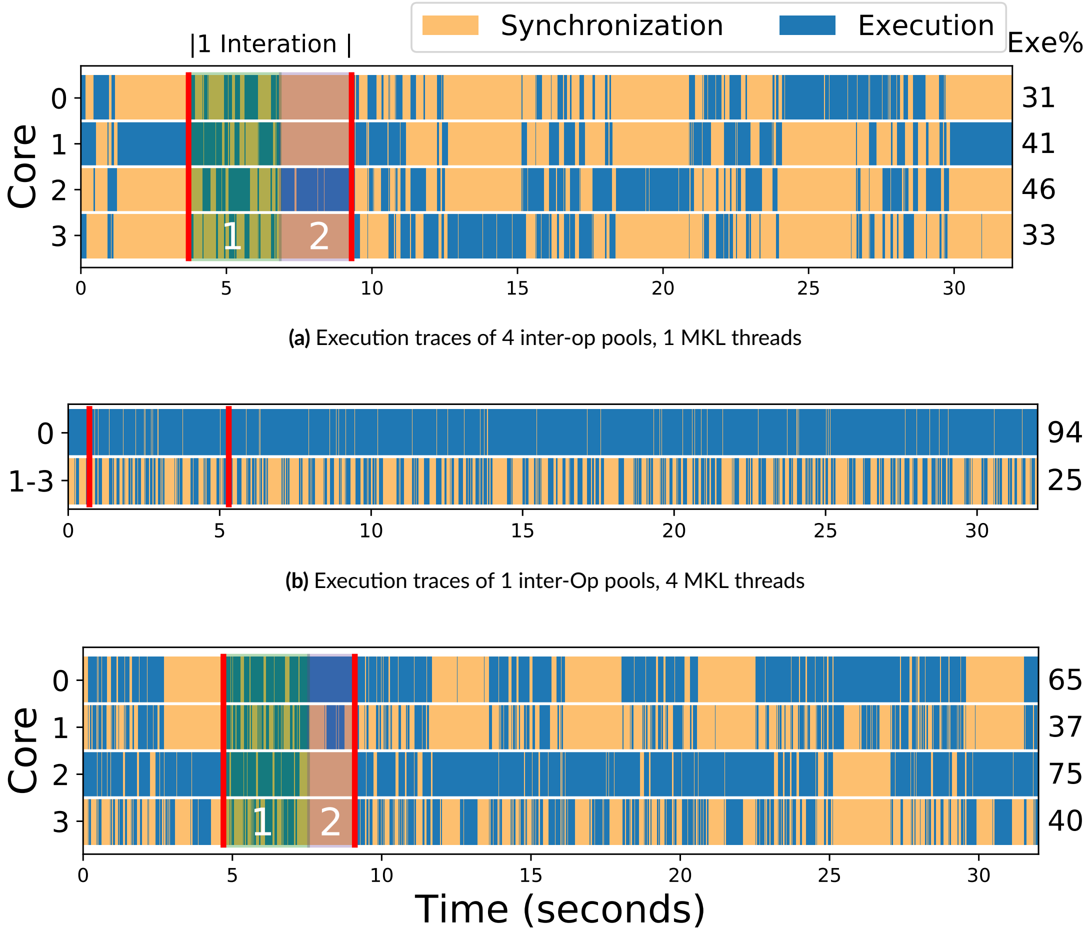


Figure 3.8: Execution time breakdown of four cases.

Caffe2 operators are single-threaded, and other cores are stalled by core 0, as shown by the long synchronization time in the rightmost three bars of Figure 3.8 and the traces of cores 1–3 in Figure 3.9b.

The third case, using two pools, each with two threads, is a better balance. It reduces synchronization time in both area 1 and area 2 compared to the first case. Area 1 is improved because the number and size of convolutions in each inception branch is not even, as shown by Figure 3.6b, and allocating one core for each branch (the first case) makes the core running the smallest branch wait for a long time. Area 2 is improved because its operators are sequential and compute-intensive, and having more cores improves performance.

OPTIMIZATION OPPORTUNITY: Operators in a complex model come in different sizes and have different dependencies. Fixing each thread pool size usually incurs synchronization overhead because of work imbalance. Thus there is an opportunity to implement a global thread pool, allowing the scheduler to determine dynamically how many threads to schedule for each operator. For example, in the traces of Figure 3.6, providing area 1 with two pools of two threads each and area 2 with one pool of four threads can lead to higher performance.



(a) Execution traces of 4 inter-op pools, 1 MKL threads

(b) Execution traces of 1 inter-Op pools, 4 MKL threads

(c) Execution traces of 2 inter-op pools, 2 MKL threads

Figure 3.9: Execution traces of three cases in Figure 3.8. Color-coded areas 1 and 2 correspond to the operators in Figure 3.6.

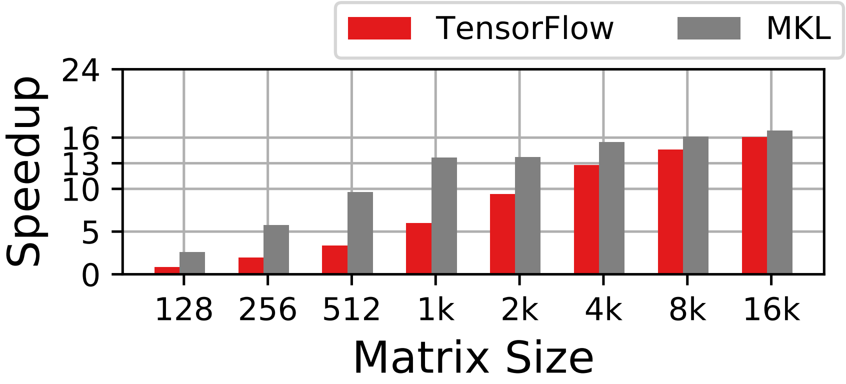


Figure 3.10: Speedup from using 24 MKL threads instead of one. TensorFlow exhibits lower speedups than MKL.

3.5 Operator Design

Operators are building blocks of DL frameworks, providing basic semantics through high-level language (like Python) APIs to ease the development process for framework users. As shown in Figure 3.1, a workload built with a DL framework involves library kernels and framework native computation. In this section, we show that efficient operator design can speed up framework native computation, and yields up to 4.2× performance improvement for real-world models.

IMPLEMENTATIONS OF FRAMEWORK NATIVE OPERATORS: We first describe framework native operators, the operators that do not use library kernels. Native operators handle control flow as well as tensor reshaping, broadcasting, and preprocessing. Some, like those for control flow or input image preprocessing, are necessary. Others can fairly be described as a framework programmability tax. The overhead stems from preparing inputs for library kernels, or computing how to parallelize a given workload in the main thread. One example of the latter kind is Eigen::ParallelFor, used by TensorFlow.

IMPLEMENTATIONS OF COMPUTE-INTENSIVE OPERATORS: A framework operator must sometimes do more than simply pass arguments to library kernels. Data preparation is often required, for example. Taking matrix multiplication (MatMul) as an example, we list two implementations below. Note that those implementations do not cover all possible cases, but we do find hints of those implementations adopted in popular frameworks, and we will talk about it in the experimental results. In the context of deep learning, x is an input matrix of size [batch size × number of activations] and w is a weight matrix of size [activations in current layer × activations in next layer]. We assume MatMul is the interface of a framework operator, and MATMUL is the corresponding library kernel, e.g., in MKL.

```
Sec 3.5.1)  MatMul1(x, w):

                data_prep(x, w)

                return MATMUL(x, w)

Sec 3.5.2)  MatMul2(x, w):

                // Reshape x, w into bx and bw

                for bx, bw in x, w:

                    threadpool.run(MatMul1,

                    bx, bw)

                return threadpool.join_results()

Sec 3.5.3)  MatMul3(x, w):

                data_prep(x, w)

                return MATMUL(wT, xT)
```

MatMul1 conducts data preparation and passes the matrices to the library kernel. MatMul2 uses an additional thread pool that we call the intra-operator thread pool, to distinguish it from the MKL thread pool. The operator splits the matrices into smaller ones, and passes those small matrices to the intra-op thread pool. The intra-op thread pool then executes multiple copies of MatMul1

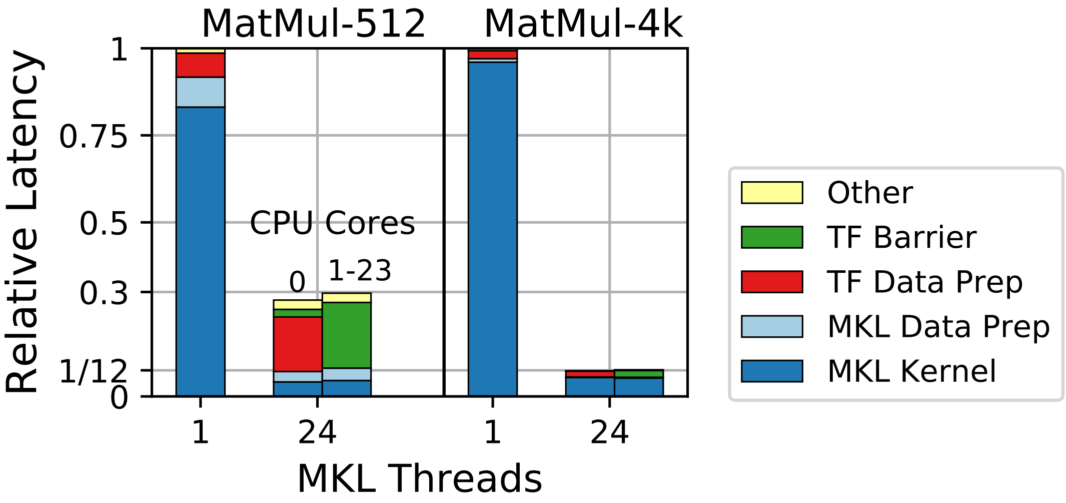


Figure 3.11: Run-time breakdown for all CPU cores. Data preparation overhead causes the poor scalability in Figure 3.10.

in parallel. This way data_prep() of the whole matrix can be parallelized. MatMul3 transposes the matrices, which can improve the performance when the two dimensions of a matrix is largely different, e.g., batch size >> activations or vise versa, assuming the library kernel focuses on parallelizing one dimension over another. Note the transposing overhead can be minimized with optimizations, thus we do not include the transposing computation in MatMul3. Before and after the library call, the data formatting and results gathering work is part of the programmability tax. We will study the three implementations in the following subsections.

3.5.1 MKL Threads

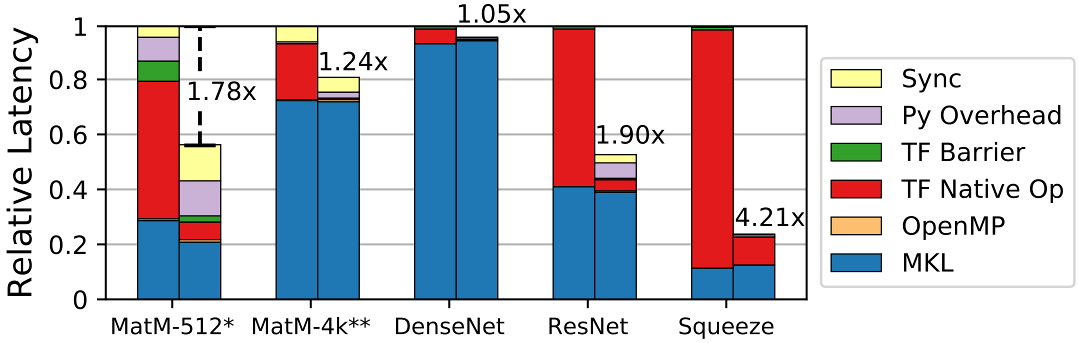
By analyzing the overhead and scalability of the first operator implementation, MatMul1, we show that both the TensorFlow (TF) operator and the MKL kernel suffer from data preparation overhead, which prevents them from scaling linearly with the number of CPU cores. The results here can also apply when convolution operators are converted to MatMuls using `im2col()`. We use the *large* platform in Table 3.1.

PERFORMANCE SCALING: Both TF and MKL have scaling issues, and TF is slightly worse. Figure 3.10 shows the speedup of using 24 MKL threads over using one, for both TF operators and MKL kernels. The matrices are squared and represented by one dimension. The total number of floating-point operations is the cube of that number. Figure 3.10 shows that the speedup of TF is always lower than that of MKL, especially for small matrices. TF speedup is comparable to MKL when matrices are larger than 4k. The maximum speedup achievable is about 16×, which is lower than the number of cores, 24.

CAUSES OF THE POOR SCALABILITY: Our profiling methodology reveals that data preparation overhead causes suboptimal performance scaling. We pick two variants of MatMul, MatMul-512 and MatMul-4k, that operate on medium- and large-size matrices, respectively. MatMul-512 represents the fully-connected (FC) layers from YouTube[51] and Facebook recommendation[145,79] models, while MatMul-4k represents the FC layers in Transformer[193]. Figure 3.11 shows the run-time breakdown of all CPU cores running the MatMuls, using 1 and 24 MKL threads. With multiple threads, the thread tasked with lengthy TF data preparation is the main thread, labeled CPU Core 0. The latency of each MatMul workload is normalized to that of using one MKL thread.

The TF parts of Figure 3.11 show that TF's scaling issue is caused by framework overhead, due mainly to TF data preparation for MKL kernels. Using one MKL thread, MatMul-512 spends over 10% of its time in TF data preparation; using 24 MKL threads, the overhead exceeds 72%. Overhead is much lower for MatMul-4k: less than 3% in both cases. Without TF overhead, speedup can clearly be much greater. The MKL parts of Figure 3.11 show that MKL data preparation causes the scaling issue for MKL kernels. MKL kernel execution time is roughly 1/24 of the original run-time for MatMul-512. Speedup drops with time spent in MKL data preparation.

THE ROLE OF FRAMEWORK DESIGN: The Amdahl's law bottleneck of DL frameworks is non-negligible. The overhead a MatMul with size $n \times n \times n$ scales linearly with n ($O(n)$), while the



* Similar size with the FC layers in YouTube and Facebook recommendation models.
** Similar size with the FC layers in Transformer.

Figure 3.12: Run-time breakdown of TensorFlow workloads with 1 (left bar) and 24 (right bar) intra-op threads. Both cases use 24 MKL threads on a 24-core CPU.

number of floating-point operations scales cubically ($O(n^3)$). Thus the speedup of large MatMuls (e.g., 4k) is closer to ideal speedup. Realistically, however, the most commonly-used FC layers are not always large enough. Actually smaller ones are common in commercial workloads including YouTube's[51] recommendation model (of size 256 to 1k) and Facebook's[145,79] (of size 64 to 512). Especially when hardware platforms are upgraded to higher floating-point computation capability, we need even larger matrices to amortize the overhead. Thus it is key to focus optimization efforts on mitigating framework overhead.

3.5.2 Intra-Operator Threads

After decades of optimizing the GEMM kernel, the performance bottleneck has shifted to the overhead of using such kernels, e.g., the data preparation overhead in Figure 3.11. A natural approach to reducing the overhead in framework design is to parallelize the framework native computation, with an intra-operator thread pool implemented at the framework level, as MatMul2 does. We show that when library kernels are using FMA units, intra-op threads improve performance by utilizing other computational units on the same physical core, thereby benefiting from Intel's hyperthreading

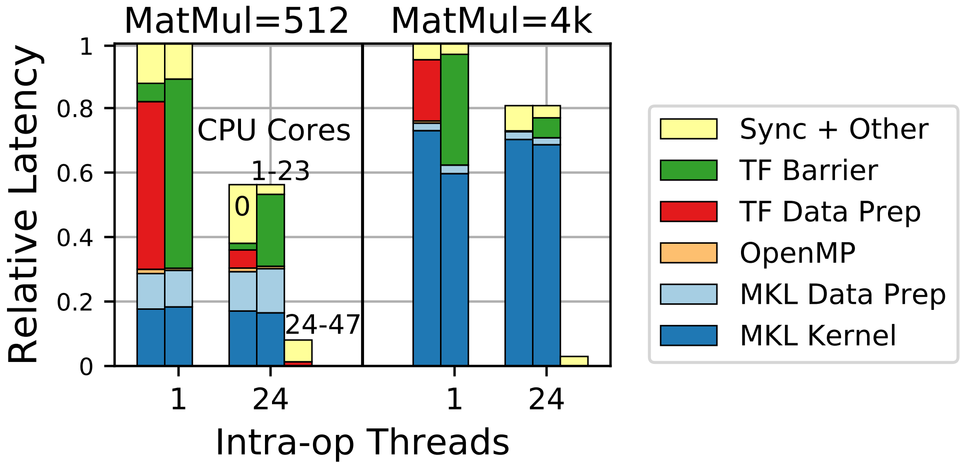


Figure 3.13: Run-time breakdown of all CPU cores. Intra-op threads parallelize the overhead in cores 24-47.

technology. We use the *large* platform from Table 3.1.

PERFORMANCE IMPROVEMENT: We first show how much performance improvement intra-op threads can yield and where it comes from. Figure 3.12 summarizes speedup and time breakdown when using 1 (left bar) and 24 (right bar) intra-op threads. Both cases use 24 MKL threads. MatMul-512 and MatMul-4k are the same operators as in previous sections. Using 24 intra-op threads reduces the execution time of TF native operators, while that of other parts stays similar. The speedup ranges from 1.05× (DenseNet) to 4.21× (SqueezeNet).

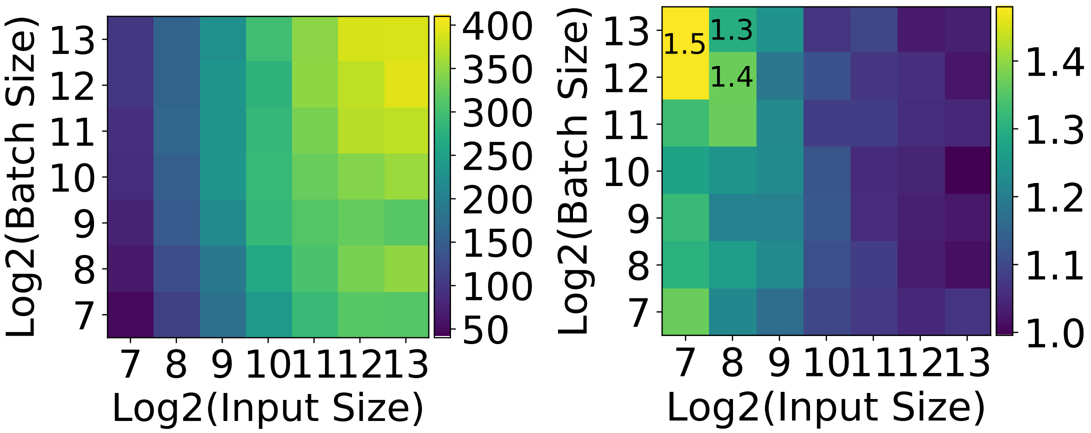
Workloads bottlenecked by TF native operators benefit more from intra-op threads. Such workloads, including MatMul-512 and SqueezeNet, have small to medium MatMul or convolution operations, because TF native operators are likely to consume larger fractions of computation time. For example, SqueezeNet has a small percentage of MKL computation since it is designed to have fewer parameters than AlexNet by using many small (1x1) convolution kernels[101].

PROGRAMMABILITY TAX: Figure 3.12 also quantifies the framework programmability tax. We estimate the tax using the non-MKL fractions, since they are not compute-intensive and can be largely optimized if written with high-performance language/code as MKL kernels. After optimizing with intra-op threads, the programmability tax ranges from 1.3% (DenseNet) to 63% (MatMul-512). SqueezeNet (47%) is higher than ResNet-50 (26%). MatMul-4k (11%) is slightly smaller. This is the price we are paying for using frameworks.

FULL-SYSTEM PROFILING: Our profiling methodology visualizes the execution of every core on the CPU platform to expose the reasons for performance improvement. Figure 3.13 shows the time breakdown for all 48 hyperthreads on the *large* platform. We focus on the two MatMuls, since they are the simplest workloads. Because cores 24-47 are not active when using one intra-op thread, the third bar is omitted for that case. Core 0 of each case is the same as in Figure 3.12.

Figure 3.13 shows that with 24 intra-op threads, TF data preparation is distributed to cores 24 through 47. (The bottom of the third bar for MatMul-512 with 24 intra-op threads shows a tiny TF data preparation cost.) That shortens TF data preparation time in core 0, so that the TF barrier time of cores 1 to 23 is shortened. With only one intra-op thread, cores 1 to 23 spend about 60% (MatMul-512) and 40% (MatMul-4k) of time waiting in barrier, which is not optimal for using computing resources.

HYPERTHREADING: Using intra-op threads takes advantage of Intel's hyperthreading technology by colocating an intra-op thread and an MKL thread on the same physical core. Since they need different hardware resources, they can execute in parallel without contention. The critical path is the MKL thread. The intra-op thread adds no execution time to the overall workload. For example, in Figure 3.13, logical cores 0 and 24 are on the same physical core. Core 0 executes mostly MKL floating-point operations with FMA units, which core 24 does not need. Even without hyperthreading, the implementation of intra-op threads improves performance through parallelization. In that case, the physical core's critical path combines the intra-op thread with the MKL thread.



(a) TF-MKL GFLOPS

(b) TF-Intel Speedup

Figure 3.14: (a) The FLOPS of TensorFlow compiled with MKL. FLOPS increases faster with large input size. (b) Intel version of TensorFlow with MKL. Operators with large batch size gets higher speedup.

3.5.3 Transposing Matrices

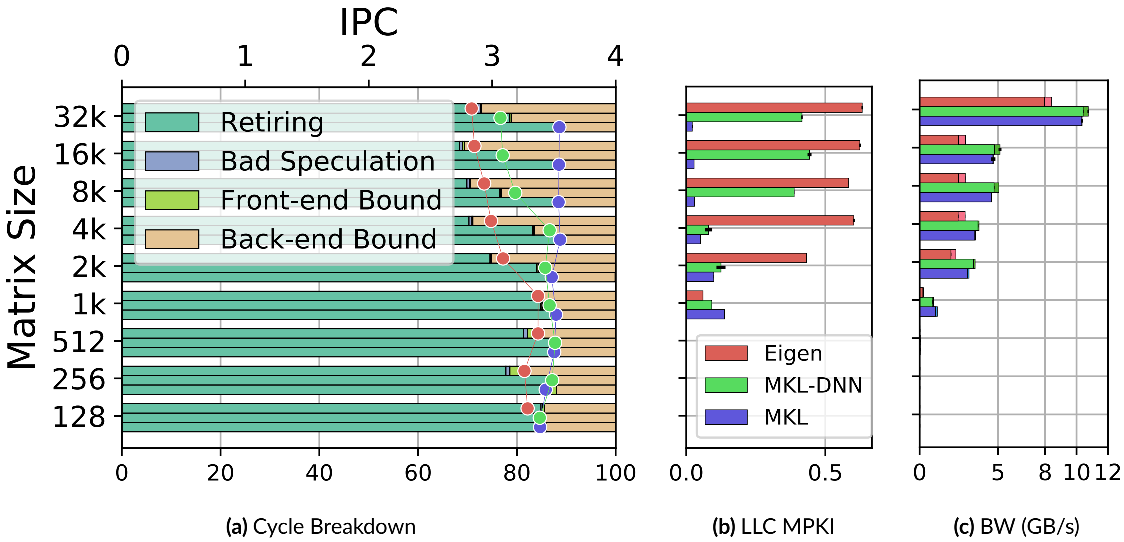
This section shows that if the performance of an operator implementation is more sensitive to one matrix dimension over another, sometimes transposing the matrices leads to better performance. To simplify the problem, we assume two dimensions of a MatMul operator, input size and batch size. We studied MatMuls with input size = batch size in the previous section. This section uses the *small* platform in Table 3.1 to avoid the scalability overhead and focusing on the matrix dimensions.

We study the performance of TF-MKL and TF-Intel. TF-MKL is TensorFlow compiled with MKL. TF-Intel is applied a set of Intel-specific optimizations, such as replacing TensorFlow operations with Intel optimized ones, eliminating unnecessary data layout conversions, and operation fusion[6]. It also mentions matrix transposing may improve performance, but details are not disclosed in the vendor proprietary library. The experimental results here show that for MatMuls with certain sizes, TF-Intel likely adopts the third MatMul implementation discussed at the beginning of Section 3.5.

Figure 3.14 studies the performance sensitivity of TF-MKL and TF-Intel on input sizes and batch sizes. Figure 3.14a shows the FLOPS of TF-MKL MatMuls with different batch and input sizes. The FLOPS increases with larger batch and input sizes, and the highest FLOPS is the MatMul at the upper right corner. It is observable that the FLOPS increases faster with an increasing input size, because color shifting is more obvious from left to right, compared to from bottom to the top. The fact that TF-MKL performance is more sensitive to input sizes exposes optimization opportunities for workloads with large batch size and small input size.

To compare TF-MKL with TF-Intel, each grid of Figure 3.14b shows the speedup of TF-Intel over TF-MKL on the same MatMul operator. The speedup is larger than 1 for MatMul input sizes less than 1k. The speedup increases with larger batch size for workloads with small input sizes, which is likely due to TF-Intel's transposing the matrices of such workloads.

The highest speedup (1.5$\times$) is achieved by an MatMul with very small input size (128) and very large batch size ($\geq$4k). The input size (activations) of Facebook and YouTube's recommendation models ranges from 64 to 1k, with an exception layer of 5k[51,145,79]. Using TF-Intel, the training performance of smaller layers can be significantly improved.



(a) Cycle Breakdown

(b) LLC MPKI

(c) BW (GB/s)

Figure 3.15: Comparison of three matrix multiplication libraries. (a) Cycle breakdown (bottom axis) and IPC (top axis) for matrices of various sizes. MKL has the highest retiring ratio and IPC, because it is the least back-end bound. (b) MKL has the lowest LLC misses per kilo instructions (MPKI). (c) The memory bandwidth consumption. MKL's prefetching is the most effective: almost all memory traffic is prefetching.

3.6 Library Choice

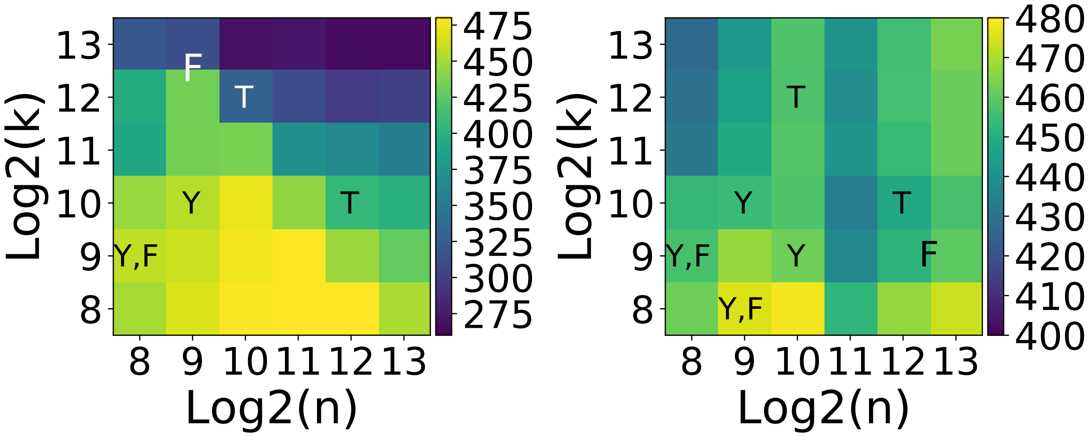
Thanks to the mathematics and thread pool libraries, deep learning framework developers do not have to implement every basic functions from scratch. However there are many choices for the same functionality, and the choice of libraries can have a large impact on performance. Thus to study frameworks, we need to understand the performance differences among existing libraries the reasons why they differ. In this section we study libraries for machine learning and thread pools. We show that optimization can improve a GEMM kernel's performance by up to 25%, owing to more efficient data prefetching. We also compare three popular thread pool libraries. We find that robust thread pools such as Eigen and Folly are better able to keep production-critical workloads running with little variation, thus investing in sophisticated implementations is worthwhile for service providers.

3.6.1 Machine Learning Library

SETUP: We compare MKL, MKL-DNN, and Eigen with GEMM (general matrix multiplication) microbenchmarks on the *small* platform (Table 3.1) to expose architectural bottlenecks, while minimizing the scaling overhead on large systems. The microbenchmarks perform general matrix multiplication (GEMM) with matrices of different sizes. GEMM is the most often used machine learning kernel. It supports all vision models, all deep recommendation models, and many sequence to sequence models. It consumes 42% of machine learning cycles in Facebook datacenters[150].

TOP-LEVEL ANALYSIS: We conduct top-down analysis[213] for single-threaded GEMM kernels with a variety of matrix sizes. Figure 3.15a shows the cycle breakdown (stacked bars) and IPC (dots). The three bars shown for each matrix size are for Eigen, MKL-DNN, and MKL, from top to bottom. Cycles are broken down into retiring and stalling due to bad-speculation, front-end, and back-end bottlenecks. By definition, instructions per cycle (IPC) is proportional to retiring cycles. Floating-point operations per second, which is not shown here, is consistent with IPC and the retiring ratio because the three libraries have comparable numbers of dynamic instructions. For GEMM, MKL performs the best, followed by MKL-DNN. The performance difference is larger for large matrices (about 0.7 IPC), and it is minimal for matrices with sizes less than 2k. (That is why we use the *small* platform. Large platforms are not easily stressed, even with matrices of size 32k.) With matrices larger than 4k, about 25% of cycles are back-end bound for Eigen and MKL-DNN.

CAUSES: The back-end bottleneck is caused by last-level-cache (LLC) misses, shown as LLC misses per thousand instructions (MPKI) in Figure 3.15b. Eigen and MKL-DNN have much higher LLC MPKI than MKL. Owing to the latency of moving data from main memory, the higher LLC miss rate reduces workload performance. The LLC miss rate difference is caused by the aggressiveness and effectiveness of data prefetching, shown by the memory traffic in Figure 3.15c, where the right ends of the bars show memory traffic incurred by LLC misses. MKL's memory traffic is close to that



(a) batch size (m) = 64

(b) batch size (m) = 1024

Figure 3.16: FLOPS of MKL matrix multiplication by varying three dimensions of a matrix multiplication, m, k and n. When m is mapped to the batch size dimension, (a) m=64, represents typical inference workloads. (b) m=1024, represents typical training workloads. T: Transformer; Y: YouTube recommendation model; F: Facebook recommendation model.

of MKL-DNN, and its much lower LLC miss rate shows that MKL's software prefetching is more *effective*. Eigen's memory bandwidth is the lowest, and its LLC miss rate is the highest, indicating that Eigen does not prefetch as aggressively as the others. Still, it is impressive that a portable lightweight library like Eigen can perform about as well as MKL for medium and small matrices.

OTHER KERNELS: We compare the GEMM kernels to demonstrate how and why kernel performance can vary. For other kernels, MKL-DNN likely outperform other libraries, because it is a library targeting deep learning with DL-specific optimizations. For example, MKL-DNN uses the GEMM kernel from MKL for its better performance. The optimizations include operation fusion, batch normalization folding, and filter caching for convolutions[31]. Besides, MKL-DNN includes MKL as a submodule to use its GEMM kernels, which outperform other libraries as shown in Figure ??. As a result, MKL-DNN reportedly outperforms other frameworks[31], so more frameworks have started to support MKL-DNN.

MKL PERFORMANCE SENSITIVITY: A matrix multiplication kernel takes two matrices of sizes

$m \times k$ and $k \times n$ as inputs, and outputs a matrix of size $m \times n$. Figure 3.15 studies the performance of square matrices with $m = k = n$. Here we show the FLOPS of MKL running non-square matrices in Figure 3.16, by varying the three dimension sizes. Intuitively, we assume m is the batch size dimension in deep learning. Figure 3.16a shows batch size $=$ 64, representing typical inference workloads[150,79]; Figure 3.16b shows batch size $=$ 1024, representing typical training workloads[193,145].

The total floating point operations of a matrix multiplication is $m \times k \times n$, and the computation is embarrasingly parallel. Intuitively speaking, increasing size of any dimension should increase the FLOPS. However Figure 3.16 shows that the performance of MKL does not increase monotonically with matrix sizes. It is likely caused by different prefetching or matrix blocking mechanisms dynamically chosen based on matrix size. Especially the scaling in Figure 3.16b looks like the heatmap is split into four blocks from the very middle point, indicating possible four choices of optimization. We also annotate the matrix sizes of Transformer[193], YouTube's[51] and Facebook's[145,79] recommendation models. Those realistic matrix sizes are not in the best, nor in the worst performed spots of MKL.

3.6.2 Thread Pool Library

We compare three thread pools, one simple implementation using std::thread,[†] and the thread pools in Eigen[63] and Folly[13]. The Eigen thread pool is used in TensorFlow[63]. Folly is a C++ library open-sourced by Facebook[13]. Our microbenchmark creates a thread pool and starts 10*k* tasks that increment the value of a globally shared variable. We benchmark in two scenarios: (1) setting the number of threads to be the same as the number of physical cores, and (2) using many more threads than the number of cores. We use the *small* platform from Table 3.1.

Figure 3.17 compares the overall latency of running 10k micro tasks. In both cases, Folly outperforms Eigen and Eigen outperforms std::thread. When thread pool size is four, Folly outperforms Eigen by 16%, and Eigen outperforms std::thread by 24%. When thread pool size is 64, greatly ex-

[†] Std::thread is compiled to pthread on POSIX platforms.

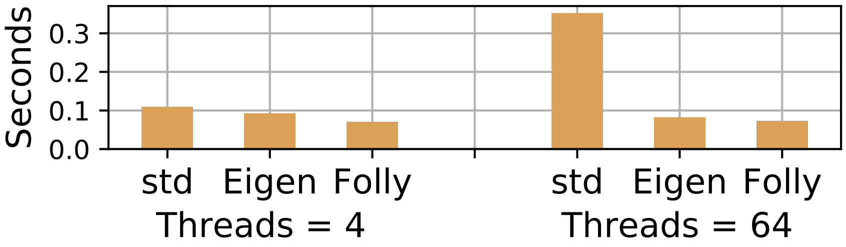


Figure 3.17: Thread pool overhead, measured as time to run 10k micro tasks. Folly outperforms std::thread and Eigen.

ceeding the number of CPU cores, Folly and Eigen perform consistently well as with four threads, and oversubscribing the system does not drastically increase the overhead. But the overhead of std::thread increases by over 3×, with every CPU core spending about 60% of its time in synchronization.

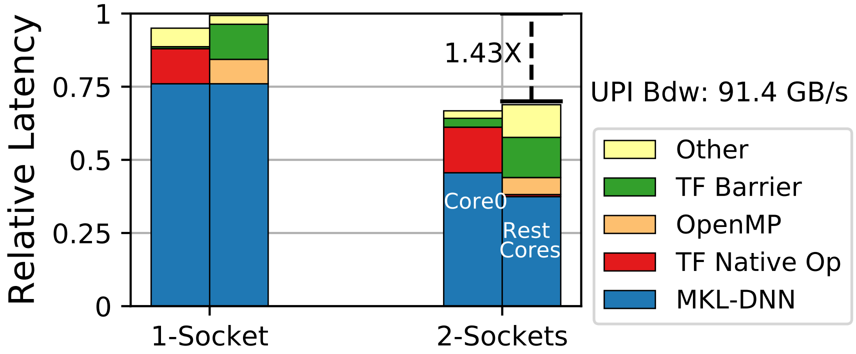


Figure 3.18: A two-socket platform speeds up ResNet-50 by 1.43×. The bottleneck is the UPI bandwidth, increasing TF data preparation time as part of the TF native operator time.

3.7 Beyond One Socket

In previous sections we have explored the framework designs on one-socket CPUs. In this section, we study how those design features can be applied to scale out the workloads beyond one socket. Unsurprisingly, the bottleneck of a two-socket system is the UPI bandwidth between sockets. We study scaling one operator to two sockets and scheduling multiple operators to different sockets, as scaling-out versions of intra- and inter-op parallelism studies. The experiments are conducted on the *large* and *large.2* platforms from Table 3.1.

3.7.1 Data Parallelism

We study data parallelism by setting the numbers of intra-op and MKL threads to the total number of physical cores and the number of inter-op threads to one.

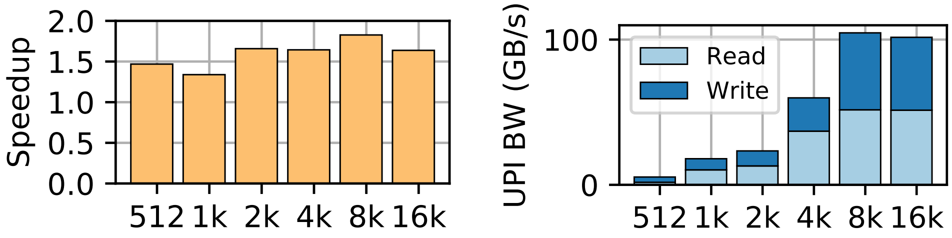
RESNET PERFORMANCE: Figure 3.18 shows the execution time breakdown of ResNet-50 running on one- and two-socket platforms. The latter speeds up ResNet by 1.43×, less than the two-fold hardware increase. The bottleneck is that UPI traffic peaks at 91.4GB/s, compared to the theoretical maximum of 120GB/s. UPI saturation increases the latency of TF native operators on a two-socket platform, which now includes both data preparation and transfer time between sockets. MATMUL PERFORMANCE: To test the limit of the *large.2* platform, we conduct microbenchmarking using TensorFlow MatMul operators. Similarly, the UPI bandwidth is the bottleneck for large MatMuls. Figure 3.19 shows two-socket speedup and corresponding UPI bandwidth consumption. The speedup and UPI throughput increase with larger MatMul sizes, and peak for MatMul-8k. For MatMul-16k, speedup decreases and bandwidth saturates, indicating empirically the maximum UPI bandwidth is around 100GB/s for such workloads.

The speedup of a workload is determined by its intrinsic parallelism and UPI bandwidth saturation. Figure 3.20 shows the time breakdown of MatMuls running on one- and two-socket platforms. For medium MatMul sizes like 512, the poor scalability is caused by the limited parallelism of the workload, which cannot hide data preparation overhead. For larger MatMuls, 4k and 8k, the data preparation time of both TF and MKL increases on the two-socket platform because of UPI saturation. MatMul-8k has the best balance of intrinsic parallelism and UPI throughput. It leads to the highest speedup (1.8×, i.e., 44% less execution time than with just one socket), which is close to perfect scaling.

3.7.2 Model Parallelism

We study model parallelism of a two-socket platform by using two inter-op pools, one per CPU socket. Model parallelism improves performance significantly when the parallel operators are on critical paths and have similar sizes, as with multiple embedding operators in neural collaborative filtering (NCF). Performance and model parallelism mechanisms will be discussed in the next section.

Model parallelism does not always improve performance. One example is the inter-op parallelism from training workloads, as in Section 3.4. Assigning gradient and weight sum operators one socket each causes workload imbalance between two sockets when batch size is large. Two-socket platforms are not beneficial when the intra-op thread pool is not implemented at the framework level, since



(a) Two-Socket Speedup

(b) UPI Traffic

Figure 3.19: (a) Speedup of a two-socket platform over one socket. (b) Measured peak UPI bandwidth consumption on the two-socket platform is close to 100GB/s.

the workload bottleneck is single-threaded operators.

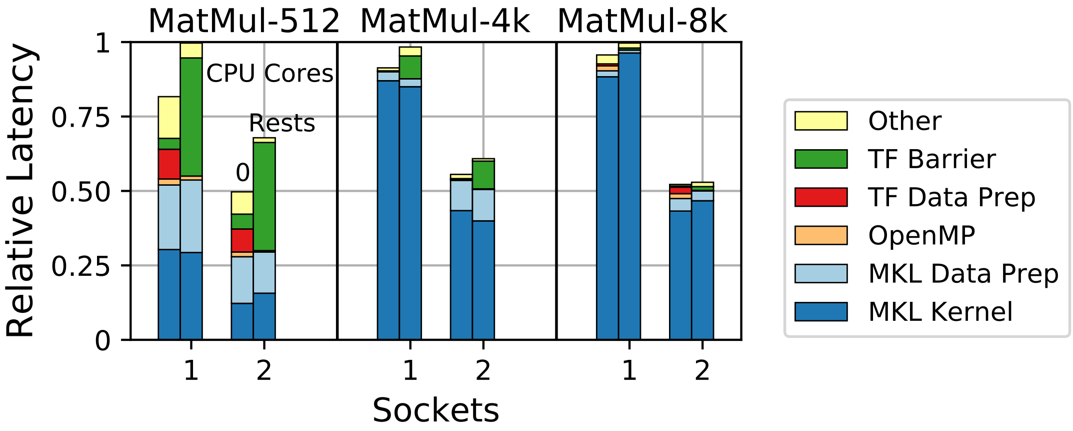


Figure 3.20: Run-time breakdown of all CPU cores. MatMul-512's parallelism cannot hide data preparation time, blunting its scalability. UPI saturation limits scalability of MatMul-4k and 8k.

3.8 Framework Design Tuning

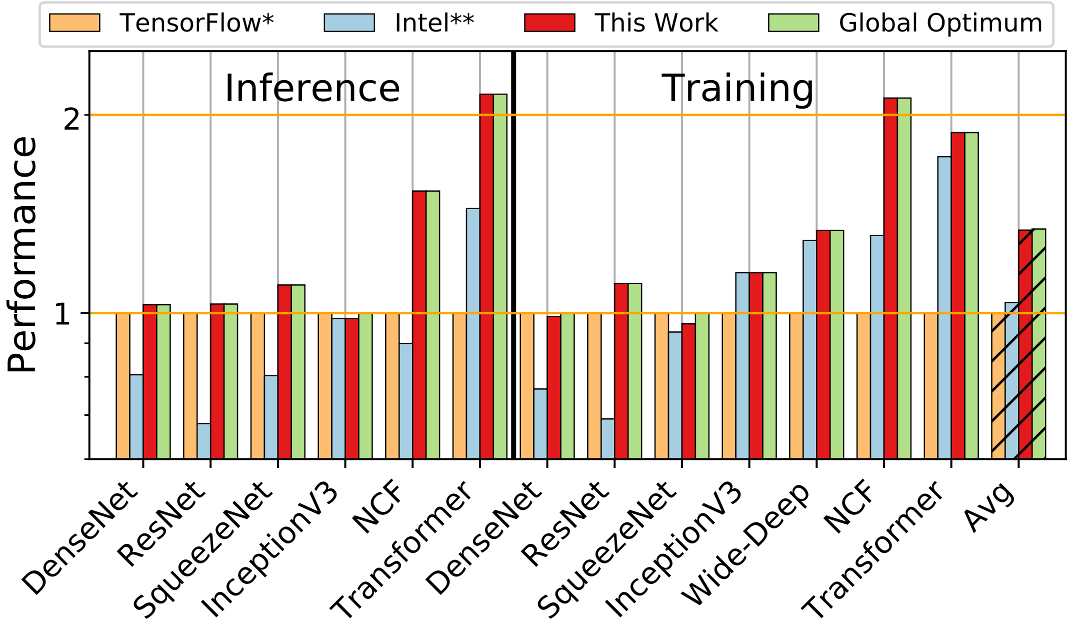
At the outset, we identified five design features: scheduling mechanism, operator implementation, math library, thread pool library, and the parallelism mechanism for platforms larger than one socket. Our analysis has shown that the most effective setting for users to determine is the proper number of inter-op pools based on the model architecture. Intra-op parallelism configurations follow from that setting.

DEFINITIONS: To determine the number of inter-op pools for a model, we need its *average width*, which quantifies its inter-op parallelism. (Section 3.4.1 and Figure 3.4 mention the maximum width.) The average width of a model is the floor of the ratio of the total number of (heavy) operators divided by the maximum number of layers. A *heavy operator* is a compute-intensive or embedding operator that usually takes significantly longer execution time than other operators. Examples are the Conv operators in Figure 3.6, as opposed to lightweight math operators, which are not considered. The average model width of Figure 3.6b is $\lfloor \frac{7}{3} \rfloor = 2$.

GUIDELINES: The number of inter-op pools (p) is chosen to be the average model width. After p is chosen, we choose the numbers of MKL and intra-op threads such that the entire system is split into p partitions without redundant threads (Section 3.4.2 and Figure 3.7). Therefore, the number of MKL threads and the number of intra-op threads for each thread pool should be equal to the total number of physical cores on the system divided by p. That way one MKL thread and one intra-op thread can share the same physical core. MKL threads can use the FMA units and the intra-op threads can use other units via hyperthreading (Section 3.5.2 and Figure 3.13).

Dense	Squeeze	ResNet	IncepV3	W/D	NCF	Trans
1	1	1	2	3	4	4

Table 3.2: Average model width, i.e., the number of pools selected for Figure 3.21 based on our guidelines. Intra-op and MKL threads = total physical cores divided by those numbers.



* TensorFlow: Intra-op/MKL threads = physical cores; inter-op pools = sockets.
** Intel: Intra-op/MKL threads = physical cores per socket; inter-op pools = sockets.

Figure 3.21: Performance using the recommended TensorFlow settings[75] (baseline), Intel blog[20], our work, and global optimum obtained by exhaustive search. Our work outperforms Intel and Tensorflow suggestions and nearly closes the gap between the state of the art and the global optimum.

EVALUATION SETUP: We integrate our guidelines with TensorFlow v1.13. (TensorFlow refers to inter-op pools as inter-op parallelism threads.) We will open source the TensorFlow plugin to the public, to let users apply our guidelines automatically. The size of the design space encompassing the numbers of MKL, intra- and inter-op threads is the cube of the number of logical cores. For the *large.2* system, that means $96^3 = 884,736$ design points. Our guidelines suggest picking only *one* of those $884,736$ possibilities.

We evaluate our guidelines by applying the rules to a fresh set of workloads and by performing the evaluation on a different platform from that used to develop the guidelines. Our analysis uses microbenchmarks and vision models that run with images; for evaluation, we add Inception v3[182],

the wide-deep recommendation model[47], the neural collaborative filtering model (NCF)[89], and Transformer[193], covering recommendation and translation workloads. The bulk of our analysis uses platforms *small* and *large* from Table 3.1; here we evaluate the guidelines on the *large.2* platform, the largest AWS bare metal instance.
SPEEDUP: Figure 3.21 summarizes the speedup of this chapter over TensorFlow[75] (baseline) and Intel[20] recommended settings. It also compares our settings' performance to the global optimum obtained by exhaustively sweeping the design space. TensorFlow suggests setting the number of MKL and intra-op threads to the physical core count, and inter-op pools to the socket count. Intel suggests setting MKL and intra-op threads to the number of physical cores per socket, and inter-op pools to the socket count. Our analysis shows that TensorFlow suggests more threads than needed, and Intel's setting is suitable for models with an average width of two.

Overall, our performance guidelines perform consistently better than the settings recommended by Intel and TensorFlow. Our method bridges the performance gap between those state-of-the-art settings and the global optimum for all evaluated workloads except Inception inference and SqueezeNet training. In those two cases, our guidelines achieve 95% of the performance offered by the global optimal setting. On average, this chapter achieves the same performance as the global optimum, and 1.34× and 1.29× better performance than TensorFlow's and Intel's suggestions, respectively.

We summarize the average model width in Table 3.2. It is the same as the number of inter-op pools in use. The models shown have average width between one and four, which is diverse. The numbers of MKL and intra-op threads for each model is the total number of physical cores (48) divided by the model width. For example, the setting for the W/D (wide and deep) model is 3 inter-op pools, 16 MKL threads, and 16 intra-op threads, which is also the globally optimal setting. The difference in speedup for different models can be observed from the difference between our settings and the baseline settings. We are able to get higher speedup for models with embedding tables including wide-deep, NCF and transformer. It is because those models are less compute-intensive and have abundant inter-op parallelism, and smartly tuning becomes especially important. On the other hand, most vision models do not have much inter-op parallelism, which makes our settings similar to baseline settings. Thus such models have relatively lower speedup.

The performance guides from Intel and TensorFlow are general, aiming to make it easy for most users get reasonable performance, and they perform reasonably well. For vision models, TensorFlow's settings perform as well as the global optima and our guidelines, while Intel's does not perform well for the vision models except for Inception, because Intel's setting favors models with inter-op parallelism that other vision models do not have. Intel's settings perform better than TensorFlow's for recommendation and translation models. The latter have several parallel embedding operators, thus their average width is no less than two. The default TensorFlow setting performs much worse than both the Intel and TensorFlow recommendations. TensorFlow naively sets all parameters—MKL threads, intra-op threads, and inter-op pools—to the number of logical cores. As pointed out in our earlier analysis, this is sub-optimal. Thus TensorFlow users who run only one model and one session at a time should at least set the number of inter-op pools to one instead of using the default setting.

3.9 Summary

We presented detailed evaluation and analysis of key design features and the role of parallelism in a machine learning framework, focusing on scheduling, operator implementation, and library back ends. To maximize parallelism, we proposed simple guidelines for tuning framework parameters, distilled from detailed domain-specific design feature knowledge and analysis. We demonstrated the usability and the additional performance improvement of this approach by integrating and evaluating our methodology with TensorFlow. On average, our method outperformed the suggested settings from Intel and TensorFlow performance guides by 1.29× and 1.34×, respectively, across a set of real-world DL models.

Science advances based on data, not hunches.

David Brooks

4

A Systematic Methodology for Performance Analysis

Training deep learning models is compute-intensive and there is an industry-wide trend to improve performance. To systematically compare deep learning systems, we introduce a methodology comprised of a set of analysis techniques and parameterized models. It can be applied to analyze various hardware and software systems, and is a very good complement to traditional methods.

4.1 Introduction

With the end of Moore's law, academic and industrial research efforts have shifted from general-purpose processors to domain specific architectures (DSAs)[91]. Deep learning, which has revolutionized many application domains[174,100,16,206], is a promising field for DSAs[56]. New customized training hardware, software stacks, and optimization tools are being developed to support ever more sophisticated deep learning models. Thus there is a great need to concurrently develop a systematic and scientific methodology for comprehensive performance analysis of hardware and software systems customized for deep learning.

The rapid evolution of deep learning models and corresponding hardware and software platforms requires new analysis techniques that go beyond simply running today's well-known deep learning models on individual platforms. A systematic methodology must expose interactions between hardware and software platforms across the spectrum of model attributes (e.g., hyperparameters), so that the resulting insights can be applied to future models. The methodology itself needs a fast development cycle to rapidly target new platforms, and it should include large enough models to stress the limits of emerging platforms.

Recent analysis efforts have been limited to relatively small collections of seemingly arbitrary DNN models[151,15,43,183]. The development of such suites is very time-consuming. It took half a year to release MLPerf v0.6, and months to add a new model. Even so, the shelf life of such models is seldom more than a couple of years. Moreover, using a collection of individual models such as ResNet-50[87] and Transformer[193] can lead to misleading conclusions. For example, Transformer is a large FC model that trains 3.5× faster on the Tensor Processing Unit (TPU) than on a GPU, yet focusing on this single model would not reveal the severe TPU memory bandwidth bottleneck that arises with FCs larger than 4k nodes.

We propose a comprehensive performance evaluation methodology that combines parameterized deep learning benchmarks with systematic analysis techniques. We introduce *ParaDnn*, a tool that generates thousands of parameterized multi-layer models, including fully-connected models, convolutional neural networks, and recurrent neural networks, with model parameter sizes that vary by almost five orders of magnitude, far beyond the range of existing benchmarks. Systematic analysis techniques then learn the sensitivity of performance to model hyperparameters and explore various dimensions of the design space. We show that this parameterized analysis methodology *complements* the use of real-world workloads (e.g., MLPerf), leading to insights that traditional approaches either cannot expose or cannot fully explain.

We conduct case studies in three diverse performance evaluation scenarios: homogeneous platforms, heterogeneous platforms, and software stacks. We hope to motivate researchers to apply our methodology to other platforms. In Section 4.4, we analyze and compare two generations of homogeneous specialized platforms, TPU v2 and v3. Our methodology provides insights for designing and upgrading ML accelerators in production-scale systems. In Section 4.5, we perform cross comparison of three architectures (CPU, GPU, and TPU) that span the continuum between general purpose processors and specialized accelerators, and the methodology reveals individual strengths and weaknesses of each platform. In Section 4.6, we explore the performance evolution of specialized software stacks, TensorFlow and CUDA.

It is important to identify limitations of the study. This chapter presents a benchmarking methodology, which is able to reveal optimization opportunities in current architecture and system designs, as they provide valuable lessons for future design. Optimization details are beyond its scope. The analysis focuses on training and not inference. We do not study accuracy, the performance of multi-GPU platforms or 256-node TPU systems, which may lead to different conclusions. We intentionally leave these topics to future work, as each deserves in-depth study. Section 4.7 discusses these and other limitations of the study, which also motivate future work.

4.2 Methodology

Current deep learning (DL) performance analysis methods have limitations in terms of the insights they are able to reveal. They often leverage two distinct types of benchmark suites: real-world suites such as MLPerf[151], Fathom[15], BenchNN[43], and BenchIP[183], and micro-benchmark suites, such as DeepBench[163] and BenchIP. Each real-world suite contains a handful of popular DL models spanning a variety of model architectures. Such suites have a long development cycle. Their shelf-life is unknown since they only contain today's deep learning models, which may become obsolete as DL models evolve rapidly. Further, they fail to reveal deep insights into interactions between DL model attributes and hardware performance, since the benchmarks are sparse points in the vast space of deep learning models. Micro-benchmark suites exercise basic operations (e.g., matrix multiplication or convolution) in neural networks, but they cannot simulate complex dependencies between different operations in end-to-end models.

To *complement* existing performance analysis methods, we introduce a systematic methodology, composed of a tool, ParaDnn, and a set of analysis methods. ParaDnn has the advantages of the above approaches, with the goal of providing large "end-to-end" models and parameterizing the models to explore a much larger design space of DNN model attributes. ParaDnn supports future performance analysis by covering current and possible *future* applications, and conveying analysis results with model hyperparameters such that the analysis can be extrapolated to future models. In this chapter, we will show that our methodology can stress the upper and lower bounds of hardware and software systems in various dimensions, including floating-point computation capability, memory bandwidth, inter-chip bandwidth, and host-device balance. For cross-platform comparisons, the methodology can also discover cases favoring one platform over another and describe the DL hyperparameters of such cases. With ParaDnn, such studies can be conducted on new platforms comprehensively, quickly, and conveniently. The utility of parameterized analysis is not limited to those cases, and one goal of this chapter is to motivate application of our methodology to new platforms.

4.2.1 ParaDnn

We first introduce ParaDnn, a tool that generates parameterized end-to-end models to run on target platforms. ParaDnn creates models encompassing fully-connected models (FC), convolutional neural networks (CNN), and recurrent neural networks (RNN). The models are parameterizable, so ParaDnn models are equal to or greater in size compared to today's real-world models. For example, a single end-to-end CNN model from ParaDnn contains a mixture of many different layers with different sizes of convolution, batch normalization, pooling, and FC layers. The complexity of ParaDnn workloads is comparable to that of real-world models (e.g., ResNet-50 and Transformer), as will be shown in Figure 4.1. Insights about hardware performance sensitivity to model attributes allow interpolating and extrapolating to future models of interest. These insights could not be discovered with either the small point space exploration of the real-world suites or microbenchmarks, which do not capture inter-operation dependencies as ParaDnn does. The model types of ParaDnn cover 95% of Google's TPU workloads[111], all of Facebook's deep learning models[86,79,145], and eight out of nine MLPerf models[151] with the exception of minigo, the reinforcement learning model.

FULLY-CONNECTED MODELS: FC models comprise multiple fully-connected layers. The architecture is

$$\text{Input} \rightarrow [\text{Layer}[\text{Node}]] \rightarrow \text{Output},$$

where [Layer] means the number of layers is variable. We can sweep the number of layers, the number of nodes per layer, and the numbers input and output units of the datasets.

CONVOLUTIONAL NEURAL NETWORKS: CNN models are residual networks. The architec-

Variable	Layer	Nodes	*Input*	*Output*	Batch Size
Min	4	32	2000	200	64
Max	128	8192	8000	1000	16384
Inc	×2	×2	+2000	+200	×2

(a) Fully Connected Models

Variable	Block	Filter	*Image*	*Output*	Batch Size
Min	1	16	200	500	64
Max	8	32	300	1500	1024
Inc	+1	64	+50	+500	×2

(b) Conv. Neural Nets: Residual and Bottleneck Blocks

Variable	Layer	Embed	*Length*	*Vocab*	Batch Size
Min	1	100	10	2	16
Max	13	900	90	1024	1024
Inc	+4	+400	+40	×4	×4

(c) Recurrent Neural Networks: RNN, LSTM, GRU

Table 4.1: The ranges of the hyperparameters and dataset variables (*italic*) chosen in this chapter.

ture of ParaDnn CNNs is

$$\text{Input} \rightarrow [\text{Residual/Bottleneck Block}] \times 4 \rightarrow \text{FC} \rightarrow \text{Output}.$$

A residual network contains four groups of blocks[87]. Each can be a residual block or a bottleneck block, followed by a fully-connected layer. Residual blocks have two convolutional layers and two batch normalization layers, while bottleneck blocks have three of each. ParaDnn treats the minimum number of filters as a variable, and it doubles in every group. An input image is square with three channels, represented by its length.

RECURRENT NEURAL NETWORKS: RNNs are comprised of multiple layers of basic RNN, LSTM, or GRU cells:

$$\text{Input} \rightarrow [\text{RNN/LSTM/GRU Cell}] \rightarrow \text{Output}.$$

Each token of the input sequence is embedded within a fixed length vector, and the length of the vector is the embedding size. We sweep the number of layers and the embedding size. The variables in the dataset include the maximum length per input sequence and vocabulary size.

RANGE OF HYPERPARAMETERS AND DATASETS: We choose the range of hyperparameters and datasets to cover the real models (Section 4.2.2), and make sure the design space is tractable. Table 4.1 summarizes how hyperparameters are swept. We focus on large batches, and extremely small batches may lead to different conclusions. By default, this chapter uses CNNs with bottleneck blocks and basic RNNs.

4.2.2 Real-World Models

In addition to ParaDnn, we study six real-world models. We show that ParaDnn and those models are complementary—ParaDnn explores a larger design space, and real models represent several currently popular design points.

This chapter focuses on TensorFlow, the native framework for TPU. We include two of the three workloads in TensorFlow from MLPerf[151], i.e., Transformer[193] and ResNet-50[87]. We also select other real-world workloads[162], including RetinaNet[135], DenseNet[100], MobileNet[99], and SqueezeNet[101]. We refer to them as real workloads/models.

Figure 4.1 shows the numbers of trainable parameters across all workloads to quantify the sizes of the models. The ParaDnn workloads are shown as ranges and the real workloads as dots. ParaDnn covers a large range of models, from 10k to nearly a billion parameters. Transformer is the largest real FC, and RetinaNet is the largest real CNN. The small models, SqueezeNet and MobileNet, are typical of models targeting mobile applications.

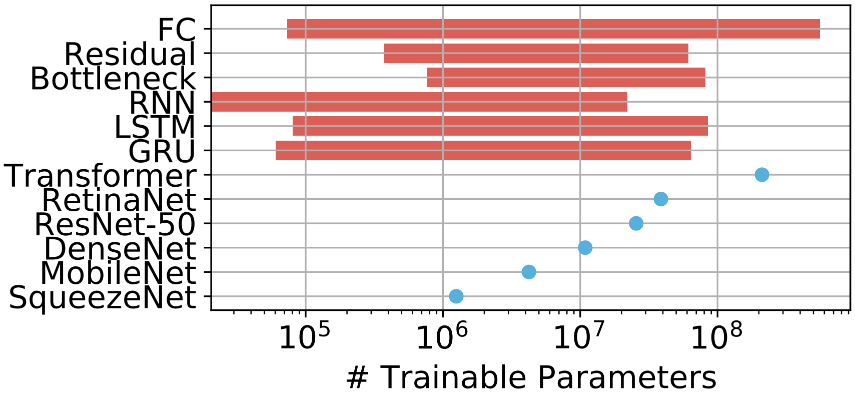


Figure 4.1: The numbers of trainable parameters for all models.

4.2.3 Analysis Methods

ParaDnn enables a set of analysis methods, that can quantify, compare, and visualize the DL design space in various dimensions. We apply those methods after running all ParaDnn workloads on platforms under study to collect performance metrics of interest. All analysis methods and results distinguish ParaDnn from suites of individual models, because the real-world suites do not support sensitivity analysis, and ParaDnn covers a much larger design space.

HEAT MAP: With ParaDnn, we can measure performance sensitivity to hyperparameters. Heat maps are an intuitive approach. A heat map uses colors to show how a performance metric of interest responds to model hyperparameters (on *x*- and *y*-axes). The rate of color change across the map reflects the sensitivity of performance to hyperparameters.

QUANTIFICATION WITH LINEAR REGRESSION: Table 4.1 shows five hyperparameters of each model type under study, and a heat map can only visualize two hyperparameters. Also, observing sensitivity via heat maps is more qualitative than quantitative. We propose to use linear regression (LR) to quantify the sensitivity. We train a LR model using hyperparameters to predict performance, and use the weights of the hyperparameters as a measure of sensitivity. Other metrics including T- and F-test may be used for this purpose[96], but they only report positive values of importance. LR reports the signs of the weights, indicating positive or negative correlations. Note that this LR model is not for prediction. Section 4.4.1 presents a detailed example.

ROOFLINE MODEL: Roofline models are useful to study memory and computation bottlenecks[204,111]. A roofline represents the upper bound of floating-point operations per second (FLOPS) for workloads with different arithmetic intensity. Roofline model analysis shows that ParaDnn's models range from extremely bandwidth-bound to compute-bound. Such a range is hard to achieve with existing real models, especially to reach the limits of TPUs. Section 4.4.2 presents detailed introduction.

OTHER DESIGN SPACE EXPLORATION METHODS: Roofline models study the design space of FLOPS and arithmetic intensity. We study the design space in other dimensions as well, by visualizing ParaDnn and real-world models on scatter plots with various *x*- and *y*-axes. Section 4.4.4 studies FLOPS and data infeed time. Section 4.5 studies model size and speedup.

BOX PLOTS: We use box plots to summarize the performance of each ParaDnn model type. Box plots show that the performance of a ParaDnn model type spans a large range, and they highlight the risk of overly optimizing hardware and software systems for certain models.

4.3 Hardware Platforms

Our selection of hardware reflects the latest configurations widely available in cloud platforms at paper submission time. Platform specifications are summarized in Table 4.2.

CPU PLATFORM: The CPU is an n1-standard-32 instance from Google Cloud Platform with Skylake architecture. It has 16 cores and 32 threads. It has large memory (120 GB) and lowest peak flops (2 TFLOPS) among the three. GeekBench 4 produced the bandwidth measurement.

GPU PLATFORM: The GPU is an NVIDIA V100 in a DGX-1 GPU platform that contains 8 V100 packages (SXM2) connected via 300 GB/s NVlink 2.0 interconnect. We currently measure the performance of a single SXM2 node. One node has 16 GB of memory and 900 GB/s memory bandwidth. A V100 has 640 tensor cores and is able to run mixed precision training using float16 to compute and float32 to accumulate, making its peak performance 125 TFLOPS.

TPU PLATFORM: We use Cloud TPU v2 instances to which we were given academic access in February 2018. Each TPU board contains four TPU packages (the default configuration)[54]. One package contains 2 cores and one core has one matrix unit (MXU). A Cloud TPU v2 platform supports 180 TFLOPS at peak. Memory size is 8 GB per core, or 64 GB per board, with 2400 GB/s overall memory bandwidth. TPU v2 supports mixed precision training using bfloat16 and float32. TPU v3 has twice the number of MXUs and twice the HBM capacity per core of v2[74]. Its memory bandwidth has not been disclosed, but empirical results show that it has increased by 1.5×. TPU v3 has a peak of 420 TFLOPS, 2.3× greater than v2.

This is the first research paper to study TPU v2/v3, which supports training, while TPU v1 only runs inference[111]. To enable training, TPU v2 supports more operations than matrix multiplication such as gradient and various optimizer operations. It also carries more pressure on the memory system, since weights are accessed twice more in the backward pass. Also, TPU v2 has scalar/vector units, which do not exist in v1. TPU v2 has MXUs of size 128×128 with 32- or 16-bit data types; v1

			Mem	*Mem Bdw*	Peak
Platform	*Unit*	*Version*	*(GB)*	*(GB/s)*	*FLOPS*
CPU	1 VM	Skylake	120	16.6	2T SP†
GPU	1 Pkg	V100	16	900	125T
TPU	1 Board (8 cores)	v2	64	2400	180T
TPUv3		v3	128	3600*	420T

† 2FMA × 32Single-Precision × 16Cores × 2GHz = 2 SP TFLOPS

* Estimated based on empirical results (Section 4.4.5).

Table 4.2: Hardware platforms under study.

has 256×256 and 8 bits.

UNDERSTANDING TPU MEMORY SIZE: The TPU implements data parallelism by splitting each batch of training data evenly among the 8 cores. Every TPU core keeps a whole copy of the model. Therefore memory size per core determines the maximum model supported (Sec 4.5.1), while total memory determines the maximum batch size (Sec 4.5.2).

COMPARISON RATIONALE: One V100 package and one TPU board (4 packages) are the minimal units available. On Cloud TPU, distribution of computation across its four packages happens automatically, while multi-GPU performance depends largely on user's implementation. Conclusions here do not apply to systems with multiple GPUs or TPU boards[41].

4.4 TPU Performance Implications

Google has been using TPUs for their large-scale production systems, including Search, Translate, and Gmail. Analyzing the architecture of such systems can provide valuable insights into future deep learning accelerator design. In this section, we use our methodology to study the performance characteristics of TPU v2 and v3[54,74], with a focus on v2, from the computation capability of the core to system balance. We show that ParaDnn can reveal system bottlenecks in a more comprehensive way than real-world models by probing upper and lower system limits. Based on such observations, we discuss possible steps to improve TPU performance, which can be generalized to other deep learning accelerator systems.

The following is a summary of our key observations and insights enabled by our methodology:

- FLOPS (Section 4.4.1): TPU makes good use of the parallelism exposed by batch size and model width, but parallelism due to model depth is under-exploited, suggesting opportunities for model pipelining[34].

- Memory bandwidth (Section 4.4.2): Memory bandwidth is the performance bottleneck of many models. Even highly-optimized compute-bound models show a significant fraction of memory-bound operations (13% in ResNet-50). Improving memory access for such operations is key to further performance improvement.

- Multi-chip overhead (Section 4.4.3): Communication overhead in a multi-chip system is non-negligible (up to 13% for CNNs with sizes similar to ResNet-50) but can be amortized with large batch sizes. Reducing the communication overhead can lead to performance gain.

- Host-device balance (Section 4.4.4): Smaller models are more likely to be data-infeed-bound from hosts.

- TPU v3 (Section 4.4.5): The maximum speedup of TPU v3 over v2 is up to 3×, exceeding the 2.3× FLOPS increase. TPU v3 benefited from its doubled memory capacity (which allows twice the batch size of v2) as well as increased memory bandwidth.

4.4.1 FLOPS Utilization

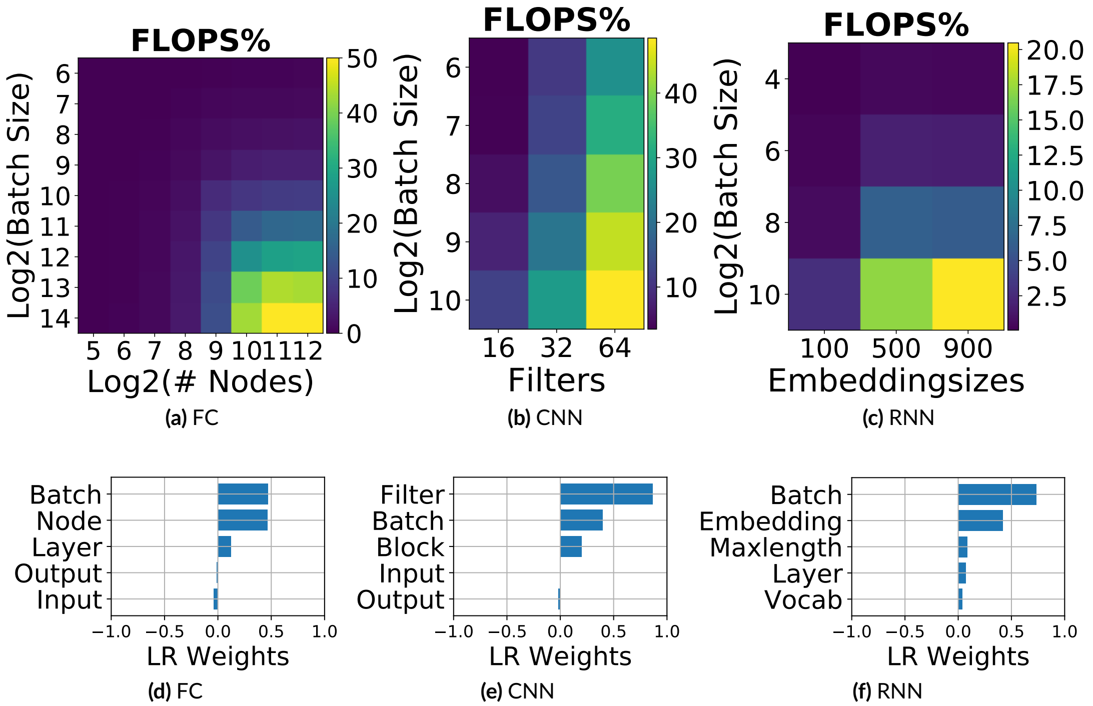
We use our methodology to study the TPU's floating-point operations per second (FLOPS) utilization, which is the ratio of workload average FLOPS to platform peak FLOPS, measuring how efficiently the computation capacity of a platform is used. We measure the FLOPS of ParaDnn models sweeping hyperparameters listed in Table 4.1. To visualize FLOPS, we use heat maps.

HEAT MAPS: Figures 4.2(a)–(c) present heat maps of FLOPS utilization for FC, CNN, and RNN ParaDnn models. For each model type, we choose two hyperparameters (as described below) that affect FLOPS utilization the most, sweeping their ranges to create a map grid while keeping other hyperparameters fixed. FLOPS utilization of all three model types increases with batch size, indicating that the TPU is capable of leveraging the parallelism within a batch. FLOPS utilization of FCs also increases with node count per FC layer; that of CNNs also increases with filter count; and that of RNNs, with embedding size. So the TPU also exploits parallelism within the widths of the models.

QUANTIFYING WITH LINEAR REGRESSION: To quantify these effects, we use the weights of a linear regression (LR) model. For FC, the LR model is

$$\begin{aligned}\text{FLOPS} = \; & w_0 \times \text{layer} + w_1 \times \text{node} + \\ & w_2 \times \text{input} + w_3 \times \text{output} + w_4 \times \text{batch size},\end{aligned}$$

where w_0–w_4 are hyperparameter weights. When training the LR model, we normalize weights to the same scale, so that weight reflects importance. For example, a positive w_1 value shows that node



(a) FC **(b)** CNN **(c)** RNN

(d) FC **(e)** CNN **(f)** RNN

Figure 4.2: (a)–(c) ParaDnn's FLOPS utilization and (d)–(f) its sensitivity to hyperparameters.

count affects performance positively.

Figures 4.2(d)–(f) show the LR weights of the model hyperparameters. Batch size and model width have the highest absolute weights, shown on the *x*- and *y*-axes in Figures 4.2(a)–(c). Figure 4.2(d) shows that the FLOPS utilization of FCs is largely affected by batch size and node count, while layer count, and output and input unit counts do not matter as much. Similarly, Figure 4.2(e) shows that filter count and batch size are most important for CNNs. For RNNs, utilization is most affected by batch and embedding sizes.

TAKEAWAYS: ParaDnn enables systematic study of hyperparameter sensitivity and shows that it is natural for a ML system to utilize parallelism arising from large batch size and model width. It is especially intuitive to map batch size and model width to the two dimensions of systolic arrays. Meanwhile, parallelism opportunities opened by large numbers of layers remain to be explored via model parallelism[55,107] and pipelining[34], while such techniques may change model numerics.

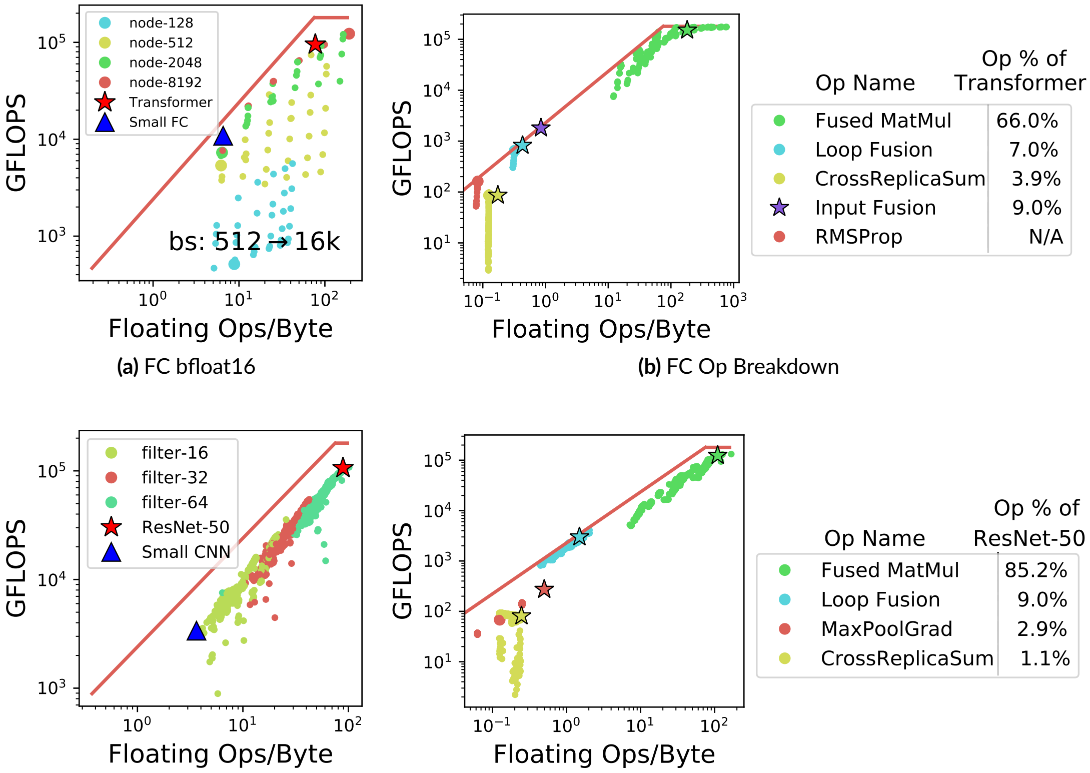
4.4.2 Roofline Model Analysis

The computation capacity of the TPU's core is only one source of its performance. Memory bandwidth also has a significant impact. In this section, we apply the roofline model[204] to ParaDnn FCs and CNNs to analyze the TPU's computation and memory bandwidth. We omit RNN models because the TPU profiler reports incorrect numbers for RNN memory bandwidth.

THE ROOFLINE MODEL: Figure 4.3 shows the roofline plots. The y-axis is FLOPS and the x-axis is arithmetic intensity, i.e., floating-point operations per byte transferred from memory. The roofline (the red line in Figure 4.3) has of a slanted part and a horizontal part. It represents the highest achievable FLOPS at a given arithmetic intensity. Any data point (x, y) on the slanted part has $\frac{x}{y}$ = memory bandwidth. The horizontal part is the hardware peak FLOPS. A workload or operation (a point in Figure 4.3) close to the slanted roofline is memory-bound; one close to the horizontal part is compute-bound. A workload or operation not close to the roofline stresses neither memory interconnect nor compute units. Figures 4.3(a) and 4.3(c) show all the ParaDnn FCs and CNNs (dots) plus Transformer and ResNet-50 (stars). Figures 4.3(b) and 4.3(d) show all the operation breakdowns. The triangles in Figures 4.3(a) and 4.3(c) are selected memory-bound models.

The design space shown with roofline models indicates that ParaDnn is a superset of the real-world models. ParaDnn models span a much larger range in the design space, from extremely memory-bound to compute-bound. Therefore performance analysis with ParaDnn can comprehensively test the limits of platforms in both extremes. An exception is that some operations of Transformer do not align closely with those of FCs. This results from a choice in this chapter: ParaDnn uses the RMSProp optimizer, keeping nodes per layer uniform for FCs, while Transformer uses the *adafactor* optimizer and has layers with 4k, 2k, and 512 nodes.

PARADNN ANALYSIS: We first discuss the insights enabled by ParaDnn, of which the real-world



Op Name	Op % of Transformer
● Fused MatMul	66.0%
● Loop Fusion	7.0%
● CrossReplicaSum	3.9%
☆ Input Fusion	9.0%
● RMSProp	N/A

Op Name	Op % of ResNet-50
● Fused MatMul	85.2%
● Loop Fusion	9.0%
● MaxPoolGrad	2.9%
● CrossReplicaSum	1.1%

(a) FC bfloat16

(b) FC Op Breakdown

(c) CNN bfloat16

(d) CNN Op Breakdown

Figure 4.3: TPU rooflines for FCs and CNNs. (a) and (c): ParaDnn and real-world models. (b) and (d): their operation breakdown. The ridge point of the roofline has $x = 75$. The legends also show the total execution time percentages of corresponding operators in Transformer and ResNet-50.

models are a subset. Figure 4.3(a) shows that large batch sizes make FCs more compute-bound, and more nodes make FCs more memory-bound. That is because FCs with more nodes need to transfer more weights/activations from the memory, and large batch sizes increase the computation per weight/activation transferred, i.e, the arithmetic intensity. Specifically, FCs with $\geq$ 2k nodes per layer and $\geq$ 8k batch size are compute-bound. Transformer is close to compute-bound and it uses 4k batch size, so it overlaps with FCs having 4k batch sizes. Figure 4.3(c) shows that models close to ResNet-50 are compute-bound, while a majority of the CNNs are bottlenecked by memory bandwidth. The CNNs' higher FLOPS comes from higher arithmetic intensity caused by more filters.

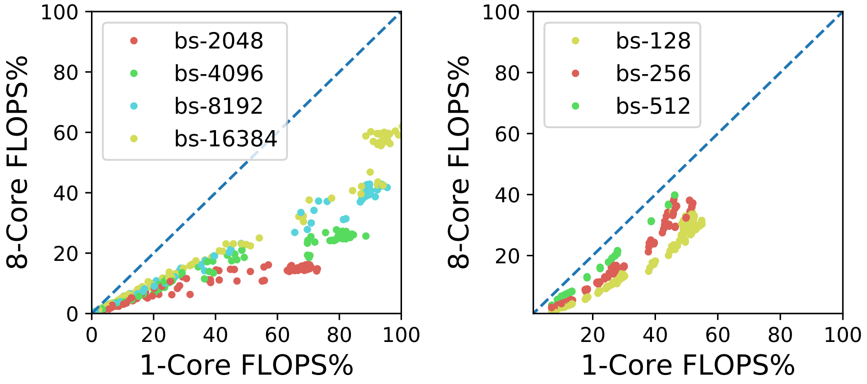
When memory bandwidth is the bottleneck, the way to increase FLOPS is to increase arithmetic intensity.

Figures 4.3(b) and 4.3(d) show the TensorFlow operations that take more than 1% of the workload execution time and more than 0 FLOPS. The arithmetic intensity of such operations can be as low as 0.125.* The TensorFlow breakdown in Figure 4.3 is generated after operation fusion, which combines and executes several operations together for higher efficiency. In Figures 4.3(b) and 4.3(d), the only compute-bound operation is large fused MatMul (MatMul fused with other operations), so a compute-bound model needs large MatMuls. Other operations are closer to the slanted line, constrained by memory bandwidth. Transformer and ResNet-50 are compute-bound (Figures 4.3(a) and 4.3(c)) because they have compute-bound MatMuls (Figures 4.3(b) and 4.3(d)).

REAL-WORLD MODEL ANALYSIS: ParaDnn and real-world models are complementary. By analyzing ParaDnn, we explore the design space and reach the limits of platforms. Analyzing real-world models puts the design space study into realistic context by highlighting popular representative designs. The tables in Figure 4.3 show the operation breakdown of Transformer and ResNet-50, and indicate that even compute-bound models contain a noticeable fraction of memory-bound operations. Transformer has three memory-bound operations: (1) input fusion (9.0%), which includes multiply, subtract, and reduce; (2) loop fusion (7.0%), which consists of control flow operations (e.g., select and equal-to); and (3) CrossReplicaSum (3.9%), which sums up the values across multiple weight replicas. These three operations contribute 19.9% of the total execution time. (12.3% of the execution time is for data formatting, which has no arithmetic intensity or TPU FLOPS.) ResNet-50 has memory-bound loop fusion (9%), MaxPoolGrad (2.9%), and CrossReplicaSum (1.1%), which sums to 13%, showing the need for end-to-end optimization for DL accelerators.

TAKEAWAYS: ParaDnn explores the design space and stresses platform limits; real-world models

* An activation accumulation operation (CrossReplicaSum in TensorFlow) uses float32 even with bfloat16 model weights. In this case, the arithmetic intensity is $1/(2 \times 4 \text{ bytes}) = 0.125$, i.e., one floating-point addition for every two data points loaded.



(a) FC

(b) CNN

Figure 4.4: Communication overhead in a multi-chip system is non-negligible, but is reduced with large batch sizes.

represent the currently important design points. ParaDnn shows that the design space is composed of very diverse models, from extremely memory-bound to compute-bound; real-world models show that even compute-bound models contain non-negligible fractions of memory-bound operations (19.9% for Transformer and 13% for ResNet-50), which suggests that memory bandwidth can affect other ML systems originally designed to optimize computation. Researchers can test system memory-boundness with ParaDnn. Approaches for speeding up memory-bound operations include caching[91], operation fusion[12,45,164], aggressive data quantization[25], and compression[81,136].

4.4.3 Multi-Chip Overhead

Computing speed and memory bandwidth of a TPU core are not the only factors affecting training performance, because typical large-scale systems use multiple chips[55]. This section evaluates the scalability of a multi-chip TPU system with ParaDnn. We quantify the multi-chip overhead by comparing the FLOPS utilization of 1-core (x-axis) and 8-core TPUs (y-axis) in Figure 4.4. If there were no multi-chip overhead, FLOPS utilization of 1-core and 8-core should be the same, i.e., all points should lie on the dashed line in Figure 4.4 showing $x = y$.

ParaDnn allows us to explore models with a wide range of communication overhead. Figure 4.4 shows that an 8-core TPU exhibits noticeably lower FLOPS utilization than a 1-core TPU, reflecting significant inter-core communication overhead. For FC, the maximum FLOPS utilization in an 8-core TPU is 62%, compared to 100% in a 1-core TPU. Multi-chip overhead is less noticeable in CNNs, with FLOPS utilization decreasing from 55% in the 1-core TPU to 40% in the 8-core. It is worse for FCs because there are more weights to synchronize across cores than for CNNs.

Our analysis method indicates that large workloads can amortize the parallelism overhead, and it highlights batch size as the key hyperparameter that affects communication overhead. Increasing batch size reduces the FLOPS utilization gap by increasing computation without increasing weight synchronization. On the 8-core TPU, FCs need at least 16k batch size to achieve more than 50% FLOPS utilization. Specifically, FCs with $\geq$ 256 nodes and $\leq$ 512 batch size run faster on a TPU with one core than on one with eight. Thus we consider FCs with larger than 1024 batch size in Figure 4.4. Based on Amdahl's law, the maximum non-parallel fraction of the workloads is up to 60% for FC and up to 40% for CNN. Using the largest batch size shown in Figure 4.4, the 90th-percentile of non-parallel fractions are 16% for FC and 8.8% for CNN.

TAKEAWAYS: With diverse ParaDnn models, we observe that communication overhead in multi-chip systems is non-negligible even for large FCs and CNNs. Using large batch size can reduce overhead by increasing parallel computation without increasing weight transfers. Possible optimizations include relaxed synchronization, model parallelism[55], gradient compression[136], and weight pruning and compression[81].

4.4.4 Host-Device Balance

Previous subsections have focused on the performance of the accelerator itself. We now turn to "data infeed," the process of preparing and moving input data to the TPU board. ParaDnn in other sections uses data synthesized from CPU hosts, which avoids most of the data infeed overhead. Here

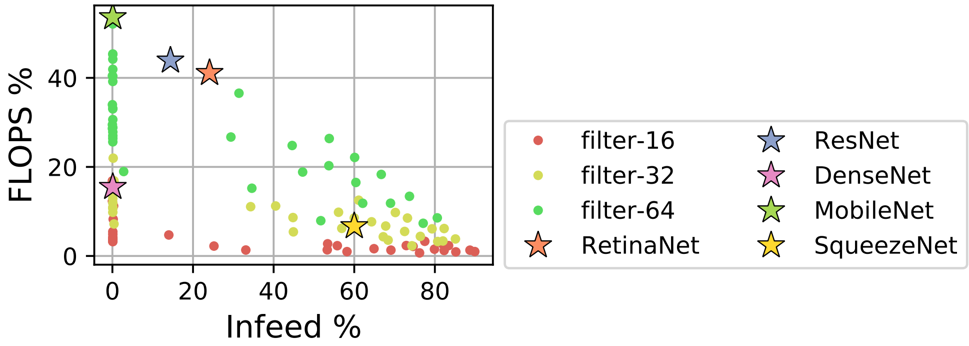


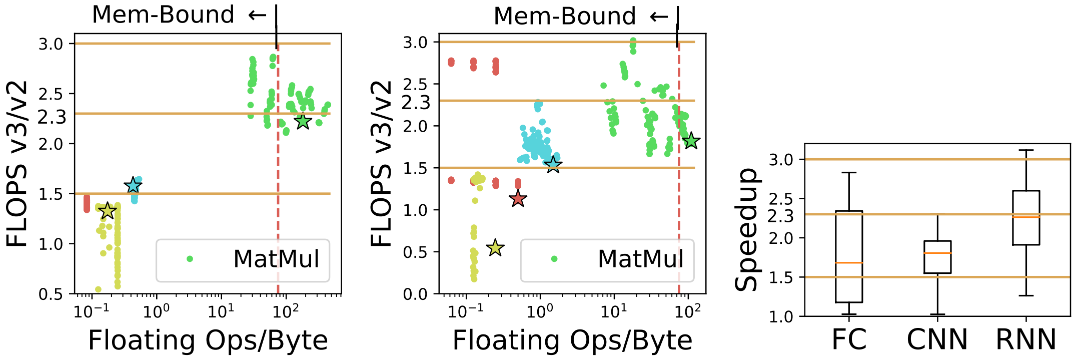
Figure 4.5: The FLOPS utilizations and data infeed percentages of ParaDnn (dots) and real-world (stars) CNNs.

we use ParaDnn CNN models with the ImageNet dataset[123].

The TPU system includes a CPU host and a TPU device[74]. For image datasets, the host fetches images from the network, decodes and preprocesses them, and feeds them to the device. We refer this as data preparation. The device then performs training computation on the images. Data infeed includes network overhead, host compute, and transfer between host and device.

For each ParaDnn CNN and the ImageNet dataset, we use the TPU profiler to collect FLOPS utilization and infeed time percentage, which is the fraction of time the accelerator spends waiting for data. Figure 4.5 shows the results as dots, along with the real-world CNNs as stars. ParaDnn models are very diverse, ranging from 0 to 50% FLOPS utilization and 0 to 90% infeed time. ParaDnn shows that many CNNs have significant infeed time and that larger CNNs tend to have lower infeed time. Large CNNs, those with more filters and/or more layers, are the most suitable for the TPU system, because the accelerator spends more time training each image and CPU infeed time per image is fixed. The high-performance TPU system targets large workloads. Consistent with the communication overhead study in Section 4.4.3, small workloads do not have enough parallelism to utilize the TPU efficiently.

Some real models show opportunities to optimize for data-infeed bottlenecks. SqueezeNet has



(a) FC Ops **(b)** CNN Ops **(c)** TPU v3 over v2

Figure 4.6: Speedup of TPU v3 over v2 for (a) FC operations, (b) CNN operations, and (c) ParaDnn models. The red line (75 ops/byte) is the inflection point in the TPU v2 roofline (Fig 4.3).

the highest infeed time because it is designed to accommodate mobile devices by using small numbers of filters per layer. While MobileNet also targets mobile devices, it has one convolution layer that uses up to 1k filters, eliminating time lost to data infeed. RetinaNet, ResNet, and SqueezeNet show potential for improvement through system optimizations. The performance without data preparation shows that resolving the infeed bottleneck can lead to 37%, 34%, and 180% performance improvement, respectively.

TAKEAWAYS: ParaDnn shows the design space of FLOPS and data-infeed time, and reveals that large workloads with abundant parallelism are not host-bound on large accelerator systems. Real-world models show that the performance of some workloads can be improved. When designing an accelerator system, scaling performance of the CPU host to match the accelerator is crucial for utilization of the accelerator's computation resource.

4.4.5 TPU v3

In this section, we systematically quantify the differences between TPU v2 and v3. Figure 4.6 compares the two using ParaDnn (dots), plus ResNet and Transformer (stars). Batch size for v3 is twice that for v2, thanks to its doubled memory capacity. Figures 4.6(a) and 4.6(b) use a variation of the roofline model, showing arithmetic intensity on the *x*-axis and operation speedup on the *y*-axis. Data point colors representing operation types are consistent with those in Figures 4.3(b) and 4.3(d). As a reference, the red dashed line is the inflection point in the TPU v2 roofline from Figure 4.3, where arithmetic intensity is 75 ops/byte (180 TFLOPS / 2.4 TB/s). The operations on the left of the red line are memory-bound; those on the right are compute-bound. We group the operations in four classes, as follows.

COMPUTE-BOUND OPS: The peak FLOPS of TPU v3 is 2.3× that of v2, so the performance of compute-bound operations is improved by about 2.3× on v3. Such operations are on the right of the red dashed line in Figure 4.6(b).

MEMORY-BOUND OPS (2× BATCH SIZE): The maximum speedup of the memory-bound operations (mainly the MatMuls in Figures 4.6(a) and 4.6(b)) is 3×. Tripled speedup comes from doubled batch size (owing to doubled memory capacity) and memory bandwidth improvement. The memory bandwidth increase of v3 over v2 has not been officially disclosed, but we can estimate it. Doubled batch size means doubled arithmetic intensity. On the slanted line of a roofline model, that means doubled FLOPS, because the ratio of FLOPS to arithmetic intensity is fixed. Switching from v2's roofline to v3's thus increases FLOPS by twice the bandwidth improvement. So the 3× overall speedup suggests that v3 bandwidth improvement over v2 is $3/2 = 1.5\times$, to 3.6 TB/s.

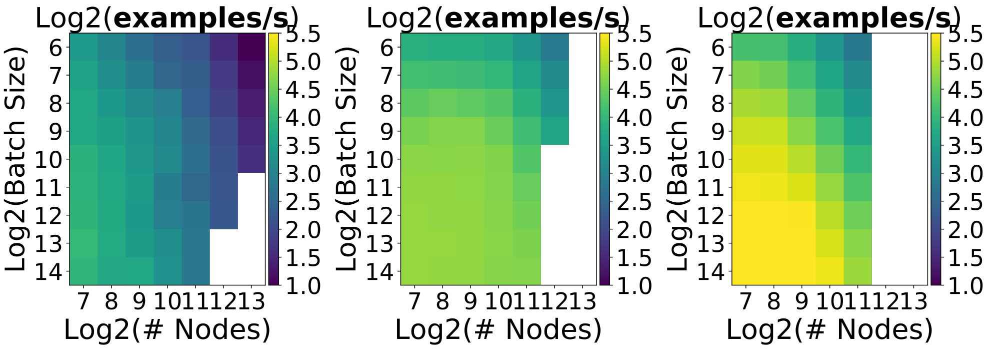
OTHER MEMORY-BOUND OPS: The 1.5× bandwidth improvement estimate is corroborated by the 1.5× speedup of other memory-bound operations, represented by the non-MatMul FC operations (such as CrossReplicaSum, RMSProp, and control flow operations) in the lower left corner of Figure 4.6(a). The performance of those operations does not increase with larger batch size, as shown by the vertical alignment of each operation type in Figure 4.3(b). Thus the 1.5× speedup in Figure 4.6(a) is from bandwidth improvement.

BOUNDARY CASES: The compute-bound MatMuls in Figure 4.6(b) become memory-bound on

TPU v3, so the speedup is $< 2.3\times$. Such operations have arithmetic intensity between 75 and 117, because the roofline inflection point of v3 is at $x = 420/(6 \times 1.5) = 117$. CrossReplicaSum (yellow dots) is slowed down on TPU v3, which may be because of more replicas across more MXUs.

END-TO-END MODELS: In Figure 4.6(c) the maximum speedups are 2.83× (FC), 2.31×(CNN), and 3.11×(RNN). Speedup increases with model width (second column of Table 4.1), and the maximum speedup is achieved by the largest width. FCs with close to 3× speedup are dominated by memory-bound MatMuls. Exceptions are RNNs with more than 3×; these have the largest embedding size (900), indicating that TPU v3 optimizes large embedding computations.

TAKEAWAYS: ParaDnn allows examining new platforms with a wider range of workloads, from memory-bound to compute-bound, than using real-world models alone. Comparing TPU v3 to v2 as an example, ParaDnn can show the system upgrade benefiting operations with different arithmetic intensity. TPU v3 shows three main levels of speedup: 2.3× for compute-bound operations, 3× for memory-bound MatMuls, and 1.5× for other memory-bound operations. This is the result of its 2.3× FLOPS, 2× memory capacity, and 1.5× memory bandwidth.



(a) CPU **(b)** GPU **(c)** TPU

Figure 4.7: Examples/second of ParaDnn FC models with fixed layer (64). Larger memory allows CPU to run the largest model.

4.5 Cross-Platform Comparison

In this section, we show ParaDnn's utility in cross-platform comparison, with CPU, GPU, and TPU, exemplars along the continuum between general purpose processors and specialized accelerators. ParaDnn shows the sensitivity of speedup to model hyperparameters, allowing users to choose platforms based on model characterizations that happen to have been reported. ParaDnn also reveals the fundamental architectural differences between the platforms and shows the trade-offs between flexibility and specialization. There are scenarios where each of the platforms is valuable:

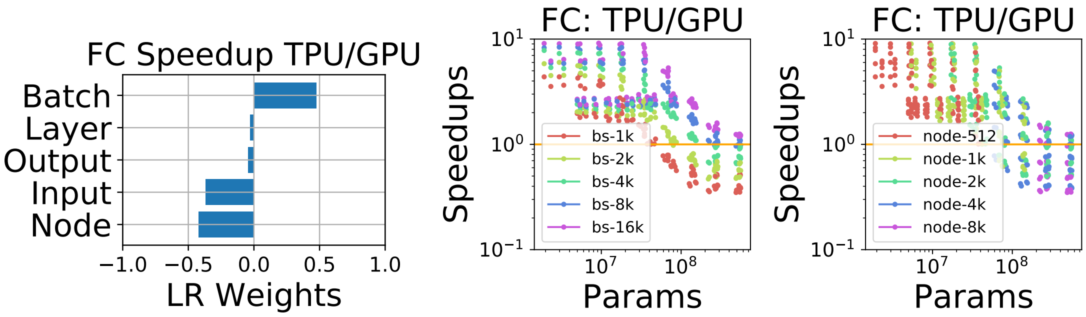
- The TPU is highly-optimized for large batches and CNNs, and has the highest training throughput.
- The GPU is more flexible and programmable for irregular computations, such as small batches and non-MatMul operations. Training of large FC models benefits from its sophisticated memory system and higher bandwidth.
- The CPU has the best programmability, so it achieves the highest FLOPS utilization for RNNs, and it supports the largest model because of its high memory capacity.

4.5.1 Fully-Connected DNNs

Examples/second measures the number of examples trained per second, a proxy for end-to-end performance. Heat maps in Figure 4.7 compare ParaDnn FCs for three platforms, with varying node counts and batch sizes but fixed layer count (64). We use LR weights from Section 4.4.1 to quantify the hyperparameter effects. Layer and node counts have negative weights because it is more time-consuming to train larger models with many layers and nodes. Batch size greatly improves throughput on GPU and TPU, but not the CPU because the parallelism available with small batch sizes can fully utilize a CPU.

In Figure 4.7, the white squares indicate models that encounter out-of-memory issues. It is interesting to note that only the CPU supports the largest models, and the GPU supports larger models than the TPU. This is because every hardware core keeps one copy of the model, so memory per core determines the largest model supported, as explained in Section 4.3. The CPU has the highest memory per core (120 GB), and the GPU (16 GB) is higher than the TPU (8 GB). While TPUs and GPUs may draw more attention, as of today the only choice for extremely large models is the CPU, which supports all model sizes. For example, Facebook uses dual-socket CPU servers with large memories to train ranking models (FC networks)[86]. That fact highlights the need for model parallelism and pipelining on the GPU and TPU[55,107,34] to allow those powerful platforms tocan support larger models..

TPU OVER GPU: To further investigate the best hardware platform for FC models, we analyze TPU over GPU speedups. Figure 4.8(a) plots the linear regression weights across FC hyperparameters for TPU over GPU speedup. Figures 4.8(b)–4.8(c) show the design space of FCs as scatter plots, with numbers of model parameters on the *x* axis and speedups on the *y* axis. To display the effects of hyperparameters, we color-code data points to reflect batch size (Figure 4.8(b)) and node count per layer (Figure 4.8(c)). Overall, 62% of the FCs perform better on the TPU.



(a) LR Weights

(b) Batch Size

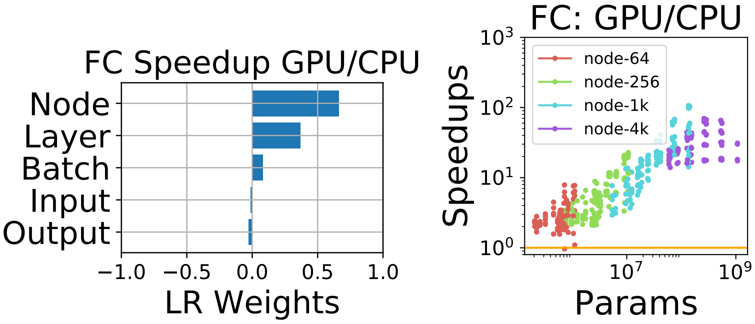
(c) Node

Figure 4.8: (a) Sensitivity analysis of TPU over GPU speedups. Speedups color-coded by (b) batch size and (c) FC nodes per layer.

The TPU is well suited for large batch training because systolic arrays excel at increasing throughput[124]. The positive weight of batch size in Figure 4.8(a) and the horizontal color bands in Figure 4.8(b) indicate that large batch size is the key to higher speedup. This suggests that the TPU MXUs, implemented with systolic arrays, need large batches to reach full utilization. The GPU is a better choice for small batches, because it executes computation in warps, so it packs small batches and schedules them on stream multiprocessors more easily[148].

GPU is a better choice for large models, suggesting that it is optimized for large FC memory reuse and streaming requirements. This is shown by the negative weights of node count, layer count, and input size in Figure 4.8(a) and the trend in Figure 4.8(c), corroborated by the overall negatively-correlated trend of speedup with parameter count in Figure 4.8. FCs have minimal weight reuse and large models have more weights, so they put a lot of pressure on the memory system. The GPU has a more mature memory system and higher memory bandwidth than the TPU, which makes it better-suited to the memory requirements of large FCs.

GPU OVER CPU: Figure 4.9(a) shows the LR weights of GPU-over-CPU speedup. Figure 4.9(b) shows the design space color-coded by node count. The GPU is a better platform for large FCs because its architecture can better exploit the parallelism available with large batches and models. Re-



(a) LR Weights

(b) Node

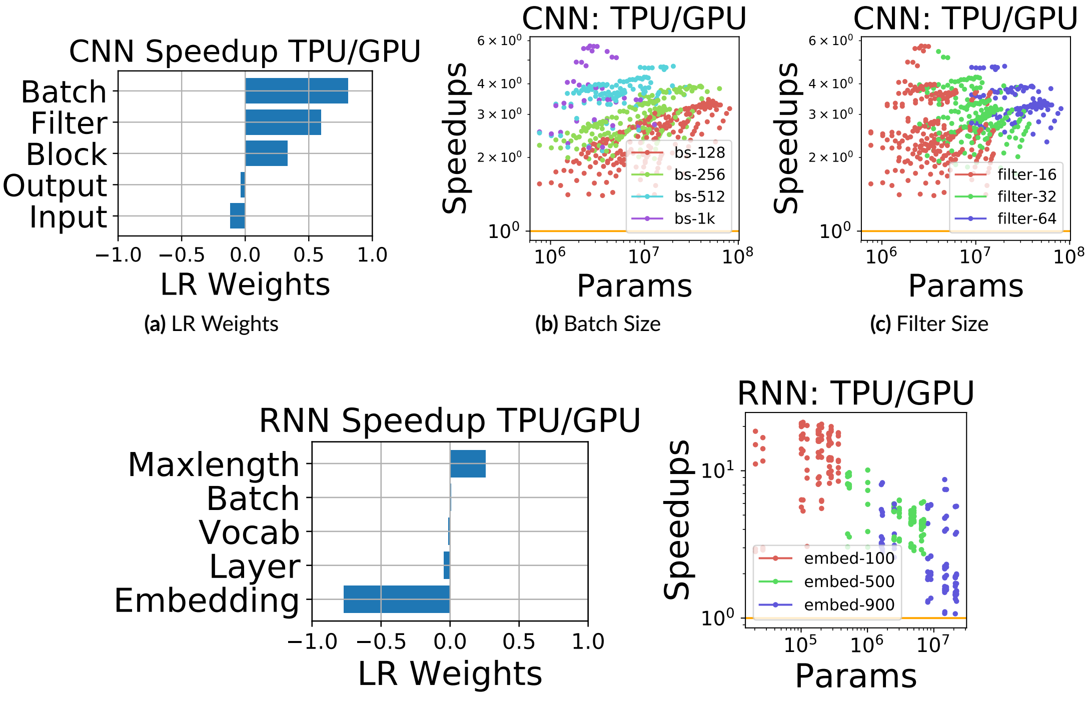
Figure 4.9: (a) The sensitivity analysis of (b) GPU over CPU speedups for FCs.

call from Figure 4.8 that that large FCs prefer the GPU over the TPU. So the GPU is the best platform for large FCs, but models with large batches perform best on the TPU.

4.5.2 CNN and RNN

We now describe the speedup of ParaDnn CNNs and RNNs. Since our conclusions for CPUs are similar to those in the previous section, we omit them in the interest of brevity.

CNN: Figures 4.10(a)–4.10(c) show the speedups of the TPU over the GPU. All CNNs perform better on the TPU. Batch size is still the key to better TPU-over-GPU speedup for CNNs, shown by its positive LR weight in Figure 4.10(a) and the increasing speedup with batch size in Figure 4.10(b). The TPU is the best platform for large CNNs, suggesting that its TPU architecture is highly optimized for the spatial reuse characteristics of CNNs. This is shown by the positive weights in Figures 4.10(a) and 4.10(c), where models with more filters and blocks have higher speedups. It is different from Section 4.5.1, showing that TPU is not preferred for large FCs. This suggests that TPU handles large CNNs than large FCs because CNNs reuse weights but FCs seldom do, which results in greater memory traffic. The GPU is a feasible choice for small CNNs. These conclusions only apply to a single GPU; multi-GPU cases may be different.



(a) LR Weights **(b)** Batch Size **(c)** Filter Size

(d) LR Weights **(e)** Embedding Size

Figure 4.10: The sensitivity analysis and speedups of TPU over GPU for (a)–(c) CNNs and (d)–(e) RNNs.

RNN: Figures 4.10(d)–4.10(e) show the speedup of TPU over GPU. We display the embedding size in Figure 4.10(e) because the magnitude of its weight is the greatest in Figure 4.10(d). Embedding size has negative weight, and embedding computation is more sparse than matrix multiplication. This suggests that the TPU is less flexible for doing non-MatMul computations than the GPU. The TPU is better at dense computations like MatMuls. Even so, RNNs are still up to 20× faster on the TPU. Optimizing non-MatMul computations is another opportunity for TPU enhancement.

4.5.3 Overall Comparison

This section summarizes the speedup of TPU over GPU and the FLOPS utilization of all ParaDnn and real models. We use box plots (Figure 5.8) to summarize each ParaDnn model type because performance has a wide range. The bar in the box shows the median. The upper and lower boundaries of the box are 9th and 91st percentiles. The upper and lower bars outside of the box are 2nd and 98th percentiles. Outliers are shown as dots. We do not show the results of using CPUs to train CNNs, because it is extremely time consuming and unlikely to contribute additional insights.

TPU OVER GPU SPEEDUP: Figure 5.8(top) summarizes the TPU over GPU speedups of all models. Note that the real models use larger batch sizes on TPU than on GPU. The speedup of TPU over GPU depends heavily on the nature of the workload measured. The speedup of parameterized models varies widely, from less than 1 to 10×, while the speedup of real workloads ranges from 3× (DenseNet) to 6.8× (SqueezeNet). ParaDnn represents a more complete view of potential workloads, and each real workload represents the concerns of certain users. Benchmarking platforms with two kinds of workloads offer a more systematic understanding of their behavior than those with only one kind.

To further compare the TPU with the GPU while relaxing the constraint on the GPU's software stack, we also include the speedup relative to GPU performance of ResNet-50, reported in NVIDIA's Developer Blog[39] (annotated as NVIDIA in Figure 5.8(top)). Note that NVIDIA's version of ResNet-50 uses unreleased libraries, and we were unable to reproduce the results. The speedup using ResNet-50 from Google is 6.2×, compared with 4.2×, which suggests software optimization can significantly impact performance.

FLOPS UTILIZATION: Figure 5.8(bottom) shows the FLOPS utilization of all workloads and platforms. On average, the maximum FLOPS utilization of the TPU is 2.2× that of the GPU for all CNN models, and the ratio is 3× for RNNs. The TPU FLOPS utilization of Transformers is consistent with FCs with 4k batch size, as in Figure 4.2. For RNNs, the TPU has less than 26% FLOPS utilization and the GPU has less than 9%, while the CPU has up to 46% because of its better programmability. RNNs have more irregular computations than FCs and CNNs, due to the temporal dependency in the cells and the variable-length input sequences. Advanced RNN optimizations

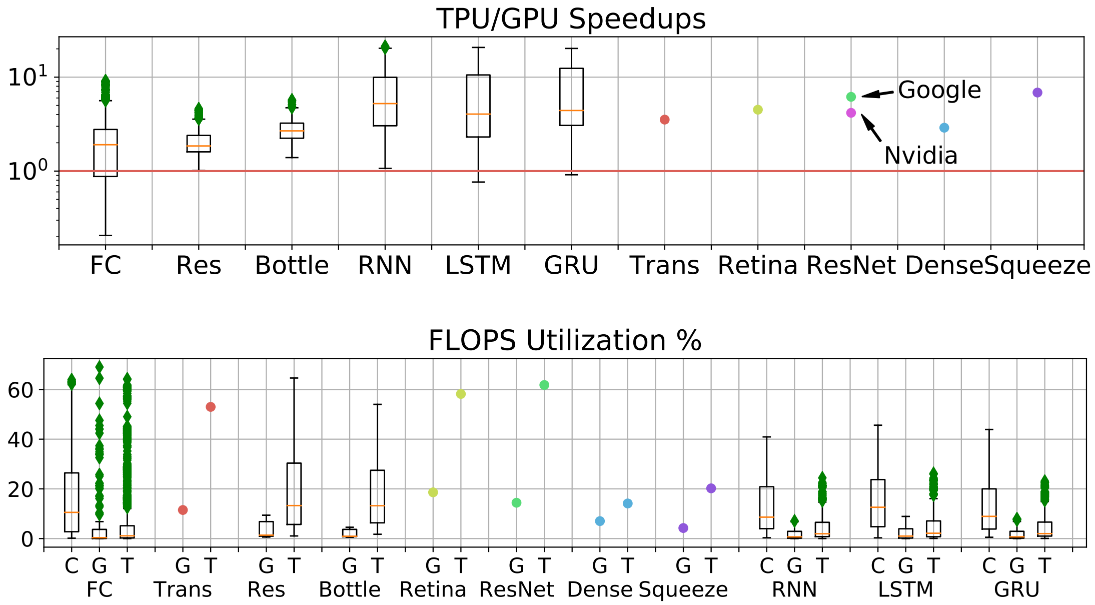


Figure 4.11: (Top) TPU over GPU speedups of all workloads. (Bottom) FLOPS utilization comparison for all platforms.

may be able to increase utilization on the GPU and the TPU. Real models with more filters have higher FLOPS, which is why ResNet-50 and RetinaNet have higher FLOPS than DenseNet and SqueezeNet.

4.6 Software Stack Advances

Custom DL hardware opens opportunities for dramatic software optimizations. ParaDnn is also useful for comparing software performance, by analyzing the performance of different TensorFlow (TF) and CUDA versions. We study data type quantization with software versions, because it depends on software support. Software versions are summarized in the legends of Figure 4.12. ParaDnn allows more comprehensive analysis of software updates than real models. It can also reveal software optimization focus (e.g., TF 1.9 optimizes small batches); we omit these details for brevity.

4.6.1 TensorFlow Versions and TPU Performance

The compiler for the TPU is XLA[128], shipped with TF. Figure 4.12(top) shows TPU speedups obtained by running TF 1.7 to 1.12, treating 1.7 with float32 as the baseline. Moving from TF 1.7 to 1.12 improves performance for all ParaDnn models. Although FC and CNN encounter performance regression with TF 1.8, TF 1.9 fixes this anomaly and improves overall performance. RNN performance is not improved much until TF 1.11. TF 1.11 shows 10× speedup for RNNs. Transformer, ResNet-50, and RetinaNet are improved continuously over TF updates. Interestingly, SqueezeNet is improved starting from TF 1.11, while the performance of DenseNet and MobileNet see little benefit. In the 7 months (222 days) between the release of TF 1.7.0 (03/29/2018) and that of TF 1.12.0 (11/05/2018), software stack performance was improved significantly. The 90th-percentile speedup of TPU is 7× for FC, 1.5× for Residual CNN, 2.5× for Bottleneck CNN, 9.7× for RNN, and 6.3× for LSTM and GRU.

Bfloat16 enables significant performance improvement for ParaDnn FCs and CNNs. 90th-percentile speedups are up to 1.8× for FC and Bottleneck CNN, and 1.3× for Residual CNN. TPU can support doubled batch sizes with 16 bits. Transmitting fewer bits also relieves bandwidth pressure, speeding up memory-bound operations. Larger performance increases may be possible with further

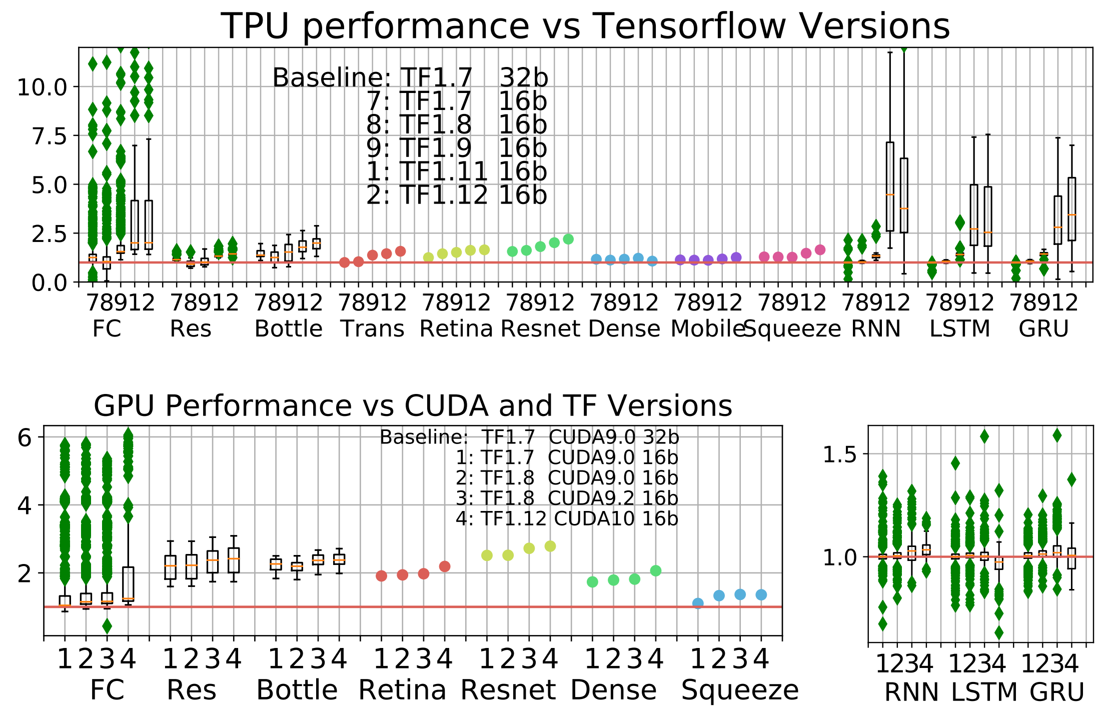


Figure 4.12: (Top) TPU performance with TensorFlow updates. (Bottom) GPU performance with CUDA and TF updates.

bitwidth reductions.

4.6.2 CUDA Versions and GPU Performance

Figure 4.12(bottom) shows GPU performance across versions of CUDA and TF. The baseline is TF 1.7 and CUDA 9.0 with float32. TF 1.8 does not improve GPU performance. By lowering memory traffic and enabling larger batch sizes, bitwidth reduction can speed up CNNs by more than 2×. We note that CUDA 9.2 speeds up ResNet-50 significantly more (8%) than other real workloads ($< 1\%$), and it speeds up ParaDnn CNNs more than FCs or RNNs. CUDA 10 speeds up other models, but not SqueezeNet. It significantly speeds up ParaDnn FCs and CNNs, but not RNNs. The overall 90th-percentile improvement for FCs, CNNs, and RNNs is up to 5.2×, 2.9×, and 8.6%, respectively.

CUDA updates have less impact than TF updates do on the TPU, likely because CUDA and GPU platforms have greatly matured since becoming popular before 2010, while TPU v2 for training was only announced in May 2017.

4.7 Limitations

SCOPE: This chapter does not study DL inference, cloud overhead, multi-node systems, accuracy, or convergence. Specifically, NVIDIA's eight-node DGX-1 or Google's 256-TPU systems are not studied. We intentionally leave these topics to future work, as each deserves in-depth study. They need different metrics such as latency, and different setups such as tuning numbers of hardware nodes, inter-node bandwidth, and synchronization mechanisms. Cloud overhead including virtualization and resource allocation may be more acute and brings up more research questions.

The validity of extrapolating training throughput to time-to-accuracy remains an open question. Recent work studied the number of training steps to accuracy as a function of batch sizes[168]. It shows that very large batch size results in sub-linear scaling, and the best batch size depends largely on the model and optimizer. In a multi-node system, synchronization becomes more complicated, which results in different convergence behavior.

TRACTABILITY: To keep the tractable, we constrain the parameters in this chapter, including the ParaDnn hyperparameters (Table 4.1) and the TPU iterations. We focus on large batches, as the platforms were designed for large batch training, and extremely small batches may lead to different conclusions. TPU performance of ParaDnn models should be considered lower bounds, because We use the RMSProp optimizer, and SGD with momentum performs faster than RMSProp. Also, the datasets in ParaDnn do not model data infeed overhead. We also limit the number of TPU iterations to 100 per loop, to allow completion of experiments given limited time and cloud resources. This means the TPU device communicates with the host every 100 training steps. To match the performance of the real models reported in the literature, the real models communicate every 1000 training steps, leading to slightly higher performance than ParaDnn, given similar hyperparameters and datasets. The performance difference is about 2% for ResNet-50.

4.8 Related Work

This chapter presents a performance analysis methodology that includes a tool, ParaDnn. The set of models generated by ParaDnn is not designed to replace other benchmark suites, but to complement existing suites to study the design space more comprehensively, as discussed in Section 4.2. DL suites that are complementary to with our methodology include CortexSuite[186], TonicSuite[84], Sirius[85], Fathom[15], DAWNBench[49], and MLPerf[151]. Benchmarks have been the driving force for compiler and architecture design for decades, and notable examples include the SPEC CPU[92] and PARSEC multiprocessor benchmarks[32]. In the same spirit as parameterized benchmarks, synthetic benchmarks are common, such as BenchMaker[110], SYMPO[71], AI Matrix[203], and[119,189,180,167,166]. Our use of DL models to compare up-to-date platforms, Google's TPU v2/v3, and NVIDIA's V100 GPU, distinguishes this chapter from previous cross-platform comparisons[172,24,122,42,88].

4.9 Summary

This chapter presents a comprehensive performance analysis methodology and its utility in deep learning. We conduct case studies to analyze and compare two generations of specialized platforms (TPU v2/v3), three heterogeneous architectures (TPU, GPU, and CPU), and two specialized software stacks (TensorFlow and CUDA). The methodology is complementary to traditional performance analysis approaches. This chapter also motivates application of our methodology to other hardware and software systems.

5

Demystifying Bayesian Inference Workloads

In this chapter, we present BayesSuite, a collection of seminal Bayesian inference workloads. We characterize BayesSuite across a variety of current-generation processors and find significant diversity. Manually tuning and deploying such workloads requires deep understanding of workloads and hardware. To address these challenges, we introduce a scheduling and optimization mechanism that can be plugged into a system scheduler. We also propose a computation elision technique that further improves the performance. Our proposed techniques provide 5.8× speedup on average.

5.1 Introduction

Recent advances in deep learning have captivated the scientific community. Systems based on neural network models have defeated world champion Go players[173], surpassed humans at image classification tasks[123], and advanced the state of the art for speech recognition[16]. However, neural networks are not the end-all solution and in many cases are not applicable. Deep learning requires massive datasets for training, is prone to overfitting, and is not conducive to reasoning about causality.

Bayesian inference is another branch of machine learning technique that complements deep learning in many ways. Bayesian inference thrives when data is limited, and its models are more interpretable, making it possible to understand how and why decisions are made. These benefits stem from the ability to combine prior knowledge with new observations.

Bayesian inference is a popular topic among machine learning researchers. Among top machine learning conferences (NIPS, ICML, and KDD), over 200 Bayesian inference papers have been published each year since 2014 and the number is steadily increasing. Notable milestones for Bayesian inference include industrial applications[18,108,149], Bayesian program learning for generalization of visual concepts from as few as one example[127], and the development of an intuitive physics engine that aids physical scene understanding[29,80].

As with deep learning, Bayesian inference models are computationally demanding, requiring attention from the hardware and systems community to improve performance and facilitate innovation. To enable systems research in Bayesian inference and to understand the architectural implications of these models, we present *BayesSuite*: a collection of seminal, representative Bayesian inference workloads. BayesSuite draws from rich application domains (ranging from economics to biology) in which Bayesian inference has been demonstrated to excel. We rigorously characterize BayesSuite on general-purpose processors found in contemporary datacenter servers. In doing so, we provide an academic understanding of the computational characteristics of a wide range of

Bayesian inference workloads, including performance bottlenecks that are amenable to optimization.

Our analysis leads to two major conclusions. First, while Bayesian inference workloads show no obvious architectural bottlenecks on single-core machines, we find that variations in the Bayesian models reveal higher sensitivity to server architecture on multicore systems. Specifically, the performance of the workloads with complex probability distribution between the observed data and the underlying features causes contention in the last-level cache (LLC). The workloads with less complicated models result in smaller working set sizes and thus tend to be more compute-bound. Leveraging these observations, we developed a scheduling and optimization mechanism that analyzes Bayesian inference jobs and automatically identifies the server configuration most likely to maximize its performance.

Second, we find that the workloads entail substantial redundant computation in the form of sampling iterations. Thus, eliding unnecessary computation through convergence detection can improve performance without reducing accuracy. We developed an intelligent mechanism that dynamically determines when to terminate a job to reduce latency and save energy without jeopardizing model accuracy.

As Bayesian inference continues to transition from academic to commercial use, proof-of-concept models need to be refined and tuned into industrial-grade code capable of performing at scale. We envision that our characterizations and proposed techniques can facilitate the deployment of Bayesian inference as a generic web service, similar to the "deep learning as a service" paradigm provided by Google Cloud Machine Learning Engine, Microsoft Azure Machine Learning, and Apache MXNet on Amazon Web Services.

This chapter makes the following contributions:

- BayesSuite: a benchmark collection of state-of-the-art Bayesian inference models for research on performance optimization by computer architects and system designers.

- A detailed characterization of the BayesSuite workloads on datacenter server architectures. We identify key bottlenecks to performance scaling and present insights on performance and energy trade-offs.

- Mechanisms that automatically provision hardware resources for specific Bayesian inference jobs in order to optimize performance and power efficiency. Across BayesSuite, we achieve an average speedup of 5.8×.

5.2 Bayesian Inference

In this section, we briefly go over the concept of Bayesian modeling and inference to familiarize readers with the algorithms. This section serves only as a primer; a thorough treatment is beyond the scope of this chapter. For more details on Bayesian inference, readers can refer to[72].

5.2.1 Bayesian Inference

Probabilistic models describe the data that could be observed from a system and use probability theory to describe the uncertainty or noise associated with the model. In supervised learning, such as deep learning, models are trained using labeled data and the uncertainty or noise is not explicitly modeled. In a situation where we do not have enough labeled data or when we are trying to create an uncertain relationship between observed data of different types, we can construct a Bayesian model and perform inference to learn what we want given some data. This process can be done by using Bayes' theorem, shown as

$$P(\vartheta|D) = \frac{P(D|\vartheta)P(\vartheta)}{P(D)}, \tag{5.1}$$

where D is the evidence, or the observed data and ϑ is the hypothesis whose probability is updated on the new data D. We refer to $P(\vartheta|D)$ as the posterior probability, the probability of a hypothesis given the observed evidence. $P(D|\vartheta)$ is the probability of observing D given ϑ and is called the likelihood. $P(\vartheta)$ and $P(D)$ are referred to as the prior probability and marginal likelihood, respectively. Because $P(D)$ does not depend on ϑ, the posterior probability of the hypothesis given evidence is proportional to its likelihood and prior probability.

Bayesian inference uses Equation (5.1) to find the posterior distribution. Analytically computing the conditional probability distribution over variables of interest becomes intractable as the number of variables increases and the complexity of the model grows. Our work focuses on large and complex models for which exact inference is impractical. Instead we use approximate inference that is tractable and still produces satisfactory results.

5.2.2 Inference Algorithm

In this section, we illustrate the Bayesian inference algorithm for computing the posterior distribution. The workloads in this chapter have different models and data, but apply the same inference algorithm. Common approximate Bayesian inference algorithms include sampling and optimization techniques. This chapter focuses on one of the sampling methods; a variant of the Hamiltonian Monte Carlo algorithm (HMC)[19], nicknamed the No-U-Turn Sampler (NUTS)[95]. NUTS autotunes the Hamilton parameters including the step size and number of steps. It is implemented in the Stan[72] probabilistic programming framework, the framework used in this chapter. We describe the more intuitive Metropolis-Hastings algorithm to illustrate the important computational characteristics, which are shared with NUTS.

ALGORITHM: Assume we have a model ϑ and observed data D. The ϑ can be replaced with an arbitrary user-defined model. The likelihood of the data given model ϑ, $P(D|\vartheta)$, and the prior probability of model ϑ, $P(\vartheta)$, are known. The goal is to estimate the posterior probability $P(\vartheta|D)$. Algorithm 1 shows a naive Metropolis-Hastings algorithm with multiple Markov chains doing sampling. A Markov chain consists of a sequence of samples, and the current sample depends on the previous one. q determines the probability of new sample ϑ' given previous sample $\vartheta(t-1)$. u is a random number drawn from a uniform distribution. In line 4 of Algorithm 1, Metropolis-Hastings draws samples from arbitrary probability distribution, often called proposal density, which results in a random-walk behavior. NUTS explores high-dimensional space by building a set of likely candidate points recursively, which eliminates random-walk behavior exhibited by the Metropolis-Hastings algorithm. In the NUTS implementation of Stan, the acceptance rate in line 5 is found by averaging acceptance probability across the entire candidate set. While each iteration of NUTS tends to be

Algorithm 1 Metropolis-Hastings Algorithm. An example of sampling posterior $P(\vartheta|D)$, given observed data D, prior $P(\vartheta)$, and likelihood $P(D|\vartheta)$.

```
1:  for chain from 1 to nchain do
2:      $\vartheta(0) \sim init$
3:      for t from 1 to n do
4:          $\vartheta' \sim q(\vartheta|\vartheta(t-1))$
5:          r = $\frac{P(\ \ ')P(D|\ \ ')}{P(\ \ (t-1))P(D|\ \ (t-1))}$
6:          $u \sim uniform(0, 1)$
7:          if $u < min\{r,\ 1\}$ then
8:              $\vartheta(t) = \vartheta'$
9:              $P(D|\vartheta(t)) = P(D|\vartheta')$
10:         else
11:             $\vartheta(t) = \vartheta(t-1)$
12:         end if
13:     end for
14:     Collect Samples
15: end for
```

more computationally expensive, it explores the target distribution much more efficiently, resulting in faster convergence.

COMPUTATION: Algorithm 1 has an outer loop over the Markov chains and an inner loop that does sampling. Each new sample is kept or discarded based on the Metropolis-Hastings rule in lines 7–12. Because the current sample depends on the previous sample, the inner loop is sequential.

There are two key computation characteristics. The first is the *sampling* in line 4, which is defined by specific models, and the computation of *acceptance rate* in line 5, which involves computations such as likelihood computation iterating over all observed data. The second characteristic is the *loop* structure. The outer loop drives the Markov chains. Its iterations are independent, so the chains can run on different cores in parallel.

OTHER ALGORITHMS: Other popular practical algorithms include variational inference, which approximates probability densities though optimization. However, these techniques do not output posterior distributions as sampling algorithms do, and do not have guarantees to be asymptotically exact. They are not as robust as sampling algorithms and need carefully crafted models and data types to avoid numerical issues, which are sometimes unavoidable. We selected NUTS as it has been widely used in the Stan community, which gives us access to a rich collection of workloads to study. We briefly discuss the performance of HMC together with NUTS in Section 5.4.1.

Table 5.1: A summary of BayesSuite workloads.

Name	*Model*	*Application*	*Cite*	*Data*
12cities	Poisson Regression	Lowering speed limits save pedestrian lives?	[23]	[3]
ad	Logistic Regression	Advertising attribution in the movie industry	[5]	[131]
ode	Friberg-Karlsson Semi-Mechanistic	Solving ordinary differential equations of non-linear systems	[140]	[27]
memory	Hierarchical Bayesian	Modeling memory retrieval in sentence comprehension	[147]	[147]
votes	Hierarchical Gaussian Processes	Forecasting presidential votes	[5]	[4]
tickets	Logistic Regression	Do police officers alter the ticket writing to match departmental targets?	[22]	[2]
disease	Logistic Regression	Measuring the continually worsening progression of Alzheimer's disease	[158]	[104]
racial	Hierarchical Bayesian	Testing for racial bias in vehicle searches	[175]	[175,8]
butterfly	Hierarchical Bayesian	Estimating butterfly species richness and accumulation	[209]	[61]
survival	Cormack-Jolly-Seber	Estimating animal survival probabilities	[117]	[1]

5.3 BayesSuite: Bayesian Inference Workloads

In this section, we present BayesSuite[*]: a Bayesian inference benchmark suite with models and datasets representing real-world use cases. We study Bayesian inference workloads developed in Stan[72]. Each BayesSuite workload consists of a model and data, both of which are fed to the NUTS inference algorithm implemented in the framework.

Workloads were selected to cover key application domains in which Bayesian inference has excelled, leveraging important models and real datasets, and to have diverse execution behaviors. The workloads are selected from StanCon 2017[5], StanCon 2018[10], Knitr[209], and BPA[117]. Below we briefly introduce the workloads. Table 5.1 summarizes the models, applications, sources, and data for each workload. We also list the corresponding publications of the workloads. If the work has not been

[*] Souce Code: https://github.com/Emma926/BayesSuite

published, we list the corresponding source.

12cities: Shows that lowering speed limits saves pedestrian lives. Uses Poisson regression on data for 12 cities obtained from FARS[3], the Fatality Analysis Reporting System maintained by the National Highway Traffic Safety Administration.

ad: Quantifies the effectiveness of various advertising channels for the movie industry. Survey data combining demographics with chosen advertising channels are fitted into a logistic regression model.

ode: Builds ordinary differential equations (ODE) to quantitatively study how drug compounds circulate in and affect the patient's body. Margossian et al. applied the Friberg-Karlsson semi-mechanistic model to this nonlinear system[140].

memory: Models the human mechanism for memory retrieval in sentence comprehension[147]. Data was collected via experiments measuring recall accuracy and latency after participants were asked to memorize words or numbers of letters. This workload implements a direct access model based on a content-addressable memory system[142].

votes: Forecasts presidential elections in all states of the US from 2020 to 2028 using historical election data from 1976 to 2016. A Gaussian process model is applied to the observed votes. Gaussian processes are very good at modeling observations over a continuous domain such as space or time.

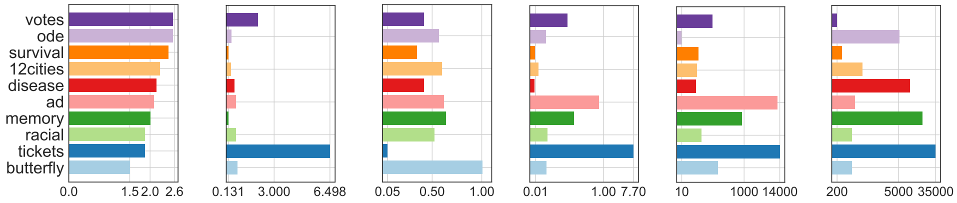
tickets: Investigates whether the New York Police Department manages officers with productivity targets, which contravenes New York state law. A generative model is proposed to describe how officers write traffic tickets. The trained model indicates that the officers alter their ticket writing substantially to match departmental targets.

disease: Models the progression of Alzheimer's disease, which is described by biomarkers and eventual loss of memory and decision-making functions. It is useful for clinical and biological purposes to understand the order of biomarkers' deterioration and their distributions for various stages of the disease. This model uses I-splines to model the monotonically increasing progression and is fitted with real patient data.

racial: Tests for racial bias in vehicle searches by police. Simoiu et al. developed a new statistical test of discrimination, the threshold test[175]. The test uses a hierarchical latent Bayesian model, and is applied to a dataset of 4.5 million police stops in North Carolina. It is found that when searching minorities, officers apply lower standards of evidence than when searching whites.

butterfly: Uses a hierarchical Bayesian model developed by Dorazio et al. to estimate butterfly species richness and accumulation[61]. Statistical estimation is necessary due to the difficulty of collecting data in grasslands with small habitat fragments in south-central Sweden, where the study was conducted. Predictions show that sample locations could be reduced by half without affecting the estimation.

survival: Cormack-Jolly-Seber (CJS) models estimate animal survival from capture-recapture data collected by capturing, tagging, and releasing animals in the population being studied. Feeding data on recapture rates into the CJS model allows survival rate to be estimated.



(a) IPC **(b)** I-Cache MPKI **(c)** Branch MPKI **(d)** LLC MPKI **(e)** Bandwidth (MB/s) **(f)** Total Run-Time (s)

Figure 5.1: Runtime statistics of BayesSuite.

5.4 Performance Analysis

In this section, to understand the execution behavior of Bayesian inference on general-purpose microprocessors, we conduct system- and architecture-level analysis of BayesSuite. Through single core performance analysis, we show that most BayesSuite workloads have higher computational efficiency than conventional sequential CPU benchmarks like SPEC CPU2006, except for some outliers suggesting possible computational bottlenecks (Section 5.4.1). By studying multicore performance, we further find that the bottleneck for BayesSuite multicore scaling is last-level cache (LLC) size (Section 5.4.2).

5.4.1 Single Core Performance

In this subsection, we study the single core performance of BayesSuite. The experiments are conducted on a modern CPU, the Intel® Core™ i7-6700K, which has four physical cores, 4.2 GHz single-thread frequency, an 8 MB last-level cache, and 34 GB/s max memory bandwidth. The performance characteristics are collected with performance counters. The sampling does not affect the results because the workloads behave consistently throughout the execution. We use RStan, the R interface of Stan, with version 2.17.3[9] and the R version is 3.4.4.

WORKLOAD DIVERSITY: The varying complexities of inference models and datasets among the workloads lead to variations in runtime behavior. Figure 5.1 shows several key dynamic characteristics of the workloads, including instructions per cycle (IPC), instruction cache misses per thousand instructions (MPKI), branch misprediction MPKI, last-level cache (LLC) MPKI, average memory bandwidth, and total execution time.

Total execution time and average bandwidth vary significantly across benchmarks, and the workloads also differ at the architecture level. For instance, the IPC of *votes* is more than 1.7× that of *butterfly*. Similar variation in the other microarchitecture characteristics demonstrate BayesSuite's diversity.

BENIGN ARCHITECTURAL BEHAVIOR: Although Bayesian inference workloads differ in absolute behavior, they tend to employ CPU microarchitecture efficiently, as suggested by the IPC values in Figure 5.1a. Instruction-level parallelism is greater in Bayesian inference workloads than in traditional sequential applications like event-driven web servers[219] and most SPEC CPU2006 benchmarks. Higher IPC results from efficient instruction supply and data feeding. Specifically, for most workloads the instruction cache MPKI (Figure 5.1b) and branch misprediction MPKI (Figure 5.1c) are several times lower than for SPEC CPU2006[105] and datacenter workloads[112]. Similarly, the LLC miss rate of Bayesian inference workloads is also insignificant, corroborated by the low bandwidth requirement (hundreds of MB/s for most of the workloads shown in Figure 5.1e, compared to tens of GB/s reached in server systems). The low data bandwidth indicates that the working set of Bayesian inference workloads can largely fit in on-chip memory.

COMPUTATION BOTTLENECKS: Although the average performance of BayesSuite is benign, there are some special cases. The i-cache and LLC MPKI of *tickets* is higher than that of other BayesSuite workloads. By studying multicore behaviors in Section 5.4.2, we will show more workloads suffering from architectural bottlenecks in details. The execution times of *tickets*, *memory*, *disease* and *ode* are much higher, which is not intrinsic but an algorithmic artifact. We will examine it in Section 5.6.

PERFORMANCE OF HMC: The single-core performance characteristics of HMC are very similar to NUTS. As a result we do not include the HMC data and focus on NUTS in the rest of this chapter. The IPC of HMC ranges from 1.5 to 2.7. The LLC MPKI of *tickets* is 8.3, and that of the other workloads is below 1 MPKI. The memory bandwidths of *ad* and *tickets* are over 12 GB/s and that of *memory* is close to 10 GB/s. The memory bandwidths of other workloads are all below 100 MB/s.

ARCHITECTURAL IMPLICATION: The benign architectural behavior of BayesSuite workloads on a modern CPU, together with the CPU's general-purpose programmability, suggests that the CPU is a good execution target for Bayesian inference. The observed cache bottleneck is the exception, which we will analyze in the next section. We will discuss GPUs and specialized hardware accelerators in Section 5.7.

5.4.2 Architectural Bottleneck

In this subsection, we analyze the performance bottlenecks of BayesSuite. We find that the multicore scalability of BayesSuite is strongly correlated with last-level cache (LLC) size.

PARALLELISM OPPORTUNITY: Sampling algorithms are inherently parallel in that the computations of different chains are independent. The `for` loop at line 1 of Algorithm 1 can be completely parallelized, providing opportunities for performance improvement using multiple cores. We sweep the number of CPU cores used while keeping 4 Markov chains as suggested in[35], and iterations as defined in the original applications.

PERFORMANCE ANALYSIS: As shown in the previous section, branch misses are minimal, the i-cache is private to each core, and memory bandwidth correlates to LLC miss rates. Therefore we focus on the LLC miss rate in this section and show the IPC, LLC MPKI, and speedup in Figure 5.2. The workloads are sorted by the LLC MPKI of 4 cores. We use speedup and IPC as performance metrics.

Speedup is typically the most important performance metric for users. Across BayesSuite, the

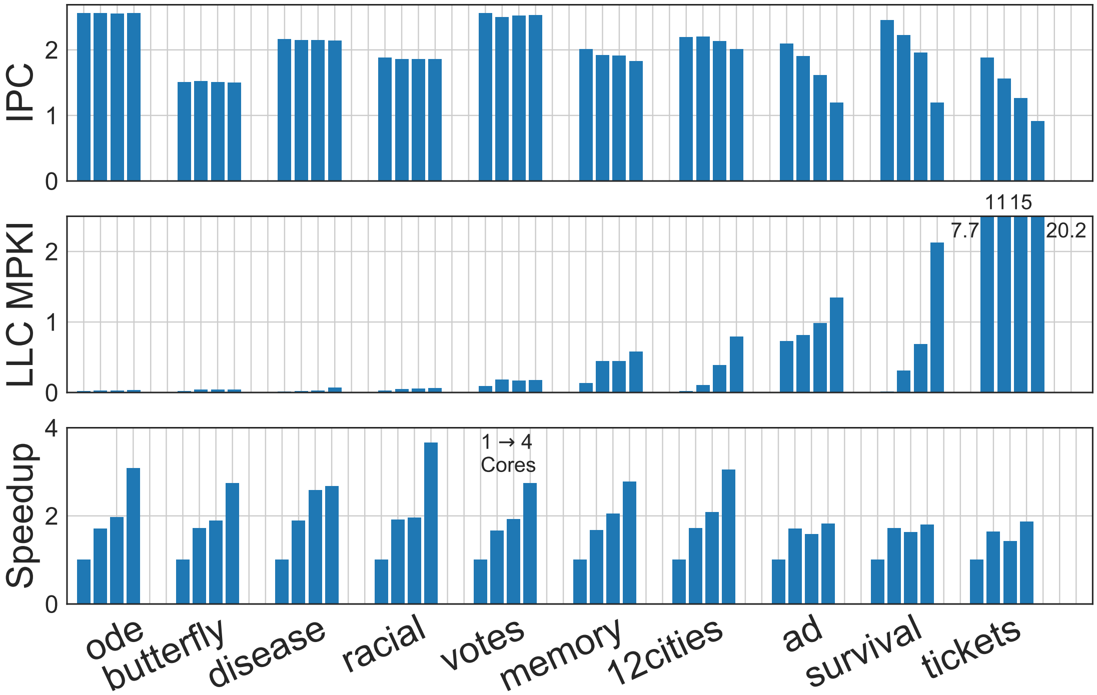


Figure 5.2: The IPC, LLC miss rates, and speedups of running the workloads on from 1 core to 4 cores of a Skylake processor.

speedup using four cores is less than 4 because the execution times of the 4 chains are not equal and the 4-core execution time depends on the slowest chain, which will be explained in Section 5.6. We observe that the performance scaling is constrained by LLC misses. This is because when using one core, the four chains are executed sequentially and only one chain needs to fit into the LLC at a time. On the other hand, when using four cores with each core executing one chain, the global working set becomes 4× larger and sometimes does not fit in the LLC. This causes more frequent accesses to off-chip memory, thereby limiting performance scalability.

In Figure 5.2, when 4-core LLC MPKI is larger than 1, the speedup does not scale linearly with the number of cores, as with *ad*, *survival* and *tickets*, whose maximum speedups are less than 2. *tickets* has especially frequent LLC misses, up to 7.7 MPKI for 1 core and 20 MPKI for 4 cores. The high

LLC miss rate leads to up to 25 GB/s memory bandwidth for the three workloads, which is not shown here.

The speedup is affected not only by LLC miss rates but also by the fact that multicore latency is constrained by the slowest chain. Thus, we also compare IPC values, which helps to see how the LLC affects the efficiency of the architecture. As the number of cores in use increases, workloads such as *memory*, *12cities*, *ad*, *survival*, and *tickets* have lower IPC and more LLC misses. Increased working set size leads to more frequent LLC misses because every chain fetches data independently. The resulting performance degradation reduces IPC.

ARCHITECTURAL IMPLICATION: LLC size is the major architectural bottleneck for BayesSuite workloads. Therefore, distinguishing workloads with large LLC needs before execution is valuable for computing resource management.

5.5 Bottleneck Resolution

In the previous section, we showed that BayesSuite workloads are constrained by LLC size, making it important to identify workloads with high LLC needs. To avoid the bottleneck, we first demonstrate our LLC miss prediction, using static indicators extracted from the model and data (Section 5.5.1). We then show that prediction can facilitate a speedup of 1.16× through scheduling of BayesSuite workloads on appropriate platforms (Section 5.5.2).

5.5.1 Last Level Cache Miss Prediction

We find 4-core LLC miss rates can be predicted using a static feature, the *modeled data* size. Modeled data are the observed data required for finding a likelihood distribution. More specifically, modeled data are used to compute the acceptance rate in line 5 of Algorithm 1. A larger modeled data size translates to more computation and possibly a larger working set size. Note that the exact sizes of modeled data (on the order of KBs) are not working set sizes (on the order of MBs), but are only proportional to them, because there are more computations and intermediate variables in the inference algorithm, such as the likelihood and the Hamiltonian computation, and the automatic tuning in NUTS.

Figure 5.3 plots 4-core LLC miss rates against corresponding modeled data sizes. Points with labels suffixed `-h` and `-q` are for runs using half and a quarter of the original modeled data, respectively. We find that modeled data sizes are positively correlated with the 4-core LLC miss rates. Particularly for workloads with LLC MPKI larger than 1, modeled data size accurately predicts LLC miss rate.

For the workloads with LLC MPKI less than 1, the correlation is weaker, so the points with y-axis less than 1 do not form a straight line. That is because when the LLC miss rate is low, it is more affected by factors such as the memory prefetcher, the design of the LLC, including its size and associativity, the structure of the cache hierarchy, and the replacement policy.

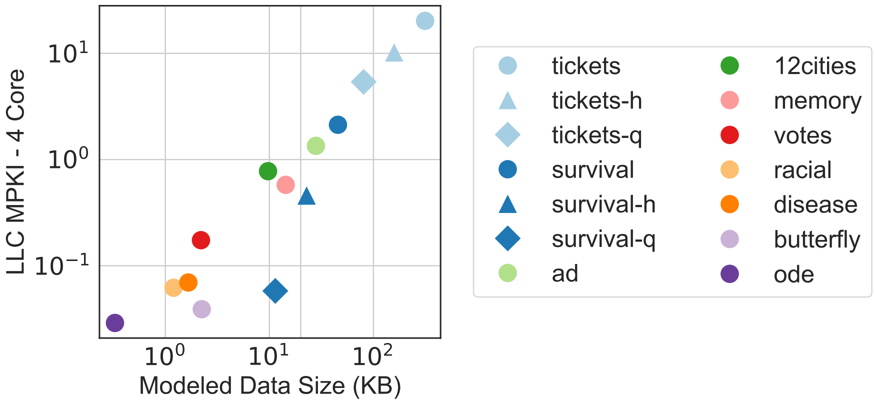


Figure 5.3: LLC miss rate prediction. Points with labels suffixed -h and -q are for runs using half and a quarter of the original modeled data, respectively.

ARCHITECTURAL IMPLICATIONS: Based on Figure 5.3, workloads with larger than 1 LLC MPKI including *tickets*, *survival*, and *ad* can be identified and predicted by setting a proper threshold for modeled data size. Resource management mechanisms can use the prediction to make better use of available computing resources. The threshold can be adjusted accordingly when applied to other machines.

5.5.2 Evaluation

In this section, we show that choosing proper platforms based on LLC miss prediction achieves 1.16× speedup in BayesSuite compared to using one platform alone.

Experimental Setup

We use two contemporary Intel CPU platforms in our evaluation: Broadwell (E5-2697A v4), and Skylake (i7-6700K), each with distinct specifications. We summarize the specifications in Table 5.2, including microarchitecture, technology, peak frequency, physical core count, LLC size, memory bandwidth, and thermal design power (TDP).

We use the Broadwell server as our baseline because it was launched in 2016, later than the Skylake machine, and it is a server-class machine and the Skylake machine is desktop-class. This work shows that better performance can be achieved by adding an older, desktop machine. The Broadwell processor has a large LLC size (40 MB) with only modest peak CPU frequency (3.6 GHz). In contrast, the Skylake processor has a high CPU frequency but small LLC size. We show that they naturally complement each other for BayesSuite workloads.

Table 5.2: A summary of experiment platforms.

Codename	Processor #	Microarch	Tech (nm)	Turbo (GHz)	Cores	LLC (MB)	Bdw (GB/s)	TDP (W)
Skylake	i7-6700K	Skylake	14	4.2	4	8	34.1	91
Broadwell	E5-2697A v4	Haswell	14	3.6	16	40	78.8	145

Performance Comparison

According to the models presented in Section 5.5.1, the LLC-bound workloads are *ad*, *survival*, and *tickets*. In order to optimize performance, we choose to run them on Broadwell for its large LLC and other workloads on Skylake. We use Broadwell as the baseline and we achieve 1.16× speedup by adding a Skylake machine.

In Figure 5.4, we compare the speedup over Broadwell, IPC, and LLC MPKI of the platforms running with 4 cores. Speedup shows the end-to-end performance. IPC shows the throughput and performance regardless of frequency. LLC miss rate is used to explain speedup and IPC differences.

Skylake outperforms Broadwell on all workloads other than *ad*, *survival*, and *tickets*. Broadwell outperforms Skylake in speedup and IPC on those three workloads because its larger (40 MB) LLC leads to lower LLC miss rates. The IPC of *memory* and *12cities* is higher on Broadwell, also due to the much lower LLC miss rates, but Skylake's high frequency gives it a slightly better overall speedup than Broadwell.

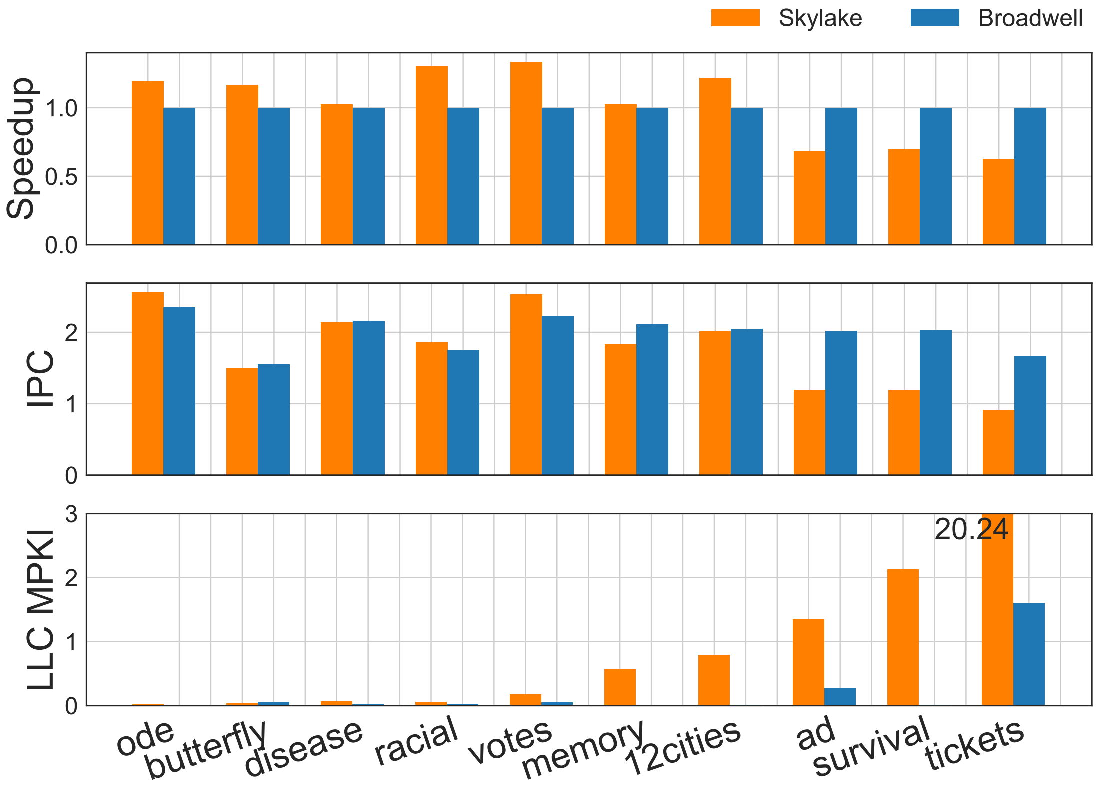


Figure 5.4: Performance comparison of the platforms.

ARCHITECTURE IMPLICATION: Following predictions by models in Section 5.5, we run *ad*, *survival*, and *tickets* on Broadwell and other workloads on Skylake. This yields an average speedup of 1.16×. LLC size and frequency are the key factors determining the performance of BayesSuite workloads.

5.6 Algorithm Convergence

Previous sections analyze the performance and architectural bottlenecks of BayesSuite. In this section, we study algorithmic aspects, the convergence and result quality of BayesSuite benchmarks. The number of sampling iterations (line 3 in Algorithm 1) is selected by users, and we find that all BayesSuite workloads have redundant iterations. We propose runtime convergence detection for users who want quick inference results with minimal overhead (Section 5.6.1). We then show that with convergence detection, BayesSuite workloads reach better design points and save 70% energy, on average. Overall, we speed up BayesSuite by 5.8× with convergence detection and LLC miss prediction (Section 5.6.2).

5.6.1 Convergence Study

Convergence detection is closely related to the number of chains used. Multiple chains prevent converging to local optima, and complex models prefer more chains. Convergence detection is based on the Gelman-Rubin diagnostic ($\hat{R}$)[73], which quantifies sample variations within and between chains. A smaller $\hat{R}$ indicates more consistent samples, and when $\hat{R}$ reaches 1, chains have converged completely. As suggested by Brooks et al.[35], we typically use 4 chains. Because several iterations are needed to warm up the chains, we only use the second half of the samples for inferring the posterior distribution[35]. A value of $\hat{R}$ less than 1.1 is taken as indicating convergence[35].

We study the convergence process and confirm that when using multiple chains, once $\hat{R}$ is less than 1.1, the results (posterior distributions) have good quality. To judge quality, we estimate the ground truth by running each benchmark with twice as many iterations as were initially specified by the model developer. To evaluate the intermediate results, we compute the KL divergence (a measure of how much one distribution diverges from another[93]) between intermediate results and the ground truth. A smaller KL divergence indicates that the result is closer to the ground truth.

We conduct a convergence study for BayesSuite and find that on average, the workloads have over 70% excess iterations. As an example, we show the convergence of *12cities* in Figure 5.5. The blue line is $\hat{R}$ and the green line shows KL divergence. With more iterations, the KL divergence decreases monotonically, showing that the results are getting closer to the ground truth. The trace of $\hat{R}$ fluctuates because the four independent chains are exploring different regions of the space. When they are sampling from the same region, $\hat{R}$ gets small; when one chain happens to jump out of that region, $\hat{R}$ increases. At the 600th iteration (orange dots), $\hat{R}$ is less than 1.1 for the first time and the KL divergence is minimal, indicating that the results are close to the ground truth. The original number of iterations of *12cities* is 2000. We find it converges after 600 iterations, eliminating 70% of the sampling iterations as unnecessary.

Reducing excess iterations can increase performance, but the iteration reduction percentage does not directly translate to latency saving. That is because the time required to auto-tune Hamiltonian parameters in NUTS may vary depending on the position of the Markov chain. Within a single chain, the latency per iteration is smaller after the chain converges, and different chains have different latencies. When multiple chains run in parallel, the overall latency is constrained by the slowest chain. For example, for *12cities* with 4 chains of 2000 iterations, the latency ratio of the slowest to the fastest chain is 1.7. The latency of 2000 iterations is 865 seconds and that of 600 iterations is 401 seconds, thus the latency is reduced by 53%. For the same reason, we should expect the average latency savings to be less than the iteration reductions.

RUNTIME CONVERGENCE DETECTION: Detecting convergence at runtime can be implemented by dynamically computing $\hat{R}$ in the framework, such as Stan in this case. Instead of executing a preset number of iterations, as in line 3 of Algorithm 1, the workload exits each iteration when it is determined to have converged. This detection is optional, for statisticians whose interests are in developing new models and would like to choose the number of iterations and test the model. But convergence detection can give quick results for those interested in using existing models with their

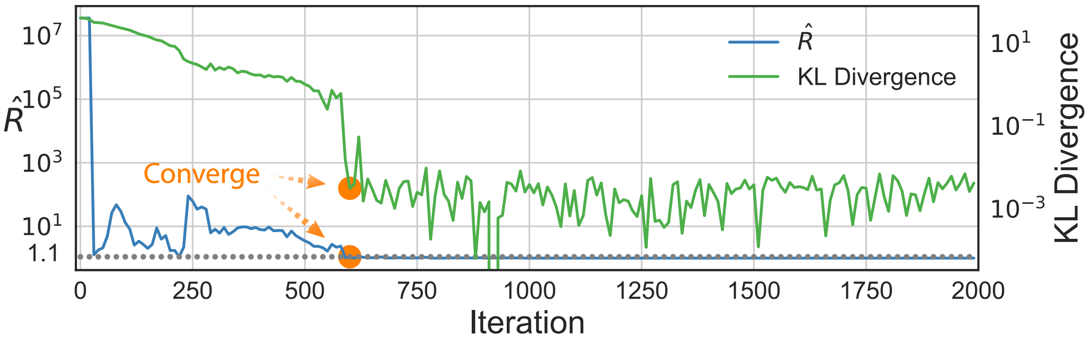


Figure 5.5: The convergence process of *12cities* in log scale. The blue line is $\hat{R}$, for detecting convergence. The green line is the KL divergence between the current result and ground truth. The orange dots mark the convergence point.

own data.

OVERHEAD ANALYSIS: We implement the computation of $\hat{R}$ in C++, based on the algorithm in[73]. We consider the worst case by taking the maximum number of iterations in BayesSuite (2000) and half of the samples for inference (i.e., 1000 data points of 4 chains). That takes 0.06 seconds on a single core of Skylake, which is minimal. In reality, optimizations can be applied to reduce the overhead.

ARCHITECTURAL IMPLICATION: By our analysis, the overhead of convergence detection is minimal, and the savings are huge.

5.6.2 Design Space Exploration

We evaluate the convergence detection mechanism using design-space exploration (DSE) techniques and compare the optimization decision with other (suboptimal) design points in terms of latency and power savings.

Our DSE setup considers three major parameters: the number of CPU cores, the number of chains, and the number of iterations. We show the design space of 4 representative workloads on Skylake as a case study in Figure 5.6. The blue stars are original user settings. The triangle markers

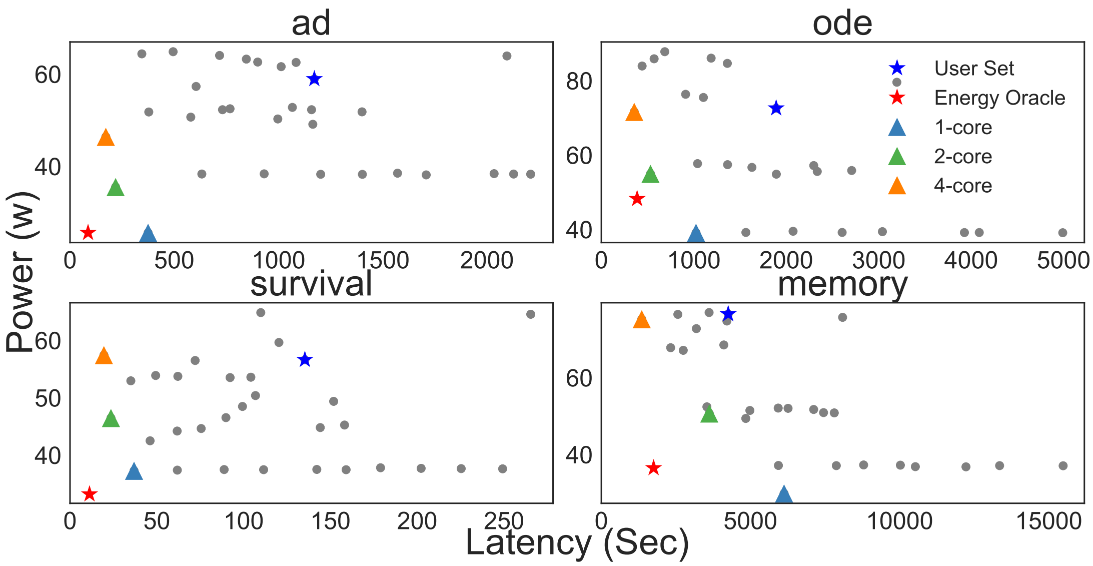


Figure 5.6: Design space exploration examples.

denote the design points that are achievable with convergence detection under 1, 2, and 4 cores. We refer to the design point that has the lowest energy consumption as the *energy oracle*, as denoted by red stars in the figure.

We find that the energy oracle design points always use only 1 or 2 chains and a small number of iterations while user-specific settings always use 4 chains with a much higher number of iterations. Although they have small KL divergence, without knowing the ground truth a priori, it is infeasible to use only 1 or 2 chains; hence, the oracle.

ENERGY SAVINGS: The triangles that we choose are much closer to the red stars than the original user settings. When applying this technique, a scheduler can be programmed to choose one of the triangles to optimize power or latency. In this section, we choose to use energy as the cost function, considering both latency and power.

We summarize our energy savings for 10 workloads on two platforms in Figure 5.7. The savings are in percentage relative to the original user settings. The average energy saving is 70% across 2 plat-

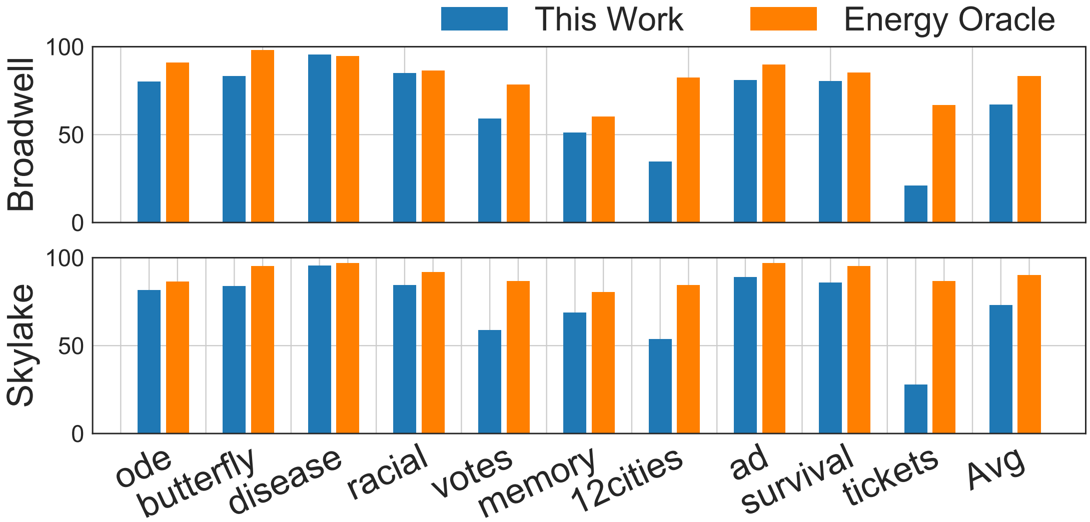


Figure 5.7: A summary of energy savings of our design points and the energy oracle.

forms and 10 workloads.

ARCHITECTURAL IMPLICATION: BayesSuite workloads reach better design points with convergence detection.

5.6.3 Overall Speedup

We present the overall speedup resulting from the techniques in this chapter: convergence detection in Section 5.6.1 and selecting the best platform in Section 5.5. In Figure 5.8 we show the overall speedups of BayesSuite over the baseline. *ad*, *survival*, and *tickets* are on Broadwell and the rest of the workloads are on Skylake. *ode* and *memory* achieve higher speedups than the energy oracle, which can be explained using Figure 5.6. *ode* and *memory* use 4 cores (orange triangles) on Skylake, and have lower latency than the red stars, the oracle points. The same explanation applies to *disease*.

With the techniques proposed, the average speedup of BayesSuite is 5.8×, and the oracle average speedup is 6.2×, over the baseline with no convergence detection running on the Broadwell server.

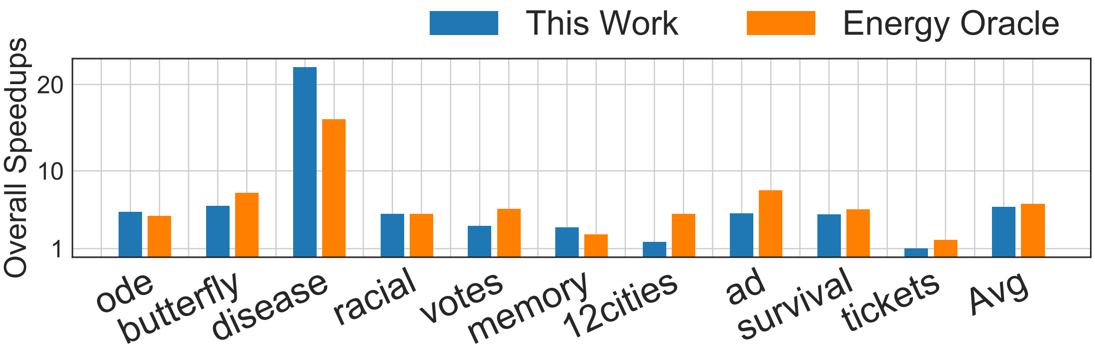


Figure 5.8: Overall speedup of the techniques proposed in this chapter. Note that the oracle points are with respect to energy, not performance.

5.7 Implications for Future Acceleration

Intelligent scheduling on today's server processors can readily provide performance improvement for Bayesian inference jobs. Pushing the efficiency a step further requires us to apply hardware specialization. Previous work on hardware specialization only focuses on a specific type of model. In this section, based on the insights that we gained from analyzing and improving workload efficiency on CPUs, we examine opportunities to accelerate Bayesian inference using specialized hardware such as GPUs, FPGAs, and ASICs, in preparation for an accelerator-centric system to further speed up Bayesian inference workloads.

We first discuss the choice of different acceleration approaches. We argue that a programmable SIMD architecture augmented with special functional units is a good accelerator style that matches well with a wide range of Bayesian inference workloads (Section 5.7.1). We then discuss the memory system requirements on LLC and i-cache (Section 5.7.2).

5.7.1 Hardware Choice

The first and foremost question is which accelerator style, e.g., single instruction multiple data (SIMD) or coarse-grained reconfigurable architecture (CGRA), is a good fit for Bayesian inference workloads. We find that these workloads exhibit both coarse-grained and fine-grained parallelism.

CHAIN-LEVEL PARALLELISM: Both coarse-grained and fine-grained parallelism exist in sampling algorithms. Coarse-grained parallelism typically manifests at the chain level (line 1 in Algorithm 1), which can be captured by a multicore CPU as we discussed in Section 5.4.2. To better exploit the chain-level parallelism, the key is to address the LLC bottleneck inhibiting CPU core scaling shown in Figure 5.2.

COMPUTATION PARALLELISM: Within a chain, there are opportunities for parallelism in the computation. For example, in the acceptance rate computation iterating through a series of observed data (line 5 in Algorithm 1), the computation of each observed data point can be conducted in parallel. At a finer grained level, BayesSuite workloads contain a diverse collection of vector and matrix operations beyond matrix multiplication, indicating the importance of architectural support for such operations. Therefore the workloads can benefit from the parallelism of SIMD-style hardware like GPUs.

VARIABLE SAMPLING PARALLELISM: The sampling of variables (line 4 in Algorithm 1) provides parallelism opportunities as well, which can benefit from SIMD hardware or specialized accelerators. When presenting the models as graphs, in which the model variables are nodes and variable dependencies are edges, the variables at the same layer can be sampled in parallel. Previous work on FPGAs and ASICs exploits the parallelism[109,69]. With multiple sampling units on chip, the sampling latency can be shortened.

It is beneficial to implement the sampling units as accelerators. The current implementation of sample drawing needs cumulative distribution functions (CDFs) of corresponding distributions. Thus the implementation depends on individual distributions. We study the distributions in BayesSuite and find the most popular distributions are Gaussian and Cauchy. It is worth building accelerators for the most popular distributions. The CDFs use functions with values stored in lookup tables, such as the error function *erf* (Gaussian) and arctangent function *atan* (Cauchy), which introduces overhead to the system and trades off precision for efficiency. Sampling accelerators can help to reduce system overheads by having their own scratchpad memory or private cache.

PARALLELISM IN OTHER ALGORITHMS: Finally, we note that different inference algorithms exhibit different opportunities for parallelism. For instance, sampling algorithms such as the one studied in this chapter are general but sequential. Exact inference has more parallelism but the complexity is exponential. Some recent work combines sampling and exact inference to get the parallelism of exact inference and the linear complexity of sampling [138,60,160]. The general idea is to do exact inference with a subset of the data within the MCMC sampling iterations. Such algorithms, exposing ample parallelism, are promising to explore in future work.

NEED FOR PROGRAMMABILITY: As we discussed, the workloads have very diverse models, requiring different combinations of matrix, vector, and scalar operations, as well as different preferences for distributions. Thus, to accelerate Bayesian inference workloads, we need to program the models.

5.7.2 Memory System

To reduce the overhead of transferring data between LLC and main memory, the LLC should be large enough to contain the whole working set. According to results in Figure 5.4, an LLC of 2 MB per core (8 MB / 4 cores on Skylake) is large enough for workloads other than *ad*, *survival*, and *tickets*. An LLC smaller than 10 MB per core (40 MB / 4 cores on Broadwell) is enough to hold *ad* and *survival*. Workloads like *tickets* need larger LLC. However, with larger datasets applied to Bayesian models, simply scaling up the LLC is not the solution. Instead, the inference algorithm should be tuned to subsample the data such that the working set fits the LLC. Figure 5.3 can be used to estimate the proper sub-sampled data size.

In addition to LLC size, a 32 KB i-cache constrains the performance of *tickets*, as shown in Figure 5.1, and leads to high LLC miss rates in Figure 5.4. Thus the hardware needs i-caches larger than 32 KB to better serve workloads like *tickets*.

5.8 Related Work

In this section, we compare and summarize the previous work related to Bayesian models, inference algorithms, hardware advancement of Bayesian inference, probabilistic programming, and workload characterization.

BAYESIAN MODELS: In addition to the BayesSuite workloads, Bayesian inference has been applied to image classification[127,165], semantics analysis[77], language learning[210], program synthesis[127,66,159], intuitive physics[80,29,28,202], and structure learning[90,116,184]. A Bayesian approach that is sometimes called Bayesian neural networks (BNN)[146] is being applied to deep learning to learn weight distributions. These models are known to achieve better results by using optimization techniques, rather than more general and easy-to-use approaches like NUTS. It is important to note that models can have varying results, depending on the inference algorithm used.

INFERENCE ALGORITHMS: Exact inference is often intractable as it has exponential complexity, while sampling is linear with regard to the number of samples[139]. This paper focuses on NUTS, a variant of Hamiltonian Monte Carlo that requires no hand-tuning of step size and number of steps[95]. It is a turnkey sampling method that can be used in a black-box fashion and has been adapted by Stan as the default inference engine. Other sampling algorithms include Gibbs sampler, Hamiltonian Monte Carlo, slice sampling, sequential Monte Carlo, and particle Markov chain Monte Carlo[19]. Variational inference is an optimization algorithm that tends to be fast but has no guarantee on convergence to global optima[195].

We discussed the combination of sampling and exact inference in Section 5.7.1[138,60,160]. Those hybrid algorithms have the potential to benefit from parallel hardware such as GPUs and we plan to explore them in future work.

HARDWARE ADVANCEMENTS: Efforts have also been made to speed up Bayesian inference with specialized hardware. The BAMBI project[†] proposed hardware implementations of probabilistic computations[69,38,50]. Mansinghka et al. have built stochastic circuits and evaluated them with Markov Random Fields and the Dirichlet Process Mixture Model[139,109]. Some of the acceleration work has been done as parallel implementations on GPUs[216,70,198] and scalable CPUs[144,207]. Furthermore, there are FPGA and ASIC implementations accelerating BNNs[36] and Markov Random Fields on perceptual applications[120,121]. [196] uses a novel device to implement the sampling procedure.

These projects primarily focus on speeding up a specific type of model or application and often require substantial effort for programming GPUs or designing the hardware. Our work is the first in the literature to introduce Bayesian inference to the architecture community as a key machine learning technique and to reveal computational bottlenecks across a suite of benchmarks on commodity datacenter CPUs.

PROBABILISTIC PROGRAMMING: Some probabilistic programming frameworks focus mostly on language design and expressiveness[139], and some provide efficient sampling for a certain subset of models[152,94,134]. Infer.NET focuses on variational approximation[197]. TensorFlow Probability[11] and Edward[188] support distributed and parallel training on top of TensorFlow, and Pyro[7] is based on PyTorch. We choose Stan mainly because of its unmatched popularity in the science community.

WORKLOAD CHARACTERIZATION: People develop benchmark suites to facilitate the advancement of architecture[92,161,33] or to introduce important workloads[15,194,186], as BayesSuite does. We use static workload features to estimate dynamic characteristics, similar to[58], and we are different by focusing on a key bottleneck of Bayesian inference. Therefore our static analysis has minimal overhead.

[†] Bottom-up Approaches to Machines dedicated to Bayesian Inference: www.bambi-fet.eu

5.9 Summary

The advances enabled by deep learning have overshadowed other aspects of the vast field of machine learning. Bayesian inference is a particularly important machine learning technique that complements deep learning in many domains. This paper introduces BayesSuite, a suite of Bayesian inference workloads to help bridge the gap between Bayesian inference researchers and computer architects. Through detailed characterization, we show that these workloads exhibit diverse behaviors that call for a variety of processor architectures. They also entail inherent redundancy that leads to execution inefficiency. We propose a scheduling mechanism and a computation elision technique to automatically speed up Bayesian inference workloads. Experiments and evaluations conducted on real systems show that we speed up BayesSuite workloads by 5.8× on average.

'Since when,' he asked,

'Are the first and last line of any poem

Where the poem begins and ends?'

Seamus Heaney

6

Future Directions

Performance analysis has been driving software and hardware advancements for decades. The surge of machine learning applications and specialized software and hardware systems brings performance analysis extra challenges. This dissertation is one of the first efforts in the computer architecture and machine learning community to systematically analyze software and hardware designs for machine learning. We first analyzed the repositories of traditional CPU benchmark suites, SPEC and Geekbench, and showed the maximum information available from and limitations of such suites. We then dived into deep learning, and thoroughly studied the performance impact of key framework features, and proposed a set of simple guidelines to achieve higher training and inference speedup. To further assist performance analysis and extract deeper insights, we proposed a systematic analysis methodology that can be applied to various hardware and software systems. We demonstrated the utility of this methodology in deep learning, by analyzing specialized platforms (TPU v2/v3), heterogeneous platforms (TPU, GPU and CPU), and software stacks (TensorFlow and CUDA). As the first step towards understanding emerging applications, we analyzed Bayesian inference workloads and proposed several mechanisms to improve the performance.

As a popular domain, deep learning performance analysis is crucial and needs to be conducted regularly. Meanwhile, architects need to be aware of emerging workloads. Here we summarize several future research directions.

6.1 Deep Learning

For deep learning performance analysis, future research directions can be extended from workloads, software and hardware dimensions. The three analysis methods presented in the thesis can be applied to those directions to extract different insights.

6.1.1 workload

At workload level, new model types can be studied by releasing and analyzing their state-of-the-art implementations, key operators, and parameterized models. For example, recommendation models with large embedding tables open up many research opportunities. Its importance in production and memory intensiveness makes it attract much attention recently[65,51,145,79]. Gupta et al.[79] profile Facebook's recommendation models and release a synthetic model with parameterized attributes. More studies can be done to explore model design space and stress system limits, similar to Chapter 4. And there may be a range of model attributes that work better on a given system than others.

The parameterized analysis in Chapter 4 focuses on training workloads. Inference workloads may provide different insights because batch sizes are usually much smaller and only activations need to fit in on-chip memory. In such cases, latency, rather than throughput, would be the most crucial metric; network latency would be another system knob that researchers can experiment on; virtualization overhead may not be a big issue for large-batch training, but it may be important for inference; memory capacity may be less important for cross comparison.

6.1.2 Software

At software level, systematic performance analysis shall be conduct at more aspects to make better trade-offs in specific scenarios. For compilation styles, just-in-time (JIT) and ahead-of-time (AOT) are two major choices. JIT compiles and interprets scripts dynamically, which therefore incurs some runtime overhead, while AOT does that statically. Since having the information of a whole computational graph, AOT may enable better optimizations, while JIT may only optimize for local subgraphs. On the other hand, if deploying to edge devices, AOT generates one binary file for one model, while JIT can interpret all scripts with one interpreter.

As one of compilation optimizations, operator fusion is crucial to reduce memory traffic and improve performance. Current fusion techniques are mainly based on heuristics learned from optimizing current workloads, such as ResNet-50. Operator fusion may benefit from parameterized analysis methodology for systematic studies, with properly constructed parameterized workloads. Systematic analysis can enable more general optimization guidelines.

Frameworks with opaque (implemented) operators are studied in Chapter 3. Other framework types provide research opportunities, such as the ones automatically generate optimized operators including Tensor Comprehension[192] and TVM[45]. Such frameworks have unique design features, leading to different performance implications, and therefore they need to be studied in-depth.

6.1.3 hardware

At system and architecture level, analysis can be performed for more platforms, such as multi-node systems, edge devices, and latest AI hardware platforms. Frameworks of more hardware platforms can also be studied, such as that of the GPU and TPU, similarly to Chapter 3 analyzing CPU frameworks. For most compute-intensive FC and CNN models, Figure 5.8 shows TPU's FLOPS utilization is lower than 60% and GPU's is lower than 20%. The fact indicates that there seems to be large fractions of overhead, possibly originating from data preparation or synchronization from and to vector/matrix units. Detailed analysis is needed to reveal root causes and approach solutions. Further, such analysis may be conducted for software stacks of other novel hardware platforms.

6.2 Bayesian Inference

This dissertation presents the first analysis research for Bayesian workloads (Chapter 5). There are plenty of research opportunities for performance analysis of Bayesian inference. We discuss from workloads, software and hardware perspectives.

6.2.1 Workloads

The models presented in Chapter 5 are small enough to run on one CPU chip. Usually larger models are used to solve large-scale problems with a cluster of CPUs or GPUs, and they especially need performance analysis and optimizations. Thus the next important step from my perspective is the systematic analysis for large-scale Bayesian workloads that solves real problems such as document classification or gene sequencing. To study large-scale workloads, parameterized methodology can be applied to complement existing workloads and stress system limits, which involves the development of parameterized Bayesian benchmarks.

Latent Dirichlet Allocation (LDA) is one example of such models. It is usually used for document classification with large datasets such as New York Times and Wikipedia. As an important type of model, many algorithmic and architectural efforts have been made for its optimization. It is sometimes a trade-off to choose among LDA algorithms and platforms. Here we present the preliminary results of ongoing research in our group to motivate further studies.

Figure 6.1 shows the per-core runtime characteristics of five LDA algorithms on a four-core CPU. From the earliest publication to the latest, the five algorithms are a simple LDA[76], sparse LDA[212], alias LDA[133], light LDA[215], and FTree LDA[214]. Each algorithm (workload) has two types of threads, sampling threads and updater threads. In Figure 6.1, we use one updater thread, and one and three sampling threads, which corresponds to the left two bars, and right four bars for each workload. The updater threads are the shaded bars in Figure 6.1(a) and orange bars in Figure 6.1(b) and (c). We conduct top-down analysis[213], and show the FLOPS and LLC miss rates.

The top-level analysis in Figure 6.1(a) shows that the workloads on the right have more instruction retiring cycles, and less backend-bound cycles. The FLOPS in Figure 6.1(b) shows that the workloads with higher instruction retiring ratios have higher FLOPS. The LLC MPKI in Figure 6.1(c) shows that the workloads with lower backend-bound ratios have lower LLC miss rates. To summarize, the results show that more sophisticated algorithms have more architectural bottlenecks. On the other hand, more sophisticated ones are developed for faster convergence, e.g., FTree claims to sample with logarithm complexity[214]. This example of LDA shows an interesting trade-off of convergence speed and architectural bottlenecks.

6.2.2 Software

Bayesian frameworks contain inference engines, and support expression of generative models. In Chapter 5 we focused on Stan, because it provides more workloads than other frameworks. With the adoption of Bayesian techniques in deep learning research, more and more Bayesian frameworks are extended from popular deep learning frameworks, such as TensorFlow probability[11], and Pyro[7] (from Pytorch). Deep learning frameworks are not at a stable stage yet, and Bayesian frameworks are even further from being stable. Therefore architects need to stay tuned, while be willing to take risks by devoting time and efforts to working on immature frameworks.

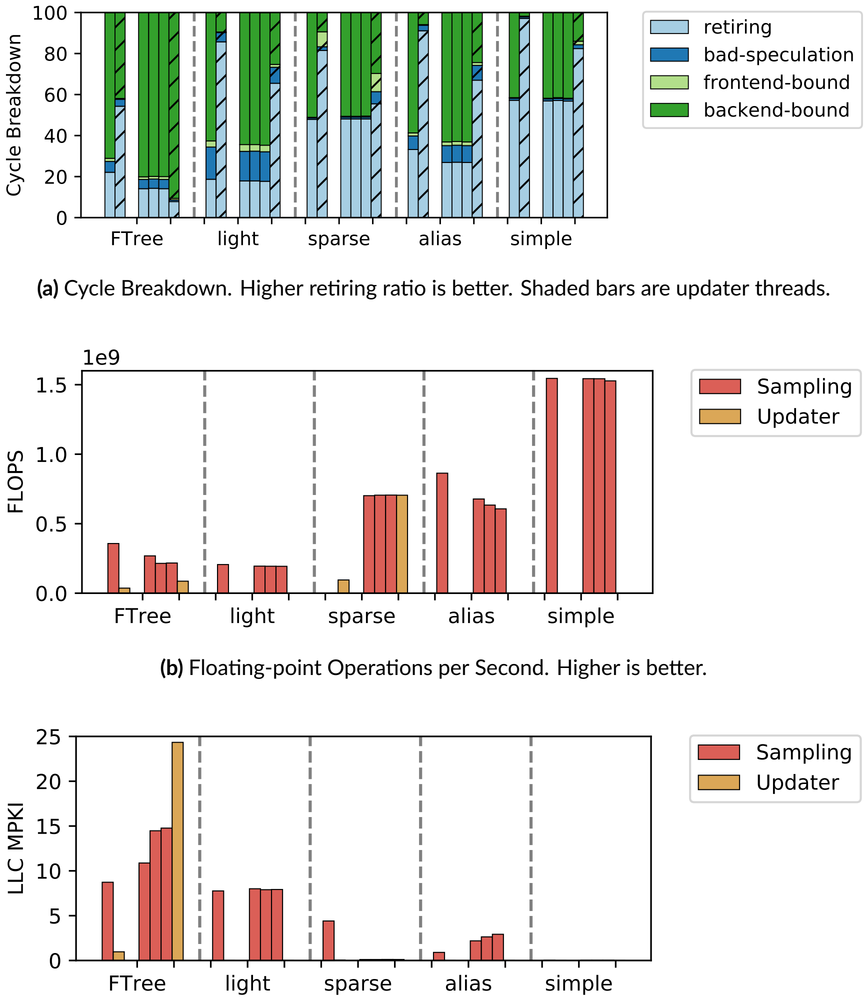
6.2.3 Hardware

To improve the performance of an emerging workload, the most efficient way is to take advantage of existing systems. That is why we started the analysis research on CPUs in Chapter 5. Larger CPU systems, multi-node CPUs, and warehouse-scale computers are the next promising playground.

Another exciting research direction is generic Bayesian frameworks implemented for GPUs. The Stan community has been discussing and implementing Stan for GPU since 2017 [*]. Implementing specific Bayesian models on GPUs is not a new topic, but providing a generic GPU framework for a variety of workloads is very complicated. It involves many design choices including libraries, floating-point precisions, levels of parallelism, and so on. Similar research and implementations can also be done with other frameworks such as TensorFlow Probability and Pyro.

Accelerator or system-on-chip (SoC) can be designed after getting a thorough understanding of Bayesian workloads. Conventional wisdom is to accelerate random number generators and vector and matrix operations. On the other hand, we believe the acceleration of new workloads needs to be conducted across the stack, i.e., we need to pick suitable inference algorithms for specific architectures. Just like deep learning took off again because of imageNet dataset[59], stochastic gradient descent (SGD), and GPU. As of now, Bayesian inference may be waiting for the best combination of algorithm and hardware platform.

[*] https://discourse.mc-stan.org/t/stan-on-the-gpu/326/10



(a) Cycle Breakdown. Higher retiring ratio is better. Shaded bars are updater threads.

(b) Floating-point Operations per Second. Higher is better.

(c) LLC Misses per Kilo Instructions. Lower is better.

Figure 6.1: The runtime characteristics of five LDA algorithms, using one and three sampling threads, and one updater thread.

References

[1] (2011). *Complete code and data files of book "Bayesian population analysis using WinBUGS"*. http://www.vogelwarte.ch/de/projekte/publikationen/bpa/complete-code-and-data-files-of-the-book.html.

[2] (2015). Parking or moving violation tickets in New York City between january 2014 and december 2015. https://raw.githubusercontent.com/jauerbach/Seventy-Seven-Precincts/master/data/tickets.csv.zip.

[3] (2017). Fatality analysis reporting system. https://www.nhtsa.gov/research-data/fatality-analysis-reporting-system-fars.

[4] (2017). Historical presidential election data (1976-2016). https://github.com/stan-dev/stancon_talks/blob/master/2017/Contributed-Talks/08_trangucci/data_pres_forecast/pres_vote_historical.RDS.

[5] (2017). Stancon 2017. http://mc-stan.org/events/stancon.

[6] (2017). TensorFlow optimizations on modern Intel® architecture. https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture.

[7] (2017). Uber AI labs open sources Pyro, a deep probabilistic programming language. http://eng.uber.com/pyro/.

[8] (2018). A dataset of 4.5 million stops conducted by the 100 largest local police departments in North Carolina. https://github.com/stan-dev/stancon_talks/blob/master/2018/Contributed-Talks/11_simoiu/north_carolina.RData.

[9] (2018a). *Modeling Language User's Guide and Reference Manual, Version 2.17.0*. http://mc-stan.org/users/documentation/.

[10] (2018b). Stancon 2018. http://mc-stan.org/events/stancon2018/.

[11] (2018). TensorFlow probability. https://www.tensorflow.org/probability/.

[12] (2018). TensorFlow: Using JIT compilation. https://www.tensorflow.org/xla/jit.

[13] (2019). Folly: Facebook open-source library. https://github.com/facebook/folly.

[14] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., & Zheng, X. (2016). TensorFlow: a system for large-scale machine learning. In *OSDI*, volume 16 (pp. 265–283).

[15] Adolf, R., Rama, S., Reagen, B., Wei, G.-Y., & Brooks, D. (2016). Fathom: reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on* (pp. 1–10).: IEEE.

[16] Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. (2016). Deep speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning* (pp. 173–182).

[17] Amodei, D. & Hernandez, D. (2018). AI and compute. *OpenAI Blog*.

[18] Analytics, R. (2014). Google uses R to calculate ROI on advertising campaigns. http://blog.revolutionanalytics.com/2014/09/google-uses-r-to-calculate-roi-on-advertising-campaigns.html.

[19] Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine learning*, 50(1-2), 5–43.

[20] Anju, P. (2018). Tips to improve performance for popular deep learning frameworks on CPUs. *Intel Developer Zone*.

[21] Ardalani, N., Lestourgeon, C., Sankaralingam, K., & Zhu, X. (2015). Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture* (pp. 725–737).: ACM.

[22] Auerbach, J. (2017). Are New York City drivers more likely to get a ticket at the end of the month? *Significance*, 14(4), 20–25.

[23] Auerbach, J., Eshleman, C., & Trangucci, R. (2017). A hierarchical model to evaluate policies for reducing vehicle speed in major American cities. *arXiv preprint arXiv:1705.10876*.

[24] Bahrampour, S., Ramakrishnan, N., Schott, L., & Shah, M. (2016). Comparative study of Caffe, Neon, Theano, and Torch for deep learning. In *ICLR*.

[25] Banner, R., Hubara, I., Hoffer, E., & Soudry, D. (2018). Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046*.

[26] Barham, P. & Isard, M. (2019). Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS 19)* (pp. 177–183).: ACM.

[27] Baron, K. T., Gastonguay, M. R., Bioavailability, A., SS, I., & ADDL, M. (2015). Simulation from ODE-based population PK/PD and systems pharmacology models in R with mrgsolve. *Omega*, 2, 1x1.

[28] Bates, C., Battaglia, P., Yildirim, I., & Tenenbaum, J. B. (2015). Humans predict liquid dynamics using probabilistic simulation. In *CogSci*.

[29] Battaglia, P. W., Hamrick, J. B., & Tenenbaum, J. B. (2013). Simulation as an engine of physical scene understanding. *Proceedings of the National Academy of Sciences*, 110(45), 18327–18332.

[30] Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1), 65–98.

[31] Bhuiyan, A., Abuzaina, M., Hasabnis, N., Ammbashankar, N., Amin, F., Fu, S., & Subramanian, B. (2019). Improving TensorFlow inference performance on Intel Xeon processors. *Intel AI Blog*.

[32] Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (pp. 72–81).: ACM.

[33] Bienia, C. & Li, K. (2009). PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[34] Blog, G. A. (2019). Introducing GPipe, an open source library for efficiently training large-scale neural network models. https://ai.googleblog.com/2019/03/introducing-gpipe-open-source-library.html.

[35] Brooks, S., Gelman, A., Jones, G., & Meng, X.-L. (2011). *Handbook of Markov chain Monte Carlo*. CRC press.

[36] Cai, R., Ren, A., Liu, N., Ding, C., Wang, L., Qian, X., Pedram, M., & Wang, Y. (2018). VIBNN: Hardware acceleration of Bayesian neural networks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 476–488).: ACM.

[37] Campanoni, S., Brownell, K., Kanev, S., Jones, T. M., Wei, G.-Y., & Brooks, D. (2014). Helix-rc: an architecture-compiler co-design for automatic parallelization of irregular programs. *ACM SIGARCH Computer Architecture News*, 42(3), 217–228.

[38] Canillas, R., Laurent, R., Faix, M., Vaufreydaz, D., & Mazer, E. (2016). Autonomous robot controller using bitwise Gibbs sampling. In *The 15th IEEE International Conference on Cognitive Informatics and Cognitive Computing. IEEE.*

[39] Case, L. (2018). Volta Tensor Core GPU achieves new AI performance milestones. *Nvidia Developer Blog*.

[40] Cerabras (2019). Cerebras wafer scale engine: An introduction.

[41] Chao, C. & Saeta, B. (2019). Cloud tpu: Codesigning architecture and infrastructure. *Hot Chips*.

[42] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)* (pp. 44–54).: Ieee.

[43] Chen, T., Chen, Y., Duranton, M., Guo, Q., Hashmi, A., Lipasti, M., Nere, A., Qiu, S., Sebag, M., & Temam, O. (2012). BenchNN: On the broad potential application scope of hardware neural network accelerators. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on* (pp. 36–45).: IEEE.

[44] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., & Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

[45] Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). TVM: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*.

[46] Chen, X. E. & Aamodt, T. M. (2011). Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3), 10.

[47] Cheng, H.-T. (2016). Wide and deep learning: Better together with TensorFlow. *Google AI Blog*. https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html.

[48] Cherkassky, V. & Ma, Y. (2004). Comparison of loss functions for linear regression. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 1 (pp. 395–400).: IEEE.

[49] Coleman, C., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., & Zaharia, M. (2017). DAWNBench: An end-to-end deep learning benchmark and competition. *Training*, 100(101), 102.

[50] Coninx, A., Bessière, P., & Droulez, J. (2017). Quick and energy-efficient Bayesian computing of binocular disparity using stochastic digital signals. *International Journal of Approximate Reasoning*, 83, 400–412.

[51] Covington, P., Adams, J., & Sargin, E. (2016). Deep neural networks for YouTube recommendations. In *Proceedings of the 10th ACM conference on recommender systems* (pp. 191–198).: ACM.

[52] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4), 303–314.

[53] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., & Vetter, J. S. (2010). The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (pp. 63–74).: ACM.

[54] Dean, J. (2017). Recent advances in artificial intelligence and the implications for computer system design. *Hot Chips*.

[55] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems* (pp. 1223–1231).

[56] Dean, J., Patterson, D., & Young, C. (2018). A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2), 21–29.

[57] Delimitrou, C. & Kozyrakis, C. (2013). Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, volume 48 (pp. 77–88).: ACM.

[58] Delimitrou, C. & Kozyrakis, C. (2014). Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (pp. 127–144).: ACM.

[59] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 248–255).: Ieee.

[60] Ding, N., Fang, Y., Babbush, R., Chen, C., Skeel, R. D., & Neven, H. (2014). Bayesian sampling using stochastic gradient thermostats. In *Advances in neural information processing systems* (pp. 3203–3211).

[61] Dorazio, R. M., Royle, J. A., Söderström, B., & Glimskär, A. (2006). Estimating species richness and accumulation by modeling species occurrence and detectability. *Ecology*, 87(4), 842–854.

[62] Dubitzky, W., Granzow, M., & Berrar, D. P. (2007). *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media.

[63] Eigen (2019). Eigen thread pool. https://bitbucket.org/eigen/eigen/src/default/unsupported/Eigen/CXX11/src/ThreadPool/.

[64] Ein-Dor, P. & Feldmesser, J. (1987). Attributes of the performance of central processing units: a relative performance prediction model. *Communications of the ACM*, 30(4), 308–317.

[65] Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K., Cidon, A., & Katti, S. (2018). Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922*.

[66] Ellis, K., Solar-Lezama, A., & Tenenbaum, J. (2015). Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems* (pp. 973–981).

[67] Eyerman, S., Eeckhout, L., Karkhanis, T., & Smith, J. E. (2009). A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2), 3.

[68] Eyerman, S., Hoste, K., & Eeckhout, L. (2011). Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on* (pp. 216–226).: IEEE.

[69] Faix, M., Laurent, R., Bessière, P., Mazer, E., & Droulez, J. (2016). Design of stochastic machines dedicated to approximate Bayesian inferences. *IEEE Transactions on Emerging Topics in Computing*.

[70] Ferreira, J., Lanillos, P., & Dias, J. (2015). Fast exact Bayesian inference for high-dimensional models. In *Workshop on Unconventional computing for Bayesian inference (UCBI), IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[71] Ganesan, K., Jo, J., Bircher, W. L., Kaseridis, D., Yu, Z., & John, L. K. (2010). System-level max power (SYMPO)-a systematic approach for escalating system-level power consumption using synthetic benchmarks. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on* (pp. 19–28).: IEEE.

[72] Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2014). *Bayesian data analysis*, volume 2. Chapman & Hall/CRC Boca Raton, FL, USA.

[73] Gelman, A. & Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical science*, (pp. 457–472).

[74] Google (2018). Cloud TPU system architecture. *Google Cloud Documentation*. https://cloud.google.com/tpu/docs/system-architecture.

[75] Google (2019). TensorFlow performance guide. https://www.tensorflow.org/guide/performance/overview. TensorFlow Documentation.

[76] Griffiths, T. L. & Steyvers, M. (2004). Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1), 5228–5235.

[77] Griffiths, T. L., Steyvers, M., & Tenenbaum, J. B. (2007). Topics in semantic representation. *Psychological review*, 114(2), 211.

[78] Gupta, U., Dai, S., & Zhang, Z. (2015). Rosetta: A realistic benchmark suite for software programmable fpgas. In *Suite of Embedded Applications and Kernels Workshop (SEAK)*.

[79] Gupta, U., Wang, X., Naumov, M., Wu, C.-J., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Jia, B., Lee, H.-H. S., et al. (2019). The architectural implications of facebook's dnn-based personalized recommendation. *arXiv preprint arXiv:1906.03109*.

[80] Hamrick, J., Battaglia, P., & Tenenbaum, J. B. (2011). Internal physics models guide probabilistic judgments about object dynamics. In *Proceedings of the 33rd annual conference of the cognitive science society* (pp. 1545–1550).: Cognitive Science Society Austin, TX.

[81] Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*.

[82] Hartstein, A. & Puzak, T. R. (2002). The optimum pipeline depth for a microprocessor. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA)* (pp. 7–13).: IEEE Computer Society.

[83] Hasabnis, N. (2018). Auto-tuning TensorFlow threading model for CPU backend. *arXiv preprint arXiv:1812.01665*.

[84] Hauswald, J., Kang, Y., Laurenzano, M. A., Chen, Q., Li, C., Mudge, T., Dreslinski, R. G., Mars, J., & Tang, L. (2015a). Djinn and tonic: DNN as a service and its implications for future warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 43 (pp. 27–40).: ACM.

[85] Hauswald, J., Laurenzano, M. A., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R. G., Mudge, T., Petrucci, V., Tang, L., et al. (2015b). Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 50 (pp. 223–238).: ACM.

[86] Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., & Wang, X. (2018). Applied machine learning at Facebook: A datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on* (pp. 620–629).: IEEE.

[87] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

[88] He, Q., Zhou, S., Kobler, B., Duffy, D., & McGlynn, T. (2010). Case study for running HPC applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (pp. 395–401).: ACM.

[89] He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web* (pp. 173–182).: International World Wide Web Conferences Steering Committee.

[90] Heckerman, D., Geiger, D., & Chickering, D. M. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3), 197–243.

[91] Hennessy, J. L. & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[92] Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 1–17.

[93] Hershey, J. R. & Olsen, P. A. (2007). Approximating the Kullback Leibler divergence between Gaussian mixture models. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4 (pp. IV–317).: IEEE.

[94] Hershey, S., Bernstein, J., Bradley, B., Schweitzer, A., Stein, N., Weber, T., & Vigoda, B. (2012). Accelerating inference: towards a full language, compiler and hardware stack. *arXiv preprint arXiv:1212.2991*.

[95] Hoffman, M. D. & Gelman, A. (2014). The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1), 1593–1623.

[96] Hogg, R. V., McKean, J., & Craig, A. T. (2005). *Introduction to mathematical statistics*. Pearson Education.

[97] Hoste, K., Eeckhout, L., & Blockeel, H. (2007). Analyzing commercial processor performance numbers for predicting performance of applications of interest. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (pp. 375–376).: ACM.

[98] Hoste, K., Phansalkar, A., Eeckhout, L., Georges, A., John, L. K., & De Bosschere, K. (2006). Performance prediction based on inherent program similarity. In *Parallel Architectures and Compilation Techniques (PACT), 2006 International Conference on* (pp. 114–122).: IEEE.

[99] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[100] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. In *CVPR*, volume 1 (pp.3).

[101] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

[102] Ipek, E., De Supinski, B. R., Schulz, M., & McKee, S. A. (2005). An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing* (pp. 196–205).: Springer.

[103] İpek, E., McKee, S. A., Caruana, R., de Supinski, B. R., & Schulz, M. (2006). Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (pp. 195–206).: ACM.

[104] Jack Jr, C. R., Bernstein, M. A., Fox, N. C., Thompson, P., Alexander, G., Harvey, D., Borowski, B., Britson, P. J., L. Whitwell, J., Ward, C., et al. (2008). The Alzheimer's disease neuroimaging initiative (ADNI): MRI methods. *Journal of Magnetic Resonance Imaging: An Official Journal of the International Society for Magnetic Resonance in Medicine*, 27(4), 685–691.

[105] Jaleel, A. (2010). Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www.glue.umd.edu/ajaleel/workload*.

[106] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., & Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (pp. 675–678).: ACM.

[107] Jia, Z., Zaharia, M., & Aiken, A. (2018). Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*.

[108] Jitwasinkul, B., Hadikusumo, B. H. W., & Memon, A. Q. (2016). A Bayesian belief network model of organizational factors for improving safe work behaviors in thai construction industry. *Safety science*, 82, 264–273.

[109] Jonas, E. M. (2014). *Stochastic architectures for probabilistic computation*. PhD thesis, Massachusetts Institute of Technology.

[110] Joshi, A., Eeckhout, L., & John, L. (2008). The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop* (pp. 1–11).

[111] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on* (pp. 1–12).: IEEE.

[112] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., & Brooks, D. (2015). Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on* (pp. 158–169).: IEEE.

[113] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., & Brooks, D. (2016). Profiling a warehouse-scale computer. *ACM SIGARCH Computer Architecture News*, 43(3), 158–169.

[114] Kanev, S., Xi, S. L., Wei, G.-Y., & Brooks, D. (2017). Mallacc: Accelerating memory allocation. *ACM SIGOPS Operating Systems Review*, 51(2), 33–45.

[115] Karkhanis, T. S. & Smith, J. E. (2004). A first-order superscalar processor model. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on* (pp. 338–349).: IEEE.

[116] Kemp, C. & Tenenbaum, J. B. (2008). The discovery of structural form. *Proceedings of the National Academy of Sciences*, 105(31), 10687–10692.

[117] Kéry, M. & Schaub, M. (2011). *Bayesian population analysis using WinBUGS: a hierarchical perspective*. Academic Press.

[118] Ketkar, N. (2017). Introduction to PyTorch. In *Deep Learning with Python* (pp. 195–208). Springer.

[119] Kim, K., Lee, C., Jung, J. H., & Ro, W. W. (2014). Workload synthesis: Generating benchmark workloads from statistical execution profile. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on* (pp. 120–129).: IEEE.

[120] Ko, G. G. & Rutenbar, R. A. (2017). A case study of machine learning hardware: Real-time source separation using Markov Random fields via sampling-based inference. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 2477–2481).

[121] Ko, G. G. & Rutenbar, R. A. (2018). Real-time and low-power streaming source separation using Markov Random field. *ACM Journal on Emerging Technologies in Computing Systems*, 14(2), 17:1–17:22.

[122] Kothari, K. (2011). Comparison of several cloud computing providers. *Elixir Comp. Sci. & Engg.*

[123] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 1097–1105).

[124] Kung, H.-T. (1982). Why systolic architectures? *IEEE computer*, 15(1), 37–46.

[125] Labs, P. (2019). GeekBench v4. https://www.geekbench.com/.

[126] Lagar-Cavilla, A., Ahn, J., Souhlal, S., Agarwal, N., Burny, R., Butt, S., Chang, J., Chaugule, A., Deng, N., Shahid, J., et al. (2019). Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 317–330).: ACM.

[127] Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, 350(6266), 1332–1338.

[128] Leary, C. & Wang, T. (2017). XLA: TensorFlow, compiled. *TensorFlow Dev Summit*.

[129] Lee, B. C. et al. (2006). Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (pp. 185–194).: ACM.

[130] Lee, K., Rao, V., & Arnold, W. C. (2018). Accelerating facebook's infrastructure with application-specific hardware.

[131] Lei, Victor Sanders, N. & Dawson, A. (2016). Advertising attribution modeling in the movie industry. https://github.com/stan-dev/stancon_talks/blob/master/2017/Contributed-Talks/03_lei/ad_attribution.Rmd.

[132] Li, A., Zong, X., Kandula, S., Yang, X., & Zhang, M. (2011). CloudProphet: towards application performance prediction in cloud. (pp. 426–427).

[133] Li, A. Q., Ahmed, A., Ravi, S., & Smola, A. J. (2014). Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 891–900).: ACM.

[134] Li, L. & Russell, S. J. (2013). *The blog language reference*. Technical report, Technical Report UCB/EECS-2013-51, EECS Department, University of California, Berkeley.

[135] Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017a). Focal loss for dense object detection. *arXiv preprint arXiv:1708.02002*.

[136] Lin, Y., Han, S., Mao, H., Wang, Y., & Dally, W. J. (2017b). Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.

[137] Lomont, C. (2011). Introduction to Intel advanced vector extensions. *Intel White Paper*, (pp. 1–21).

[138] Maclaurin, D. & Adams, R. P. (2015). Firefly Monte Carlo: Exact MCMC with subsets of data. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[139] Mansinghka, V. K. (2009). *Natively probabilistic computation*. PhD thesis, Massachusetts Institute of Technology.

[140] Margossian, C. & Gillespie, W. R. (2016). Stan functions for Bayesian pharmacometric modeling. In *Journal of Pharmacokinetics and Pharmacodynamics*, volume 43 (pp. S52–S52).: SPRINGER/PLENUM PUBLISHERS 233 SPRING ST, NEW YORK, NY 10013 USA.

[141] marketsandmarkets.com (2018). Artificial intelligence market worth $190.61 billion by 2025 with a growing cagr of 36.6%.

[142] McElree, B. (2000). Sentence comprehension is mediated by content-addressable memory structures. *Journal of psycholinguistic research*, 29(2), 111–123.

[143] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

[144] Namasivayam, V. K. & Prasanna, V. K. (2006). Scalable parallel implementation of exact inference in Bayesian networks. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1 (pp. 8–pp).: IEEE.

[145] Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. (2019). Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*.

[146] Neal, R. M. (2012). *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media.

[147] Nicenboim, B. & Vasishth, S. (2016). Models of retrieval in sentence comprehension: A computational evaluation using Bayesian hierarchical modeling. *arXiv preprint arXiv:1612.04174*.

[148] Nickolls, J. & Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30(2).

[149] Ohlssen, D. (2016). An industry perspective of the value of Bayesian methods. https://pharmacy.ucsf.edu/sites/pharmacy.ucsf.edu/files/ohlssen.pdf.

[150] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., Pino, J., Schatz, M., Sidorov, A., Sivakumar, V., Tulloch, A., Wang, X., Wu, Y., Yuen, H., Diril, U., Dzhulgakov, D., Hazelwood, K., Jia, B., Jia, Y., Qiao, L., Rao, V., Rotem, N., Yoo, S., & Smelyanskiy, M. (2018). Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*.

[151] Patterson, D. (2018). MLPerf: SPEC for ML. https://rise.cs.berkeley.edu/blog/mlperf-spec-for-ml/.

[152] Pfeffer, A. (2016). *Practical Probabilistic Programming*. Manning Publications Co.

[153] Phansalkar, A., Joshi, A., Eeckhout, L., & John, L. K. (2005). Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on* (pp. 10–20).: IEEE.

[154] Phansalkar, A., Joshi, A., & John, L. K. (2007a). Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th annual international symposium on Computer architecture* (pp. 412–423).: ACM.

[155] Phansalkar, A., Joshi, A., & John, L. K. (2007b). Subsetting the SPEC CPU2006 benchmark suite. *ACM SIGARCH Computer Architecture News*, 35(1), 69–76.

[156] Phansalkar, A. S. (2007). *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. The University of Texas at Austin.

[157] Piccart, B., Georges, A., Blockeel, H., & Eeckhout, L. (2011). Ranking commercial machines through data transposition. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on* (pp. 3–14).: IEEE.

[158] Pourzanjani, A. A., Bales, B. B., Petzold, L. R., & Harrington, M. (2018). Flexible modeling of Alzheimer's disease progression with I-splines.

[159] Pu, Y., Miranda, Z., Solar-Lezama, A., & Kaelbling, L. (2018). Selecting representative examples for program synthesis. In *International Conference on Machine Learning* (pp. 4158–4167).

[160] Quiroz, M., Kohn, R., Villani, M., & Tran, M.-N. (2018). Speeding up MCMC by efficient data subsampling. *Journal of the American Statistical Association*, (just-accepted), 1–35.

[161] Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y., & Brooks, D. (2014). Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 110–119).: IEEE.

[162] Repository, T. M. (2018). https://github.com/tensorflow/tpu. *Github*.

[163] Research, B. (2017). Deepbench https://github.com/baidu-research/DeepBench.

[164] Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Nadathur, S., Olesen, J., et al. (2018). Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*.

[165] Salakhutdinov, R., Tenenbaum, J., & Torralba, A. (2012). One-shot learning with a hierarchical nonparametric Bayesian model. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning* (pp. 195–206).

[166] Saleem, M., Mehmood, Q., & Ngomo, A.-C. N. (2015). Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference* (pp. 52–69).: Springer.

[167] Schaffter, T., Marbach, D., & Floreano, D. (2011). GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods. *Bioinformatics*, 27(16), 2263–2270.

[168] Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*.

[169] Sharkawi, S., Desota, D., Panda, R., Indukuru, R., Stevens, S., Taylor, V., & Wu, X. (2009). Performance projection of HPC applications using SPEC CFP2006 benchmarks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (pp. 1–12).: IEEE.

[170] Shelepov, D., Saez Alcaide, J. C., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S., & Kumar, V. (2009). HASS: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43, 66–75.

[171] Shepelow, D. & Fedorova, A. (2008). Scheduling on heterogeneous multicore processors using architectural signatures. In *WIOSCA workshop of the 35th annual international symposium on Computer architecture (ISCA)*.

[172] Shi, S., Wang, Q., Xu, P., & Chu, X. (2016). Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on* (pp. 99–104).: IEEE.

[173] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484–489.

[174] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676), 354.

[175] Simoiu, C., Corbett-Davies, S., Goel, S., et al. (2017). The problem of infra-marginality in outcome tests for discrimination. *The Annals of Applied Statistics*, 11(3), 1193–1216.

[176] Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[177] Singh, K., İpek, E., McKee, S. A., de Supinski, B. R., Schulz, M., & Caruana, R. (2007). Predicting parallel application performance via machine learning approaches. *Concurrency and Computation: Practice and Experience*, 19(17), 2219–2235.

[178] Sriraman, A., Dhanotia, A., & Wenisch, T. F. (2019). SoftSKU: Optimizing server architectures for microservice diversity at scale. In *Proceedings of the 46th International Symposium on Computer Architecture* (pp. 513–526).: ACM.

[179] Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., & Hwu, W.-m. W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127.

[180] Stroobandt, D., Verplaetse, P., & Van Campenhout, J. (2000). Generating synthetic benchmark circuits for evaluating CAD tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9), 1011–1022.

[181] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (pp. 1–9).

[182] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (pp. 2818–2826).

[183] Tao, J.-H., Du, Z.-D., Guo, Q., Lan, H.-Y., Zhang, L., Zhou, S.-Y., Xu, L.-J., Liu, C., Liu, H.-F., Tang, S., Chen, W., Liu, S.-L., & Chen, Y.-J. (2018). BenchIP: Benchmarking intelligence processors. *Journal of Computer Science and Technology*, 33(1), 1–23.

[184] Tenenbaum, J. B., Kemp, C., Griffiths, T. L., & Goodman, N. D. (2011). How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022), 1279–1285.

[185] Thirey, B. & Hickman, R. (2015). Distribution of euclidean distances between randomly distributed gaussian points in n-space. *arXiv preprint arXiv:1508.02238*.

[186] Thomas, S., Gohkale, C., Tanuwidjaja, E., Chong, T., Lau, D., Garcia, S., & Taylor, M. B. (2014). CortexSuite: A synthetic brain benchmark suite. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on* (pp. 76–79).: IEEE.

[187] tractica.com (2018). Deep learning chipset market to reach $66.3 billion by 2025.

[188] Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., & Blei, D. M. (2016). Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*.

[189] Turki, M., Mehrez, H., Marrakchi, Z., & Abid, M. (2012). Towards synthetic benchmarks generator for CAD tool evaluation. In *Ph. D. Research in Microelectronics and Electronics (PRIME), 2012 8th Conference on* (pp. 1–4).: VDE.

[190] Van den Steen, S., De Pestel, S., Mechri, M., Eyerman, S., Carlson, T., Black-Schaffer, D., Hagersten, E., & Eeckhout, L. (2015). Micro-architecture independent analytical processor performance and power modeling. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on* (pp. 32–41).: IEEE.

[191] Van den Steen, S., Eyerman, S., De Pestel, S., Mechri, M., Carlson, T. E., Black-Schaffer, D., Hagersten, E., & Eeckhout, L. (2016). Analytical processor performance and power modeling using micro-architecture independent characteristics. In *IEEE Transaction on Computers*, volume 65.

[192] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., & Cohen, A. (2018). Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*.

[193] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (pp. 5998–6008).

[194] Venkata, S. K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., & Taylor, M. B. (2009). SD-VBS: The San Diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (pp. 55–64).: IEEE.

[195] Wainwright, M. J. & Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2), 1–305.

[196] Wang, S., Zhang, X., Li, Y., Bashizade, R., Yang, S., Dwyer, C., & Lebeck, A. R. (2016a). Accelerating Markov random field inference using molecular optical Gibbs sampling units. In *Proceedings of the 43rd International Symposium on Computer Architecture* (pp. 558–569).: IEEE Press.

[197] Wang, S. S. J. & Wand, M. P. (2011). Using Infer.NET for statistical analyses. *The American Statistician*, 65(2), 115–126.

[198] Wang, Y., Qian, W., Zhang, S., Liang, X., & Yuan, B. (2016b). A learning algorithm for Bayesian networks and its efficient implementation on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 27(1), 17–30.

[199] Wang, Y. & Wang, D. (2013). Towards scaling up classification-based speech separation. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(7), 1381–1390.

[200] Wang, Y. E., Wei, G.-Y., & Brooks, D. (2019a). Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*.

[201] Wang, Y. E., Zhu, Y., Ko, G. G., Reagen, B., Wei, G.-Y., & Brooks, D. (2019b). Demystifying bayesian inference workloads. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 177–189).: IEEE.

[202] Watters, N., Zoran, D., Weber, T., Battaglia, P., Pascanu, R., & Tacchetti, A. (2017). Visual interaction networks: Learning a physics simulator from video. In *Advances in Neural Information Processing Systems* (pp. 4539–4547).

[203] Wei, W., Xu, L., Jin, L., Zhang, W., & Zhang, T. (2018). AI matrix-synthetic benchmarks for DNN. *arXiv preprint arXiv:1812.00886*.

[204] Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), 65–76.

[205] Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., et al. (2019). Machine learning at Facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (pp. 331–344).: IEEE.

[206] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[207] Xia, Y. (2010). *Exploration of parallelism for probabilistic graphical models*. University of Southern California.

[208] Xie, S., Girshick, R., Dollár, P., Tu, Z., & He, K. (2017). Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (pp. 1492–1500).

[209] Xie, Y. (2016). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.15.1.

[210] Xu, F. & Tenenbaum, J. B. (2007). Word learning as Bayesian inference. *Psychological review*, 114(2), 245.

[211] Yang, A. (2019). Deep learning training at scale spring crest deep learning accelerator (intel nervana nnp-t).

[212] Yao, L., Mimno, D., & McCallum, A. (2009). Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 937–946).: ACM.

[213] Yasin, A. (2014). A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 35–44).: IEEE.

[214] Yu, H.-F., Hsieh, C.-J., Yun, H., Vishwanathan, S., & Dhillon, I. S. (2015). A scalable asynchronous distributed algorithm for topic modeling. In *Proceedings of the 24th International Conference on World Wide Web* (pp. 1340–1350).: International World Wide Web Conferences Steering Committee.

[215] Yuan, J., Gao, F., Ho, Q., Dai, W., Wei, J., Zheng, X., Xing, E. P., Liu, T.-Y., & Ma, W.-Y. (2015). Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web* (pp. 1351–1361).: International World Wide Web Conferences Steering Committee.

[216] Zheng, L., Mengshoel, O., & Chong, J. (2012). Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization. *arXiv preprint arXiv:1202.3777*.

[217] Zheng, X., John, L. K., & Gerstlauer, A. (2016). Accurate phase-level cross-platform power and performance estimation. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE* (pp. 1–6).: IEEE.

[218] Zheng, X., Vikalo, H., Song, S., John, L. K., & Gerstlauer, A. (2017). Sampling-based binary-level cross-platform performance estimation. In *Proceedings of the Conference on Design, Automation & Test in Europe* (pp. 1713–1718).: European Design and Automation Association.

[219] Zhu, Y., Richins, D., Halpern, M., & Reddi, V. J. (2015). Microarchitectural implications of event-driven server-side web applications. In *Proceedings of International Symposium on Microarchitecture*.