論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Cross-Architecture Performance Prediction Using Collaborative Filtering
著者(和文)	Salaria Shweta
Author(English)	Shweta Salaria
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第11318号, 授与年月日:2019年9月20日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,脇田 建,額田 彰,横田 理央
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第11318号, Conferred date:2019/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO INSTITUTE OF TECHNOLOGY

Cross-Architecture Performance Prediction Using Collaborative Filtering

Author:
Shweta SALARIA

Supervisor:
Prof. Satoshi MATSUOKA

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Department of Mathematical and Computing Sciences

August 27, 2019

Abstract

Over the last decade, in response to the slowing or end of Moore's law, the High Performance Computing (HPC) community has turned towards heterogeneous systems. Performance prediction of scientific applications across systems becomes increasingly important in today's diverse computing environments. A wide range of choices in execution platforms pose new challenges to researchers in choosing a system which best fits their workloads and administrators in scheduling applications to the best performing systems. While previous studies have employed simulation- or profile-based prediction approaches, such solutions are time-consuming to be deployed on multiple platforms. To address this problem, we use two collaborative filtering techniques to build analytical models which can quickly and accurately predict the performance of workloads across different multicore systems. The first technique leverages information gained from performance observed for certain applications on a subset of systems and use it to discover similarities among applications as well as systems. The second collaborative filtering based model learns latent features of systems and workloads automatically and use these features to characterize the performance of applications on different platforms. We evaluated both the methods using 30 workloads chosen from NAS Parallel Benchmarks, BOTS and Rodinia benchmarking suites on ten different systems. Our results show that such collaborative filtering methods can make predictions with RMSE as low as 0.6 and with an average RMSE of 1.6.

The graphic processing units (GPUs) have become a primary source of heterogeneity in today's computing systems. With the rapid increase in number and types of GPUs available, finding the best hardware accelerator for each application is a challenge. For that matter, it is time consuming and tedious to execute every application on every GPU system to learn the correlation between application properties and hardware characteristics. To address this problem, we extend our previously proposed collaborating filtering based modeling technique, to build an analytical model which can predict performance of applications across different GPU systems. Our model learns representations, or embeddings (dense vectors of latent features) for applications and systems and uses them to characterize the performance of various GPU-accelerated applications. We improve state-of-the-art collaborative filtering approach based on matrix factorization by building a multi-layer perceptron. In addition to increased accuracy in predicting application performance, we can use this model to simultaneously predict multiple metrics such as rates of memory access operations. We evaluate our approach on a set of 30 well-known micro-applications and seven NVIDIA GPUs. As a result, we can predict expected instructions per second value with 90.6% accuracy in average.

Acknowledgements

First and foremost, I'd like to thank my supervisor, Prof. Satoshi Matsuoka. For the superb mentorship and stellar guidance he has provided towards my thesis and my growth as a researcher, I will be eternally grateful.

I also gratefully and humbly acknowledge the efforts of my direct collaborators, Dr. Aleksandr Drozd, Dr. Artur Podobas and Dr. Hideyuki Jitsumoto. Their contributions have been instrumental in developing the research topics presented in this document.

I give my sincerest thanks to all the members of Matsuoka lab, students and staff, for the tireless assistance they have provided at the different stages of this research and its presentation.

And to my friends and family who have kept me balanced, my appreciation knows no bounds.

Contents

List of Figures

List of Tables

Dedicated to my parents…

Chapter 1

Introduction

1.1 Motivation

Modeling and predicting computer system performance has been and still is a crucial component in our road towards better, faster and more reliable computer systems. With Moore's law on its last legs and heterogeneity a part of our everyday life, predicting performance across systems is perhaps more important now than ever before.

Knowing how to extrapolate or project performance across systems is useful across the entire hardware stack, ranging from small embedded systems, all the way up to large distributed high-performance machines. Cloud-services such as Amazon [4] or Microsoft [39] can assess user requirements using predictive tools and deliver neither more nor less than what the customers and their applications require, subject only to the agreed-upon quality-of-service protocols. Cloud-users can similarly evaluate and predict their cloud needs and pinpoint the exact service to invest in. System administrators and users can (prior to purchase) approximate the impact of replacing system components. Software designers can similarly estimate how their applications run on next-generation hardware, whereas researchers use predictive models as a tool to see future trends and potential pitfalls.

Graphics Processing Units (GPUs) are today the de-facto source of performance in High-Performance Computing (HPC), and the vast majority of current top supercomputers [60] include them in their system setup. These powerful devices are explicit vector machines, whose programming model allows the programmer to leverage the large amount of parallelism they offer. Unlike general-purpose processors, which focus on exploiting instruction-level parallelism (and focus on latency), GPUs focus on thread-level parallelism, masking/hiding long-latency operations (e.g. external memory accesses) by time-sharing computational resources. Despite their similar programming model, GPUs are constantly undergoing architectural changes across generation (e.g. mixed-precision Arithmetic Logic Units, per thread program counters, diverse amount of floating-point units), which makes their performance non-trivial to reason around and *predict*.

Predicting GPU performance is a challenging and hard task. Despite sharing a programming model (CUDA [45] and OpenCL [58]), their architectural differences between generations can be substantial. Furthermore, with the advent and popularity of Deep-Learning, the type and target audience of GPUs is diversifying. As a result, GPUs specialized in inference, training, gaming, and scientific computing are emerging [46]. Given this vast array of current and emerging GPU types, *how do users choose which to invest in?* Today, most users blindly buy the fastest next-generation accelerator for their workload, which is more than likely not the most optimal choice. There is a need for a *simple* and effective performance model that assist users in choosing accelerators suitable for the workloads they care about.

1.2 Problem Statement

Predicting or modeling performance has been researched ever since the 1960s [51]. Unfortunately, the vast majority of work attempts (with more or less success) to develop *system-specific* models that are unique to a particular system. These models require either detailed knowledge about hardware [48] (e.g. pipeline stacks, reorder buffer sizes, etc.) or rely on a cherry-picked set of hardware counters (e.g. [53]), which may or may not exist across architectures.

Processor performance prediction has historically leveraged hardware performance counters to extrapolate expected application or system behavior. Numerous studies have collected large amount of hardware counter information and tried to model application behavior as a function of these. Early on modeling was performed by regression (linear or otherwise), while lately deep-learning is being applied. Given the large amount of research carried out with performance prediction in mind, what drives us to seek an alternative solution to the problem?

There are two main reasons why *any* method based on amassing information from hardware performance counters is sub-optimal. The first drawback is the labor involved in constructing the models, which typically involved running several microbenchmarks [53, 17, 2] that stress the system in different ways (that is, isolate various features), and then manually regressing the model to the data collected. It is a lengthy and tedious labor process that in the end only yields a *machine-specific* performance model. The second drawback is that the machine-specific models based on the hardware counter-based method are inapplicable across systems. The number and type of hardware performance counters can vary significantly between systems to the point where many proposed performance models are unusable on modern hardware. Furthermore, hardware counters are often unique to a particular system or instruction set architecture (ISA) and even counters named similar can have ambiguous meanings across ISAs, vendors and architectures.

Other methods exists, but are either orthogonal or complementary to our approach, or come with problems of their own. One approach is to simulate the performance on a hypothetical machine using simulators such as gem5 [44] or simics [50] as it was done for x86-64 Hammer prior to hard existing [16]). The simulators provides reasonable good accuracy and enough details to understand what is happening to the system. Unfortunately, they are also many orders of magnitudes slower and machine specification does not exist for all processors. Faster and more abstract analytic models such as LogP [13], Roofline model [54]) allow for fast estimation of performance, but are often hard to use without detailed knowledge of the algorithm at hand. Other methods include estimating performance inside the compiler [12] or using detailed and thus difficult to obtain machine specifications such as reorder buffer sizes and pipeline-depths [48]. We considered these methods complementary to the one we present in the paper, as they can be incorporated into our proposed method to potentially further improve accuracy.

1.3 Proposal and Contributions

To address the previously mentioned challenges, we present the following contributions:

1. We perform an in-depth analysis of two scientific benchmarks using a cloud platform and two supercomputers. We find that cloud instances exhibit good compute performance with low performance variability after optimizing the benchmarks with number of threads and MPI ranks. These instances prove to be a good fit for compute-intensive and memory-intensive application with little communication.The performance exhibited by these cloud instances is comparable to our in-house supercomputers. Thus, they can be adopted as a replacement to on-premises hardware deployments for such workloads.

2. With the increasing heterogeneity in systems, the problem of modeling performance of scientific applications across different systems has become more important now than ever before. We show the feasibility of using collaborative filtering (CF) methods for predicting performance using a pilot study. We construct performance datasets and implement two CF based methods to predict performance across multi-core processors.We show the capability of matrix factorization based CF to uncover latent features that explain hardware-program interactions using statistical methods.

3. We extend our previous matrix factorization based prediction approach to Graphics Processing Units (GPUs). We evaluate a set of 30 applications on 7 different generations of GPUs ranging from Kepler to Volta.

4. We design and propose a neural network architecture to learn complex hardware-program interactions to further improve the accuracy of our prediction model. We show that augmenting multiple performance metrics affect the prediction accuracy.

1.4 Thesis Outline

The document is organized as follows. We discuss the context of this work by giving some background knowledge in Chapter 2. We then evaluate performance of some scientific benchmarks on different systems and discuss the need to focus on cross-architecture performance models. We follow by the body of the work in the next three chapters.Chapter 4 relates our model and analysis to predict performance across different processors on multicore systems. Chapter 5 presents our exisiting model solution to model GPU performance. This Chapter also introduces a new model using multilayer perceptron in order to improve the prediction accuracy.Finally, we make concluding remarks in Chapter 6 along with future directions.

Chapter 2

Background

2.1 Recommender Systems

Recommender Systems (RSs) are software tools and methods that provide suggestions to a user for items that can be of use. These suggestions can be related to various decision-making processes, such as what items to buy, what movie to watch, or what news to highlight in a user's social media account. "Item" is what the system recommends to users. A RS focuses on a specific type of item and three things related to the item in question: 1) the item's design, 2) the item's graphical user interface (GUI) and 3) the recommendation technique that is used to generate the recommendations. The recommendations are then customized to provide useful and effective suggestions for that specific type of item.

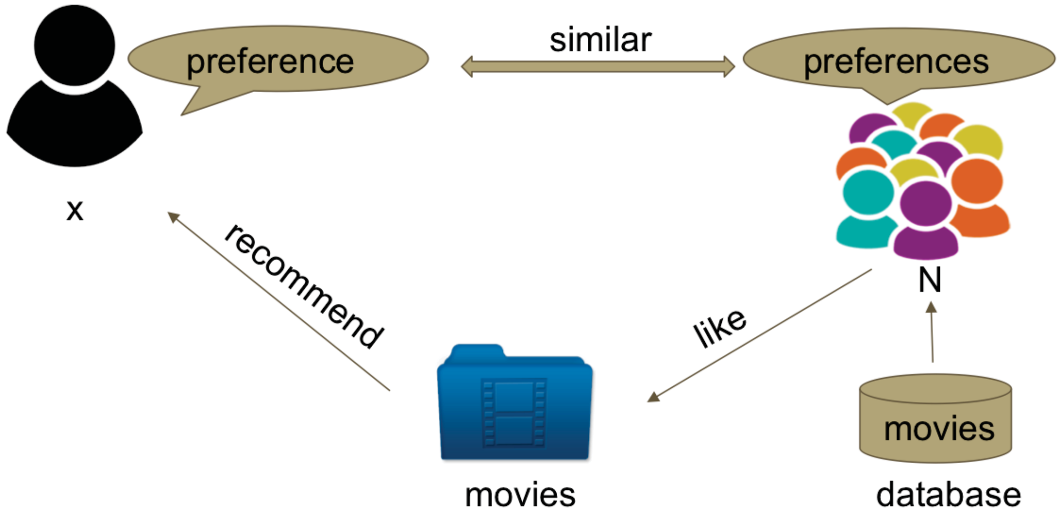


FIGURE 2.1: Movie Recommender System

RSs are designed for users who lack sufficient personal experience or competence to evaluate the potentially overwhelming number of alternative items. A case in point is a movie recommender system in figure 2.1 that assists users to select a movie to watch.

2.2 Collaborative Filtering

Collaborative filtering techniques [3] are the most popular and widely implemented techniques in building recommender systems. They generate automatic predictions (filtering) about the interests of a user by collecting preferences or information from many users (collaborating) present in the system. One of the most publicized applications of collaborative filtering is the Netflix challenge. The goal of the challenge was to improve Netflix's movie recommendation system, Cinematch by providing valid movie recommendations to its users. The prize was won by BellKor's Pragmatic Chaos [64, 36, 8] after improving Cinematch by over 10% using a combination of different collaborative filtering techniques.

The collaborative filtering techniques are grouped in two general classes of neighborhood and model-based methods. In neighborhood-based collaborative filtering, the user-movie ratings stored in the system are directly used to predict ratings for new movies. It can be done in two ways known as user-based or movie-based collaborative filtering. User-based systems evaluate the interest of a user u given a movie m. The evaluation is performed by identifying neighbors of u that have similar rating pattern as u, and looking at how those very neighbors, called v, rated m. Identifying which neighbors belong to u is done by looking at how well shared items of u and v correlated with each other.

Movie-based approaches take a different stand and predict the rating of a user u for a movie m by looking at how u rated movies similar to m. In this way, two movies are similar if several users found them similar. In contrast to neighborhood-based methods, which use the stored ratings directly in the prediction, model-based approaches use these ratings to learn a predictive model. The core idea is to model the user-movie interactions with factors representing the latent characteristics of the users and movies in the system. The model is then trained using the available data and later used to predict missing ratings of users for movies.

2.2.1 Neighborhood-based Recommendation

k-Nearest Neighbor (kNN) is an intuitive approach to implement collaborative filtering. For a given application and system, we want to predict its performance score. kNN works by identifying the k closest points (nearest neighbors) of either the application or system. The underlying idea is that similar applications (or systems) tend to cluster together based on their similar performance scores. This neighborhood approach is highly dependent on defining an appropriate similarity metric. We use Pearson correlation to measure the similarity between applications as well as systems. We implement two types of neighborhood-based collaborative filtering: application-based collaborative filtering and system-based collaborative filtering.

User-based Rating Prediction

We first construct a $N_a \times N_a$ matrix and populate the matrix by calculating similarity for each pair of applications using the Pearson correlation. The Pearson correlation coefficient, $sim(i, j)$ between any two applications i and j is a measure of strength of the linear relationship between them. The value of coefficient ranges from -1 to 1. A coefficient of -1 indicates a perfect negative linear relationship between applications, a coefficient value of 0 indicates no relationship while a coefficient of 1 indicates a positive linear relationship. It is calculated using the formula below:

Here, $S(a_1)$ is the set of all systems for which the performance score of application a_1 is known; $S(a_2)$ is the set of all systems for which the performance score of application a_2 is known; $S(a_1) \cap S(a_2)$ is a set of systems for which scores of both applications a_1 and a_2 are known.

We want to predict the performance score for an application a on a system s. We begin by finding the k most similar applications for which the performance scores on the system s is known. We denote these neighbors of application a by $N_s(a)$. Then, we find the predicted score by taking the weighted average using the performance scores of k applications and their similarity values w.r.t system s. The prediction of a performance score p_{as} using application-based collaborative filtering is obtained as:

$$\hat{p}_{as} = \bar{p}_a + \frac{\sum\limits_{k \in N_s(a)} sim(a,k)(p_{ks} - \bar{p}_k)}{\sum\limits_{k \in N_s(a)} |sim(a,k)|}$$

The prediction accuracy depends on k, the number of nearest neighbors. So, we try different value of k in the training set and evaluate the corresponding performance by measuring the RMSE.

Movie-based Rating Prediction

We construct a $N_s \times N_s$ similarity matrix for N_s number of systems. We calculate the similarity values between each pair of systems and then apply kNN method to predict the performance score as:

$$\hat{p}_{as} = \bar{p}_s + \frac{\sum\limits_{k \in N_a(s)} sim(s,k)(p_{ak} - \bar{p}_k)}{\sum\limits_{k \in N_a(s)} |sim(s,k)|}$$

2.2.2 Matrix Factorization

Matrix factorization models are the most popular implementations of model-based collaborative filtering. The core idea is to uncover latent features or factors among applications and systems that explain the known performance scores. These models map both applications and systems to a joint latent factor space of dimensionality r, such that application-system interactions are modeled as inner products in that space.

The input to the model is a sparse matrix A, with one row per application and one column per system. The first step in a matrix factorization technique is to decompose A into U, λ and V such that:

$$A \approx U \lambda V^T$$

Given the matrix A (n applications, m systems), we factorize it into $n \times r$ matrix U (n applications, r latent features), $r \times r$ diagonal matrix λ (strength of each latent feature), and $m \times r$ matrix V (m systems, r latent features). Figure 4.1 illustrates this

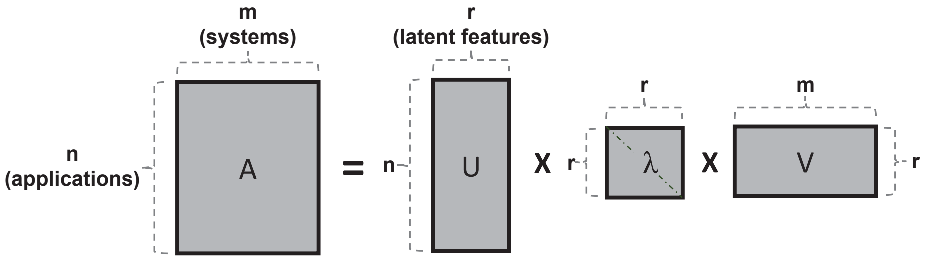


FIGURE 2.2: Factorization of matrix A into sub matrices U, λ and V using the latent features.

idea. The λ diagonal matrix contains the similarity concepts identified by the matrix factorization technique. The U matrix is interpreted as the application-to-feature similarity matrix, while the V matrix is the system-to-feature similarity matrix. For instance, two or more applications can be similar in one latent feature (they both benefit from a large L3 cache) but different in others (only one benefits from high clock frequency). These latent features contain the singular values which are sorted in decreasing order. The application-to-feature matrix represents the strength of the associations between an application and the r latent features. In this way, each application is described using a r-dimensional space. Similarly, the system-to-feature matrix identifies each system using a r-dimensional space.

Let us denote $Q_{n\times r} = U$ and $P^T_{r\times m} = \lambda V^T$. The prediction of an application a on a system s can be calculated as the dot products as their vectors:

$$\begin{aligned}\hat{p}_{as} &= q_s.p_a^T \\ &= \sum_r q_{sr}.p_{ar}\end{aligned}$$

Now we may obtain P and Q by applying any dimensionality reduction technique such as Singular Value Decomposition (SVD) to the matrix A.

2.3 Latent features

The performance of a program on a given system is guided by the complex interactions between the program's properties and system's characteristics. The most common approach taken towards building analytical models is to first perform a detailed characterization of applications on systems and collect various performance metrics such as execution time, hardware counters [63, 24]. Second, it needs to explicitly list all the features that can capture the inter-dependencies between the application and system at hand such as clock rate, cache size, floating point operations per second (FLOPS) etc. Most previous works use an intuitive approach when selecting the number of features in order to determine a set of good explanatory features. However, missing just one crucial feature from these carefully handpicked features can greatly (and negatively) affect the prediction accuracy. Third, we need to test each possible subset of those features finding the one which minimizes the error. This is an exhaustive search of the feature space, and is computationally expensive and unfeasible for all but the smallest of feature sets. Feature selection algorithms come handy when it comes to finding the most impactful feature subsets but it comes with an extra effort of selecting the appropriate algorithm and its parameters.

While considering tens or more of applications and systems, manually defining features for each set of application and system is practically impossible. For building a cross-architecture predictive model – such as the one we are targeting – we focus on two things: 1) We do not want to manually define feature for each application and system 2) we want our model to learn the features automatically from the known performance of a subset of applications executed on a subset of systems and leverage this information to predict performance.

2.4 Embeddings

The concept of representation learning is grounded in the idea that often the information needed to characterize or classify high dimensional data can be found in a low-dimensional manifold, or mapped into a dense vector. For instance, natural language processing systems use word embeddings to represent (embed) words in a continuous vector space where semantically similar words have similar vector representations (embeddings)[40]. Recommender systems employ similar representations to describe its entities (e.g. users and items) and call them as *latent features*.

We assume that there is a number of important features, called latent features, which characterize systems and applications. We use a machine learning model to learn these features as a by-product of predicting known runtime metrics. In our model, we have application and system feature vectors. Two or more applications can be similar in one latent feature (they both benefit from high memory bandwidth) but different in others (only one benefits from high core clock frequency).

Chapter 3

Performance Evaluation

3.1 Introduction

High performance computing (HPC) systems are fundamental to solving complex scientific problems by utilizing the high computation power of thousands of processors and high-throughput networks. However, the use of HPC systems for Big Data problems is becoming more common across HPC centers. Big Data refers to the diverse, complex, and massive data sets that can contain structured, semi-structured and unstructured data. These data sets are difficult to be stored, processed, and analyzed by traditional database technologies. Hence, we seek a set of new technologies and advanced data analytics methods for data distribution, management, and processing. The trend in the recent decade has been to extend the application of HPC systems beyond the computationally-intensive scientific domain to data-intensive domains, or the domain of "Big Data". While HPC has its roots in solving compute-intensive scientific and large-scale distributed problems, Big Data problems have been proven to benefit from typical HPC environments: high processing power, low-latency networks, and non-blocking communications [23]. Also, such environments are used for running some typical data analytics problems such as graph processing and its application in cybersecurity, social networks, medical informatics etc. There have been various approaches proposed for employing HPC to support data-intensive applications e.g. Map-Reduce-MPI [33], Pilot-MapReduce [35] etc.

HPC has given the most significant contribution to scientific discovery and is expanding its applications to analytics. However, the use of HPC systems is limited to scientists and researchers who have access to supercomputing labs and centers. With HPC extending its application space from scientific applications to big data analytics, the increasing demand for resources in HPC centers may not be met immediately. Hence, it is important to study alternative HPC solutions and the extent to which these solutions can be employed for different classes of computing and their applications.

Public cloud computing has gained significant popularity over the past decade. It exhibits pay-as-you-go facilities, elasticity, flexibility, usability, and scalability, which have been attracting clients to use these environments as a cost effective measure to run their applications or businesses. Cloud environments can help in bringing HPC access to users who can't afford complex setup and huge infrastructure costs. While Cloud for HPC has always been a point of concern with the HPC community, there has been growing efforts by cloud service providers to build new solutions that target HPC audiences. Amazon Elastic Compute Cloud (Amazon EC2), the leading infrastructure as a service (IaaS) provider, is the most studied cloud platform to evaluate the capabilities of their instances for running HPC applications. They announced the new generation of compute-optimized instances, C4 [7] in 2015 with the aim of enabling HPC in cloud. These instances are supposed to offer the power of an HPC system while leveraging the flexibility of the cloud, ideal for the true HPC-Big Data convergence. To the best of our knowledge, these C4 instances have yet to be tested yet, which is a motivating factor of this work.

This work assesses the performance trade-off when using the latest generation of HPC-like clouds versus dedicated HPC systems. For this evaluation, we compare the performance of a traditional HPC scientific mini-application, NICAM, and a traditional Big Data benchmark, Graph500, on TSUBAME2.5 supercomputer, TSUBAME-KFC supercomputer, and Amazon EC2 C4 instances.

3.2 Related work

There have been several studies in the past that examine the feasibility of using public clouds for high performance computing with different goals and focuses. The Magellan Report on Cloud Computing Science [61] in 2011 was a two-year study to investigate the potential role of cloud computing in addressing the computing needs for DOE Office of Science (D0E), particularly for mid-range computing and future data-intensive workloads. It assessed the performance of various scientific applications on Amazon cloud and systems at two DOE labs and concluded that scientific applications with minimal communication and I/O were best suited for clouds. There was much focus on evaluating cloud platforms for tightly-coupled, MPI-based applications [22, 27, 42]. Also, there was a study to assess the cost-effectiveness of cloud environments for running scientific workloads [25]. With the announcement of Amazon EC2 Cluster Compute Instances (CCI) in 2011, there was a study for comparing in-house cluster with EC2 CCI for MPI applications [67, 65]. Since then, Amazon EC2 has introduced many different instances intended for high performance science and engineering applications. They offered the latest generation of compute-optimized instances, C4 instances in 2015 with the highest performing processors in all the instances. Capabilities of these instances for different HPC as well as for data-intensive workloads have yet be studied.

3.3 Methods

3.3.1 Machines Used in Study

We present the specifications of TSUBAME2.5, TSUBAME-KFC, and Amazon C4 instance in Table 3.1.

TSUBAME2.5

TSUBAME2.5 is a production supercomputer operated by Global Scientific Information and Computing Center (GSIC), Tokyo Institute of Technology. It consists of more than 1400 thin compute nodes interconnected by 2xQDR InfiniBand i.e. 80 Gbps network. Each TSUBAME2.5 node has two Intel Xeon X5670 processors, NVIDIA Tesla K20X, and 54 GB of local memory. The compute nodes share a scalable storage system constructed with Lustre parallel file system with a capacity of 7.13 PB. Codes were compiled on TSUBAME2.5 using gcc 4.3.4 and OpenMPI 1.6.5.

TSUBAME-KFC

TSUBAME-KFC is a prototype for next-generation TSUBAME 3.0 supercomputer operated by Global Scientific Information and Computing Center (GSIC), Tokyo Institute of Technology. It is composed of forty-two compute nodes interconnected by 4xFDR InfiniBand i.e. 54 Gbps network. Each compute node is equipped with two Intel Xeon E5-2620 v2 processors (Ivy Bridge EP) and four NVIDIA Tesla K80 GPU boards. These compute nodes share 1.1 TB SSD-based storage system. Codes were compiled on TSUBAME-KFC using gcc 4.8.5 and OpenMPI 1.6.5.

Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides Infrastructure as a Service (IaaS), enabling the creation and management of virtual machines. We used the latest generation of compute-optimized instances, c4.8xlarge instances in our cloud experiments. Each c4.8xlarge instance has an Intel Xeon E5-2666 v3 (Haswell) 2.9 GHz & 3.5 GHz with Intel Turbo Boost, processors with 60GB memory and 36 vCPUs. These instances are interconnected with 10 Gigabit Ethernet and offer Elastic Block Store (EBS) [6]. EBS are block-level storage volumes and they can be attached to any running instance within the same availability zone.

We created a virtual EC2-based cluster using our customized Amazon Machine Image (AMI) [5] on c4.8xlarge instances for our experiments. For shared file system, we attached an Amazon EBS volume to one of the instances. On top of the EBS volume, we built a standard Linux ext4 file system that was exported via NFS to all of the virtual cluster nodes. We ran all our experiments in the Asia Pacific (Tokyo) region. Also, we launched our instances using a placement group which ensures that the virtual machines were created within one region only.

3.3.2 Applications Used in Study

NICAM-DC-MINI

NICAM-DC-MINI is a Fiber miniapp that is developed and maintained at RIKEN Advanced Institute for Computational Science [37, 38]. It is a subset of NICAM-DC

TABLE 3.1: Specification of the evaluated platforms.

Specs	TSUBAME2.5	TSUBAME-KFC	Amazon EC2 c4.8xlarge instance
Cores	12 Physical	12 Physical	36 Virtual
Memory	54 GB	64 GB	60 GB
Processor	Intel Xeon X5670 (Westmere-EP)	Intel Xeon E5-2620v2 (Ivy bridge)	Intel Xeon E5-2666v3 (Haswell)
Core Frequency	2.93 GHz	2.10 GHz	2.9 GHz
Network	80 Gbps (2XQDR Infiniband)	54 Gbps (4xFDR Infiniband)	10 Gbps Ethernet
Storage	7.13 PB (Lustre parallel file system)	1.1 TB (Local SSD) and 30 TB (NFS over 10 Gigabit Ethernet)	Elastic Block Store

application and contains the minimum computational procedures to run the baroclinic wave test case Jablonowski [26]. Jablonowski is a well-known benchmark of atmospheric general circulation model reproducing unsteady baroclinic wave oscillation. It is a stencil code on icosahedral grid with the explicit method of Runge-Kutta. The icosahedral grid is divided into blocks and distributed among nodes. Each node then transfers the boundary data to the nearest-neighbor nodes. The algorithm pattern of NICAM-DC-MINI corresponds to the structured grid of Berkeley Dwarfs [9].

Graph500 benchmark

Graph500 [20] is a data-intensive benchmark based on large-scale graph analysis. It performs breadth-first searches on weighted, undirected large graphs generated by scalable data generator based on a Kronecker graph. Graph500 consists of two kernels: The first kernel constructs an undirected graph used by the second kernel. The second kernel performs a breadth-first search (BFS) of the graph from a randomly chosen source vertex in the graph. Both the kernels are timed and this benchmark uses the performance metric Traversed Edges Per second (TEPS) to compare the benchmark performance across multiple architectures, programming models, and frameworks. TEPS is measured by benchmarking the second kernel.

3.3.3 Experiment Setup for NICAM-DC-MINI

We have performed weak scaling tests using the Jablonowski test case of NICAM-DC-MINI on all three systems. The default test case is an 11-day simulation with 224 km horizontal grid spacing and 792 time steps. For our experiments, we reduced the simulation time by setting the number of time steps to 264. The weak scaling test cases of Jablonowski are provided for 10, 40, and 160 MPI processes. The simulation size for each test case is depicted in Table 3.2.

TABLE 3.2: Weak scaling test cases of Jablonowski.

# of MPI ranks	# of horizontal grids (total)	total # of grids
10	11,560	947,920
40	46,240	3,791,680
160	184,960	15,166,720

Amazon EC2 C4 instances use custom Intel Xeon E5-2666 v3 (Haswell) processors as stated by Amazon. /proc/cpuinfo on a c4.8xlarge instance results in 2 sockets and 9 cores per socket. Also, Amazon has mentioned the breakdown of 36 vCPUs as 18 cores with each core running two hyperthreads [7]. But since it is a virtualized environment, we evaluated both 10 and 20 MPI process per c4.8xlarge instance. Both TSUBAME2.5 and TSUBAME-KFC have 12 physical cores per node, so we used 10 MPI process per node in our experiments. We profiled the miniapp using ScoreP 2.0.2 [62]. The total execution time of the miniapp is broken down into: application (App) time, stall time and MPI time. Stall refers to the time spent on MPI_WaitAll and MPI time is time spent on all other MPI calls. We have measured performance variability in all the evaluated platforms by performing 5 trials of our experiments on each platform.

3.3.4 Experiment Setup for Graph500 benchmark

Implementations and Problem Size

The Graph500 benchmark provides four implementations: simple, replicated, replicated-csc, one-sided. All these four implementations use level-synchronized BFS, which means that all vertices at a given level of the BFS tree must be processed before any

TABLE 3.3: BFS time with different MPI ranks and OpenMP threads in TSUBAME2.5

# of MPI ranks	16	8	8	8	4	4	2	2	1	1
OpenMP threads	1	1	2	3	3	6	6	12	24	12
Total threads	16	8	16	24	12	24	12	24	24	12
BFS time	0.79	0.84	0.68	0.57	0.61	0.56	0.60	0.56	0.55	0.62

TABLE 3.4: BFS time with different MPI ranks and OpenMP threads in TSUBAME-KFC

# of MPI ranks	16	8	8	8	4	4	2	2	1	1
OpenMP threads	1	1	2	3	3	6	6	12	24	12
Total threads	16	8	16	24	12	24	12	24	24	12
BFS time	3.65	4.05	3.63	2.61	2.76	2.56	2.73	2.49	2.33	2.69

vertices from a lower level in the tree are processed. Each of these implementations has four phases: graph generation, graph construction, BFS, validation. The graph size is determined by two inputs: Scale and edgefactor. Scale is the logarithm base of the number of vertices and edgefactor is the ratio of the graph's edge count to its vertex count. The total number of vertices (N) and edges (M) are given by 2^{Scale} and edgefactor*N. There are different problem classes defined with respect to Scale parameter. Scale 26 comes under the problem class Toy (level 10) and it requires 10^{10} bytes and 17GB of memory.

We tested the Graph500 benchmark simple implementation and observed that each of its 64 BFS took longer than that of replicated_csr and replicated_csc implementations. This is because the simple implementation is single-threaded and thus BFS can't be parallelized. The Graph500 benchmark one-sided implementation makes use of Remote Memory Access (RMA) or one-sided communication introduced by MPI-2 standard. For MPI versions without RMA support, the implementation enables emulation of one-sided functionality with some overhead. For these reasons, we tested only the replicated_csr and replicated_csc implementations in our performance evaluation. Since both implementations use MPI and OpenMPI, we conducted experiments to determine the optimal number of MPI ranks and OpenMP threads on TSUBAME2.5, TSUBAME-KFC, and Amazon EC2 C4 instances. Each TSUBAME2.5 node has two sockets for 6-core Intel Xeon X5670 processors with hyperthreading enabled and each node in TSUBAME-KFC has dual sockets

TABLE 3.5: BFS time with different MPI ranks and OpenMP threads in AWS EC2 C4 instances

# of MPI ranks	16	16	8	8	8	4	4	2	2	1	1
OpenMP threads	1	2	4	2	1	9	4	9	18	36	18
Total threads	16	32	36	16	8	36	16	18	36	36	18
BFS time	1.52	1.52	1.43	1.53	2.79	1.34	1.51	1.34	1.33	1.24	1.44

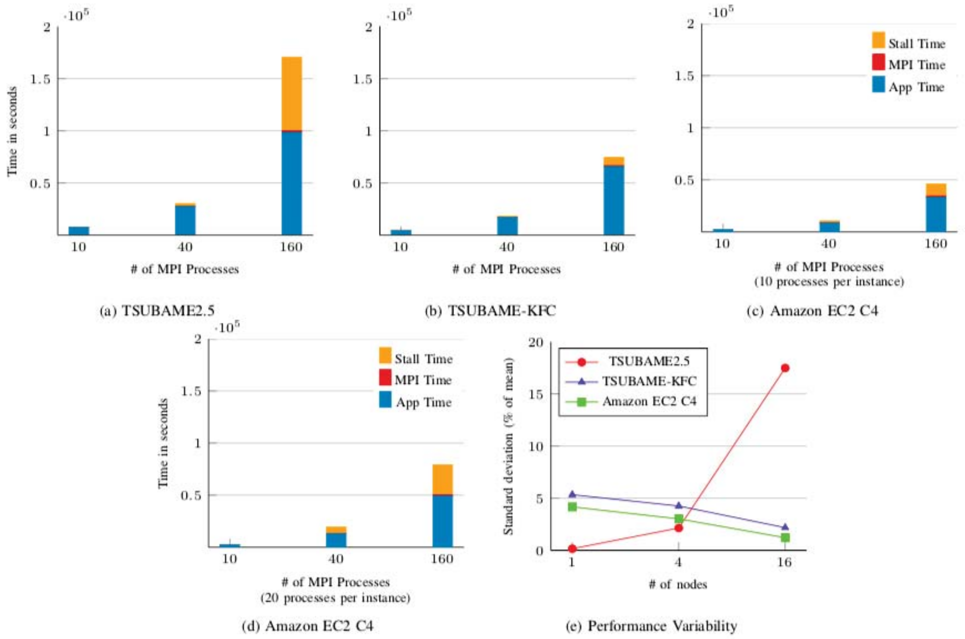
for 6-core Intel Xeon E5-2620v2 processors with hyperthreading enabled, so the total number of threads were 24 on both the platforms. We tested different MPI ranks and OpenMP threads with replicated_csr implementation on a node with Scale 24 on TSUBAME2.5 and with Scale 26 on TSUBAME-KFC as shown in Table 3.3 and 3.4. The results showed that 1 MPI rank and 24 threads achieved the best result for BFS time in both the environments. We used the same configuration for replicated_csc implementation on both the platforms.

Since Amazon EC2 C4.8xlarge instances has 36 virtual cores, we used different combinations of MPI ranks and OpenMP threads on one c4.8xlarge instance with Scale 26 as shown in Table 3.5. We repeated the experiments five times considering the performance variability caused by virtualization in account. The results are presented as an average of multiple runs which shows that 1 MPI rank and 36 OpenMP threads achieved the best BFS time. Also, we used these combination of MPI and OpenMP for running replicated_csc implementation on Amazon EC2 C4.

We used Graph500 benchmark version 2.1.4 and OpenMPI 1.6.5 for our experiments on the three evaluated platforms. We wanted to profile the execution time of the second kernel i.e. each of 64 breadth-first searches in each run. For that, we used mpiP 3.4.1 [29] which is a lightweight profiling library for MPI applications.The run time of BFS time is divided into two components: 1) App Time 2) MPI Time. MPI Time is the time spent on MPI calls while App Time is calculated by deducting MPI Time from the total time. We disabled the validation check at the end of each BFS run. For strong scaling experiments, we chose Scale 26 for our experiments since it's the largest scale that a node in TSUBAME2.5, TSUBAME-KFC and a C4 instance in AWS EC2 can handle. For weak scaling experiment, we used Scales 26, 27, 28, 29, 30, and 31 for 1, 2, 4, 8, 16, and 32 nodes, respectively. This ensures that the graph size handled by each node remains the same since the size of the graph doubles with each larger scale.

3.4 Results

3.4.1 NICAM-DC-MINI Results



(a) TSUBAME2.5 (b) TSUBAME-KFC (c) Amazon EC2 C4

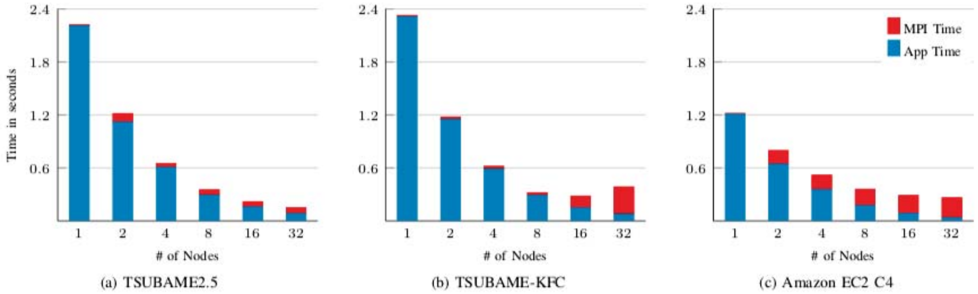
(d) Amazon EC2 C4 (e) Performance Variability

FIGURE 3.1: Execution breakdown for weak scaling experiments with NICAM-DC-MINI and performance variability on TSUBAME2.5, TSUBAME-KFC, and Amazon EC2 c4.8xlarge instances

Figure 3.1 shows the weak scaling results in all the three tested environments. The execution time on each system represents the average of 5 runs on that system. In case of 10 MPI process per c4.8xlarge instance in Figure **??**, Amazon EC2 C4 performance was almost 2x faster than TSUBAME-KFC. The total execution time was dominated by the application time which was the time spent on computation for running the simulation. TSUBAME2.5 was 2x slower than TSUBAME-KFC and up to 4x slower than Amazon EC2 C4 with 10 MPI process per instance. On running 20 MPI ranks per c4.8xlarge instance in Figure **??**, we found that the communication becomes the dominant factor with increased nodes. As we double the number of MPI process per instance, there is up to 4x increase in MPI time in Amazon EC2 C4. However, the compute performance by c4.8xlarge instances was still better than both TSUBAME2.5 and TSUBAME-KFC. The overall performance of TSUBAME2.5

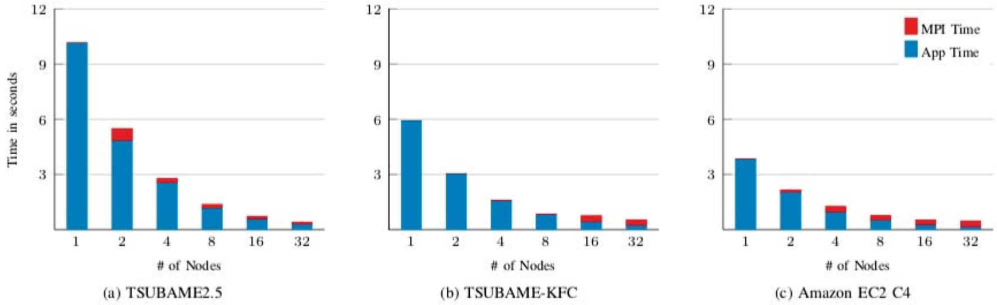
was up to 3x slower than using 20 MPI process per c4.8xlarge instance. The results in Figure **??** shows performance variability as percentage of ratio of the standard deviation to the average of total time on each platform. c4.8xlarge instances exhibit up to 4% standard deviation whereas TSUBAME-KFC was characterized by up to 5% performance variance. TSUBAME2.5 shows an increase in standard deviation as we increased MPI processes, with 17.5% for 16 nodes.

3.4.2 The Graph500 Benchmark Results



(a) TSUBAME2.5 (b) TSUBAME-KFC (c) Amazon EC2 C4

FIGURE 3.2: Execution breakdown for strong scaling experiments with Graph500 benchmark replicated_csr implementation on TSUBAME2.5, TSUBAME-KFC, and AWS EC2 c4.8xlarge instances



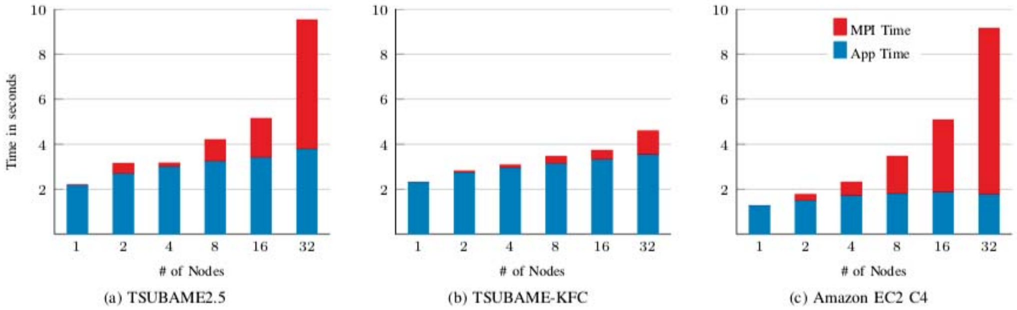
(a) TSUBAME2.5 (b) TSUBAME-KFC (c) Amazon EC2 C4

FIGURE 3.3: Execution breakdown for strong scaling experiments with Graph500 benchmark replicated_csc implementation on TSUBAME2.5, TSUBAME-KFC, and AWS EC2 c4.8xlarge instances

Strong Scaling Results

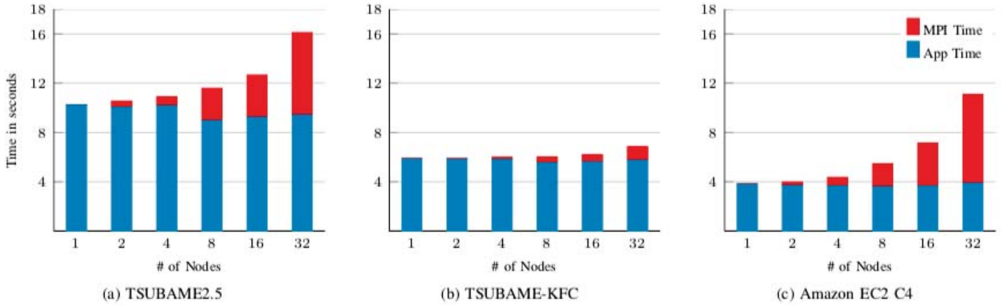
This section presents the strong scaling results for both replicated_csr and replicated_csc implementations. We ran our experiments using Scale 26 and the graphs are shown in Figures 3.2 and 3.3. Comparing replicated_csr on both the evaluated platforms, we can see that the BFS App Time is reduced to half with increasing number of nodes in all three environments. However, the MPI time is increased due to the increased communication among MPI processes in nodes. There is a significant MPI time increase in Amazon EC2 C4 instances than TSUBAME-2.5 and TSUBAME-KFC. The BFS App Time in Amazon EC2 C4 instances is almost half than on TSUBAME2.5 and TSUBAME-KFC. But, the increase in MPI time in Amazon EC2 C4 instances with increased nodes makes the total time comparable to the total time on both TSUBAME2.5 and TSUBAME-KFC. In replicated_csc implementation, the overall performance of Amazon EC2 C4 instance was slightly better than the others except for 32 nodes.

Weak scaling Results



(a) TSUBAME2.5 (b) TSUBAME-KFC (c) Amazon EC2 C4

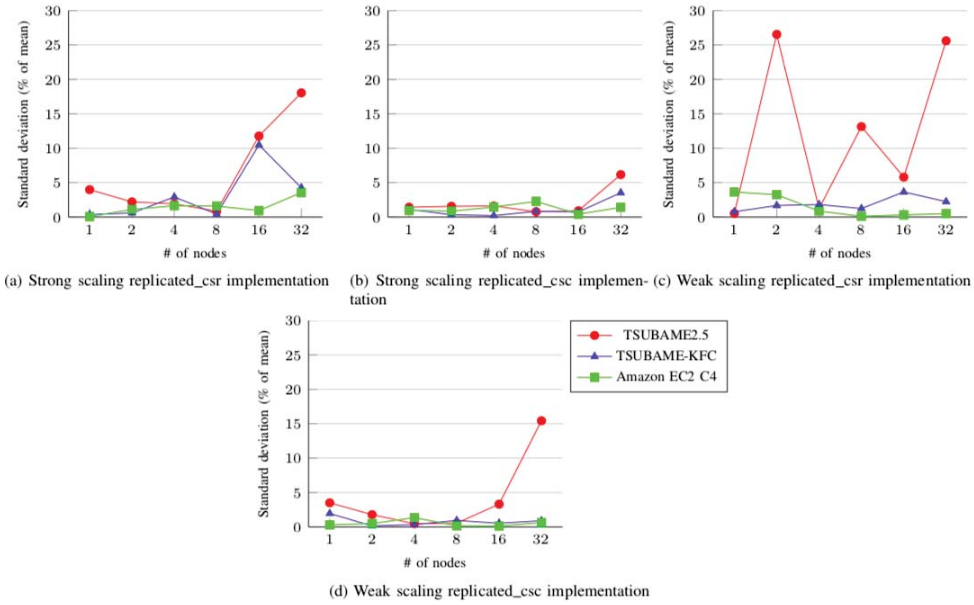
FIGURE 3.4: Execution breakdown for strong scaling experiments with Graph500 benchmark replicated_csr implementation on TSUBAME2.5, TSUBAME-KFC, and AWS EC2 c4.8xlarge instances



(a) TSUBAME2.5 (b) TSUBAME-KFC (c) Amazon EC2 C4

FIGURE 3.5: Execution breakdown for strong scaling experiments with Graph500 benchmark replicated_csc implementation on TSUBAME2.5, TSUBAME-KFC, and AWS EC2 c4.8xlarge instances

We present the weak scaling results of both replicated_csr and replicated_csc in this section. Results from these experiments are shown in Figures 3.4 and 3.5. In case of replicated_csr implementation, the execution time is increased as the number of nodes increases in TSUBAME2.5, TSUBAME-KFC and Amazon EC2. While there is considerable increase in BFS MPI time as the number of nodes increase in Amazon EC2, BFS App time in AWS is still lower than BFS App time in TSUBAME2.5 and TSUBAME-KFC. In case of using 16 and 32 nodes for Scale 30 and Scale 31 respectively, the communication becomes the dominant factor and AWS performance drops. However, the performance of these two scales on AWS is comparable to TSUBAME2.5 performance. In replicated_csc results, the performance pattern was almost similar to replicated_csr and there is significant MPI performance drop for 16 and 32 nodes in TSUBAME2.5.



(a) Strong scaling replicated_csr implementation (b) Strong scaling replicated_csc implementation (c) Weak scaling replicated_csr implementation

(d) Weak scaling replicated_csc implementation

FIGURE 3.6: Performance variability for strong and weak scaling experiments with Graph500 replicated_csr and replicated_csc implementations on TSUBAME2.5, TSUBAME-KFC, and Amazon EC2 c4.8xlarge instances

Figure 3.6 shows the performance variability for strong and weak scaling experiments with Graph500 replicated_csr and replicated_csc implementations on the three evaluated platforms. It corresponds to 5 trials of experiments on TSUBAME2.5 and 3 trials on both TSUBAME-KFC and Amazon EC2 C4. We could not conduct additional 2 trials on TSUBAME-KFC because of the recent system update. TSUBAME2.5 shows high variability with 16 and 32 nodes in almost all the cases. However, the graph of weak scaling experiments with replicated_csr implementation shows two peaks on using 2 and 32 nodes and a small peak on using 8 nodes in TSUBAME2.5. Our initial investigation attributed this behavior to variability in execution time for MPI_Allgather and MPI_Allreduce routines used in the BFS phase. Further study is needed to establish its root cause by using a detailed profiling tool. TSUBAME-KFC exhibit less than 5% variability in all the cases except for 16 nodes in strong scaling experiments with replicated_csr implementation whereas Amazon EC2 C4 instances show less than 5% variability for both the implementations.

3.5 Discussion

3.5.1 Application Time

The difference in application times among the three evaluated platforms can be characterized by different Intel processors' microarchitectures used by the systems. Haswell processors on Amazon EC2 C4 instances are successor to Ivy bridge and Westmere-EP microachitectures on TSUBAME-KFC and TSUBAME2.5 respectively. Haswell microachitecture includes support for Advance Vector Extensions version 2.0 (AVX2) as compared to AVX instruction on Ivy Bridge microarchitecture. Using 10 MPI processes per C4 instance, the application time was 2x faster and 3x faster than TSUBAME-KFC and TSUBAME2.5 respectively. However, with 20 MPI processes per C4 instance, the application time was slowed down by up to 1.5x. The performance on using 10 and 20 MPI processes on 1 instance is mostly the same. But, the increased communication induced by doubling MPI processes per instance while using 4 and more instances affected the performance. In Graph500 results, Amazon EC2 was 2x faster in application time than TSUBAME-KFC. For replicated_csr implementation, TSUBAME2.5 and TSUBAME-KFC show comparable performance but for replicated_csc implementation, TSUBAME-KFC was almost 2x faster than TSUBAME2.5.

3.5.2 Communication Time

While TSUBAME-KFC uses FDR Infiniband with the communication performance of a maximum of 54 Gbps, Amazon EC2 C4 instances offers 10 Gbps Ethernet network. As expected, TSUBAME-KFC outperforms Amazon EC2 C4 in both cases of using 10 and 20 MPI processes per C4 instance with NICAM-DC-MINI miniapp. TSUBAME2.5 with dual-rail QDR Infiniband network is expected to exhibit better performance than Amazon EC2 C4 instances. However, TSUBAME2.5's communication time for 16 nodes was 2.5x slower than Amazon EC2 C4 with 20 MPI processes per instance. We need further study to establish the reason behind this performance difference. In strong scaling results with Graph500 implementations, since BFS is parallelized by OpenMP threads and we use one MPI process per node and c4.8xlarge instance, the overall performance has lesser effect from communication time than non-MPI time. The performance of non-MPI time in Amazon EC2 C4 helps in achieving the overall performance comparable to TSUBAME2.5 and TSUBAME-KFC for replicated_csr implementation. In weak scaling tests, since the communication is getting dominant with larger nodes, Amazon EC2 C4 performance drops. TSUBAME2.5 and TSUBAME-KFC exhibit comparable communication performance in BFS MPI time in strong scaling results. However, it was not the same case with weak scaling results. Figures **??**, **??** and Figures **??**, **??** show the difference in BFS App to BFS MPI ratio with increasing number of nodes. We plan to look into the communication performance of TSUBAME2.5 to verify the results.

3.6 Summary

We performed the evaluation of an HPC mini-app, NICAM-DC-MINI and data-intensive benchmark, Graph500 on three systems: 1) our in-house production supercomputer TSUBAME2.5, 2) the energy-efficient TSUBAME-KFC supercomputer and 3) AWS EC2's latest generation of compute-optimized instances, C4. We conducted this study in an attempt to measure the capabilities of C4 instances for representative HPC and Big Data applications. Our finding demonstrates that Amazon EC2 C4 instances exhibit good compute performance with low performance variability. The 10 Gbps Ethernet network in these instances is the chief bottleneck for scaling the workloads. So, C4 instances can be a good fit for compute-intensive and memory-intensive application with little communication. We optimized the Graph500 benchmark for C4 instances by finding the optimal number of threads and MPI ranks and achieved performance comparable to our in-house supercomputers. Similarly, we achieved good performance using both 10 and 20 MPI ranks per C4 instance for NICAM-DC-MINI. Thus, C4 instances can be adopted as a replacement to on-premises hardware deployments for such workloads.

For future work, we want to perform a detailed analysis on low performance variability in C4 instances to see if it corresponds to customized Intel Haswell processors and/or less virtualization overheads. Also, we intend to run different workloads corresponding to different motifs (or dwarfs) and ogres on more diverse architectures and computing hardware from other public cloud providers.

Chapter 4

Predicting CPU Performance

4.1 Introduction

Modeling and predicting computer system performance has been and still is a crucial component in our road towards better, faster and more reliable computer systems. With Moore's law on its last legs and heterogeneity a part of our everyday life, predicting performance across systems is perhaps more important now than ever before.

Knowing how to extrapolate or project performance across systems is useful across the entire hardware stack, ranging from small embedded systems, all the way up to large distributed high-performance machines. Cloud-services such as Amazon [4] or Microsoft [39] can assess user requirements using predictive tools and deliver neither more nor less than what the customers and their applications require, subject only to the agreed-upon quality-of-service protocols. Cloud-users can similarly evaluate and predict their cloud needs and pinpoint the exact service to invest in. System administrators and users can (prior to purchase) approximate the impact of replacing system components. Software designers can similarly estimate how their applications run on next-generation hardware, whereas researchers use predictive models as a tool to see future trends and potential pitfalls.

Predicting or modeling performance has been researched ever since the 1960s [51]. Unfortunately, the vast majority of work attempts (with more or less success) to develop *system-specific* models that are unique to a particular system. These models require either detailed knowledge about hardware [48] (e.g. pipeline stacks, reorder buffer sizes, etc.) or rely on a cherry-picked set of hardware counters (e.g. [53]), which may or may not exist across architectures.

A different approach, which is also the approach we take and propose, is to leverage machine learning to build the predictive model. Unlike system-specific models, we put no formal requirement (although we do empirically evaluate its impact in section 4.2.3) regarding how many benchmarks to execute. The user provides existing data and our machine learning algorithm fills in the gaps. Our proposed approach allows users to remain oblivious regarding what data need to be extracted and its potential impact on the prediction. Unlike previous work, our approach allows us to be less obsessed by system-details, and instead let the machine learning process recognize the underlying features linking systems and applications together.

To summarize, we contribute with:

- Design and implementation of performance models based on two types of collaborative filtering (kNN and matrix factorization),
- Empirically evaluate our predictive models on a set of 30 benchmarks spanning three well-used benchmark suites and 10 different systems, showing that our predictive models can predict instructions per second (IPS) with RMSE as low as 0.6 and with an average RMSE of 1.6.

4.2 Motivation and Related work

Processor performance prediction has historically leveraged hardware performance counters to extrapolate expected application or system behavior. Numerous studies have collected large amount of hardware counter information and tried to model application behavior as a function of these. Early on modeling was performed by regression (linear or otherwise), while lately deep-learning is being applied. Given the large amount of research carried out with performance prediction in mind, what drives us to seek an alternative solution to the problem?

There are two main reasons why *any* method based on amassing information from hardware performance counters is sub-optimal. The first drawback is the labor involved in constructing the models, which typically involved running several microbenchmarks [53, 17, 2] that stress the system in different ways (that is, isolate various features), and then manually regressing the model to the data collected. It is a lengthy and tedious labor process that in the end only yields a *machine-specific* performance model. The second drawback is that the machine-specific models based on the hardware counter-based method are inapplicable across systems. The number and type of hardware performance counters can vary significantly between systems to the point where many proposed performance models are unusable on modern hardware. Furthermore, hardware counters are often unique to a particular system or instruction set architecture (ISA) and even counters named similar can have ambiguous meanings across ISAs, vendors and architectures.

Other methods exists, but are either orthogonal or complementary to our approach, or come with problems of their own. One approach is to simulate the performance on a hypothetical machine using simulators such as gem5 [44] or simics [50] as it was done for x86-64 Hammer prior to hard existing [16]). The simulators provides reasonable good accuracy and enough details to understand what is happening to the system. Unfortunately, they are also many orders of magnitudes slower and machine specification does not exist for all processors. Faster and more abstract analytic models such as LogP [13], Roofline model [54]) allow for fast estimation of performance, but are often hard to use without detailed knowledge of the algorithm at hand. Other methods include estimating performance inside the compiler [12] or using detailed and thus difficult to obtain machine specifications such as reorder buffer sizes and pipeline-depths [48]. We considered these methods complementary to the one we present in the paper, as they can be incorporated into our proposed method to potentially further improve accuracy.

4.2.1 Fast and Accurate Prediction

The key requirement to predict an application's performance across different systems is to identify the relationship between the application and system at hand and their inter-dependencies. Previous work limit themselves to predicting performance of a single system, which demands tedious and detailed characterization of applications on that system. The resulting performance model is often unique to that system and is not portable across other systems, forcing users with the task of repeating the characterization process for all systems they are interested in.

We focus on building a prediction model that does not require an application to be executed on all target systems. Many modeling-based schemes rely on executing the workloads on a target system multiple times and carefully handpicking the features to build the prediction model. As there could be tens or hundreds of thousands of such features in the data set, the analysis of such a large number of feature set is impossible. Feature selection algorithms which help users in selecting impactful features are not perfect, and even missing one crucial feature can lead to a bad performance model. Our aim is to develop a prediction model which leverages information it already has seen about the associations between applications and systems. The model then uses this information to express the performance of a new application on a target system as a combination of known applications. We build our prediction models using *collaborative filtering*.

4.2.2 Collaborative Filtering

Collaborative filtering techniques [3] are the most popular and widely implemented techniques in building recommender systems. They generate automatic predictions (filtering) about the interests of a user by collecting preferences or information from many users (collaborating) present in the system. One of the most publicized applications of collaborative filtering is the Netflix challenge. The goal of the challenge was to improve Netflix's movie recommendation system, Cinematch by providing valid movie recommendations to its users. The prize was won by BellKor's Pragmatic Chaos [64, 36, 8] after improving Cinematch by over 10% using a combination of different collaborative filtering techniques.

The collaborative filtering techniques are grouped in two general classes of neighborhood and model-based methods. In neighborhood-based collaborative filtering, the user-movie ratings stored in the system are directly used to predict ratings for new movies. It can be done in two ways known as user-based or movie-based collaborative filtering. User-based systems evaluate the interest of a user u given a movie m. The evaluation is performed by identifying neighbors of u that have similar rating pattern as u, and looking at how those very neighbors, called v, rated m. Identifying which neighbors belong to u is done by looking at how well shared items of u and v correlated with each other.

Movie-based approaches take a different stand and predict the rating of a user u for a movie m by looking at how u rated movies similar to m. In this way, two movies are similar if several users found them similar. In contrast to neighborhood-based methods, which use the stored ratings directly in the prediction, model-based approaches use these ratings to learn a predictive model. The core idea is to model the user-movie interactions with factors representing the latent characteristics of the users and movies in the system. The model is then trained using the available data and later used to predict missing ratings of users for movies.

4.2.3 Pilot study: Is collaborative filtering a good match for performance prediction?

While considering whether collaborative filtering can be applied to predict performance, we first began by making an analogy between movie recommendation and performance prediction. In movie recommendation scenario, the data sets with hundreds or thousands of users, movies and hundred of thousands of ratings are readily available. While for the latter, we can start with building performance database consisting of tens of systems and applications (which is a one time effort) and the database will grow over period of time. The movie recommendation systems rely on the assumption that each user has rated a small subset of movies (for user-based collaborative filtering) and each movie has been rated by at least a small subset of users (for movie-based collaborative filtering).

Removed ratings per user	**Sparsity**	**RMSE**
5	93.99%	0.974
10	94.28%	0.969
15	94.58%	0.976
18	94.76%	0.993
19	94.82%	1.011

TABLE 4.1: Neighborhood-based collaborative filtering using MovieLens data set used in the pilot study.

In our case, we need each application to be executed on a few systems in order to uncover its hidden associations with other applications and similarly for systems. We then conducted a pilot study using a data set by MovieLens [15] which contained 100k movie ratings (1-5) from 943 users and a selection of 1682 movies. The sparsity of the data set was 93.7% in which each user had rated at least 20 movies. We started looking at how the prediction accuracy will be affected if we have fewer ratings per user. We prepared five training sets by removing a few ratings per user and predicted the ratings using a neighborhood-based collaborative filtering model. Table 4.1 illustrates the removed ratings per user, sparsity of the training set and the prediction error. We used root mean squared error (RMSE) in equation 4.1 to measure the accuracy of predictions.

We began by removing five ratings per user from the data set and got the RMSE as low as 0.974. As we further increased the sparsity of the data set, the prediction error stayed almost the same. This shows that once there are enough ratings to identify the number of clusters of users (or movies) having similar preferences (or characteristics), the collaborative filtering model is able to predict with similar accuracy. Accordingly, in case of performance prediction, the number of applications and systems should represent a good coverage of scientific workloads and diverse architectures. Our hypothesis is that collaborative filtering can similarly work to predict performance (movie ratings) given a certain computer system (movie) and application (user).

4.3 Prediction Models

We implemented two collaborative filtering techniques to see how well an application will run across different hardware systems available. In this section, we begin with the formal description of the problem, population of the data set which is then followed by the description of the two collaborative filtering techniques.

4.3.1 Problem Formulation

Let us denote the number of applications by N_a and the number of systems by N_s. We then construct an application-system matrix M of size $N_a \times N_s$. Each cell in the matrix M has value as:

$$M_{as} = \begin{cases} p_{as}, & \text{known performance score} \\ 0, & \text{not known} \end{cases}$$

So, our goal is to predict all the zero entries of the matrix M. Let $\hat{p}_{as}$ be the predicted performance score of application a corresponding to system s. We divide the performance scores into a training set p_{train} which is used to learn and a test set p_{test} used to calculate the prediction accuracy. RMSE [57] is a de-facto method to measure accuracy of collaborative filtering algorithms. We used RMSE to evaluate the accuracy of our models as:

$$RMSE = \sqrt{\frac{1}{|p_{test}|} \sum_{p_{as} \in p_{test}} (\hat{p}_{as} - p_{as})^2} \tag{4.1}$$

4.3.2 Data Normalization

We need to normalize performance scores in matrix M to map these scores to a continuous scale. We use mean-centering normalization scheme. The rationale behind using mean-centering is to determine whether a given performance score is positive or negative by comparing it to the mean score. In application-mean-centered normalization, a raw performance score p_{as} is transformed to a mean-centered score $h(p_{as})$ by subtracting to p_{as} the average $\bar{p}_a$ of the known scores from running the application on systems:

$$h(p_{as}) = p_{as} - \bar{p}_a$$

In the same way, the system-mean-centered normalization of P_{as} is as follows:

$$h(p_{as}) = p_{as} - \bar{p}_s$$

4.3.3 Neighborhood-based Model

k-Nearest Neighbor (kNN) is an intuitive approach to implement collaborative filtering. For a given application and system, we want to predict its performance score. kNN works by identifying the k closest points (nearest neighbors) of either the application or system. The underlying idea is that similar applications (or systems) tend to cluster together based on their similar performance scores. This neighborhood approach is highly dependent on defining an appropriate similarity metric. We use Pearson correlation to measure the similarity between applications as well as systems. We implement two types of neighborhood-based collaborative filtering:

application-based collaborative filtering and system-based collaborative filtering.

Application-based collaborative filtering

We first construct a $N_a \times N_a$ matrix and populate the matrix by calculating similarity for each pair of applications using the Pearson correlation. The Pearson correlation coefficient, $sim(i, j)$ between any two applications i and j is a measure of strength of the linear relationship between them. The value of coefficient ranges from -1 to 1. A coefficient of -1 indicates a perfect negative linear relationship between applications, a coefficient value of 0 indicates no relationship while a coefficient of 1 indicates a positive linear relationship. It is calculated using the formula below:

Here, $S(a_1)$ is the set of all systems for which the performance score of application a_1 is known; $S(a_2)$ is the set of all systems for which the performance score of application a_2 is known; $S(a_1) \cap S(a_2)$ is a set of systems for which scores of both applications a_1 and a_2 are known.

We want to predict the performance score for an application a on a system s. We begin by finding the k most similar applications for which the performance scores on the system s is known. We denote these neighbors of application a by $N_s(a)$. Then, we find the predicted score by taking the weighted average using the performance scores of k applications and their similarity values w.r.t system s. The prediction of a performance score p_{as} using application-based collaborative filtering is obtained as:

$$\hat{p}_{as} = \bar{p}_a + \frac{\sum\limits_{k \in N_s(a)} sim(a,k)(p_{ks} - \bar{p}_k)}{\sum\limits_{k \in N_s(a)} |sim(a,k)|}$$

The prediction accuracy depends on k, the number of nearest neighbors. So, we try different value of k in the training set and evaluate the corresponding performance by measuring the RMSE.

System-based collaborative filtering

We construct a $N_s \times N_s$ similarity matrix for N_s number of systems. We calculate the similarity values between each pair of systems and then apply kNN method to predict the performance score as:

$$\hat{p}_{as} = \bar{p}_s + \frac{\sum\limits_{k \in N_a(s)} sim(s,k)(p_{ak} - \bar{p}_k)}{\sum\limits_{k \in N_a(s)} |sim(s,k)|}$$

4.3.4 Matrix Factorization Model

Matrix factorization models are the most popular implementations of model-based collaborative filtering. The core idea is to uncover latent features or factors among applications and systems that explain the known performance scores. These models map both applications and systems to a joint latent factor space of dimensionality r, such that application-system interactions are modeled as inner products in that space.

The input to the model is a sparse matrix A, with one row per application and one column per system. The first step in a matrix factorization technique is to decompose A into U, λ and V such that:

$$A \approx U \lambda V^T$$

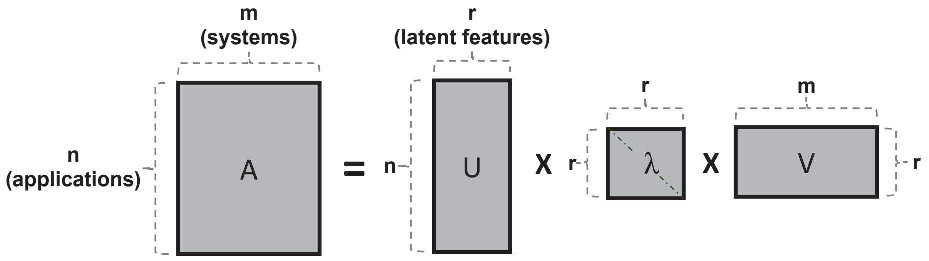


FIGURE 4.1: Factorization of matrix A into sub matrices U, λ and V using the latent features.

Given the matrix A (n applications, m systems), we factorize it into $n \times r$ matrix U (n applications, r latent features), $r \times r$ diagonal matrix λ (strength of each latent feature), and $m \times r$ matrix V (m systems, r latent features). Figure 4.1 illustrates this idea. The λ diagonal matrix contains the similarity concepts identified by the matrix factorization technique. The U matrix is interpreted as the application-to-feature similarity matrix, while the V matrix is the system-to-feature similarity matrix. For instance, two or more applications can be similar in one latent feature (they both benefit from a large L3 cache) but different in others (only one benefits from high clock frequency). These latent features contain the singular values which are sorted in decreasing order. The application-to-feature matrix represents the strength of the associations between an application and the r latent features. In this way, each application is described using a r-dimensional space. Similarly, the system-to-feature matrix identifies each system using a r-dimensional space.

Let us denote $Q_{n\times r} = U$ and $P^T_{r\times m} = \lambda V^T$. The prediction of an application a on a system s can be calculated as the dot products as their vectors:

$$\begin{aligned}\hat{p}_{as} &= q_s.p_a^T \\ &= \sum_r q_{sr}.p_{ar}\end{aligned}$$

Now we may obtain P and Q by applying any dimensionality reduction technique such as Singular Value Decomposition (SVD) to the matrix A. In our work, however, we use Stochastic Gradient Descent (SGD) [31] to iteratively optimize pairs of p_i and q_j corresponding to existing entries a_{ij}. This approach allows us to add new systems or benchmarks when needed and re-train only corresponding rows of P or columns of Q matrices. We can reformulate our model as an optimization problem in which we have to find matrices P and Q such that:

$$\min_{P,Q} \sum_{training} (p_{as} - \hat{p}_{as})^2 + \beta \left[\sum_a ||p_a||^2 + \sum_s ||q_s||^2 \right]$$

After each step, e_{as}, q_s and p_a are updated as follows:

$$\begin{aligned} e_{as} &\leftarrow p_{as} - \hat{p}_{as} \\ q_s &\leftarrow q_s + \alpha(e_{as} \cdot p_a - \beta \cdot q_s) \\ p_u &\leftarrow p_u + \alpha(e_{as} \cdot q_s - \beta \cdot p_u) \end{aligned}$$

Where α is learning rate and β is regularization parameter. The result of SGD is the dense $Q.P^T$ matrix using which we construct the missing performance scores in matrix A. When evaluating this model on our data set, we used α as 0.01 and β as 0.001.

Dwarf	Workloads
Dense Linear Algebra	Alignment, SparseLU, Kmeans, LU Decomposition (LUD), Nearest Neighbour (NN), Stream Cluster (SC), Block Tri-diagonal solver (BT), Lower-Upper Gauss-Seidel Solver(LU)
Structured Grid	Multi-Grid (MG), Scalar Penta-diagonal solver (SP), Heartwall, Hotspot2D, Hotspot3D, Leukocyte Tracking, Myocyte, SRAD
Graph Traversal	Integer Sort (IS), Fibonacci, Sort, UTS, BFS, Health
Sparse Linear Algebra	Conjugate Gradient (CG)
Spectral Methods	3D Fast Fourier Transform (FT)
Unstructured Grid	CFD Solver (CFD)
Map Reduce	Embarassingly Parallel (EP)
N-Body Methods	LavaMD
Dynamic Programming	Needleman-Wunsh
Backtrack and Branch-and-Bound	NQueens, Floorplan

TABLE 4.2: Workloads

4.4 Experimental Setup

4.4.1 Systems

We evaluated the models using ten diverse systems. We selected architectures that are well-known in both high-performance computing (HPC) and cloud infrastructure (x86-64 Xeon's, Xeon Phi's), embedded computing (ARM Cortex, Intel Atom) and general-purpose stationary systems (Intel i7). Note that the gap between HPC and embedded systems domains is shrinking, and for example the expected RIKEN Post-K [49] HPC system will amass ARM processing cores, which also motivated us to include this family of processors. The configurations of these systems are shown in Table 4.3. The heterogeneity of the machines is with respect to ISA (ARM vs x86-64), interconnect/memory hierarchy (Xeon Phi vs ARM vs Xeon) and power consumption (embedded vs desktop vs server).

4.4.2 Workloads

We used 30 workloads chosen from three well-known benchmark suites: The NAS Parallel Benchmarks (NPB) version 3.3.1 [43], Barcelona OpenMP Task Suite (BOTS)

[1] MCDRAM configured as last-level cache in cache mode.

System	ghz	sockets	cores	threadspercore	mem(gb)	LLC(MB)
Xeon E52650 v3	2.30	2	10	2	251	26
Xeon E52650 v4	2.20	2	12	2	251	31
Xeon X5650	2.67	2	6	2	46	12
Xeon E5-2699 v3	2.30	1	18	2	125	46
Xeon Phi 7295	1.50	1	72	4	94	16384[1]
Xeon Phi 7210F	1.30	1	64	4	94	16384[1]
i7-3930K	3.20	1	6	2	15	12
i7-3770K	3.50	1	8	2	15	8
Atom C2750	2.40	1	8	1	7.8	1
ARM Cortex-A7	0.90	1	4	1	0.9	0.3

TABLE 4.3: Specifications of the systems used.

[11] and Rodinia benchmark suite [55] version 3.1. Table 4.2 illustrates the workloads and their corresponding domains and dwarfs. These workloads represent ten classes of numerical methods that are widely used in scientific applications. We managed to compile eight benchmarks from NPB, eight benchmarks from BOTS, six kernels and eight applications from Rodinia on all the ten systems. Two benchmarks from BOTS (fft and strassen) and four kernels from Rodinia (backprop, b+tree, particlefilter and streamcluster) were not included as we came across compilation issues and segmentation faults on the ARM board.

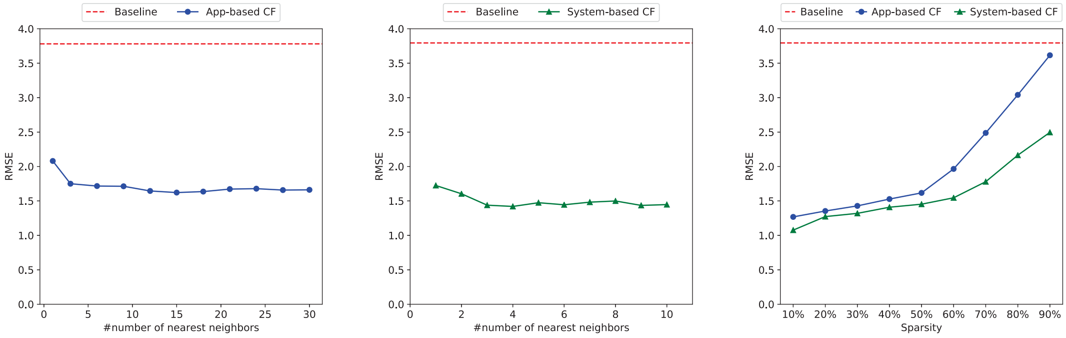
The workloads were compiled using GCC versions 6.3.0 (ARM), 5.4.0 (Intel Atom) and linked against the GCC OpenMP library (libgomp). Each workload was executed five times and the mean was taken to represent its performance. IPS (and related counters) were obtained using Linux `perf`. All applications were executed in isolation. We executed all the workloads serially as well as in parallel. For serial runs, each of the workloads was executed with 1 thread. In parallel runs, we allow each application unrestricted access to all of the processor's threads (including hyper-threads).

4.4.3 Performance Scores

We populate M with normalized scores that represents how well an application performs on a system. We use the number of instructions per second (IPS), a function of instructions per cycle (IPC, a well-known proxy for performance [34, 56, 30, 1]) and the processor clock-frequency.

Execution mode	Minimum	Maximum	Mean
Singlethreaded	0.05×10^9	13.07×10^9	3.78×10^9
Multithreaded	0.05×10^{10}	22.99×10^{10}	3.48×10^{10}

TABLE 4.4: Range of IPS values in our data set.



(A) Application-based CF (B) System-based CF (C) Sparsity vs. RMSE

FIGURE 4.2: kNN collaborative filtering with performance data corresponding to serial execution of workloads.

For the end-user, predicting application execution time is the ultimate goal. However, execution time is a poor candidate to predict simply because it can drastically vary between applications. IPS allows us to estimate runtime since the number of instructions often remains constant. Furthermore, for applications that never end (such as services), predicting execution time is impossible while predicting IPS is not. Additionally, IPS is useful by itself, as it characterizes system utilization. The range of IPS values in our data set is shown in table 4.4. We pre-process the performance scores corresponding to serial execution by dividing each IPS value by 10^9. The new scores then lie within [0.05, 13.07] and 3.78 is the mean value of the performance data. Similarly, for multithreaded execution, we divide each score by 10^{10} which results in new range of IPS values i.e. [0.05, 22.99] with 3.48 as the mean.

4.5 Results

In this section, we present and discuss the results of both collaborative filtering based models.

4.5.1 kNN

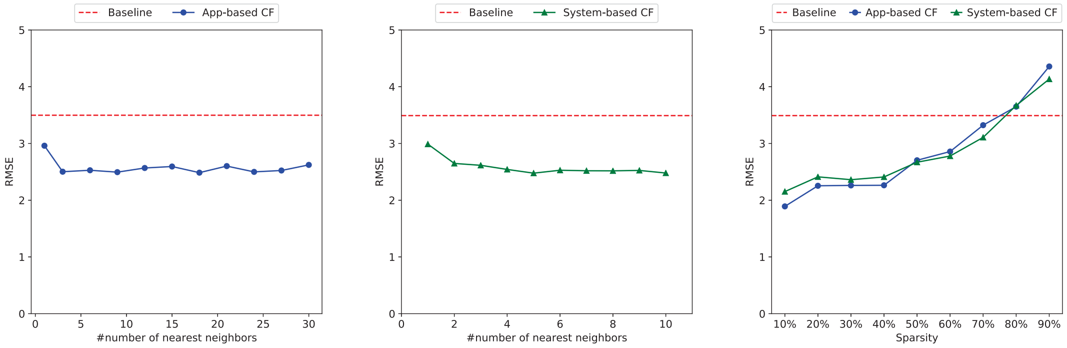
We present the results of using neighborhood-based collaborative filtering model on our data set. We used kNN as explained in section 4.3.3 on both singlethreaded (ST) and multithreaded (MT) execution of workloads on systems. Figure 4.2 illustrates the plots corresponding to ST execution. Figures 4.2a and 4.2b show the number of nearest neighbors on the x-axis and RMSE on the y-axis in application-based collaborative filtering (CF) and system-based CF respectively. The baseline shows the mean value from the training data set in each case and the sparsity of training data set is 50% in figures 4.2a, 4.2b and 4.3a, 4.3b.

The model aims to characterize the performance of a workload w on a system s using one parameter, k (number of nearest neighbors). In case of application-based CF, kNN finds the top-k most similar workloads to w for which the performance scores on the system s are known, for different values of k. As our data set consists of 30 workloads, the value of k ranges from 1 to 30. Even when $k = 1$, the model finds the most similar workload to w and predicts its performance score on system s with 45% and 55% improvement over the baseline for application-based and system-based CF respectively. We tested RMSE for different values of k and found that $k = 15$ gives the best RMSE in case of application-based CF.

In case of system-based CF, kNN finds the top-k most similar systems to s for which the performance scores on the workload w are known. The k varies from 1 to 10 for the total number of systems in our data set. $k = 4$ achieves the best RMSE in figure 4.2b. In both cases, initially the prediction improves as the model is considering additional neighbors of application (or system) and then remains almost the same after it has discovered groups of similar applications (systems).

Figure 4.2c shows the sparsity vs RMSE when $k = 15$ in application-based CF and $k = 4$ for system-based CF. As the sparsity of data set increases, the prediction suffers as kNN is not able to find similar workloads (or systems) and there is not enough information to make a good prediction. Also, system-based CF is able to make better predictions than application-based CF. Each workload w_i for $i = 1, 2, ..., 30$ can be characterized by a tuple with 10 elements, $w_i(s_1, s_2, ..., s_{10})$ where each element corresponds to one of the ten systems. Whereas, each system s_j where $j = 1, 2, ..., 10$

can be characterized by a tuple with 30 elements, $s_j(w_1, w_2, ..., w_{30})$ for a total of 30 workloads. With more workloads than systems, each system tends to have more performance scores than each workload. So, the system-based collaborative filtering method could better capture the similarities among systems even when the sparsity was 90%.

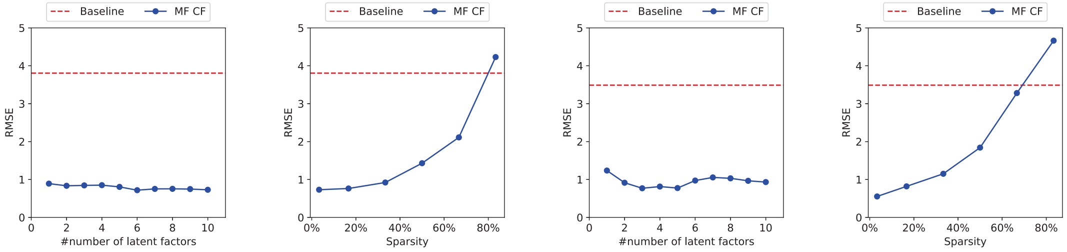


(A) Application-based CF

(B) System-based CF

(C) Sparsity vs. RMSE

FIGURE 4.3: kNN collaborative filtering with performance data corresponding to multithreaded execution of workloads.



(A) Factors vs. RMSE, ST

(B) Sparsity vs. RMSE, ST

(C) Factors vs. RMSE, MT

(D) Sparsity vs. RMSE, MT

FIGURE 4.4: MF collaborative filtering with performance data corresponding to serial and multithreaded execution of workloads.

Figure 4.3 shows the plots corresponding to MT execution of workloads. The best RMSE was observed for $k = 18$ in application-based CF in figure 4.3a. For system-based CF, $k = 5$ resulted in the best RMSE. Figure 4.3c shows sparsity vs RMSE for $k = 18$ in application-based CF and $k = 5$ in system-based CF. In this case, application-based collaborative filtering makes better prediction than the other when the sparsity of data set is below 50%. This signifies that the model identifies more similar neighbors of workloads than systems. The RMSE for both approaches meets the baseline when the sparsity is 70% and further the prediction error increases.

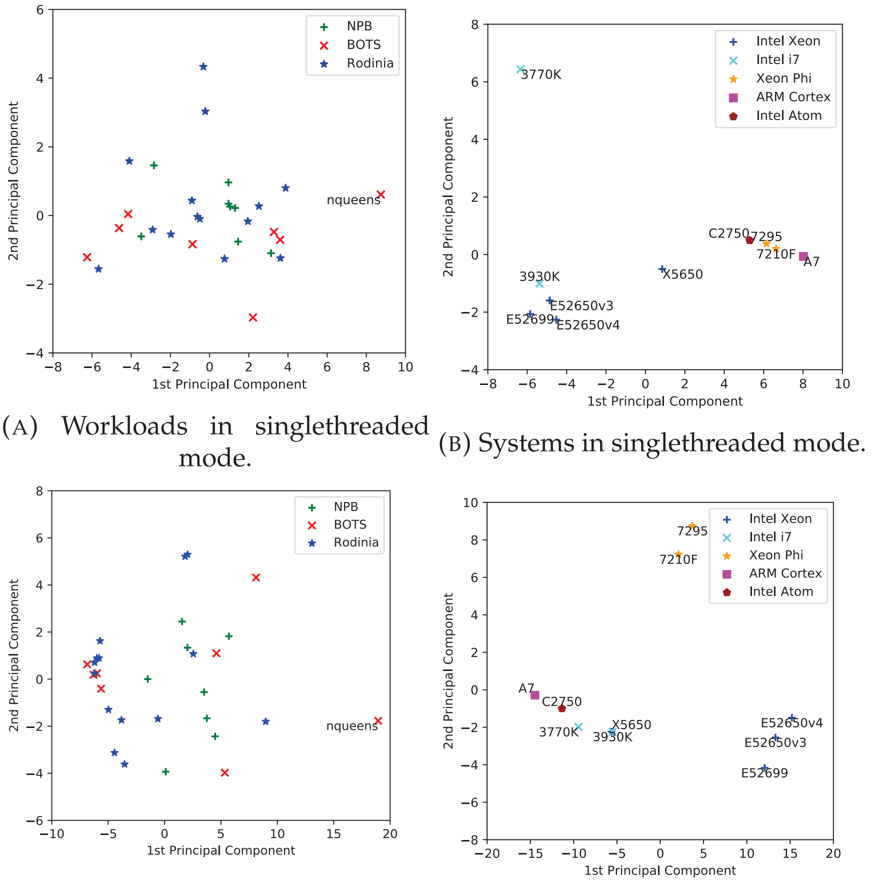
The different range of IPS values for ST and MT execution contributes to the difference observed in RMSE values for both the executions. We also want to point out that the multithreaded execution of workloads involves far more elements of uncertainty (false sharing, sub-linear scalability, lock-contention, runtime overheads etc.) that needs to be considered in future work.

We conclude by stating that our kNN model can predict IPS with RMSE as low as 1.8 even when only 30% of the data is known, which will manifest itself in that users only need to profile $\frac{1}{3}$ of their applications to use our model. Indeed, our results point towards the need to integrate more metrics when predicting performance using kNN in order to obtain even better accuracy. We plan to tackle it in future work.

4.5.2 Matrix Factorization

We present the results of matrix factorization based collaborative filtering in this section. Figure 4.4 illustrates the plots corresponding to ST and MT execution of workloads respectively. Figure 4.4a shows the prediction accuracy on varying the number of latent factors. The number of latent features is smaller than the number of workloads and systems such that each latent feature can describe relationship among two or more workloads (or systems). Since there are 30 workloads and 10 systems used, the number of latent features can range from 1 to 10. The model learns the number of latent features represented at x-axis automatically and uses them to predict IPS for the workloads and systems. Even just using one latent feature performs better than kNN and results in lower RMSE. The model automatically identifies six such latent features as shown in figure 4.4a which results in the lowest error.

Figure 4.4b shows the error vs sparsity of the dataset. The best RMSE is achieved with 10% sparsity i.e. when the matrix is 90% dense. As sparsity of the data set grows, RMSE increases. Similar results are observed for MT execution as shown in figures 4.4c and 4.4d. In figure 4.4c, initially the error decreases as the model is making predictions by adding new latent features. However, the error starts increasing when the model uses more than five such features. This is caused by overfitting of the training data. Our matrix factorization based model generated better predictions than kNN with 32% and 49% improvement over its predictions for ST and MT respectively.



(A) Workloads in singlethreaded mode.

(B) Systems in singlethreaded mode.

(C) Workloads in multithreaded mode.

(D) Systems in multithreaded mode.

FIGURE 4.5: Variance in performance data set w.r.t workloads and systems using two principal components.

4.5.3 Analysis of Learned Representations

We analyze learned representations (latent features) to validate that our model could indeed learn meaningful features, as well as to show how these representations can be used to compare systems and benchmarks.

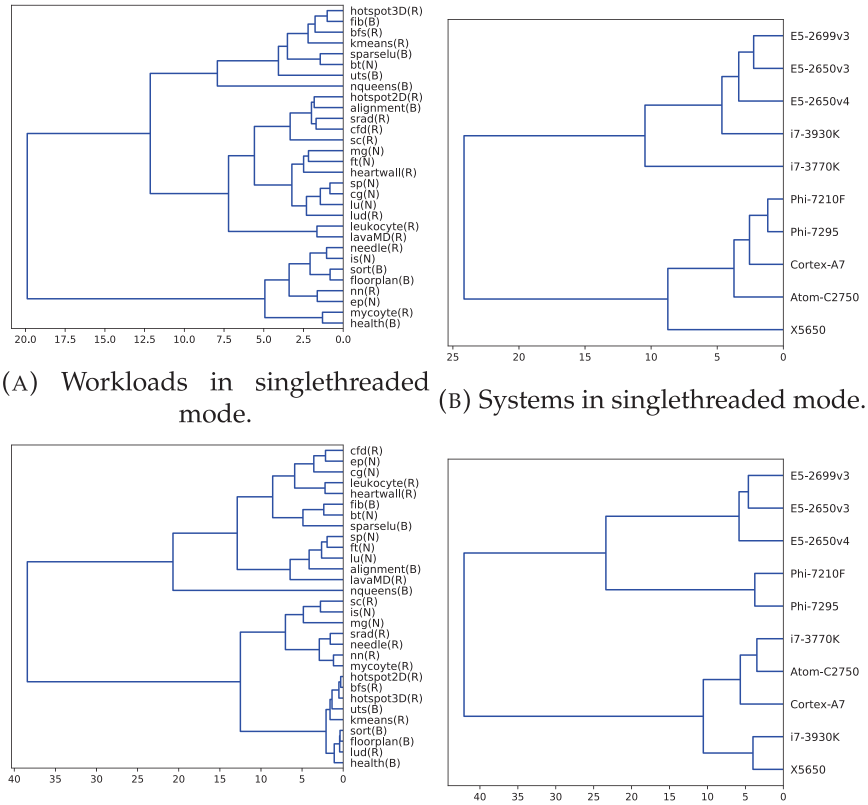
Principal Component Analysis

Principal Component Analysis (PCA) is a statistical method to find patterns in high-dimensional data sets. PCA computes n new variables, called principal components, which are linear combinations of n original variables, such that all principal components are uncorrelated. The amount of variance captured by the first component is larger than the amount of variance on the second component and so on. The dimensionality of the data set is reduced by neglecting those components with a small contribution to the variance. We used PCA to find equivalence classes of workloads as well as systems with similar characteristics.

Figure 4.5 shows two-dimensional PCA projection of learned latent features of workloads and systems for ST and MT execution respectively. Workloads which exhibit similar performance dynamics on a set of hardware platforms are grouped together in one or both dimensions. Figure 4.5a shows the workload mix using NPB, BOTS and Rodinia benchmark suites. These benchmark suites form small clusters with every cluster having similar workloads where `nqueens` from BOTS is the outlier. Since the PCA components are constructed by using one metric i.e. IPS, we think that other metrics can be used to cover the crucial differences among workloads. Also, more benchmark suites or real-world workloads can be used to ensure a wider application coverage. Selecting performance metrics to fully represent a set of workloads is a research challenge on its own and is outside the scope of this work. This work is focused on studying collaborative filtering methods to analytically characterize the performance by using a given set of workloads, systems and performance metric.

In Figure 4.5b, the systems show interesting behaviors on being grouped together based on similar performance characteristics for a set of workloads. The three Intel Xeon E5 processors used in the study are projected closer to each other while Xeon X5650, a legacy processor shows disparate performance. The Intel i7 lies closer to the Xeons while its Atom processor takes the farthest area in the space with the two Xeon Phi and ARM processor. The serial performance of the two Xeon Phi and Atom processor matches with that of ARM Cortex A7. This explains how the model is able to characterize performance of a completely different ARM processor having a different ISA using the performance seen on Intel Knights Landing, Knights Mill and Atom processors.

In Figure 4.5c, the workloads form two groups while `nqueens` remains the outlier. Most kernels and applications from Rodinia are grouped together occupying the leftmost area in the space while most benchmarks from NPB are in the middle area. Whereas benchmarks from BOTS are evenly distributed between Rodinia and NPB spaces. A thorough examination with other multithreaded benchmarks along with representative performance metrics is required to establish a single set of



(A) Workloads in singlethreaded mode.

(B) Systems in singlethreaded mode.

(C) Workloads in multithreaded mode.

(D) Systems in multithreaded mode.

FIGURE 4.6: Dendrograms showing similarity between workloads and systems for two different executions.

workloads with sufficient coverage. In figure 4.5d, PCA forms three groups of systems with similar performance characteristics for a set of workloads. One of them is all Intel Xeons, other is the two Xeon Phis and the last is with Intel i7, Intel Atom, the legacy Intel X5650 and ARM Cortex. The multithreaded performance of Intel Knights Landing and Knights Mill processors separate them from ARM and Intel Atom as earlier observed in PCA system components for ST execution in figure 4.5b. While the Intel Xeons are still grouped togther as they were earlier, the two Intel i7 processors' performance aligns with that of ARM and Intel X5650.

Hierarchical Clustering

Here we present the hierarchical clustering which discovers meaningful clusters that exist in the data. It assigns data objects to groups so that the objects in the same groups are more similar than objects in different groups. The similarity between every pair of data objects in the data set is determined using distance measures. Hierarchical clustering algorithms successively cluster objects within found clusters, producing a set of nested cluster organized as a hierarchial tree (dendrogram). We used SciPy hierarchical clustering method, linkage to perform the hierarchical clustering on our data set consisting of performance score for each pair of workload and system.

Figure 4.6a shows the three benchmark suites for ST execution. The vertical lines indicate the cluster merges while the horizontal line shows the labels (workloads) or clusters that were a part of the merge forming the new cluster. The length of the horizontal line represents the distance that needed to be bridged to form the new cluster. Thus, the magnitude of the link between any two labels (or clusters of labels) shows the measure of dissimilarity between those labels. The serial performance of *Mycoyte* and *Health* are fairly similar, while that of *Nqueens* differs significantly from the these two workloads. While *LavaMD* and *Needle* are spatially close but they belong to two different clusters in the figure. They are dissimilar than *LUD* and *Hostspot2D*. It is also evident from this dendrogram that these three benchmark suites cover different application spaces, with most clusters containing workloads from NPB, BOTS and Rodinia. Figure 4.6b shows the dendrogram for systems for ST execution of workloads. It depicts how all Xeon E5 and i7-3930K form one cluster while Xeon Phis, ARM Cortex, Atom are grouped in a different cluster. The serial performance of Intel X5650 is more similar to the cluster having Xeon Phis. This matches with the system mix plot shown using PCA components in figure 4.5b.

Figures 4.6c and 4.6d show the dendrograms for workloads and systems corresponding to MT execution. As compared to dendrogram obtained for ST execution in figure 4.6a, this has roughly two major clusters while *Nqueens* is still an outlier. The first cluster (spanning from *CFD* to *LavaMD*) shows a good mix of workloads with has 6 NPB, 3 BOTS, 4 Rodinia workloads, while the other (spanning from *Stream Cluster* to *Health*) has 2 NPB, 4 BOTS, 10 Rodinia workloads. We can verify how most Rodinia workloads are placed towards the left in the workload mix plot using PCA in figure 4.5c. The MT performance of these benchmark suites on systems shows a little bit different associations in figure 4.6d. All Intel Xeons E5 form a cluster while Xeon Phis form a separate cluster. ARM Cortex now aligns with Atom and i7-3770K processor, while the legacy processor X5650 is grouped with i7-3930K.

4.6 Conclusions

We have presented two analytical methods to predict the performance of both serial and parallel applications. These methods are based on two collaborative filtering algorithms, kNN and Matrix Factorization. kNN finds similarity among workloads (application-based collaborative filtering) and systems (system-based collaborative filtering) using Pearson correlation and uses the similarity scores to characterize the performance among other workloads and systems. Matrix factorization methods learns latent features that explain the associations among workloads and systems and the strength of every such association to make accurate predictions.

We evaluated both methods using NPB, BOTS and Rodinia workloads on ten varied systems ranging from x86 to ARM instruction set architectures. We show that both kNN and Matrix Factorization methods can accurately estimate the performance of workloads with even when the data set is up to 70% sparse. While kNN is simple and intuitive to implement, Matrix Factorization uncovers latent features among workloads and systems and makes better predictions.

These collaborative filtering methods prove to be useful in identifying suitable system(s) favourable for specific workloads. Also, these methods can complement other simulation-based prediction methods which can then characterize the performance of a system selected using such collaborative filtering approaches. Additionally we demonstrate that benchmarks and systems themselves can be analyzed and compared using feature vectors which are obtained as a by-product of matrix factorization method. Using techniques like principal component analysis and clustering, it is easy to find benchmarks and systems with similar or alternatively, abnormal performance characteristics.

In future work, we will investigate our approach for a wider application space and different systems. Also, we will identify a set of performance metrics that capture different performance components and use them to improve our analytical models.

Chapter 5

Predicting GPU Performance

5.1 Introduction

Graphics Processing Units (GPUs) are today the de-facto source of performance in High-Performance Computing (HPC), and the vast majority of current top supercomputers [60] include them in their system setup. These powerful devices are explicit vector machines, whose programming model allows the programmer to leverage the large amount of parallelism they offer. Unlike general-purpose processors, which focus on exploiting instruction-level parallelism (and focus on latency), GPUs focus on thread-level parallelism, masking/hiding long-latency operations (e.g. external memory accesses) by time-sharing computational resources. Despite their similar programming model, GPUs are constantly undergoing architectural changes across generation (e.g. mixed-precision Arithmetic Logic Units, per thread program counters, diverse amount of floating-point units), which makes their performance non-trivial to reason around and *predict*.

Performance prediction is (and will continue to be) a core pillar in computer science, and is used to assess the performance (or other metrics such as power-consumption) of a (non-) fictional system prior to acquisition. The usage of predicting system performance ranges from private user's reasoning around which cloud solution fits their performance and price-budget best, to HPC system administrators understanding what components to expand their system with, all the way to researchers attempting to map and reason around performance trends and directions. And with the end of Moore's law near [14], prediction and understanding performance is more crucial than ever.

Predicting GPU performance is a challenging and hard task. Despite sharing a programming model (CUDA [45] and OpenCL [58]), their architectural differences between generations can be substantial. Furthermore, with the advent and popularity of Deep-Learning, the type and target audience of GPUs is diversifying. As a result, GPUs specialized in inference, training, gaming, and scientific computing are emerging [46]. Given this vast array of current and emerging GPU types, *how do users choose which to invest in?* Today, most users blindly buy the fastest next-generation accelerator for their workload, which is more than likely not the most optimal choice. There is a need for a *simple* and effective performance model that assist users in choosing accelerators suitable for the workloads they care about.

Existing methods to predict GPU performance are either constrained by the programming environment (and the necessity of mapping algorithms to existing GPU features) [32, 19], or based on compiler-based approaches [10] to extract GPU-specific micro-architectural features. Such methods often work well on the targeted GPU, but are inapplicable across GPU types and architectures as these methods are *system-specific*. A different approach based on collaborative filtering (CF) was recently proposed by Salaria et al.[52], and was shown to work well on general-purpose processors (CPUs) that were diverse in both instruction set architecture (ISA) and architecture, even when the sampling data was sparse. As their approach was limited to CPUs, we ask ourselves the question whether CF-based methods can also work on the more challenging (compared to their evaluated CPUs) GPUs.

In this paper, we evaluate the CF-based prediction methods on GPU-based systems. We focus on building a prediction model that does not burden users to needlessly execute an application on all target systems. Manually defining which program properties are crucial to describe its performance on a given system is critical to build an accurate model. Our aim is to develop a model that leverages information it already has seen about the associations between applications and systems. The model then uses this information to express the performance of another application on a target system as a combination of known applications.

The main contributions of this work are as follows:

1. We show that CF can be used to capture and predict performance for GPUs.

2. We introduce a neural network architecture to learn representations of applications and systems and test whether using auxiliary training objectives can further improve its predictive power.

3. We evaluate and analyze our performance model empirically on a large and diverse set of well-known benchmarks and multiple generations of GPUs, quantifying the prediction accuracy.

5.2 Motivation and Related work

In this section, we introduce topics relevant to this work, and discuss prior work related to them.

5.2.1 Explicit Features

The performance of a program on a given system is guided by the complex interactions between the program's properties and system's characteristics. The most common approach taken towards building analytical models is to first perform a detailed characterization of applications on systems and collect various performance metrics such as execution time, hardware counters [63, 24]. Second, it needs to explicitly list all the features that can capture the inter-dependencies between the application and system at hand such as clock rate, cache size, floating point operations per second (FLOPS) etc. Most previous works use an intuitive approach when selecting the number of features in order to determine a set of good explanatory features. However, missing just one crucial feature from these carefully handpicked features can greatly (and negatively) affect the prediction accuracy. Third, we need to test each possible subset of those features finding the one which minimizes the error. This is an exhaustive search of the feature space, and is computationally expensive and unfeasible for all but the smallest of feature sets. Feature selection algorithms come handy when it comes to finding the most impactful feature subsets but it comes with an extra effort of selecting the appropriate algorithm and its parameters.

While considering tens or more of applications and systems, manually defining features for each set of application and system is practically impossible. For building a cross-architecture predictive model – such as the one we are targeting – we focus on two things: 1) We do not want to manually define feature for each application and system 2) we want our model to learn the features automatically from the known performance of a subset of applications executed on a subset of systems and leverage this information to predict performance.

5.2.2 Representation Learning

The concept of representation learning is grounded in the idea that often the information needed to characterize or classify high dimensional data can be found in a low-dimensional manifold, or mapped into a dense vector. For instance, natural language processing systems use word embeddings to represent (embed) words in a continuous vector space where semantically similar words have similar vector representations (embeddings)[40]. Recommender systems employ similar representations to describe its entities (e.g. users and items) and call them as *latent features*.

We assume that there is a number of important features, called latent features, which characterize systems and applications. We use a machine learning model to learn these features as a by-product of predicting known runtime metrics. In our model, we have application and system feature vectors. Two or more applications can be similar in one latent feature (they both benefit from high memory bandwidth) but different in others (only one benefits from high core clock frequency).

5.2.3 Collaborative Filtering

Collaborative filtering [3] is considered to be the most popular and widely implemented technique in recommender systems. It automatically generates predictions (filtering) about the interests of a user by collecting preferences or information from many users (collaborating) present in the system. One of its most publicized applications is the Netflix challenge for improving Netflix's movie recommender system, Cinematch, by providing valid movie recommendations to its users. The prize was won by BellKor's Pragmatic Chaos [64, 36, 8] after improving Cinematch by over 10% using a combination of different collaborative filtering techniques.

Model-based CF approaches transform both users and items to the same latent feature space. The latent space is then used to explain ratings by characterizing both products and users in terms of factors automatically inferred from user feedback.

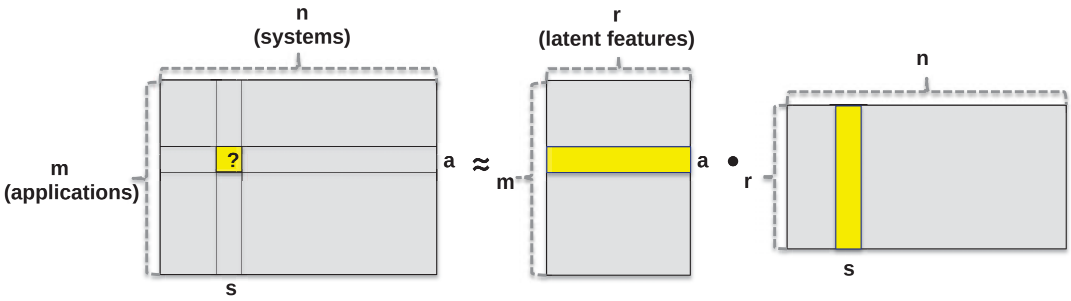


FIGURE 5.1: Mapping of applications and systems into a shared latent space using matrix factorization.

CF has recently been studied to predict performance across different processor architectures [52]. Matrix factorization (MF) is a model-based CF technique used in this work to model the interactions between applications and systems. While training, MF associates each of m applications and n systems with real-valued vectors of latent features of size r as shown in Figure 5.1. It infers these latent features automatically by uncovering hidden patterns in performance observed on systems. In order to predict performance of a benchmark a on system s, MF calculates the predicted score by taking the dot product of their latent features.

5.3 Prediction Models

We first discuss the limitations of matrix factorization and then present a neural network architecture that can learn complex application-system representations using additional training objectives.

5.3.1 Multi-Layer Perceptron Model

MF models the two-way interactions of applications and systems as a dot product of their latent features in a latent space. However, the linear combination of latent features with the same weight can limit its capability to model all application-system interactions in the low-dimensional latent space. Also, using a large number of latent features in order to capture both linear and non-linear interactions may affect the generalization of the model (e.g. overfitting the data). We address this limitation of MF by learning latent features using deep neural networks.

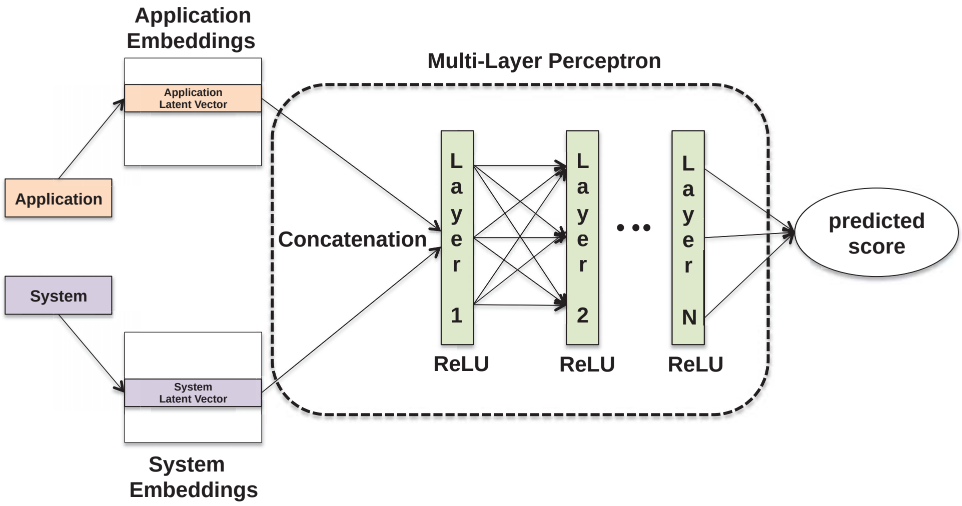


FIGURE 5.2: Multi-layer perceptron model using latent features

We propose a multi-layer perceptron (MLP) to learn the interactions between application and system latent features as shown in Figure 5.2. Each application and system in dataset is identified by a sparse vector with one-hot encoding. These sparse representations are used to create dense vectors called embeddings. The obtained application (system) embedding can be seen as the latent feature vector for application (system) in the context of matrix factorization model. The application and system embeddings are fed into a multi-layer neural network to learn the interaction between the corresponding application and system.

Let x_a and y_s denote the embeddings for application a and system s, respectively. Then, the MLP model is defined as:

$$\begin{aligned} l_1 &= (x_a, y_s) = [x_a \;\; y_s] \\ \phi(l_1) &= a_1(W_1 l_1 + b_1), \\ &\ldots\ldots. \\ \phi(l_n) &= a_n(W_n l_n + b_n), \end{aligned} \tag{5.1}$$

where W_x, b_x, a_x and (x_a, y_s) denote the weight matrix, bias vector, activation function for the xth's layer perceptron and concatenation of x_a and y_s embeddings respectively.

5.3.2 Multiple Training Objectives

In our study, we focus on predicting instructions per second (IPS) metric as the most reasonable proxy to application performance; however we can easily collect additional runtime metrics and use them for training. We choose loads per second (LPS) and stores per second (SPS).

Predicting multiple metrics is useful by itself for gaining better insight into application behavior. Additionally, we want to investigate if additional training objectives can improve performance prediction itself. Auxiliary losses have been used in machine learning models in various domains to improve statistical efficiency and to build better representations [66, 28, 41].

As with IPS, we use root mean squared error (RMSE) as an actual loss function for each of the additional metrics. We sum all losses for backpropagation and report only IPS component of total loss for fair comparison with the model which is trained with single metric.

5.3.3 Automated Architecture Search

We assume that a neural network that works better in the case of training with a single metric would not necessarily be the same as for training with multiple objectives. Although our model is elegantly simple and it is not uncommon to develop such models manually, we would like to avoid possible pitfall of subconsciously dedicating more attention to fine-tuning a model. As if it performs better, it would support our hypothesis of handpicking features. Additionally, because the problem is relatively small, we can afford to train and evaluate multiple architectures in a short time. For these reasons we perform a grid search for the best model architectures for cases of training with single and auxiliary objectives. We constrain the model to be a multi-layer feed forward network with an arbitrary number of layers and neurons in each layer.

5.4 Experimental Setup

5.4.1 Machine Specification

To demonstrate robustness of our approach, we selected GPUs as shown in Table 5.1. These hardware accelerators are commonly used in both HPC and cloud systems. The heterogeneity of these accelerators are with respect to micro-architecture (Kepler, Maxwell, Pascal and Volta) and the number of streaming multiprocessors (SM), L2 cache size, core frequency and memory bandwidth within accelerators having similar micro-architectures. We also show the type of Intel Xeon processor used along with each GPU in our experiments.

TABLE 5.1: Specifications of the GPUs used in our experiments.

GPU	arch	SMs	cores/SM	L2[1]	mem[2]	corefreq.[3]	memfreq.[4]	cpu used
K20m	Kepler	13	192	1.5	5	706	208	E5-2670v3
K20X	Kepler	14	192	1.5	6	732	250	E5-2650
K40c	Kepler	15	192	1.5	12	745	288	E5-2699v3
GTX-980Ti	Maxwell	22	128	3	6	1225	337	E5-2650v3
P100-PCIE	Pascal	56	64	4	16	1329	721	E5-2650v3
P100-SXM2	Pascal	56	64	6	16	1480	721	E5-2630v4
V100-SXM2	Volta	80	64	6	16	1530	897	Gold 6140

5.4.2 Benchmarks

We selected a diverse set of benchmarks from a variety of domains, as shown in Table 5.2. These workloads are from two well-known benchmark suites: Rodinia benchmark suite [55] version 3.3.1 and Polybench GPU version 1.0 [21].

The benchmarks were compiled using CUDA version 9.2.88 on P100-PCIE, V100-SXM2 and CUDA version 9.1.85 on all the other systems. We used nvprof [47] to collect three performance metrics which are inst_executed (instructions executed), gld_transactions (global load transactions) and gst_transactions (global store transactions). We executed each benchmark in isolation and recorded the total execution time for each benchmark.

[1] L2 size in MiB.

[2] Memory size in GiB.

[3] Core frequency in MHz.

[4] Peak memory bandwidth in GB/s.

TABLE 5.2: Workloads used in our experiments along with their domains.

Domain	**Benchmark**
Linear Algebra	gaussian, 2mm, 3mm, atax, bicg, gemm, gesummv,gramschmidt, mvt, syrk, syr2k
Data Mining & Pattern Recognition	correlation, covariance, nearest neighbor (nn), back-propogation (backprop)
Stencils	2dconvolution, 3dconvolution, fdtd-2d
Signal Processing	discrete wavelet transform 2D (dwt2d)
Image Processing	heartwall, srad, particlefilter
Simulation	hotspot2D, hotspot3D, myocyte
Graph Traversal	breadth-first search (bfs), b+tree, pathfinder
Fluid and Molecular Dynamics	lavamd
Bioinformatics	needleman-wunsh (nw)

5.4.3 Methodology

Problem Formulation.

Let M and N denote the number of applications and systems, respectively. We construct an application-system interaction matrix, $Y \in \mathbb{R}^{M \times N}$. Each cell in the matrix Y has value as:

$$y_{as} = \begin{cases} p_{as}, & \text{if application } a \text{ was executed on system } s \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

Here p_{as} indicates the observed performance score when application a was executed on system s. Our goal is to predict all the zero entries of Y.

Datasets.

We constructed three datasets with IPS, LPS and SPS values respectively for our experiments. In order to map these scores to a continuous scale, we performed z-score normalization of scores in each of the datasets. For each application a, we obtained mean score, $\bar{p}_a$ and standard deviation, σ_a in performance exhibited by the application on all the systems. The normalization of a performance score, p_{as} can be obtained as:

$$zscore(p_{as}) = \frac{p_{as} - \bar{p}_a}{\sigma_a} \tag{5.3}$$

Evaluation.

We used Chainer [59] to construct the MLP with Rectifier Linear Units (ReLU) [18] as the activation function. The network was trained to minimize RMSE and the optimizations were done by performing stochastic gradient descent (SGD)[31].

Let $\hat{p}_{as}$ be the predicted performance score of application a corresponding to system s. We divide the performance scores into a training set y_{train}, which is used to learn, and a test set y_{test}, which is used to calculate the prediction accuracy. RMSE [57] is a de-facto method to measure accuracy of CF algorithms. We used RMSE to evaluate the accuracy of our models as:

$$RMSE = \sqrt{\frac{1}{|y_{test}|} \sum_{p_{as} \in y_{test}} (\hat{p}_{as} - p_{as})^2} \tag{5.4}$$

5.5 Results

In this section, we conduct experiments with the aim of answering the following research questions:

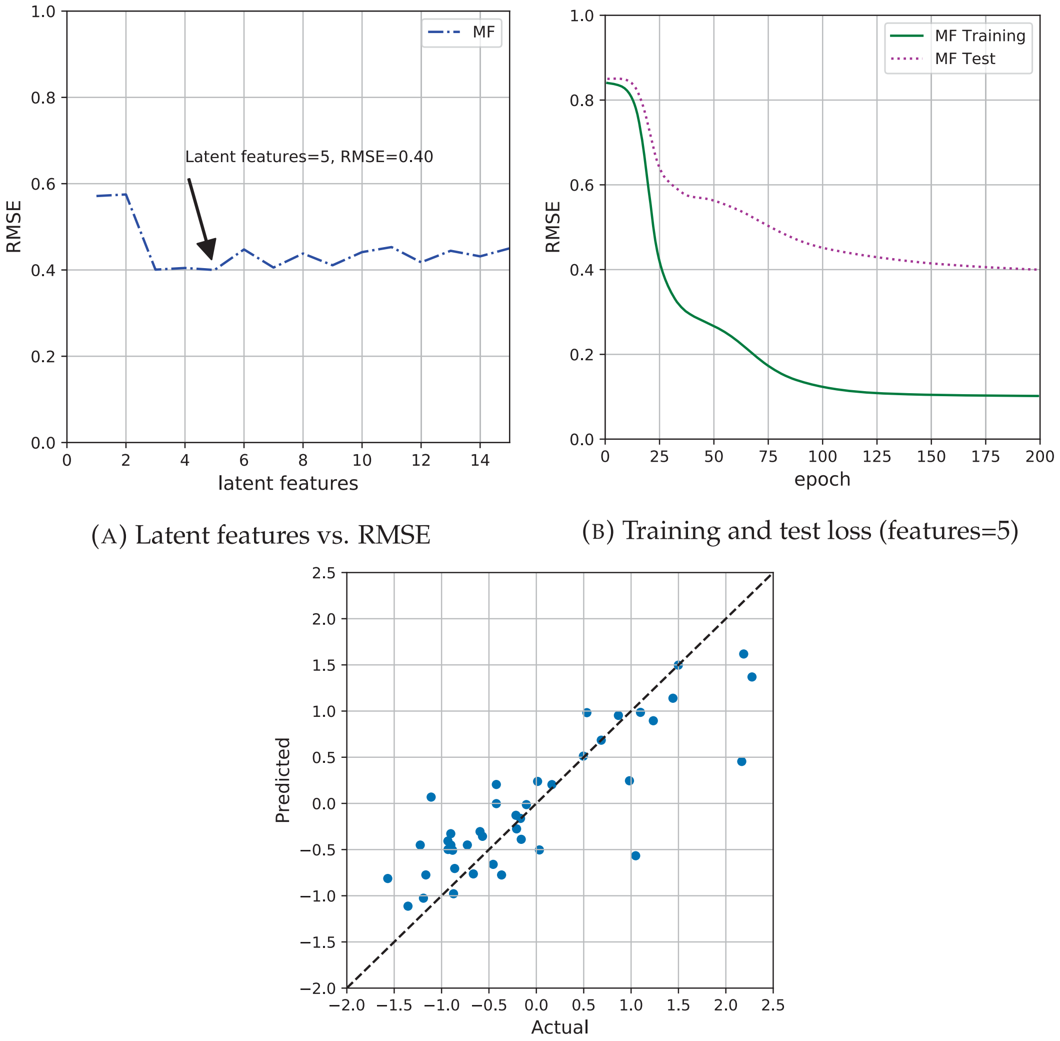
R1 Does collaborative filtering based matrix factorization approach work with GPUs?

R2 Can we improve the prediction quality by using deep neural networks?

R3 Does training with other performance metrics alongside IPS improve the prediction accuracy?

5.5.1 Performance of Matrix Factorization (R1)

Figure 5.3a shows the performance of MF with respect to the number of latent features on the IPS dataset. When we use one latent feature to represent each benchmark as well as system, the RMSE is 0.57. As we increase the features, MF projects each benchmark and system as data points in a higher dimensional space that describes the correlations between benchmarks and systems. As a result, the prediction improves by analyzing the linear associations between benchmarks and systems. The best RMSE is 0.40 when each benchmark and system is defined by five features. As we further increase the number of latent features to six, MF starts over-fitting the training data that performs poorly on the test dataset thereby increasing the error.



(A) Latent features vs. RMSE

(B) Training and test loss (features=5)

(C) Actual vs. predicted normalized IPS scores

FIGURE 5.3: Prediction performance of matrix factorization using latent features with IPS dataset.

We show the training and test losses when using five latent features in figure 5.3b. We can see that with more epochs, the training and test RMSE gradually decrease. The most effective updates are observed in the first 100 iterations for the training dataset. Although for the test dataset, the loss keeps decreasing and it starts saturating after 175 iterations at a RMSE of 0.4.

Figure 5.3c shows the scatter plot of actual vs predicted normalized IPS scores. Ideally, all the points in the plot should lie on or close to the regressed diagonal line. First, this plot tells us that the normalized actual scores lie within the range of -1.6 to 2.3. Second, MF make predictions close to the diagonal line when the actual value is greater than -1.5 and less than 1.5. While some of the values lie on the diagonal line showing accurate predictions, many fall close to the line. There are two cases when the actual score is near 1.0, that are estimated rather incorrectly. Third, when the actual value is greater than 2.0, MF underestimates the actual scores. On further inspection, we found that the three of these underestimated predictions are related to benchmarks mvt, pathfinder and backprop on K20m.

Table 5.3 shows the accuracy of MF on 42 pairs of benchmarks and GPUs selected from the test set. We can see that our model is able to predict a wide range of IPS values (with a minimum of 95.33 and a maximum of 690 000 000). The minimum IPS value corresponds to gaussian elimination application (gaussian) from the Rodinia benchmark using data file as matrix4.txt. We also measured the prediction accuracy for each benchmark and GPU pair and reported the average accuracy across all those pairs. We used the absolute value of the relative error ($\frac{Actual-Predicted}{Actual} * 100$) to evaluate the accuracy. MF achieves an average (relative) error of 15.8% and geometric mean (Gmean) of 7.4%, with minimum error of **0.02%** and maximum error of 52.31%.

Answer to R1:

We extended collaborative filtering based matrix factorization approach to seven GPUs using 30 benchmarks. Overall, MF achieves good predictions with an average error of 15.8% (**84.2%** accuracy) and geometric mean of 7.4%.

5.5.2 Performance of Multi-Layer Perceptron (R2)

We constructed MLPs with one hidden layer (MLP-1) as well as two hidden layers (MLP-2) to predict IPS as described in section 5.3.1. We tested each network with number of neurons in each layer as [2, 4, 8, 16, 32, 64], the embedding size of 1 to 15.

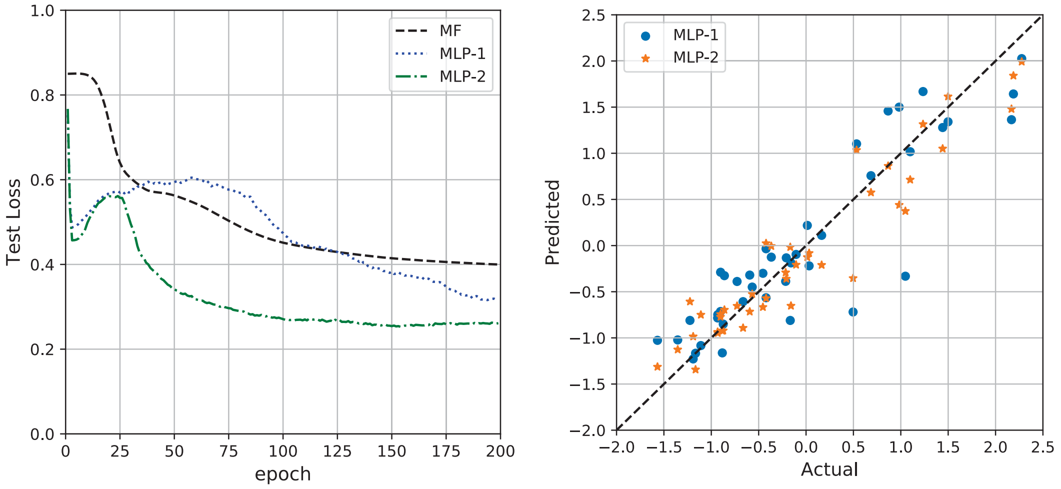
We present the findings of our experiments in Table 5.4. MF serves as the baseline performance with an RMSE of 0.40 with 5 latent features. MLP-1 with one hidden layer of 4 neurons and embedding size of 11 results in 20% decrease in RMSE thereby increasing prediction accuracy. While, MLP-2 with 32 and 8 neurons in the first and second hidden layer respectively (32→8) achieves the best RMSE of 0.25 which accounts for 37.5% improvement over the baseline.

TABLE 5.3: Accuracy of the matrix factorization model.

Benchmark	Suite	GPU	Actual IPS	Predicted IPS	Relative Error%
mycoyte	rodinia	V100-SXM2	12362329.33	12440673.99	0.63
hotspot2d	rodinia	V100-SXM2	628213.33	707013.49	12.54
mvt	polybench	K20m	2446784.0	1465271.07	40.11
gaussian	rodinia	K20m	95.33	92.74	2.72
srad1	rodinia	K40c	169454.77	184485.42	8.87
mvt	polybench	K40c	699081.14	914765.62	30.85
2mm	polybench	K20m	685834240.0	699268041.2	1.96
atax	polybench	P100-PCIE	1204320.0	1254020.08	4.13
pathfinder	rodinia	K20m	651904.0	574757.98	11.83
gemm	polybench	K20X	6277120.0	5636040.41	10.21
backprop	rodinia	K40c	805745.29	1123610.02	39.45
gesummv	polybench	P100-PCIE	1557440.0	1572621.1	0.97
backprop	rodinia	GTX980Ti	1395922.2	1397639.04	0.12
atax	polybench	K20m	1631189.33	1630874.66	0.02
bicg	polybench	V100-SXM2	941926.4	1142419.5	21.29
heartwall	rodinia	P100-SXM2	97238507.25	87104974.98	10.42
nn	rodinia	K20X	9102.8	9657.58	6.09
mvt	polybench	P100-PCIE	963456.0	1322472.2	37.26
covariance	polybench	V100-SXM2	70272888.89	57696629.18	17.9
gesummv	polybench	P100-SXM2	2076629.33	2076142.74	0.02
gramschmidt	polybench	P100-PCIE	2316.76	3166.81	36.69
fdtd2d	polybench	K20X	110436.15	135592.53	22.78
covariance	polybench	K20X	47624371.2	59948517.91	25.88
backprop	rodinia	K20m	1880072.33	1569359.24	16.53
3dconvolution	polybench	K20X	4145.14	5518.12	33.12
syrk	polybench	K20X	39819673.6	35606924.25	10.58
3dconvolution	polybench	P100-PCIE	4204.9	4497.67	6.96
mycoyte	rodinia	K20m	12547497.2	14798852.29	17.94
gramschmidt	polybench	K20m	7524.22	6787.84	9.79
b+tree	rodinia	V100-SXM2	2646457.0	2541856.97	3.95
3dconvolution	polybench	GTX980Ti	5667.48	4943.69	12.77
heartwall	rodinia	GTX980Ti	54529707.29	56063846.0	2.81
syr2k	polybench	V100-SXM2	526275925.33	333587189.11	36.61
syrk	polybench	V100-SXM2	37499699.2	41851043.74	11.6
gesummv	polybench	K20m	2054570.67	1936674.13	5.74
syrk	polybench	P100-PCIE	44900352.0	43516292.35	3.08
mvt	polybench	GTX980Ti	963481.6	1203747.72	24.94
2dconvolution	polybench	K40c	3294354.29	3665771.31	11.27
syrk	polybench	GTX980Ti	30632618.67	39903543.09	30.26
backprop	rodinia	V100-SXM2	1186206.2	1144560.7	3.51
covariance	polybench	P100-PCIE	48332996.27	61818140.88	27.9
syr2k	polybench	K20X	268763136.0	409358367.43	52.31
Average					15.8
Gmean					7.4

TABLE 5.4: Results of grid search for MLP-1 and MLP-2 parameters.

Model	Network	Features	Epoch	RMSE	↓ in RMSE
MF	-	5	199	0.40	-
MLP-1	4	11	194	0.32	20%
MLP-2	32→ 8	3	154	**0.25**	**37.5%**



(A) Test Loss

(B) Actual vs. Predicted Scores

FIGURE 5.4: Test loss and the scattered plot of actual vs predicted normalized IPS scores using MLP.

Figure 5.4 shows the test loss and actual vs. predicted normalized IPS scores for both MLP-based models. We can see that MLP-1 test loss decreases with more epochs and it starts predicting better than MF after 130 epochs in Figure 5.4a. Whereas, for MLP-2 the prediction performance on test dataset starts improving after 25 epochs. The above findings w.r.t prediction performance i.e. MLP-2 > MLP-1 > MF provide empirical evidence for the effectiveness of using deeper layers to improve prediction accuracy.

We show the advantage of using a deep network to predict IPS in Figure 5.4b. MLP-2 plots data points closer to the diagonal line than MLP-1. It is to be noted that for all the actual values underestimated by MLP-1 such as when the actual values are between 0.5 and 1.0 and also greater than 2.0, MLP-2 with just one more layer learns better and make predictions near to their actual values.

Table 5.5 shows the relative errors using MLP-1 and MLP-2 models. For fair comparison, we selected the same pairs of benchmarks and GPUs as evaluated for MF

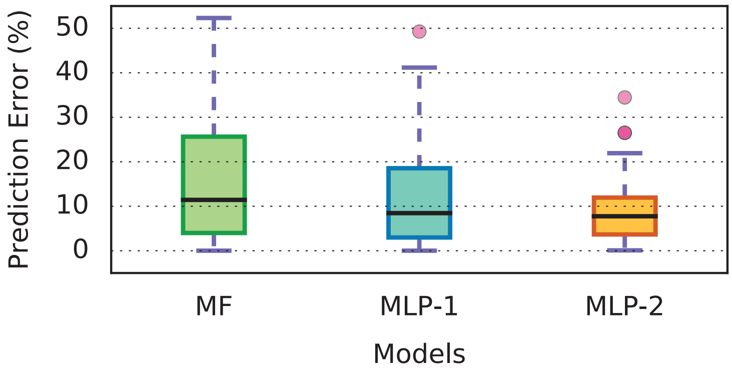


FIGURE 5.5: Accuracy of MF, MLP-1 and MLP-2 using IPS dataset

and presented the accuracy of all the three models. First, the average error as well as geometric mean show the same trend that MLP-2 $>$ MLP-1 $>$ MF. Second, overall, MLP-1 predicts better than MF specially when the relative error when using MF is greater than 20%. However, there are a few corner cases where MLP-1 underestimates the actual value when MF has an error of less than 1%. This can be attributed to those cases when a simple model is enough to describe the linear correlation between benchmark and system properties. In that case, using linear layers with ReLU as an activation unit may cause some irregularities in prediction performance. Since the main point of focus of this work is not to reason on how many and what features are important to model performance across benchmarks and systems, a model like MLP-1 which caters to the many of the cases is a better choice.

MLP-2 further decreases the large errors seen in MF and achieves the lowest average error across all the predictions. It is to note that the maximum error seen in MF is 51%, while in MLP-1 and MLP-2 are 49.26% and 34.45%. We summarize our results for MF, MLP-1 and MLP-2 in Figure 5.5. An outlier which is common to MLP-1 and MLP-2 in the box plot corresponded to a simulation application, myocyte from the Rodinia benchmark suite on V100-SXM2.

We show the prediction performance of MF, MLP-1 and MLP-2 on V100-SXM2 GPU and K40c in Figures 5.6 and 5.7 respectively. MLP-2 predicted the best in all the cases except for benchmarks myocyte and backprop on V100-SXM2.

TABLE 5.5: Relative error using MF, MLP-1 and MLP-2 on the test set.

Benchmark	Suite	GPU	Error MF%	Error MLP-1%	Error MLP-2%
srad1	rodinia	K40c	8.87	8.09	5.51
b+tree	rodinia	V100-SXM2	3.95	0.86	1.79
backprop	rodinia	GTX980Ti	0.12	19.44	4.52
gaussian	rodinia	K20m	2.72	1.96	9.25
backprop	rodinia	V100-SXM2	3.51	2.03	8.09
mycoyte	rodinia	K20m	17.94	22.69	19.99
mvt	polybench	P100-PCIE	37.26	23.04	26.55
mycoyte	rodinia	V100-SXM2	0.63	49.26	34.45
heartwall	rodinia	P100-SXM2	10.42	9.97	6.36
mvt	polybench	GTX980Ti	24.94	8.5	8.86
gesummv	polybench	K20m	5.74	3.08	7.42
gramschmidt	polybench	P100-PCIE	36.69	0.02	16.58
syr2k	polybench	K20X	52.31	1.19	15.98
syrk	polybench	P100-PCIE	3.08	3.46	6.95
gesummv	polybench	P100-SXM2	0.02	2.95	2.18
nn	rodinia	K20X	6.09	20.82	6.43
backprop	rodinia	K40c	39.45	28.29	13.3
pathfinder	rodinia	K20m	11.83	3.27	3.71
atax	polybench	P100-PCIE	4.13	0.41	4.63
covariance	polybench	V100-SXM2	17.9	8.42	3.82
3dconvolution	polybench	GTX980Ti	12.77	7.56	11.29
syrk	polybench	V100-SXM2	11.6	6.51	2.33
gramschmidt	polybench	K20m	9.79	12.51	2.31
mvt	polybench	K20m	40.11	18.82	16.11
2dconvolution	polybench	K40c	11.27	15.5	10.61
atax	polybench	K20m	0.02	2.39	3.65
3dconvolution	polybench	P100-PCIE	6.96	1.55	8.69
heartwall	rodinia	GTX980Ti	2.81	5.62	2.51
mvt	polybench	K40c	30.85	22.86	3.29
gesummv	polybench	P100-PCIE	0.97	1.35	9.38
syrk	polybench	K20X	10.58	8.0	10.93
syrk	polybench	GTX980Ti	30.26	41.17	10.62
bicg	polybench	V100-SXM2	21.29	20.23	8.97
covariance	polybench	P100-PCIE	27.9	9.15	6.33
fdtd2d	polybench	K20X	22.78	9.66	0.83
hotspot2d	rodinia	V100-SXM2	12.54	15.31	3.35
3dconvolution	polybench	K20X	33.12	17.73	26.46
covariance	polybench	K20X	25.88	7.34	0.15
syr2k	polybench	V100-SXM2	36.61	31.29	15.3
2mm	polybench	K20m	1.96	13.4	0.09
gemm	polybench	K20X	10.21	1.26	21.93
backprop	rodinia	K20m	16.53	11.58	12.17
Average			15.8	11.9	9.4
Gmean			7.4	6.3	6.0

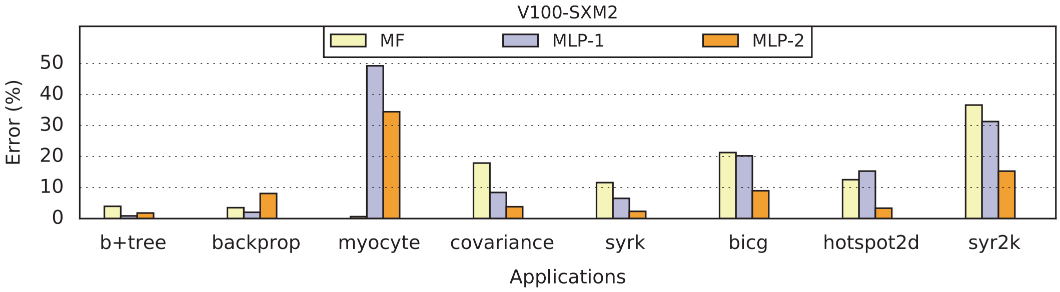


FIGURE 5.6: Evaluation of MF, MLP-1 and MLP-2 on V100-SXM2 GPU

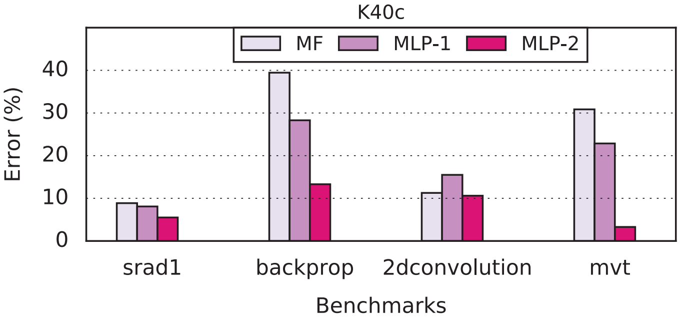


FIGURE 5.7: Performance of MF, MLP-1 and MLP-2 on K40c GPU

Answer to R2:

We investigated MLP with a curiosity to see whether using a deep network structure is beneficial to the prediction. The MLP-1 (MLP with one hidden layer) predicted with an average error of 11.9% (88.1% accuracy) and geometric mean of 6.3%. While the MLP-2 (MLP with two hidden layers) predicted with an average error of 9.4% (**90.6%** accuracy), geometric mean of 6% as compared to 84.2% accuracy, geometric mean of 7.4% achieved with MF.

5.5.3 Training with additional metrics (R3)

We augmented training data (IPS) with two additional performance metrics, LPS and SPS to determine if using multiple training objectives improve the predictions. We performed a grid search for the best neural architectures for training with additional objectives.

Table 5.6 show the results of the grid search in order to find the best model corresponding to different training metrics. Training MLP-1 with IPS and LPS, we find

TABLE 5.6: Performance of MLP when using additional training metrics, LPS and SPS.

	MLP-1				MLP-2			
Metric	Network	Features	RMSE	Error%	Network	Features	RMSE	Error%
IPS	4	11	0.32	11.9%	32→8	3	0.25	**9.4%**
IPS+LPS	32	9	0.32	12%	16→32	5	0.28	11.5%
IPS+LPS+SPS	32	12	0.33	12.2%	32→32	8	0.28	10.7%

that a network with 32 neurons and embedding size of 9 shows the similar predictive performance as with training MLP-1 with IPS. When we look at three metrics case using MLP-1, there is a slight increase in RMSE and the average error across all predictions. Overall, we can say that augmenting additional performance metrics with IPS using MLP-1 results in a similar performance.

We repeated the grid search to find the best parameters using MLP-2 as well. First, when training the model with IPS+LPS, the number of neurons increase from 16 to 32. So, it seems to be trading off the number of training metrics for depth as it goes through the layers. By using an embedding size greater than while training with only IPS, the model manages to achieve an average error of 11.5%. Similarly, for 3 metrics case, by increasing the number of neurons in the first layer and the embedding size, the average error becomes marginally lower though the RMSE remains the same.

Answer to R3:

This shows that IPS is the only metric that we need to predict performance across different benchmarks and GPUs. Using more additional optimization objectives in the predictive model does not improve accuracy of predicted IPS values in this study.

5.6 Conclusions

In this work, we demonstrated that it is possible to collect performance metrics and use collaborative filtering for GPU-based applications. We evaluated a set of 30 applications on 7 different generations of GPUs. Using the vanilla matrix factorization method of collaborative filtering resulted in 84.2% accuracy when the actual IPS has a wide range of values, with a minimum of 95.3 and a maximum of $6.9 * 10^8$. We then introduced neural network architectures to further improve the prediction accuracy. While for predicting performance, IPS it the only metric which we need, we showed that using additional optimization objectives in the predictive model (other metrics such as LPS and SPS) results in the similar accuracy of predicted IPS values. In total, we achieved 90.6% accuracy in average, with a geometric error mean of 6% with our multi-layer perceptron implementation. We showed that the confidence of predictions made varies between different kinds of applications. We leave it to future work to develop a model which can predict this uncertainty explicitly.

Chapter 6

Conclusion

6.1 Summary

Over the last decade, in response to the slowing or end of Moore's law, the High Performance Computing (HPC) community has turned towards heterogeneous systems. The trend in the recent decade has been to extend the application of HPC systems beyond the computationally intensive scientific domain to data-intensive domains, or the domain of Big Data. While HPC has its roots in solving compute-intensive scientific and large-scale distributed problems, Big Data problems have been proven to benefit from typical HPC environments: high processing power, low-latency networks, and non-blocking communications. However, the use of HPC systems is limited to scientists and researchers who have access to supercomputing labs and centers. With HPC extending its application space from scientific applications to big data analytics, the increasing demand for resources in HPC centers may not be met immediately. Hence, it is important to study alternative HPC solutions and the extent to which these solutions can be employed for different classes of computing and their applications. In the first step, we performed the evaluation of an HPC mini-app, NICAM-DC-MINI and data-intensive benchmark, Graph500 on three systems: 1) our in-house production supercomputer TSUBAME2.5, 2) the energy-efficient TSUBAME-KFC supercomputer and 3) AWS EC2's latest generation of compute-optimized instances, C4. We conducted this study in an attempt to measure the capabilities of C4 instances for representative HPC and Big Data applications. Our finding demonstrates that Amazon EC2 C4 instances exhibit good compute performance with low performance variability. The 10 Gbps Ethernet network in these instances is the chief bottleneck for scaling the workloads. So, C4 instances can be a good fit for compute-intensive and memory-intensive application with little communication. We optimized the Graph500 benchmark for C4 instances by finding the optimal number of threads and MPI ranks and achieved performance comparable to our in-house supercomputers. Similarly, we achieved good performance using both 10 and 20 MPI ranks per C4 instance for NICAM-DC-MINI. Thus, C4 instances can be adopted as a replacement to on-premises hardware deployments for such workloads.

Performance prediction of scientific applications across systems becomes increasingly important in today's diverse computing environments. A wide range of choices in execution platforms pose new challenges to researchers in choosing a system which best fits their workloads and administrators in scheduling applications to the best performing systems. While previous studies have employed simulation- or profile-based prediction approaches, such solutions are time-consuming to be deployed on multiple platforms. To address this problem, we use two collaborative filtering techniques to build analytical models, which can quickly and accurately predict the performance of workloads across different multicore systems. The first technique leverages information gained from performance observed for certain applications on a subset of systems and use it to discover similarities among applications as well as systems. The second collaborative filtering based model learns latent features of systems and workloads automatically and uses these features to characterize the performance of applications on different platforms. We evaluated both the methods using 30 workloads chosen from NAS Parallel Benchmarks, BOTS and Rodinia benchmark suites on ten different systems. Our results show that such collaborative filtering methods can make predictions with RMSE as low as 0.6 and with an average RMSE of 1.6.

The graphic processing units (GPUs) have become a primary source of heterogeneity in today's computing systems. With the rapid increase in number and types of GPUs available, finding the best hardware accelerator for each application is a challenge. For that matter, it is time consuming and tedious to execute every application on every GPU system to learn the correlation between application properties and hardware characteristics. To address this problem, we extend our previously proposed collaborating filtering based modeling technique, to build an analytical model which can predict performance of applications across different GPU systems. Our model learns representations, or embeddings (dense vectors of latent features) for applications and systems and uses them to characterize the performance of various GPU-accelerated applications. We improve state-of-the-art collaborative filtering approach based on matrix factorization by building a multi-layer perceptron. In addition to increased accuracy in predicting application performance, we can use this model to simultaneously predict multiple metrics such as rates of memory access operations. We evaluate our approach on a set of 30 well-known micro-applications and seven NVIDIA GPUs. As a result, we can predict expected instructions per second value with 90.6% accuracy in average.

Bibliography

[1] Yoongu Kim, Dongsu Han, Onur Mutlu, Mor Harchol-Balter. "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers". In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2010.

[2] Ali Karami, Sayyed Ali Mirsoleimani and Farshad Khunjush. "A statistical performance prediction model for OpenCL kernels on NVIDIA GPUs". In: *International Symposium on Computer Architecture and Digital Systems (CADS)*. 2013, 15–22.

[3] Almazro, Dhoha Shahatah, Ghadeer Albdulkarim, Lamia Kherees, Mona Martinez, Romy Nzoukou, William. "A Survey Paper on Recommender Systems". In: 2010.

[4] *Amazon Elastic Compute Cloud*. http://aws.amazon.com/ec2/.

[5] Amazon Web Services. *Amazon Machine Image*. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html.

[6] Amazon Web Services. *Elastic Block Store*. https://aws.amazon.com/ebs/.

[7] Amazon Web Services. *Now Available – New C4 Instances – Amazon Blog*. https://aws.amazon.com/jp/blogs/aws/now-available-new-c4-instances/.

[8] Andreas Toscher and Michael Jahrer. *The BigChaos Solution to the Netflix Grand Prize*. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf.

[9] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[10] Sara S. Baghsorkhi, Matthieu Delahaye, and Wen-mei WḢwu Sanjay J. Patel William D. Gropp. “An Adaptive Performance Modeling Tool for GPU Architectures”. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP’10. 2010, pp. 105–114.

[11] *Barcelona OpenMP Task Suite*. https://github.com/bsc-pm/bots.

[12] Calin Cascaval, Luiz DeRose, David A. Padua, and Daniel A. Reed. “Compile-time based performance prediction”. In: *International Workshop on Languages and Compilers for Parallel Computing*. 1999, pp. 365–379.

[13] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian and Thorsten von Eicken. “LogP: Towards a realistic model of parallel computation”. In: *ACM Sigplan Notices*. 1993, pp. 1–12.

[14] Jeff Dean, David Patterson, and Cliff Young. “A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution”. In: *IEEE Micro* 38.2 (2018), pp. 21–29.

[15] F. Maxwell Harper and Joseph A. Konstan. “The MovieLens Datasets: History and Context”. In: *ACM Transactions on Interactive Intelligent Systems (TiiS)*. 2015.

[16] Frank an der Linden. “Porting NetBSD to the AMD x86-64: A Case Study in OS Portability”. In: *BSDCon*. 2002, pp. 1–9.

[17] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena and Derek Chiou. “GPGPU performance and power estimation using machine learning”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 564–576.

[18] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Network”. In: *Proceedings of the Fourteenth International Conference on Artifical Intelligence and Statistics*. PMLR 15. 2011, pp. 315–323.

[19] Naga K. Govindaraju et al. “A Memory Model for Scientific Algorithms on Graphics Processors”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. November 2006. 2006.

[20] Graph500.org. *The Graph500 List*. http://www.graph500.org/.

[21] Scott Grauer-Gray et al. "Auto-tuning a High-Level Language Targeted to GPU Codes". In: *Innovative Parallel Computing (InPar)*. 2012.

[22] Qiming He et al. "Case Study for Running HPC Applications in Public Clouds". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. Chicago, Illinois: ACM, 2010, pp. 395–401. ISBN: 978-1-60558-942-8. DOI: 10.1145/1851476.1851535. URL: http://doi.acm.org/10.1145/1851476.1851535.

[23] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. "Towards Efficient MapReduce Using MPI". In: *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland: Springer-Verlag, 2009, pp. 240–249. ISBN: 978-3-642-03769-6. DOI: 10.1007/978-3-642-03770-2_30. URL: http://dx.doi.org/10.1007/978-3-642-03770-2_30.

[24] Sunpyo Hong and Hyesoon Kim. "An Integrated GPU Power and Performance Model". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA'10. 2010, pp. 280–289.

[25] Alexandru Iosup et al. "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 931–945. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.66.

[26] Christiane Jablonowski and David L. Williamson. "A baroclinic instability test case for atmospheric model dynamical cores". In: *Quarterly Journal of the Royal Meteorological Society* 132.621C (2006), pp. 2943–2975. ISSN: 1477-870X. DOI: 10.1256/qj.06.12. URL: http://dx.doi.org/10.1256/qj.06.12.

[27] K. R. Jackson et al. "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 2010, pp. 159–168. DOI: 10.1109/CloudCom.2010.69.

[28] Max Jaderberg et al. "Reinforcement Learning with Unsupervised Auxiliary Tasks". In: *CoRR* abs/1611.05397 (2016).

[29] Jeffrey Vetter, Chris Chambreau. *mpiP: Lightweight, Scalable MPI Profiling*. http://mpip.sourceforge.net/.

[30] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez and Joel Emer. "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)". In: *SIGARCH Computer Architecture News*. 2012.

[31] L. Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *In Proc. of COMPSTAT 2010*.

[32] Weiguo Liu and Bertil Schmidt. "Performance Predictions for General-Purpose Computation on GPUs". In: *Proceedings of 2007 International Conference on Parallel Processing*. ICPP. Xi-An, China, 2017.

[33] A. Luckow et al. "P*: A model of pilot-abstractions". In: *E-Science (e-Science), 2012 IEEE 8th International Conference on*. 2012, pp. 1–10. DOI: 10.1109/eScience.2012.6404423.

[34] Madhavi Valluri and Lizy John. "Is Compiling for Performance == Compiling for Power?" In: *Interaction between Compilers and Computer Architectures*. Springer, 2001.

[35] Pradeep Kumar Mantha, Andre Luckow, and Shantenu Jha. "Pilot-MapReduce: An Extensible and Flexible MapReduce Implementation for Distributed Data". In: *Proceedings of Third International Workshop on MapReduce and its Applications Date*. MapReduce '12. Delft, The Netherlands: ACM, 2012, pp. 17–24. ISBN: 978-1-4503-1343-8. DOI: 10.1145/2287016.2287020. URL: http://doi.acm.org/10.1145/2287016.2287020.

[36] Martin Piotte and Martin Chabbert. *The Pragmatic Theory Solution to the Netflix Grand Prize*. https://www.netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf.

[37] Naoya Maruyama. "Miniapps for Enabling Architecture-Application Co-design for Exascale Supercomputing". In: *19th Workshop on Sustained Simulation Performance*. 2014.

[38] Naoya Maruyama et al. *Fiber Miniapp Suite*. http://fiber-miniapp.github.io/.

[39] *Microsoft Azure Cloud Computing Platform Services*. https://azure.microsoft.com/en-us/.

[40] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems 26*. Ed. by C.J.C. Burges et al. Curran Associates, Inc., 2013, pp. 3111–3119.

[41] Piotr W. Mirowski et al. "Learning to Navigate in Complex Environments". In: *CoRR* abs/1611.03673 (2016).

[42] Jeffrey Napper and Paolo Bientinesi. "Can Cloud Computing Reach the Top500?" In: *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*. UCHPC-MAW '09. Ischia, Italy: ACM, 2009, pp. 17–20. ISBN: 978-1-60558-557-4. DOI: 10.1145/1531666.1531671. URL: http://doi.acm.org/10.1145/1531666.1531671.

[43] *NAS Parallel Benchmarks*. https://www.nas.nasa.gov/publications/npb.html.

[44] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti and others. "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News*. 2011, pp. 1–7.

[45] *NVIDIA Corporation*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[46] *Nvidia Turing GPU Architecture*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[47] *NVProf*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[48] P.J. Joseph, Kapil Vaswani and and M.J. Thazhuthaveetil. "Construction and use of linear regression models for processor performance analysis". In: *International Symposium on High-Performance Computer Architecture*. 2006, pp. 99–108.

[49] *Post-k Development and Introducing DLU, Fujistu*. http://www.fujitsu.com/global/Images/post-k-development-and-introducing-dlu.pdf.

[50] P.S Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner. "Simics: A full system simulation platform". In: *IEEE Computer*. 2002, pp. 50–58.

[51] Robert Fenichel, Adrian J. Grossman. "An Analytic Model of Multiprogrammed Computing". In: *AFIPS Spring Joint Computing Conference*. 1969, pp. 717–721.

[52] Shweta Salaria et al. "Predicting Performance Using Collaborative Filtering". In: *Proceedings of the 2018 IEEE International Conference on Cluster Computing*. CLUSTER. Belfast, UK, 2018, pp. 504–514.

[53] Sameh Sharkwai, Don Desota, Raj Panda, Rajeev Indukuru, Stephen Stevens, Valerie Taylor and Xingfu Wu. "Performance projection of HPC applications using SPEC CFP2006 benchmarks". In: *International Symposium on Parallel & Distributed Processing*. 2009, pp. 1–12.

[54] Samuel Williams, Andrew Waterman and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM*. 2009, pp. 65–76.

[55] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jermy W. Sheaffer, Sang-Ha Lee and Kevin Skadron. "Rodinia: A Benchmark Suite for Hetrogenous Computing". In: *International Symposium on Workload Characterization (IISWC)*. 2009.

[56] Sunpyo Hong and Hyesoon Kim. "An integrated GPU power and performance model", SIGARCH Computer Architecture News". In: ACM, 2001.

[57] T. Chai and R. R. Draxler. "Root mean square error (RMSE) or mean absolute error (MAE) - Arguments against avoiding RMSE in the literature". In: *Geosco. Model Dev.* Pp. 1247–1250.

[58] *The OpenCL Specification*. https://www.khronos.org/opencl/.

[59] Seiya Tokui et al. "Chainer: a Next Generation Open Source Framework for Deep Learning". In: *Proceedings of Workshop on Machine Learning Systems in NIPS*. 2010.

[60] *Top500*. https://www.top500.org.

[61] U.S. Department of Energy. *The Magellan Report on Cloud Computing for Science*. https://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf.

[62] VI-HPS. *Score-P*. http://www.vi-hps.org/projects/score-p/.

[63] Yao Xhang and John D. Owens. “A Quantitative Performance Analysis Model for GPU Architectures”. In: *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*. HPCA’11. 2011.

[64] Yehuda Koren. *The BellKor Solution to the Netflix Grand Prize*. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.

[65] Katherine Yelick et al. *The Magellan Report on Cloud Computing for Science*. Tech. rep. U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research (ASCR), Dec. 2011.

[66] Zhang Yuting, Lee Kibok, and Lee Honglak. “Augmenting Supervised Neural Networks with Unsupervised Objectives for Large-scale Image Classification”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML’16. New York, NY, USA: JMLR.org, 2016, pp. 612–621.

[67] Yan Zhai et al. “Cloud Versus In-house Cluster: Evaluating Amazon Cluster Compute Instances for Running MPI Applications”. In: *State of the Practice Reports*. SC ’11. Seattle, Washington: ACM, 2011, 11:1–11:10. ISBN: 978-1-4503-1139-7. DOI: 10.1145/2063348.2063363. URL: http://doi.acm.org/10.1145/2063348.2063363.