2023

Domain-Specific Optimization For Machine Learning System

Yu Chen
College of William and Mary - Arts & Sciences, cy.milky@gmail.com

Domain-specific Optimization for Machine Learning System

Yu Chen

Hangzhou, Zhejiang, China

Bachelor of Engineering, Southeast University, 2014

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May 2023

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

[illegible]

Yu Chen

Approved by the Committee, May 2023

Bin Ren

Committee Chair
Bin Ren, Associate Professor, Computer Science
College of William & Mary

[illegible]

Stathopoulos Andreas, Professor, Computer Science
College of William & Mary

Zhenming Liu

Zhenming Liu, Adjunct Professor, Computer Science
College of William & Mary

Robert Michael Lewis

Lewis Robert Michael, Associate Professor, Computer Science
College of William & Mary

Xipeng Shen

Xipeng Shen, Professor, Computer Science
North Carolina State University

ABSTRACT

The machine learning (ML) system has been an indispensable part of the ML ecosystem in recent years. The rapid growth of ML brings new system challenges such as the need of handling more large-scale data and computation, the requirements for higher execution performance, and lower resource usage, stimulating the demand for improving ML system. General-purpose system optimization is widely used but brings limited benefits because ML applications vary in execution behaviors based on their algorithms, input data, and configurations. It's difficult to perform comprehensive ML system optimizations without application specific information. Therefore, domain-specific optimization, a method that optimizes particular types of ML applications based on their unique characteristics, is necessary for advanced ML systems. This dissertation performs domain-specific system optimizations for three important ML applications: graph-based applications, SGD-based applications, and Python-based applications.

For SGD-based applications, this dissertation proposes a lossy compression scheme for application checkpoint constructions (called LC-Checkpoint). LC-Checkpoint intends to simultaneously maximize the compression rate of checkpoints and reduce the recovery cost of SGD-based training processes. Extensive experiments show that LC-Checkpoint achieves a high compression rate with a lower recovery cost over a state-of-the-art algorithm. For kernel regression applications, this dissertation designs and implements a parallel software that targets to handle million-scale datasets. The software is evaluated on two million-scale downstream applications (i.e., equity return forecasting problem on the US stock dataset, and image classification problem on the ImageNet dataset) to demonstrate its efficacy and efficiency. For graph-based applications, this dissertation introduces ATMem, a runtime framework to optimize application data placement on heterogeneous memory systems. ATMem aims to maximize the fast memory (small-capacity) utilization by placing only critical data regions that yield the highest performance gains on the fast memory. Experimental results show that ATMem achieves significant speedup over the baseline that places all data on slow memory (large-capacity) with only placing a minority portion of the data on the fast memory.

The future research direction is to adapt ML algorithms for software systems/architectures, deeply bind the design of ML algorithms to the implementation of ML systems, to achieve optimal solutions for ML applications.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

In retrospect, I realize that the first conversation between Dr. Zhenming Liu and me at McGlothlin-Street Hall was the beginning of a beautiful friendship. As a prominent advisor, brilliant colleague, and close friend, Zhenming gave me valuable guidance for my research and career, his advice will be a precious asset in my future life. I thank him for his encouragement and support during my Ph.D.

I was also fortunate to be advised by Dr. Bin Ren and Dr. Andreas Stathopoulos (yes, I have three advisors). I thank Bin for his generous help both in my research and personal life, I would not have begun my Ph.D. without his strong support. I thank Andreas for his attentive instructions and patience (e.g. corrected my stupid pronunciation over and over again, replied to my email when his flight was runaway, met with me to answer my naive questions when he got covid, wrote the paper until 3 AM because I gave him the wrong deadline, etc).

It would not be possible to finish my Ph.D. without these wise professors.

Also, I thank my dissertation committee members, Dr. Michael Lewis and Dr. Xipeng Shen for their time and constructive feedback.

Additionally, I would like to extend thanks to my collaborators Dr. Xin Jin, Dr. Xu Liu, and Dr. Ivy Peng Bo. I'm grateful to collaborate with these excellent researchers and hear their advice.

Lastly, I would like to thank Jialiang Tan, LeCun, Felicis, and Liz's unconditional love and company. I thank my family, especially my grandparents, for their trust and support.

To Yingming Li.

LIST OF TABLES

LIST OF FIGURES

Domain-specific Optimization for Machine Learning System

Chapter 1

Introduction

In recent years, machine learning (ML) has achieved huge success in extensive areas including computer vision [35, 45, 18], robotics [72, 69, 113], recommendation systems [106, 68, 117], etc. With the explosive growth of ML, massive datasets, complected algorithms, and sophisticated theories are introduced to the ML community, which leads to much higher requirements for ML systems, e.g. the model size increased by five orders of magnitude from 2018 to 2022 [126]. Appropriate optimizations on ML systems can significantly increase the performance of ML applications and reduce their resource usage. However, performing general-purpose optimizations is not the panacea because different types of ML applications have various behaviors on the systems. For example, most deep learning applications are compute-bound but graph-based applications and recommendation systems are memory-bound because of their irregular memory access patterns. Without the application information, general-purpose system optimization provides limited benefits to ML applications.

To overcome the limitation of general-purpose optimization, domain-specific optimization is applied for improving the ML systems [112, 2]. With applications' information, domain-specific optimization optimizes applications based on their unique characteristics, bridging the gap between ML applications and ML systems.

In this dissertation, we introduce three works that perform domain-specific optimizations for ML applications: (1) LC-Checkpoint, a lossy scheme to compress the checkpoints for SGD-based applications; (2) a parallel software for million-scale kernel principal component regression applications; (3) ATMem, a runtime framework to optimize the data placement on heterogeneous memory systems for graph-based applications.

1.1 Contributions

1.1.1 Efficient Construction of Checkpoints for SGD-based Application

The efficient construction of checkpoints/snapshots is a critical tool for training and diagnosing deep learning models. Producing frequent checkpoints minimizes the wasted time for restarting failed training processes and serves as breakpoints for debugging deep learning applications. The current standard practice of constructing checkpoints that saves the whole model state directly yields unmanageable burdens to file systems, even under modern distributed platforms [3, 83, 89]. Attempts of storing partial model states are also examined [108] but these works focus on recovery speed, instead of directly tackling system issues. Therefore, a method to compress checkpoints is desired for constructing checkpoints frequently.

We propose a lossy compression scheme for checkpoint constructions (called LC-Checkpoint). LC-Checkpoint simultaneously maximizes the compression rate and optimizes the recovery speed, under the assumption that SGD is used to train the model. LC-Checkpoint uses an exponent-based quantization and priority promotion to store the most crucial information for SGD to recover, and then uses a Huffman coding to leverage the non-uniform distribution of the gradient scales. Our extensive experiments show that LC-Checkpoint achieves a compression rate up to 28× and recovery speedup up to 5.77× over a state-of-the-art algorithm. (SCAR [108])

1.1.2 Parallel Software for Million-scale Exact Kernel Regression

Kernel regressions are nonlinear and interpretable models. It has wide downstream applications [17, 115, 65, 94, 61, 129, 41, 84, 142, 140] because of its robust theoretical properties, and is shown to have a close connection to the deep neural networks [60, 96]. Nevertheless, the exact regression of large-scale kernel models using currently available software has been notoriously difficult because it is both compute and memory intensive and it requires extensive tuning of hyperparameters. While in computational science distributed computing and iterative methods have been a mainstay of large-scale software, they have not been widely adopted in kernel learning.

In this work, we design and implement a kernel principal component regression software that handles training datasets with millions of observations. It leverages existing high-performance computing (HPC) techniques and develops new ones that address cross-cutting constraints between HPC and learning algorithms. It integrates three major components: (a) a state-of-the-art parallel eigenvalue iterative solver, (b) a block matrix-vector multiplication routine that employs both multi-threading and distributed memory parallelism and can be performed on-the-fly under limited memory, and (c) a software pipeline consisting of Python front-ends that control the HPC backbone and the hyperparameter optimization through a boosting optimizer. We perform feasibility studies by running the software on the entire ImageNet dataset and a large asset pricing dataset to demonstrate its efficacy and efficiency.

1.1.3 Adaptive Data Placement for Graph Application on HMS

With the active development of new memory devices, such as non-volatile memories and high-bandwidth memories [63], heterogeneous memory systems (HMS) become a promising solution for implementing large-scale memory systems with cost, area, and power limitations [62, 63, 64]. Typical HMS consists of a small-capacity high-performance memory and a large-capacity low-performance memory. Data placement on such systems plays a critical role in performance optimization. Existing efforts [36, 26, 82, 103, 134, 130, 97, 31] explored the coarse-grained data placement in applications with dense data structures; However, these approaches are inefficient for graph applications due to the irregular access patterns of graph data structures. The main challenges are: 1) coarse-grained solutions that perform whole data structure placement may put non-critical data regions (e.g. the data associated with low-degree vertices in graph processing) on the high performance memory, suffering a waste of scarce resources; 2) the effective data placement for graph applications mostly depends on the feature of the input data, i.e., the sparsity and the structure of a graph, and the query at each run.

We propose ATMem—a runtime framework for adaptive granularity data placement optimization in graph applications. ATMem consists of a lightweight profiler, an analyzer using a novel m-ary tree-based strategy to identify sampled and estimated critical data chunks, and a high-bandwidth migration mechanism using a multi-stage multi-threaded approach. The target of ATMem is to maximize the performance gain per byte, i.e., improving fast memory utilization by only placing critical data regions that yield the highest performance gains on it. ATMem is evaluated in five applications on two HMS hardware, including the Intel Optane byte-addressable NVM and MCDRAM. Experimental results show that ATMem selects 5%-18% data to be placed on high-performance memory and achieve an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM, over the baseline that places all data on the large-capacity memory. On NVM-DRAM, ATMem achieves performance comparable to a full-DRAM system with as low as 9%-54% slowdown.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we propose LC-Checkpoint, a lossy compression scheme to construct checkpoint for training and diagnosing deep learning models. In Chapter 3, we present a kernel principal component regression software that handles million-scale datasets. In Chapter 4, we introduce ATMem—a runtime framework for adaptive granularity data placement optimization in graph applications. And in Chapter 5, we describe our conclusion and discuss future research directions.

Chapter 2

On Efficient Constructions of Checkpoints

2.1 Introduction

Efficient construction of checkpoints (snapshots) has been increasingly important to deep learning research. In the arms race of developing more accurate models, researchers utilize heavier computing infrastructure and develop deeper and larger models. Without proper infrastructure support, the research process inevitably becomes fragile. For example, distributed computation fails from time to time, leading to the excessive need to re-train models [108]. Diagnosing deep learning models also evolves to a complex procedure partly because that the community has a better understanding of deep learning models and produces more rules for "debugging" them. Some common errors include gradient explosion [43], "divide by zero" [58], and dead activation. This calls for the need to construct "breakpoints," resembling those used in debugging computer programs, so that researchers can conveniently jump to the state right before the model "crashes" in the training.

Producing checkpoints frequently enables failed training process to restart with minimum wasted time, and serves as breakpoints for debugging models. So far the standard practice of constructing checkpoints is primitive. The most common practice is to save the model state directly, counting on that the backend system is sufficiently robust so that this operation does not become a bottleneck [13]. Attempts of partially storing model states are also examined [108] but these works usually focus on recovery speed, instead of directly tackling system issues.

The most pronounced technical challenge here is that deep models are usually large, so producing frequent checkpoints creates unmanageable burdens to both I/O and storage, even under modern distributed platforms [3, 83, 89]. Therefore, this leads to our question:

Research Q: *How can we compress model checkpoints?*

We specifically aim to design a *lossy* compressing scheme, addressing two criteria simultaneously. First, like standard compression problems, we need to maximize the compression rate. Second, the scheme needs to be optimized for the downstream application of *training.* When a model restarts from our lossy checkpoints, it needs to efficiently resume to the most recent state (e.g., restart from a failed process or reach the state preceding the crash).

Compression of model states is a new technical problem that requires addressing cross-cutting constraints from information theory, learning algorithm, and system design. We need to leverage statistical patterns encoded in the model state and factor in how the model states interact with a learning algorithm (more specifically, stochastic gradient type algorithms in the deep learning setting). This means neither standard lossy compression algorithms nor recently developed model compression algorithms [48, 24, 55, 81, 86] directly work in our setting. Standard lossy compression algorithms aim to minimize reconstruction error but our end goal is to enable a learning algorithm to "quickly recover." Model compression techniques aim to transform a (static) model into a simpler one while ensuring the forecasts are not perturbed much whereas in our setting we need a reliable coding scheme that functions well throughout the entire dynamic process of learning, which is an orthogonal and perhaps more challenging goal. In addition, our algorithm must be efficient and scalable so that it can be executed frequently.

Our solution. To achieve our aims, we focus on a delta-encoding scheme [93], tracking only the information on the difference between two checkpoints. Under this scheme, we examine whether we can cut the least useful information (with respect to training) from the model state, and ensure that the remaining information is amenable for compression. A perhaps surprising message here is that ℓ_2-norm reconstruction error for the "delta" appears to be an ineffective metric for minimizing the recovery time. Instead, our algorithm first removes all the parameters with inconsequential updates, and then quantizes the remainder information. These strategies resemble those used in distributed training with the goal of minimizing communication cost [7]. After we obtain the most significant information for portion of parameter updates, we represent them in suitable format and apply a Huffman coding to further compress these bits, so that the compression rate can be at the information theoretic limit. This strategy resembles recent techniques for model compression [48, 133, 99, 143, 111].

The contribution of this work includes:

- Proposal of a fundamental research question on compressing model states for training recovery.
- Characterization of a family of compression schemes that can efficiently track the learning process, based on a stylized model we develop.
- Design of a lossy coding scheme with high-compression rate that integrates both classical compression techniques and recent ones developed for distributed learning and model compression.
- Optimization of training systems that minimizes the overhead of producing checkpoints on the fly.

Our extensive evaluation demonstrates that by simultaneously leveraging techniques from distributed training and model compression, our algorithm delivers a solution (called LC-Checkpoint, LC refers to Lossy Compression) with a compression rate of **up to 28x** and superior recovering time—achieving up to 5.77× recovery speedup over a state-of-the-art algorithm (SCAR).

2.2 Our approach

We now describe our compression framework. We introduce a stylized model for the learning process to facilitate the analysis of the system design trade-off. Then we explain our design principles, determined by both the stylized model and our extensive experiments.

Our model. A "high-dimensional" vector $\mathbf{u} \in \mathbf{R}^n$ represents the model state. An iterative algorithm (e.g., stochastic gradient descent) is used to gradually move the model state vector $\mathbf{u}$ toward a local optimal point $\mathbf{u}^*$. Let $\mathbf{u}_t$ be the model state at the t-th round. In our stylized model, we assume $\mathbf{u}_t$ performs a (drifted) random walk that converges to $\mathbf{u}^*$. Specifically, we use the following process to model $\mathbf{u}_i$'s trajectory. Let $L = \|\mathbf{u}_0 - \mathbf{u}^*\|$.

$$\mathbf{u}_{t+1} = \mathbf{u}^* + \eta(\mathbf{u}_t - \mathbf{u}^*) + \epsilon_t, \tag{2.1}$$

where η and L jointly model the convergence rate of the algorithm, and ϵ_t is a random noise component to reflect the stochastic nature of SGD. When η is set to be a small constant, the model characterizes those algorithms that have linear convergence rate. When $\eta = (1 - 1/L)$, this model characterizes those algorithms whose convergence rates are $1 - 1/t$ [16]. While our model does not captures the detail of many SGD algorithms, because different SGD algorithms have different convergence rate, designing a unifying model that highlights design trade-offs requires us to make simplifying assumptions.

Our design principles. We next describe our design principles.

P1. Minimize irritation to SGD. When we design lossy compression scheme, a portion of information is inevitably lost, causing performance degradation to a learning algorithm. We find that we should not simply use ℓ_2 reconstruction error to measure degradation of SGD. This can be best illustrated by the stylized model. For simplicity, let $\mathbf{u}^* = 0$, so $\mathbf{u}_{t+1} = \mathbf{u}_t - ((1-\eta)\mathbf{u}_t + \epsilon_t)$. The delta term we want to compress is $((1-\eta)\mathbf{u}_t + \epsilon_t)$. When we use a lossy compression, it corresponds to adding an additional noise term that is a function of $\mathbf{u}_t$ and ϵ_t. So with the compression scheme, the new learning process becomes $\mathbf{u}_{t+1} = \mathbf{u}_t - ((1-\eta)\mathbf{u}_t + \epsilon_t + f(\mathbf{u}_t, \epsilon_t))$. Observing that as long as $\mathbb{E}[f(\mathbf{u}_t, \epsilon_t) \mid \mathbf{u}_t, \epsilon_t] = 0$, and $\mathrm{Var}(f(\mathbf{u}_t, \epsilon_t) \mid \mathbf{u}_t, \epsilon_t)$ is dominated (smaller than) by $\mathrm{Var}(\epsilon_t)$, then the convergence quality remains unchanged, by standard results from stochastic approximation [76, 74].

There are many constructs that satisfy the expectation and variance constraints. Let us consider an example of keeping the most significant bit of $((1-\eta)\mathbf{u}_t + \epsilon_t)$ by using standard randomized rounding [7]. Because of the nature of the rounding algorithm, the expectation is 0. In addition, because the most significant bit is kept, the information loss in rounding will not be greater than $\| ((1-\eta)\mathbf{u}_t + \epsilon_t) \|_2 = O(\mathrm{std}(\epsilon_t))$ under a mild assumption that ϵ_t's standard deviation also scales proportionally to $\|\mathbf{u}_t\|$ over time. Therefore, this rounding scheme does not affect the performance of the training algorithm. In general, the 1-bit encoding is a special case of quantization. A wide family of quantization schemes will satisfy the expectation and variance constraint. Our algorithm will explore this trade-off.

Note also when we minimize ℓ_2 reconstruction error, this corresponds to keeping top-k heaviest entries in $\mathbf{u}_{t+1} - \mathbf{u}_t$.

P2. Maximize redundancies in residual information. Our compression scheme also needs to ensure the information we keep exhibits large redundancy, as measured by entropy. This will enable us to use traditional coding schemes such as Huffman code to compress the data at the information theoretic limit.

The interplay between P1 and P2 highlights the unique structure of our compression problem. This can be best illustrated by a compression scheme called TOPN. This compression scheme keeps the largest elements in δ_t. We observe *(i)* while this scheme minimizes ℓ_2 reconstruction error, it *does not* have superior recovery time. Many other compression schemes that possess the aforementioned properties recover equally fast, as

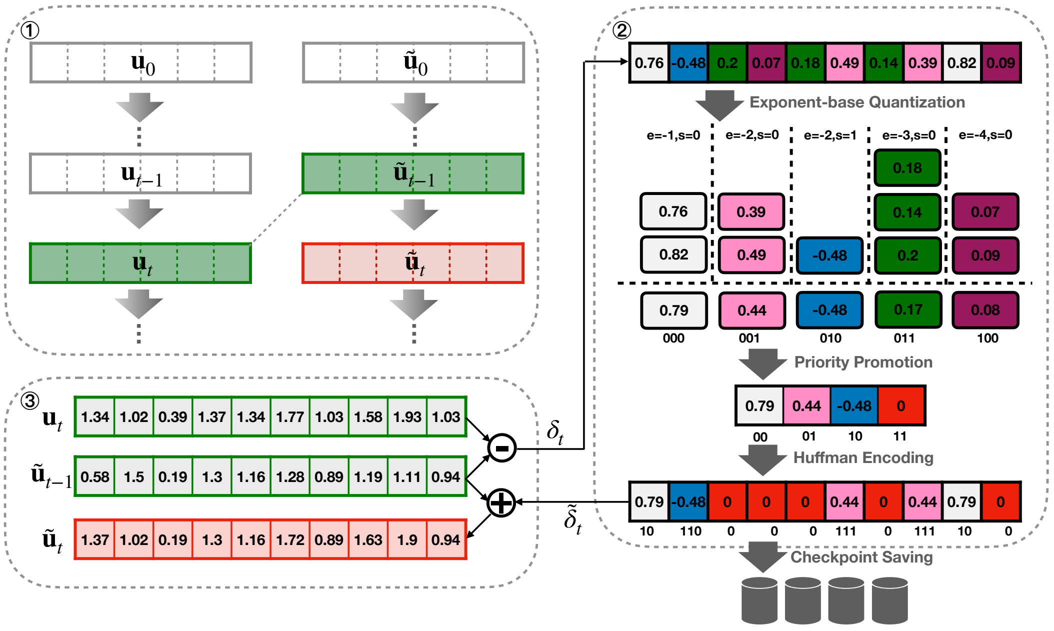


Figure 2.1: LC-Checkpoint overview.

suggested by our stylized model. *(ii)* It is difficult to perform compression for the TOPN scheme. TOPN scheme usually needs to track 10% of all the entries in δ_t to be effective. The overhead of tracking the *locations* of these elements is surprisingly high. This is because in part that the vector is not sufficiently sparse so sparse matrix representation does not help.

Our solution, on the other hand, carefully complies P1 and circumvents the need to track the locations of the entries we keep and thus achieves significantly higher compression rate.

P3. Do not use random projections and/or sketches. Notably, we discover that sketch-based randomized projection techniques [132] *harm* the compression. Roughly speaking, sketches compress information by projecting multiple numbers into one cell. While this could speed up query time, it only irritates the gradient descent algorithm in our setting. Consider a toy example in which $\mathbf{u}_t \in \mathbf{R}^2$ and the optimal point $\mathbf{u}^* = (0, 10)$. Let $\mathbf{u}_t = (5, 5)$ be the current state so the gradient is along the direction $(-1, 1)$. When we apply sketches (say CountMin sketches), it collapses the direction $(-1, 1)$ into a single

Algorithm 1 LC-CHECKPOINT-BASED SGD

```
Input: $\mathbf{u}^*$, $\mathbf{u}_0$, $\eta$
1: Initialize $\tilde{\mathbf{u}}_0 = \mathbf{u}_0$.
2: for $t = 1$ to $T$ do
3:     Update model state: $\mathbf{u}_t = \mathbf{u}^* + \eta(\mathbf{u}_{t-1} - \mathbf{u}^*) + \epsilon$
4:     Compute distance: $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1}$
5:     Quantize $\delta_t$: $\tilde{\delta}_t = \text{QUANTIZE}(\delta_t)$
6:     Compress $\tilde{\delta}_t$ by Huffman coding and save to disk
7:     Update checkpoint state: $\tilde{\mathbf{u}}_t = \tilde{\mathbf{u}}_{t-1} + \tilde{\delta}_t$
Output: $\mathbf{u}_T$, $\{\tilde{\delta}_t \mid t \in [T]\}$
```

point 0. When we make a query, the gradients for both coordinates are incorrect. Sketches are more useful when the entries in the gradient vector are heterogeneous and queries need to be answered at "line rate" (e.g., do not slow down the training [59]). Here, when a model needs to be recovered from a checkpoint, the job is less time-sensitive. Therefore, even we face heterogeneous parameters, it is more effective to carefully disentangle crucial information from inconsequential ones than using arbitrary random projections.

2.3 LC-Checkpoint-based SGD

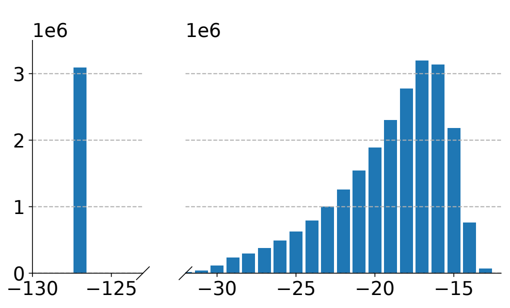
We now describe our solution LC-Checkpoint (LC refers to Lossy Compression). See Figure 2.1 for a working example and Algorithm 1 for a workflow. For simplicity, we assume that our system maintains a checkpoint $\tilde{\delta}_t$ for each iteration. We slightly abuse δ_t to refer to both the compressed data and the real vector it represents. It is straightforward to downsample our operations to construct a checkpoint every k-iterations. Our solution consists of two major components.

C1. Approximate tracking by delta-coding. At each step, our system maintains an approximation $\tilde{\mathbf{u}}_t$ of the ground-truth state. We simply set $\tilde{\mathbf{u}}_t = \mathbf{u}_0 + \sum_{i \leq t} \tilde{\delta}_i$, where $\mathbf{u}_0$ is the initial state of the model. Our system continuously maintains and updates $\tilde{\mathbf{u}}_t$ at the background (line 7 in Algorithm 1). Our major compression task is to properly track the "delta" between the approximate state and ground-truth. Specifically, the compression task for the t-th iteration is $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1}$. See ③ in Figure 2.1.

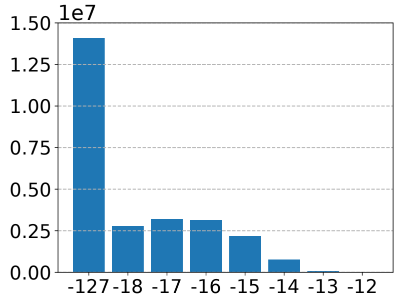
C2. Quantization and Huffman coding. This component compresses δ_t through two steps, *Step 1. Two-stage quantization.* We first perform an exponent-based quantization, and then a priority promotion operation. This operation intelligently drops inconsequential information between two consecutive states. *Step 2. Lossless compression by Huffman.* Finally, the quantized distance vector is further compressed using Huffman coding.

One can see that to reconstruct the model state at iteration t from the checkpoints, we may simply compute $\mathbf{u}_t = \mathbf{u}_0 + \sum_{i=1}^{t} \tilde{\delta}_t$.

In what follows, Section 2.3.1 discusses C2 and Section 2.3.2 discusses additional system-level optimizations.



(a) Exponent distribution of δ.



(b) Exponent distribution of $\tilde{\delta}$ (3-bit promotion).

Figure 2.2: The distribution of all elements' exponent parts in the last convolutional layer of AlexNet. When e equals -127, the element value is 0. The x-axis denotes the exponent part value, and the y-axis indicates the count of elements with this value.

2.3.1 Quantization and Huffman coding

2.3.1.1 Two-stage Quantization

LC-Checkpoint employs a novel two-stage pipeline to quantize δ_t, which consists of two main sub-steps: exponent-based quantization and priority promotion.

Exponent-based Quantization. Recall that a floating point v is represented by $v = (-1)^s \times m \times 2^e$, where s is the sign, m is the mantissa, and e is the exponent. Recall that $\delta_t = \mathbf{u}_t - \tilde{\mathbf{u}}_{t-1} \in \mathbf{R}^n$ is a high-dimensional vector we aim to encode. Our exponent-based quantization works as follows: first, it partitions entries in δ into multiple buckets according to e and s, i.e., it assigns the elements with identical exponents and signs to the same bucket. Our crucial observation from extensive experiments is that entries in $\mathbf{u}_t$ usually drift towards the same direction, so δ_t typically have the same sign. Next, our algorithm represents each bucket by the average of maximum and minimum values in the bucket.

Figure 2.1 ② shows an example, in which, δ_t is quantized into five buckets (marked with five different colors). All entries in each bucket are then represented by a unique value.

Indexing k buckets requires $\log_2 k$ bits. Because δ_t consists of n floating points, each of which uses b (e.g., $b \in \{32, 64\}$) bits, the compression rate is $r = \frac{nb}{n \log_2 k + kb}$.

For example, in Figure 2.1, δ has 10 elements (i.e., $n = 10$), each of which is represented by a single-precision floating point (i.e., $b = 32$). Thus, the original δ has nb, i.e., 320 bits in total. Exponent-based quantization uses 5 buckets (i.e., $k = 5$). Thus, after quantization, δ has $(10 \times \log 5 + 5 \times 32 = 190)$ bits. Therefore, the compressing rate (r) is 1.68 (i.e., 320/190).

It is critical to control the number of buckets k to achieve an optimal compression ratio. Fortunately, the exponent-based bucketing can control $k \leq 2^9$ for single-precision floating point elements, and control $k \leq 2^{12}$ for double-precision. [1] Our evaluation results (Section 2.4.3) confirm that usually $k < 2^5$ suffices. Figure 2.2a plots the distribution of all elements' exponent parts in the last convolutional layer of AlexNet.

Priority Promotion. We further improve the compression ratio by limiting the number of buckets with a priority promotion approach. Our crucial observation is that when $\delta_{t,i}$ is excessively close to 0 (i.e., $\tilde{\mathbf{u}}_{i,t-1}$ is close $\mathbf{u}_{i,t}$), it is more effective to batch the updates (i.e., do not update the i-th entry of δ_t until it becomes substantial). Note also this

[1] Single-precision floating point numbers use 8 bits to store e, and together with a sign bit—that is why $k \leq 2^9$. Similarly, double-precision numbers use 11 bits to store e.

is conceptually different from minimizing construction errors. Minimizing construction errors corresponds to exactly keeping track of the heaviest entries in δ_t, whereas we both remove excessively small entries and quantize large entries (as done in the previous step). Specifically, we propose x-bit priority promotion. It keeps $2^x - 1$ buckets with larger e only and merges the rest buckets into one with a unique value of 0. In other words, priority promotion updates $\tilde{\mathbf{w}}_i$ with a larger distance to $\mathbf{w}_i$ with a higher priority. It limits the index of buckets within x bits.

Figure 2.1 (Priority Promotion) uses 2-bit priority promotion to control the number of buckets under 4. It merges the green and purple buckets into a red one that is represented by a value 0. Indexing these buckets only needs 2 bits. Figure 2.2b gives a real example of 3-bit priority promotion for the last convolutional layer in AlexNet.

2.3.1.2 Huffman Coding

Finally, observing the number of elements in each bucket is highly non-uniform in most learning processes, we use Huffman coding [125] to further compress the bucket. For example, Figure 2.2a plots the distribution of all elements' exponent parts in the last convolutional layer of AlexNet. This distribution shows a skewed behavior,

thus more suitable for Huffman coding. Our crucial observation is that priority promotion *further aggravates* the skewness of this distribution (Figure 2.2b), thus marrying quantization with Huffman coding produces more than "sum of parts" benefits. Our later evaluation validates it (Section 2.4.3).

2.3.2 System Optimizations

LC-Checkpoint also comprises several novel system-level optimizations as follows:

- **Asynchronous Execution:** Because only the first step of LC-Checkpoint depends on the model state, the rest steps can run simultaneously with the next iteration of SGD computation. This asynchronous (non-blocking) execution significantly reduces the checkpoint overhead, and mitigates the blocking of model execution.

- **Checkpoint Merging:** To further reduce the recovery time, LC-Checkpoint employs a helper process to merge multiple checkpoints into *super-step* ones, periodically. In case of any system crash, LC-Checkpoint uses these *super-step* checkpoints for recovery.
- **Huffman Code Table Caching:**

  The number of buckets may stay the same from one iteration to another, specifically after priority promotion. Thus, it is possible to reuse the Huffman code table (with only a simple sort of buckets according to the number of entries in each bucket) among different iterations without any rebuilding. LC-Checkpoint comprises a lightweight cache to store the Huffman code table for each buckets count.

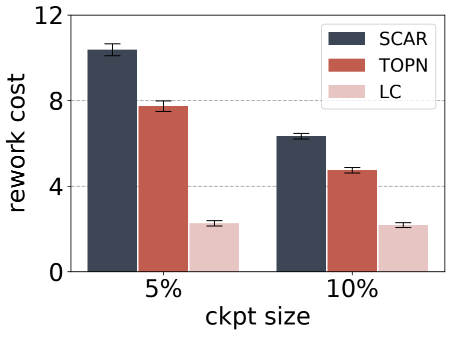
2.4 Experiments

This section evaluates LC-Checkpoint on four typical ML applications with three benchmark datasets, and compares it with previous efforts (SCAR [108] and a TOPN mechanism as mentioned in Section 2.2) on recovery (rework) cost, compression ratio, and execution overhead, demonstrating the superiority of LC-Checkpoint.

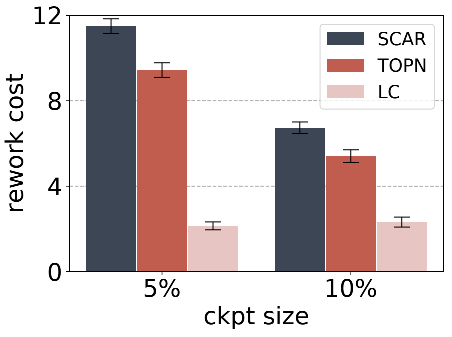
2.4.1 Methodology

Evaluation Objective: This evaluation has four main objectives: (1) comparing LC-Checkpoint' recovery (rework) cost with previous work; (2) evaluating the compression benefits brought by different approaches mentioned before; (3) specifically, validating the effectiveness of priority promotion; (4) confirming that LC-Checkpoint incurs low overhead by an experiment case study. Our work is mainly compared with two state-of-the-art efforts: SCAR [108] and a TOPN mechanism. SCAR partitions the parameters and updates one partition in each iteration to reduce the checkpoint size. The TOPN mechanism only updates the parameters with the top-n largest distances to the previous iteration. The TOPN checkpoint is stored in a compressed sparse row (CSR) format.

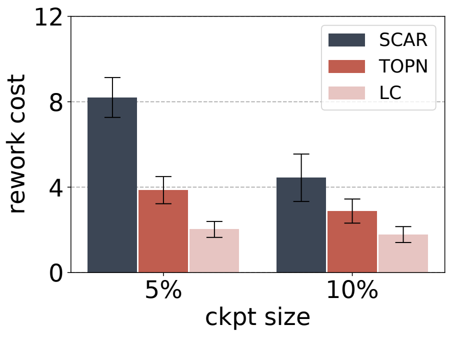
ML Applications and Datasets: LC-Checkpoint is evaluated on four typical ML ap-



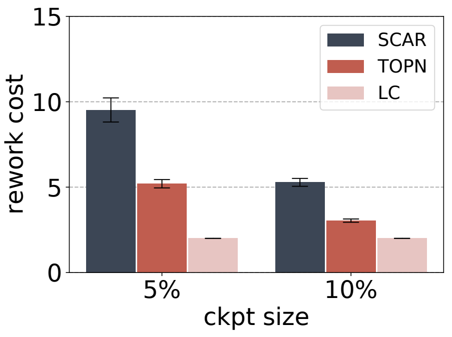
(a) MLR on MNIST.



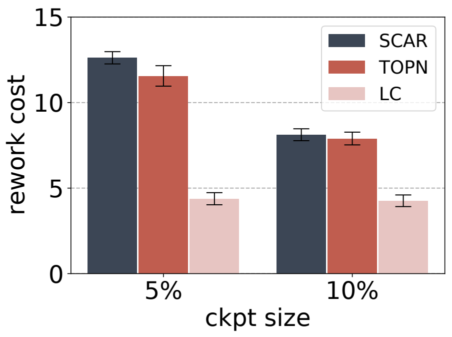
(b) LeNet on MNIST.



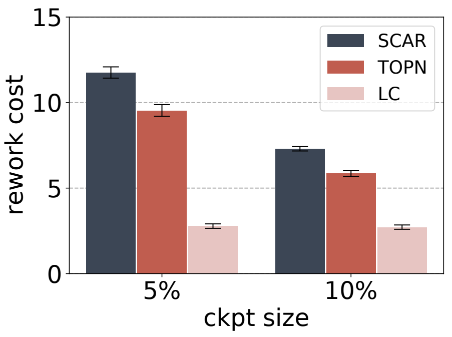
(c) AlexNet on MNIST.



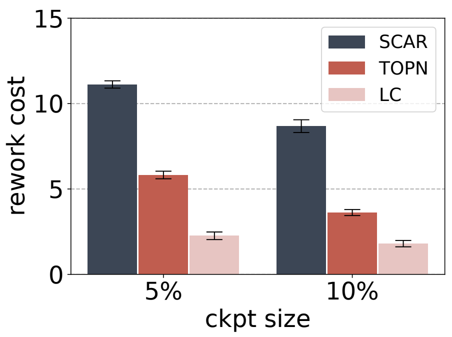
(d) MF on MovieLens.



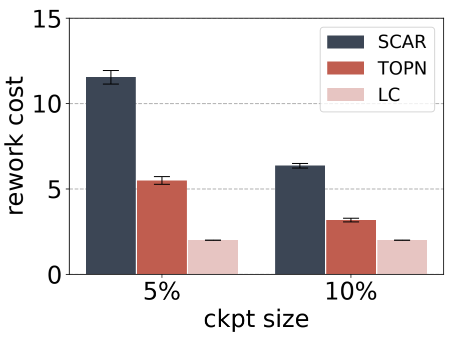
(e) MLR on FashionMNIST.



(f) LeNet on FashionMNIST.



(g) AlexNet on FashionMNIST.



(h) MF on Jester.

Figure 2.3: Rework cost comparison among LC-Checkpoint, SCAR, and TOPN. The x-axis indicates the ratio of the compressed checkpoint size over the full checkpoint size. The y-axis shows the rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 50 times.

plications: Multinomial Logistic Regression (`MLR`), LeNet-5 (`Lenet`) [78], `AlexNet` [73] and Matrix Factorization (`MF`). The first three applications are trained on `MNIST` [78] and `FashionMNIST` [137] datasets. The last one, `MF` is trained on `Jester` [42] and `MovieLens10M` [51].

Platforms and Evaluation Configurations: Our experiments are conducted on a multi-core server with an Intel Xeon Gold 6138 Skylake CPU with 40 cores, each running at 2.0 GHz, and 192 GB DDR4 memory. The training is performed on a Tesla P100 GPU with 16GB High-bandwidth Memory (HBM).

2.4.2 Recovery/Rework Cost Comparison

This section evaluates the recovery (or rework) cost of LC-Checkpoint, particularly comparing it to SCAR [108] and a TOPN mechanism[2].

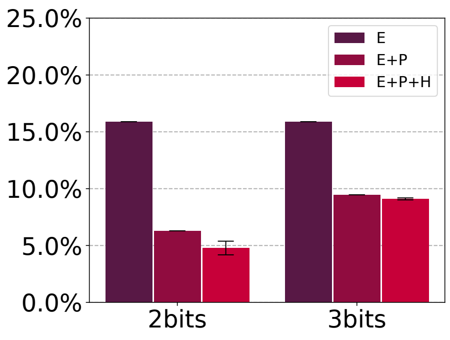
To evaluate their rework costs fairly, we use the same checkpoint size (update size) for all three methods. Two checkpoint sizes are tested: 5% and 10% of the full checkpoint size[3]. These checkpoint sizes can be set directly for SCAR and TOPN. However, LC-Checkpoint's size is determined by the data distribution and thus changed dynamically. To address this issue, LC-Checkpoint employs 2-bit and 3-bit priority promotion that control its checkpoint size at 5% and 10%. Figure 2.4 reports more details of LC-Checkpoint's checkpoint size information.

Figure 2.3 compares the rework cost of three methods, SCAR, TOPN, and LC-Checkpoint, showing that LC-Checkpoint incurs the lowest rework cost for all ML applications and datasets among them. For the 5% checkpoint test case, LC-Checkpoint outperforms SCAR by 2.88×-5.77×, and TOPN by 2.17×-4.06×, respectively. With 10% checkpoint size, LC-Checkpoint outperforms SCAR by 1.9×-4.82×, and outperforms TOPN by 1.52×-2.17×, respectively.

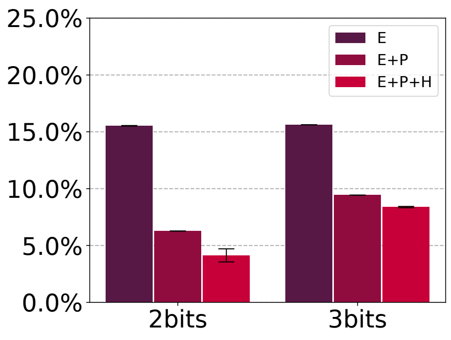
In addition, comparing two checkpoint sizes (5% v.s. 10%), LC-Checkpoint results in

[2]Rework (or recovery) cost is defined as the number of iterations from $\tilde{\mathbf{u}}_t$ to $\mathbf{u}_t$. All methods share the same SGD computation cost for each iteration.

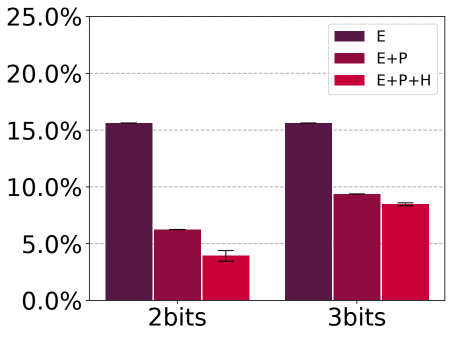
[3]Full checkpoint stores all model parameters after a specific iteration.



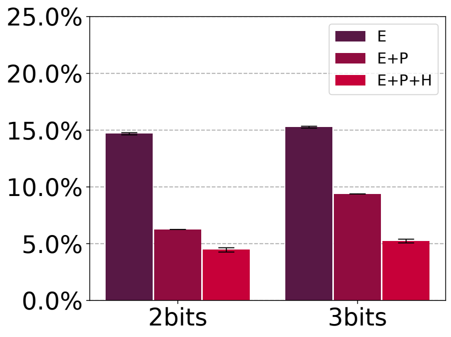
(a) MLR on MNIST.



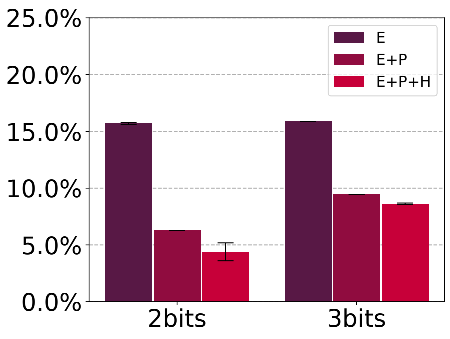
(b) LeNet on MNIST.



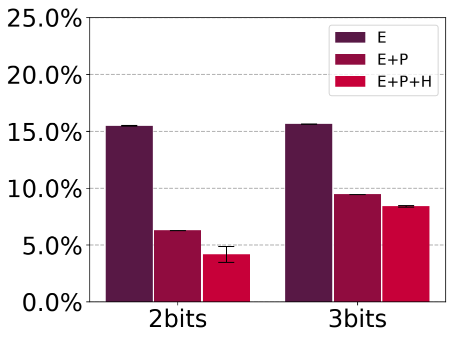
(c) AlexNet on MNIST.



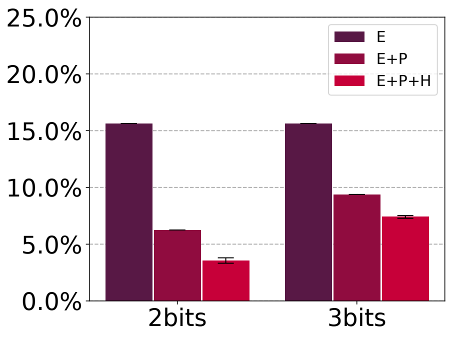
(d) MF on MovieLens.



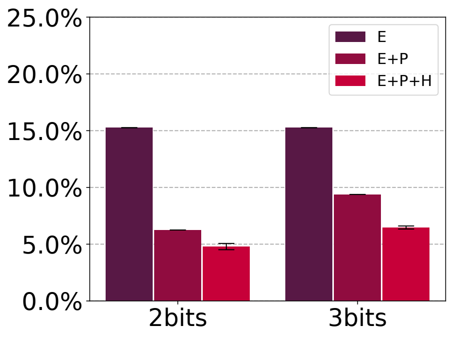
(e) MLR on Fashion-MNIST.



(f) LeNet on Fashion-MNIST.

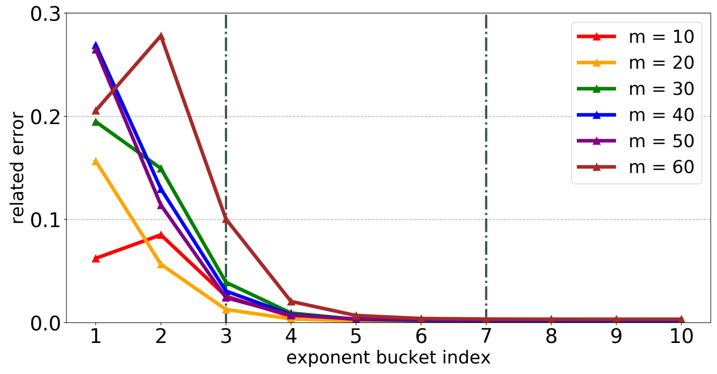


(g) AlexNet on Fashion-MNIST.

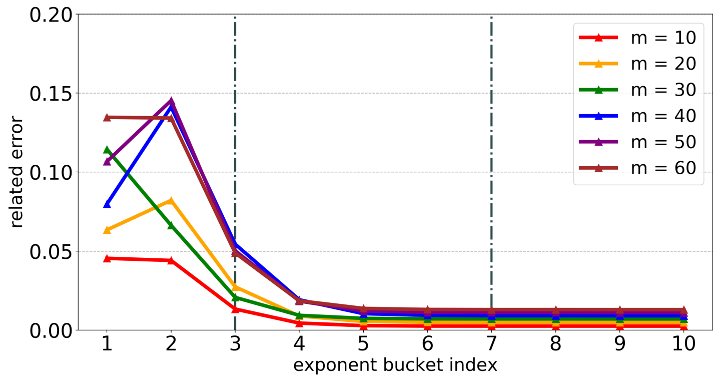


(h) MF on Jester.

Figure 2.4: The compression ratio with different compression methods. The x-axis denotes the bits count used in priority promotion, and the y-axis is the ratio of the checkpoint size after compression over the one before compression. E, P, H denote "exponent-base quantization", "priority promotion", and "Huffman coding", respectively.



(a) MLR on MNIST.



(b) MLR on FashionMNIST.

Figure 2.5: Evaluation on the priority of each exponent bucket. The x-axis denotes the id of the exponent bucket that is deleted. The y-axis shows the relative error to the ground-truth.

more stable rework cost as the checkpoint size decreasing. For example, decreasing the checkpoint size from 10% to 5%, LC-Checkpoint has a negligible rework cost increase on `LeNet` with `MNIST` (Figure 2.3b) and `AlexNet` (Figure 2.3c, 2.3g). It does not have any rework cost change for other cases. In contrast, SCAR and TOPN increase 1.6× rework cost on average as the checkpoint size changing from 10% to 5%.

2.4.3 LC-Checkpoint Compression Effect Breakdown

This section evaluates and analyzes the compression effect of different approaches mentioned before, exponent-base quantization (`E`), priority promotion (`P`), and Huffman coding (`H`). Figure 2.4 reports the compression ratios with 2-bit and 3-bit priority promotion. With all compression approaches, the ultimate checkpoint sizes (`E+P+H`) are all below 5% with 2-bits, and below 10% with 3-bits over the uncompressed full checkpoint, i.e., the compression rates are above 20× and 10×, respectively.

Exponent-base quantization yields a compression ratio of 85% on average. It proves that the exponent parts of all parameters in δ span across a small range of all values that can be represented by single precision floating-point. 15% also indicates that the bucket number $k < 2^5$, because the average bucket number can be estimated as $k = 2^{(32 \times 15\% = 4.8)}$, where 32 is the width of single precision floating-point. Priority promotion brings 9.26% extra compression ratio on average for 2-bit and 6.23% for 3-bit. For most cases, priority promotion with smaller bits yields more benefits for Huffman coding except MF (Figure 2.4d, 2.4h). This is because MF's parameters are sparse, thus Huffman coding can reach a sufficient compression ratio without aggressive priority promotion. Across all models (and datasets), Huffman coding brings 2% extra compression ratio with 2-bits priority promotion, and 1.6% with 3-bits one on average.

2.4.4 The Effectiveness of Priority Promotion

This section further discusses the effectiveness of priority promotion. It aims to prove that priority promotion is able to save the majority of high priority parameters. We prove it by showing the exponent buckets result in a larger impact on the model state when their represented unique values are further from 0 (i.e., e is larger).

Assume δ is calculated from one state $\mathbf{u}_\theta$ to another for m iterations. Then, δ_m^i is created by setting the parameters in the i-th exponent bucket to 0. The ground truth is calculated as $V_{gt} = L(\mathbf{u}_\theta + \delta_m)$ where $L(x)$ denotes the loss function. Then the relative error is calculated as:

$$E_m^i = \frac{\left\| V_{gt} - L(\mathbf{u}_\theta + \delta_m^i) \right\|_2}{V_{gt}} \tag{2.2}$$

Figure 2.5 reports the result of `MLR` with $m = 10n, n \in [1, 6]$. Both datasets (`MNIST` and

`FashionMNIST`) on varied m prove that the elements in the buckets with the top-n largest distance impact more on the model (denotes as a higher relative error when the bucket represented value is set to 0).

In addition, it is possible to preserve all *important* buckets with only a small number of index bits. For example, using 2-bit priority promotion (4 buckets with the last bucket storing 0) can easily preserve the most important buckets, and using 3-bit (8 buckets) can preserve all effective buckets. This result explains why priority promotion can compress the checkpoint with negligible accuracy loss.

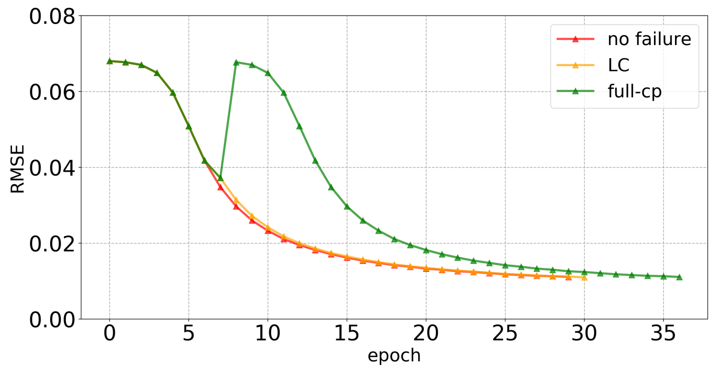


Figure 2.6: MF on MovieLens25M. The x-axis denotes the iteration and the y-axis is the model's RMSE (Root Mean Square Error).

2.4.5 A Case Study on LC-Checkpoint's Overhead

This section evaluates LC-Checkpoint's execution overhead and overall impact on the model execution using a case study, i.e., training `MF` on `MovieLens25M` [51] dataset. Each iteration costs 91 seconds on average. LC-Checkpoint employs 3-bit priority promotion, resulting in a checkpoint size below 10% (of the uncompressed full checkpoint size). Default approach creates a full checkpoint every 10 iterations. A failure is triggered at the 7-th iteration.

Figure 2.6 reports the result. LC-Checkpoint only incurs one extra iteration than the normal execution without any failure to convergence, and saves 6 iterations compared to the full checkpoint method, i.e., saving 546 seconds execution time. LC-Checkpoint introduces only less than 4 seconds (i.e., around 4%) overhead for each iteration, which is negligible.

2.5 Related Work

Fault-tolerance is a key fundamental support for ML systems. Li et al. [83] propose a runtime parameter replication approach for recovery. Tensorflow [3] employs periodic checkpoint to save the model state. Other efforts like [50, 107] aim to support strong consistency semantics. In contrast, our work relaxes the consistency guarantee of checkpoint based on the self-correcting behavior of ML applications. With a set of lossy compression mechanisms, our work can afford high frequent checkpoints, resulting in low rework cost and fine-grained model state recovery. Similarly, Qiao et al. [108] also propose a fault-tolerant solution (SCAR in our evaluation) based on weak consistency by partially updating parameters. SCAR is potential to store redundant information during checkpointing according to our evaluation, and our work aims to eliminate such redundancy by selectively saving the distance between two states.

Model compression has been proposed to reduce model storage space and accelerate model execution time, simultaneously. Weight pruning and weight quantization are two important categories of model compression.

Some popular weight pruning techniques closely related to our work are summarized as follows. Guo et al. [46] present a dynamic network surgery approach with on-the-fly connection pruning to reducing the network complexity. Dai et al. [27] combine the growth and the pruning phases in training to generate compact DNN architectures. Han et al. [49] design Deep Compression, a model compression approach by combining pruning, quantization, and Huffman coding. Mao et al. [90] carefully explore the impact of varied pruning granularity on model accuracy and propose a coarse-grained weight pruning approach. All effort above aims to prune model weights without compromising accuracy. Different from them, our work eliminates the redundancy between two checkpoints and reduces the rework cost during recovery by designing a reliable coding scheme working throughout the entire dynamic process of learning.

Weight quantization is also widely used for model compression. BinaryConnect [24] introduces the binary weight for replacing multiplication by addition and subtraction. Binarized Neural Networks [25] also use binary weights and activations to accelerate computation. Park et al. [99] propose a clustering method based on weighted entropy for weight quantization. Leng et al. [81] formulate quantization as an optimization problem and solve it by ADMM. Our approach also employs quantization to reduce the bits of parameters by designing a novel exponent-based quantization technique. Moreover, our approach emphasizes filtering the parameters with a new priority promotion method.

2.6 Summary

This work presents LC-Checkpoint, the **first** checkpoint scheme based on lossy compression to achieve the maximal compression rate and efficient recovery simultaneously. It employs a novel two-stage quantization method consisting of exponent-based quantization and priority promotion to identify and store the most critical information for SGD to recover, and leverages Huffman coding to further benefit from the non-uniform distribution of gradient scales. Our evaluation demonstrates that LC-Checkpoint achieves a compression rate up to 28× and recovery speedup up to 5.77× over the state-of-the-art algorithm (`SCAR`).

Chapter 3

Parallel Software for Million-scale Exact Kernel Regression

3.1 Introduction

Kernel learning refers to a set of learning algorithms that map the original features to a possibly infinite dimensional space and use them to learn a model with tractable/convex objectives. For example, kernel ridge and kernel principal component regression learn a linear model in the feature map, whereas the kernel support vector machine maximizes the margins of training data represented by the feature map. Because kernel learning algorithms usually enjoy sound theoretical properties, they had been extensively used in a wide range of areas such as (medical) image recognition [17, 115, 65], bioinformatics [94, 61], asset pricing [129], recommendation systems [41, 84], smart cities [142, 140], etc. Although in recent years some major downstream "users" move to use deep-learning-based models, applications that require interpretable models and robust reproducibility (e.g., different "random seeds" will not result in models with different performance) still heavily utilize kernel techniques. In addition, it was discovered recently that a neural net with infinite width is equivalent to kernel regression using the so-called neural tangent kernel (NTK) [60, 96]. Therefore, kernel techniques remain relevant for both specialized applications and a better understanding of deep learning.

This work presents the design and implementation of a software package that solves large-scale kernel principal component regressions (KPCR). Kernel principal regression and kernel ridge regression (KRR) are the two most widely used linear kernel models and they often deliver similar performance [30]. We choose KPCR over KRR because the former is more interpretable, especially when the dimension of the kept kernel map is small but the software and techniques developed here can be easily extended to KRR, which would require only a switch to a linear solver instead of an eigensolver.

Scaling up KPCR has been challenging for two reasons: (a) most software uses dense eigenvalue solvers, requiring $O(N^3)$ running time and $O(N^2)$ space, where N is the number of observations (samples); (b) parallelization, when offered, is typically performed across different problem instances rather than to allow for larger problem sizes. However, there is strong interest in techniques and software that handle datasets with a million or more samples, which is at least an order of magnitude larger than what off-the-shelf packages like scikit-learn are capable of. Such datasets are both conceptually and practically important. First, many applications in finance, medicine, and vision have benchmarks with 500K to 3 million samples [136, 135]. For example, the (original) ImageNet has 1.2 million images [29] which, to our knowledge, cannot be studied with current KRR/KPCR solvers. Second, recent investigation on deep learning demands a scalable kernel learning solver to better understand approximation error between NTK and different deep artechiture [96, 6]. To address this scaling problem, a significant stream of research has focused on reducing the time complexity of the kernel matrix through approximations [139, 32, 131, 110, 141]. The design of our software tool is independent of how the user provides the matrix-vector multiplication and therefore it allows for use of kernel approximations. In this work, however, we address the exact (non-approximated) kernel matrix as it is more general, more computationally challenging, and still needed by practitioners as it obviates the need to bound another source of error.

A software package that scales up the currently feasible problem size by more than an order of magnitude must leverage multiple existing technologies. For example, iterative methods for eigenvalues or linear systems can and have been used to bring the complexity down to $O(N^2)$ [128] but without distributed computing, the target problem sizes would still be infeasible. In addition, the software should employ techniques that build and apply the kernel matrix efficiently, include hyperparameter optimization in the pipeline, and the eigensolver performance should be tuned for KPCR.

Our solution consists of the following components: *(i)* the use of a state-of-the-art distributed-memory eigenvalue iterative solver that computes k selected eigenpairs in $O(kN^2)$, *(ii)* the development of high performance computing matrix-vector multiplication routines that employ both multi-threading and distributed memory parallelism, and, when needed, can work under limited memory by rebuilding the kernel tile-by-tile on the fly, *(iii)* the development of a software pipeline consisting of two interacting Python front-end drivers, one handling the hyperparameter optimization and the other the regression on an HPC back-end, ensuring a fault-tolerant execution. The main contribution of the work is the design of a software tool using novel algorithmic integration rather than the introduction of new algorithms.

We demonstrate the efficacy and efficiency of the software on two million-scale downstream applications. First, we apply it on 2 million observations from empirical asset pricing, a notoriously difficult ML problem with a low signal-to-noise ratio. The fast execution of our method allows us to implement a boosted KPCR which demonstrates superior performance [44]. Second, we run KPCR on the entire Imagenet dataset, which to our knowledge is the first time a linear kernel model is used to fit Imagenet. We run this as a feasibility and stress test for our HPC software by computing tens of thousands of eigenvectors. Although, because of $O(N^2)$ complexity, we cannot expect scalability to sizes beyond $O(10^7)$, our experiments demonstrate that we can enable the solution of a variety of important problems with datasets of size $O(10^5 - 10^7)$.

3.2 Preliminaries and related work

Kernel Regression. We consider the problem of fitting a real-valued function using a total number of N data points $\{\mathbf{x}_i, y_i\}_{i \leq N}$, in which $\mathbf{x}_i \in \mathbf{R}^F$ and $y_i \in \mathbf{R}$. Let $k(\mathbf{x}_i, \mathbf{x}_j)$ be a (positive semi-definite) kernel function that intuitively describes (dis)-similarities between $\mathbf{x}_i$ and $\mathbf{x}_j$. By Mercer's theorem [114], there exists a feature map $\phi(\mathbf{x})$ such that $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. Kernel regression models aim to find a linear relation $y \sim \langle \beta, \phi(\mathbf{x}) \rangle$. Note that $k(\cdot, \cdot)$ is part of a model specification and ϕ does not need to be computed explicitly. In addition, the dimension of $\phi(\mathbf{x}_i)$ could be infinite so regularization is needed. When we add a regularizer of $\lambda \|\beta\|_2^2$, the model becomes kernel ridge regression (KRR), and when we use a low rank matrix to approximate the Gram matrix $K \in \mathbf{R}^{N \times N}$, where $K_{i,j} = k(x_i, x_j)$, the model becomes kernel principal component regression (KPCR). It was recently shown that KRR and KPCR are mostly equivalent [30] but the latter often is considered more interpretable because it has a smaller number of learnable parameters. Examples of kernel functions include the Gaussian kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$, inner product kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$, and polynomial kernel $k(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + \gamma)^d$. Neural tangent kernels (NTK) are a new family of kernel functions discovered recently that approximate neural nets at the width limit [60]. Computing NTK usually is more resource hungry [96].

Kernel regression can also be used to solve classification problems using standard transformations and is found to be effective [9]. A drawback is that c kernel models need to be fitted to solve a classification with c classes.

Computational challenges. There are two major challenges.

1. Memory and computation. It takes $\Theta(N^2)$ space to store the matrix K. Fitting the model requires us to solve a large least squares system in the form $\tilde{K}\alpha = y$ (obtained from the dual of the MSE cost in β). In KRR, $\tilde{K} = K + \lambda I$, whereas in KPCR, $\tilde{K}$ is a low-rank approximation of K. A combination of the two is also possible. The solutions can be obtained by direct methods e.g., inverting K or $\tilde{K}$, or performing the SVD decomposition using the LAPACK library [8]. The cost of any of these computations is $O(N^3)$ which makes it prohibitive for massive datasets.

Iterative methods have been a central tool in large scale scientific computing [10, 12] but until recently had not received much attention in kernel learning. Based on matrix-vector multiplication, methods such as Conjugate Gradient can solve the KRR problem iteratively with complexity $O(N^2)$ [128], even for many different regularization parameters λ [38]. The Lanczos method computes k largest eigenpairs to form a low rank approximation $K = V_k \Sigma_k V_k^T$ with complexity $O(kN^2)$. Besides interpretability, the benefit of the low rank approach is that we can later solve $\alpha = V_k \Sigma_k^{-1} V_k^T y$ for many different y or even combine it with many different λ.

Recently, some packages for kernel regression have included a sequential version of the restarted Lanczos method (through the ARPACK software [80]) or the Randomized SVD method [47]. See for example [40, 100, 19, 4, 77, 91]. However, they cannot address large scale problems that will not fit into the memory of a single server or that may require a large number of compute nodes (or GPUs) to reduce execution time. Moreover, the optimization of the matrix vector multiplication is critical for performance but is typically left to the user. Other approaches avoid the solution of the entire kernel matrix by randomization, partition, gradient descent methods, or fast multipole approximations [139, 32, 131, 110, 141] but they need to bound the distance of the obtained solutions from those of the original problem, and they may still pose great computational demands. Although, our software tool could be used in this case, we focus on the exact kernel solution.

Our work takes a holistic approach to Kernel regression solvers, optimizing both distributed matrix vector multiplication for a given kernel and memory constraints, and the underlying iterative solver for these problems. We employ the use of PRIMME, one of the state-of-the-art parallel eigensolvers [120]. By adjusting block size, basis size, and other parameters, PRIMME methods can be tuned to converge nearly optimally and its MPI implementation can work on distributed memory computers. Other high quality packages include SLEPc [52] and Anasazi [11], but they are much bigger and less agile, while not performing better than PRIMME [120].

2. Hyperparameter search and boosting. In practice, extensive hyperparameter search is crucial, e.g., finding the optimal γ of the Gaussian kernel. Search methods include discrete grid search, Bayesian optimization, and learning curve based optimization [67]. When we perform a search, the data is usually split into training and validation (or testing) sets, in which the training set is used to fit a function with a specific hyperparameter set. The quality of the fitting is determined by using the validation set to compute a score. Typical scoring functions include MSE, classification error, and correlation between validation and predicted outputs. Hereafter, we use T and S to denote quantities related to the training and validation set, respectively (e.g., K_T is the Gram matrix formed from the training set). Hyperparameter search adds an extra dimension of complexity to the computational cost of fitting a kernel model (i.e., thousands of models may need to be fitted to find a reliable hyperparameter). Different models can be fitted independently in parallel, and this task based parallelism can be exploited in some packages such as TensorFlow. However, we are not aware of a software package that combines this with distributed memory, and parallel tasks, and this is the second thrust of this work. In addition, models fitted from different hyperparameters usually extract complementary signals. A boosting method consolidating predictions from multiple models usually further strengthens forecasting power and is also frequently used in practice [71].

In the following, we describe our design and implementation of a Machine Learning software pipeline that integrates hyperparameter optimization with a high performance, parallel regression solver, addressing the entire stack: kernel generation, matrix-vector, and eigensolver optimization, and evaluation.

3.3 Driver Workflow

The regression and the automatic machine learning are performed by separate drivers, the regression and evaluation driver and the hyperparameter optimization driver. Both are

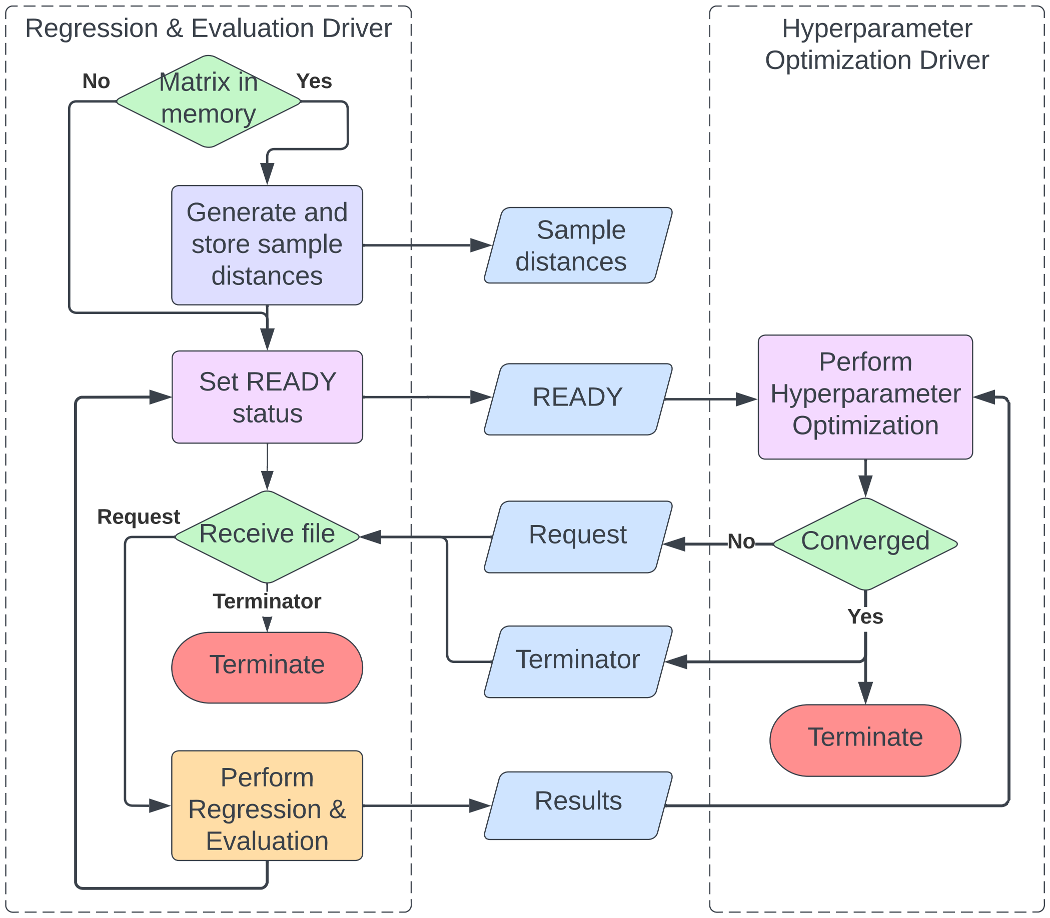

Regression & Evaluation Driver
No
Matrix in memory
Yes
Generate and store sample distances
Sample distances
Set READY status
READY
Hyperparameter Optimization Driver
Perform Hyperparameter Optimization
Request
Receive file
Request
No
Converged
Terminator
Terminate
Terminator
Yes
Terminate
Perform Regression & Evaluation
Results

Figure 3.1: Driver architecture and interaction.

written in Python and interact with each other as shown in Fig. 3.1. The hyperparameter optimization driver applies Bayesian optimization and boosting to find the best hyperparameters for the model. It is an iterative method that considers the history of evaluation scores for past choices of hyperparameters, decides which hyperparameters to evaluate next, and passes them to the regression and evaluation driver. The regression and evaluation driver trains the model for the requested hyperparameters, evaluates their fitness score, and returns those scores to the hyperparameter optimization driver to continue the optimization. We can view the hyperparameter optimization driver as a "consumer" of the regression and evaluation driver's results and as an issuer of requests for more results. The regression and evaluation driver coordinates the work requested by assigning it to one or more parallel jobs. Decoupling the drivers into two interacting but independent processes allows for a more flexible, fault tolerant, and scalable design. Under such an architecture, users can switch between different hyperparameter optimization algorithms or different evaluation methods with little effort.

3.3.1 Hyperparameter optimization driver

The hyperparameter optimization driver is initialized with user-defined configurations including Bayesian optimization parameters (e.g. hyperparameter search space and convergence criteria), and directory paths for storing results needed by the regression and evaluation driver. For each hyperparameter optimization iteration, the hyperparameter optimization driver first identifies the hyperparameters that need to be evaluated with Bayesian optimization based on existing observations (past evaluation results from regression and evaluation driver). Then it creates a request file containing the hyperparameters for which the regression and evaluation driver will compute model solutions and waits for the results. When the regression and evaluation driver finishes the evaluations and writes them to disk, the hyperparameter optimization driver locates the result files based on the request ID, adds the evaluation results to the existing observations with the boosting method and updates the Bayesian optimization.

The hyperparameter optimization driver only submits new requests when a regression and evaluation driver is available, as indicated by the existence of a READY file which contains the process ID of the evaluation driver. After submitting a request, the hyperparameter optimization driver resets the status of the regression and evaluation driver by deleting this file.

3.3.2 Regression and evaluation driver

The regression and evaluation driver is initialized with the process ID of the hyperparameter optimization driver and the paths to the training and testing data. If the entire kernel matrix can fit in available memory, the sample distances are computed and saved to disk; the regression solver will be generating the kernel matrix from the sample distances. Otherwise, and assuming the sample data can fit in the memory of one node, the regression solver will be performing the matrix-vector operations on the fly by regenerating portions of the matrix from the sample data. At this point, the regression and evaluation driver enters the request loop with the hyperparameter optimization driver, indicating that it is ready to compute a regression model by creating the READY file containing its process ID.

When ready, the hyperparameter optimization driver responds with a request file, containing a list of hyperparameters for each regression model that needs to be solved. For each hyperparameter in the request, the regression and evaluation solver (architecture seen in Fig. 3.2) computes a model solution for each rank computed by the SVD. It then computes a fitness score for each model solution. The regression and evaluation driver saves to disk the model solution with the highest evaluation score and reports the location of this information back to the hyperparmeter optimization driver in a results file for the current request. Once the regression and evaluation driver has computed the optimal model solutions for all requested hyperparameters, it finalizes the results file, creates a READY file, and awaits a new request from the hyperparmater optimization driver.

3.3.3 Benefits of a two-driver workflow and fault tolerance

It is natural to address a consumer-producer workflow with a design of separate drivers. The Bayesian optimization function can be changed to different software in any language without affecting the code of the numerical solvers and vice versa. For example, Bayesian optimization can work with partial information and update it as new results arrive. On the other hand, the regression and evaluation driver can decide to launch separate parallel jobs to evaluate different hyperparameters if the resources are available.

The hyperparameter optimization and regression and evaluation drivers implement a checkpoint system for fault tolerance. While a driver is waiting, it can periodically poll the OS with the process ID of the other driver program. If it determines that the other driver has terminated, it will save its current state and terminate as well. If the hyperparameter optimization driver has converged, or if some other error prevents it from continuing, it saves its state, creates a terminator file and terminates itself. The regression and evaluation driver will detect this file and follow its own shutdown procedure. Any running regression and evaluation solvers will finalize their results which can be consumed by the hyperparameter optimization driver when the drivers are restarted at a later time. During a restart, the regression and evaluation driver will check if there is a request that was incompletely processed. If it finds an incomplete results file for the last received request, it will solve the regressions for any outstanding hyperparameters and finalize the results file before creating the READY file.

3.4 The design of the kernel learning solver

The software architecture we described earlier is generic and can work with a variety of hyperparameter optimization and regression software, both sequential and parallel. In this section we describe what makes our package unique; the development of a high performance, parallel code that solves the regression problem and forms and evaluates its predictions. To perform at the extreme scale required for large data sets, the code must support distributed memory parallelism. This is because a single high-end server (a) does not have the memory to store dense matrix kernels of size more than a million, and (b) even if the memory is available, the execution time required on one node would be prohibitive. In addition, the code should support multi-threading on each distributed node so that many-core or GPU environments are utilized efficiently.

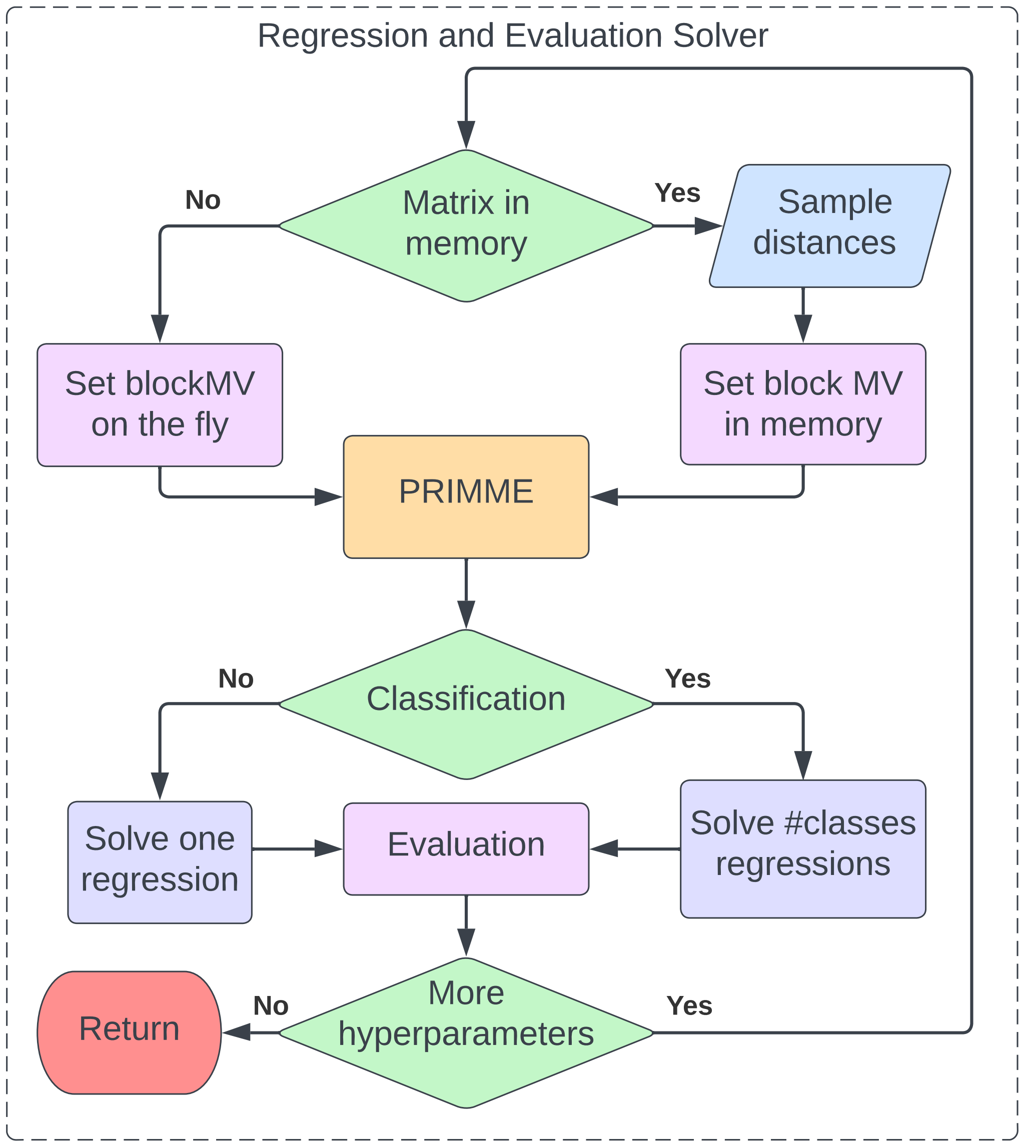


Figure 3.2: Regression and evaluation solver.

Our implementation uses the MPI+X paradigm, with heavy use of optimized LAPACK and BLAS multithreaded libraries. This allows easy transition from many-core to GPU architectures, especially since our iterative eigensolver PRIMME provides a GPU interface through MAGMA [123]. For this work, OpenMP optimizations and tuning have been performed for the target KNL architecture on the Stampede2 supercomputer at TACC [119], funded by XSEDE [124]. The code is written to support both single precision and double precision arithmetic. Mixed precision support is planned for the future.

There are three computational stages. The pre-processing stage is where the training and testing data is read to create the training and testing kernel matrices. The Truncated SVD stage uses the eigenvalue package PRIMME to compute a low-rank eigenspace that sufficiently approximates and regularizes the training kernel. The third stage solves the regression problems with the low-rank space and evaluates the score of its predictions.

3.4.1 Pre-processing

3.4.1.1 Matrix generated on-the-fly

With large scale data, the discriminating factor is the number of features F. If the number of features is small (say $F \leq 10$), the training data, $\mathbf{x}_T$, has low memory demands, $O(NF)$ numbers, and can remain in the memory of each node. The training kernel matrix, however, requires storage of $O(N^2/p)$ numbers, where p is the number of nodes in the parallel program. The matrix is used by the iterative solver in a matrix times a block of vectors multiplication kernel (hereafter called block matvec) with time complexity $O(2bN^2/p)$, where b is the size of the block. Therefore, when the number of samples N is too large to allow for storage of the entire matrix, or to do so we would need an excessive number of nodes, we can instead keep the training set in memory and recompute the matrix on the fly at every block matvec. This increases the block matvec complexity to $O(N^2F/p+2bN^2/p)$ which for $F \ll b$ is a negligible increase. With this flexibility, the code can run in much smaller processor allocations, e.g., in smaller clusters that users may have available, and achieve higher utilization. Moreover, it allows for multiple hyperparameter spaces to be explored in parallel on different partitions of supercomputers.

For this case of on-the-fly generation of K, the pre-processing is straightforward. Depending on the file system, all nodes read the training data, or one node reads it to avoid disk contention and broadcasts it to all other nodes.

3.4.1.2 Matrix generated and stored

When the number of features is large (say 10^4 or 10^5), regenerating the matrix on the fly at every iteration becomes prohibitive. In this case the matrix must be created in the pre-processing stage. We follow the usual distribution, where node j is assigned the task to generate a set of matrix rows with indices denoted I_j. To compute its local part of the matrix, node j would need to compute $k(\mathbf{x}_T(I_j), \mathbf{x}_T)$. However, because of its large size, $\mathbf{x}_T$ may not be stored in local memory. We perform this in parallel with each node storing only two sections of the data; its own $\mathbf{x}_T(I_j)$ and a communicated $\mathbf{x}_T(I_m)$. Specifically, a pipelined ring communication starts by every node j sending its data block to node $j+1$ and receiving the data block from node $j-1$ in a non-blocking fashion. At the same time it computes its local $k(\mathbf{x}_T(I_j), \mathbf{x}_T(I_j)$ part of the matrix. When the new data block arrives, the node forwards it to $j+1$, posts another receive, and computes $k(\mathbf{x}_T(I_j), \mathbf{x}_T(I, j-1)$. The algorithm continues for p steps, at which point all data blocks have been seen and recorded. Since communication time $O(NF)$ is far smaller than computation time $O(N^2F/p)$, we expect it to be completely overlapped.

There are two additional computational considerations. First, for very large F, the matrix generation cost is much larger then the time to solve the problem with the eigensolver. Thus, it would be useful if many more nodes were used to generate the matrix than to solve it. However, this would create different row distributions between the two stages. Second, the regression driver issues requests for the solution of multiple regression problems corresponding to different hyperparameters (often in the order of 100s). For each hyperparameter, a different kernel needs to be generated, which becomes computationally infeasible. One solution would be to transform the $K_{i,j}$ element of the current hyperparameter to the matrix element of the new hyperparameter, e.g., for the Gaussian kernel $K_{ij}^{\gamma_2} = \exp(\gamma_2/\gamma_1 \log K_{i,j}^{\gamma_1})$. We have found this to introduce too much floating point error, especially when storing the kernel in single precision arithmetic. A second alternative is the pre-processing step to store a copy of all distances $\|\mathbf{x}_{Ti} - \mathbf{x}_{Tj}\|$ in memory from which we can compute a kernel for any hyperparameter. This, however, doubles the memory requirements and does not address the problem of different number of nodes between matrix generation and eigensolver.

Our approach is to let the pre-processing stage write all the pair-wise distances $\|\mathbf{x}_{Ti} - \mathbf{x}_{Tj}\|^2$ to a file and not compute the kernel matrix explicitly. A node can then read the file and broadcast the rows to the all the nodes in a new partition so that they can compute locally their kernel matrix. The approach has several benefits; (a) it allows different node distributions and partitions between pre-processing and solver; (b) does not introduce extra floating point error in matrix creation; (c) the computation of the matrix from distances takes less time than a matrix vector multiplication.

3.4.2 Truncated SVD

Applying a high performance iterative eigenvalue solver to compute the Truncated SVD (TSVD) of the kernel is not a ground breaking idea. Yet, most current ML packages use either a complete SVD decomposition through LAPACK or, the most advanced ones, have been using sequential Lanczos or Randomized SVD methods [92]. PRIMME, as one of the state-of-the-art eigensolvers, has been shown to scale well to massively parallel platforms and has been used to find eigenpairs of (sparse) matrices of dimension more than a billion [120]. It is therefore a natural choice to include as the TSVD solver in our package.

Among the several methods available in PRIMME, we choose to work with block GD+K. This method provides near optimal convergence rate, much faster than randomized SVD and at least as fast as LOBPCG, depending on basis size. Because our kernel matrices are dense, any reduction in the number of iterations translates directly to less computing time. The second key issue is the implementation of an efficient and scalable block matvec. The presence of a block matvec is critical to improve the per-node computational intensity and to reduce communication latency during matvec. At the same time if the block is too large convergence of the eigensolver deteriorates. We study the choice of block size in section 3.5. Denote the block matvec as KX where X is of size $N \times b$.

3.4.2.1 Matrix generated on-the-fly

As discussed in the previous paragraph the block sizes that are optimal for our problem are very small (often much smaller than the number of eigenvalues required). This implies that the block matvec cannot benefit from state-of-the-art algorithms for matrix-matrix multiplication [75, 28]. For matrix-vector multiplication, the algorithm found in ScaLAPACK [14] uses a 2D cyclic partitioning of the matrix elements and achieves optimal weak scalability asymptotically. While such a choice of matvec should be provided in the software tool, it is not sufficient for a couple of reasons.

First, the last step of the 2D algorithm involves a global reduction which cannot be fully overlapped with computation. This may not affect the asymptotic weak scalability of the method but for smaller number of processors (a more practical regime for most users) a fully overlapped communication might give better results. Second, when the matrix is generated on-the-fly there is additional computation allowing even a simple 1D algorithm to fully overlap the communication and scale perfectly to larger numbers of processors. Therefore, we provide the alternative of a 1D approach that overlaps communication with both generation of matrix elements and their multiplication.

Each node is responsible for about N/p rows of K and stores the corresponding rows of X. In this case, we have the flexibility to generate the local matrix tile by tile, so that the tile fits in local memory and its size depends on the performance of SGEMM (the BLAS sequential matrix-matrix multiplication function).

Based on this 1D row distribution, a simple pipelined algorithm multiplies the parts of X that have arrived to the node while communicating in a non-blocking fashion the parts of X that will be needed next. We can perform the ring communication of the blocks $X(I_m)$ while overlapping the computation with only the first tile in the local set of rows. After the first tile has been computed, X is resident in its entirety on each node, and the rest of the tiles can be generated and multiplied without communication. With large number of features F this is sufficient.

A more scalable approach is to perform the multiplication of the local I_j rows in p groups of $I_1, \ldots, I_p$ columns, also through a pipelined ring communication of the block vectors $X(I_1), \ldots, X(I_p)$. The benefit is that we can overlap the entire generation of the matrix section $K(I_j, I_m)$ with communication.

It is relatively simple to model the performance of this algorithm, say for single precision arithmetic. At each step, every processor will eventually generate, regardless of tile size, a $N/p \times N/p$ section, multiply it with the Nb/p block of vectors, while communicating Nb/p words. Generating the section takes $O((N/p)^2(3F + 15))$ operations, where F is the number of features based on which the distance is calculated and 15 is an average number of flops required for computing the exponential. Assuming that this computation can achieve a rate of g_0 GFLOPs, the time to do this computation on a node is $O((N/p)^2(3F + 15)/g_0)$. Assuming SGEMM achieves a rate of g GFLOPs, the time to compute the block matvec of this section on a node is $O(2(N/p)^2 b/g)$. In our experiments we observed $g \approx 2g_0$, so we will use this relation in the model.

At the same time, we send $O(bN/p)$ single precision numbers, or $O(32bN/p)$ bits, to the next numbered processor and receive an equal amount from the previous one. Assuming that the network allows this ring communication to proceed without contention (as is the case on the fat tree topology of the Omni-Path network of Stampede2) then the communication time required for this step is $O(32bN/(pw))$, where w Gbps is the network bandwidth. The ratio of compute time over communication time is then greater than one, i.e., communication fully overlapped, if

$$p < \frac{Nw}{16g}\left(\frac{3F + 15}{b} + 1\right). \tag{3.1}$$

Notice that increasing the block size b beyond the one that achieves peak SGEMM performance is reducing the scalability of the algorithm. In fact the optimal block size should be the smallest b for which communication is fully overlapped. With respect to scalability for $N = 10^6$, for the worst case where $F = 0$ and the ratio $w/g < 10^{-3}$, full overlapping of communication should be achieved up to 62 nodes.

3.4.2.2 Matrix in memory

For smaller number of nodes, the 1D method above continues to be competitive. For larger number of processors, a 2D algorithms as the one in ScaLAPACK can be used. The stored distances can be redistributed to the nodes in any desired distribution, including the required 2D block cyclical partitioning.

3.4.2.3 Initial guesses across hyperparameters

A significant advantage of PRIMME is that it can use multiple initial guesses to converge faster to the required eigenspace [120]. Since PRIMME is called repeatedly for a sequence of hyperparameters, the eigenvectors computed from a previous K^{γ_i} can be passed as initial guesses to the $K^{\gamma_{i+1}}$. The quality of these guesses depends on the closeness of γ_i to γ_{i+1}. Therefore, hyperparameters should be processed in a sorted order. However, also the rank k_i depends on each γ_i, with smaller hyperparameters requiring a computation of much smaller ranks. For this reason, we solve the corresponding eigenproblems such that larger hyperparameters that require a larger rank k are executed first. Thus, two successive kernel matrices will have the smallest distance and the computed rank of the previous matrix will always provide enough initial guesses for the following problem. We have followed this technique with the Gaussian kernel, where solving the kernels in order of decreasing γ yields between 30-40% speedup.

3.4.2.4 Choosing the rank of TSVD

Choosing the optimal rank k of the eigenspace is a central problem in statistics and Machine Learning. The role of the eigenspace is to act as a regularizer to the noise inherent in the data, which is typically unknown. Often, experts would have some idea on a lower bound of the smallest eigenvalue to be included in the eigenspace, e.g., $\sigma_k \geq \delta$. Other times, they would relate the σ_k to the norm of $\|K\|$, e.g., $\delta = \|K\|10^{-4}$. PRIMME has an option for a user provided function that implements any desired stopping criterion [120]. Our software implements the above criteria based on a user provided δ. This functionality also allows us to combine stopping PRIMME with the evaluation of a current low rank space. If at a certain point during the iteration, k eigenpairs have converged, we can evaluate them against a set of testing data (see next subsection) and decide whether more eigenpairs need to be computed. This functionality is currently under development but it has been provisioned in the design of the software.

3.4.3 Computing predictions and evaluations

Having obtained a low rank approximation of the training kernel $K_T \approx V\Sigma V^T$, we can solve the linear regression problems $\alpha = V\Sigma^{-1}V^TY$ where Y is the single vector of training responses in regression, while in classification Y has a number of columns equal to the number of classes. As V and Y are distributed by rows, the V^TY involves one global reduction, while other computations are performed in parallel. The predictions are formed as a matrix vector multiplication with the testing kernel matrix $y_{predict} = K_S\alpha$. The algorithm for this multiplication depends on the size of K_S. If the number of rows of K_S is small, it can be distributed by columns and the matvec performed as a set of inner products with a global reduction. If the number of rows is large, we can use the same algorithm as the block matvec with K_T.

Usually, the rank yielding the optimal evaluation is not known and therefore we would like to check the predictions for many or all ranks $i = 1, \ldots, k$. To avoid recomputing the low rank regression for each i, we can update $\alpha^{(i)}$ from the solution at the lower rank $\alpha^{(i)} = \alpha^{(i-1)} + \sigma_i^{-1}V_iV_i^TY$. The computation involves only level-1 BLAS for regression and level-2 BLAS for classification, as well as one synchronization point, but the time is constant for any $i = 1, \ldots, k$.

Finally, $y_{predict}$ needs to be evaluated against the user provided known responses for the testing data y_S. Such responses could be reserved at the start or could be part of a cross validation scheme. We currently provide functions that compute the MSE and correlation of $y_{predict}$ with y_S. For classification we provide a metric that discretizes the real values in $y_{predict}$ to identify the class it corresponds to and compares this with the classes in y_s. We plan to also implement the t-statistic and Sharpe metrics. Ultimately the evaluation function depends on the specific problem and therefore a user-defined function can be provided to do the evaluation.

3.5 Evaluating performance

3.5.1 Matrix generated on the fly

When the matrix is generated on the fly during each matvec, we have to decide on the size of the tile and on the block size which determines not only the single node performance but also the performance of the pipelined matvec algorithm.

Figure 3.3 studies performance trade offs on a single KNL node. The top two subplots show the execution time and the TFLOPs achieved for a variety of block and tile sizes. Different blocks are plotted with different lines, and the block size is annotated at the ends of the line. The red vertical bars depict the time or TFLOPs of the tile generation, which is part of the matvec. Clearly, for small block sizes, most of the time is spent generating the tile, while large block sizes take more time but multiply more vectors and thus amortize the tile generation. The subplot on the right shows that the tile generation gets better performance for smaller tile sizes, while the SGEMM performance prefers tile sizes around 512 to 1280. Performance peaks with block size 128. The third subplot reports the matvec time per block vector multiplied. The best overall performance occurs with tile sizes 768 to 1280 and block sizes 128 and 256 respectively. Given similar matvec times, a smaller block should be preferred in order to reduce the overall runtime of the eigensolver.

Based on the above, we report timings of the regression and evaluation solver on one of the data sets of the stock data in Table 3.1. This is large data set of almost 2 million points, for which 150 eigenvalues are computed. We use a large block size of 150, which

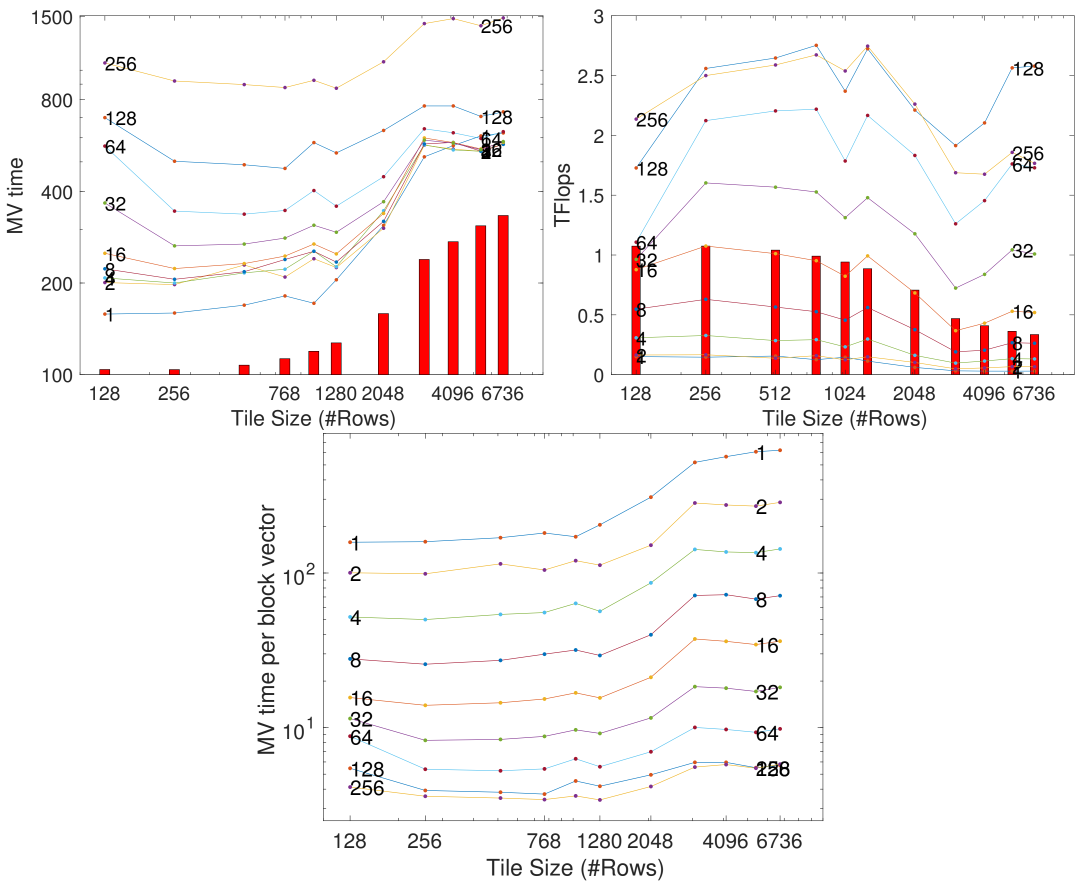


Figure 3.3: Matvec efficiency as a function of tile size and block size. The overlayed numerals indicate the block size, the red bars indicate the overhead of the tile generation.

reduces as eigenvalues converge. We see that matvec takes the vast majority of the time, but the total execution time is reasonable, allowing the solution of regression problems for a series of hyperparameters γ.

3.5.2 Matrix in memory

For the ImageNet dataset, the number of features (150528) and the size of the matrix (1281167) make the regeneration of the matrix from the training data prohibitive. Therefore, we use the methods in the software that distributes the entire matrix over the nodes. A minimum number of 128 Stampede2 nodes were necessary to store the matrix. For each new hyperparameter in the optimization, the inter-distances between points are read

Stage		Time
pre-processing	Read data once	negligible
SVD Computation	# sing. values computed	150
	# outer iterations	15
	# single vector MVs	1304
	Total time	204.2 sec
	Matvec time	189.1 sec
	Tile generation	31.8 sec
	Ortho time	1.4 sec
	AllReduce time	10.3 sec
Low rank regression and evaluation	Regress $x = V_k \Sigma_k^{-1} V_k^T y$	
	Predict and evaluate x	
	Total:	36.1 sec

Table 3.1: PRIMME execution statistics on 64 KNL nodes (4352 cores) for one of the data sets of the stock data with dimension 1996975, with $\gamma = 190$, which yields a more difficult eigenvalue problem, and with (variable) block size 150. The matrix is generated on the fly with a tile size of 1280. Low rank regression is performed incrementally per rank.

from disc and the kernel is recomputed. Next, we study the timings for each stage of this process as shown in Table 3.2.

In the pre-processing stage, we first read the training and testing data and distribute it over the nodes. Then, we run the pipelined tile generation algorithm to compute the local part of the matrix, and finally, write the local matrix of each node to the parallel file system. Clearly, the time is dominated by the generation of the matrix, while the corresponding communication is completely overlapped. Reading the training data and writing the resulting matrix tiles takes less than 10% of the pre-processing time. Because one node reads from disk and redistributes, this I/O time is not scalable as it is not reduced by using more nodes, but it is a cost that any ML method must occur on this machine. However, this operation trivially scales on hardware platforms where each node has a local disk that can hold the desired partition for I/O.

Once the kernel matrix has been generated, our optimization pipeline executes a sequence of calls to PRIMME. The cost of the hyperparameter optimization part of the program is negligible. Using the Gaussian kernel for the ImageNet data set gave the best

Stage		Time
Pre-processing	Read and distribute data	1439 sec
	Pipelined tile generation	27050 sec
	Communication overhead	<3 sec
	Write matrix tiles to PFS	410 sec
	Total:	29305 sec
SVD computation	# sing. values computed	19000
	# outer iterations	336
	# single vector MVs	73863
	Read matrix from PFS	404.4 sec
	Total PRIMME time	2665.7 sec
	Matvec time	1686.9 sec
	Ortho time	652.8 sec
	AllReduce time	416.8 sec
Low rank regression and evaluation	Regress $V_k \Sigma_k^{-1} V_k^T Y$ and evaluate per rank (size(Y,2)=1000)	9.25 sec

Table 3.2: PRIMME execution statistics on 128 KNL nodes (8704 cores) for the entire ImageNet of dimension 1281167, with $\gamma = 10^{-5}$ and block size = 256. The number of classes is 1000 so low rank regression solves Y with 1000 columns. This is performed incrementally per rank. The evaluation was stopped before rank 19000 was reached.

results for hyperparameters $\gamma \approx 10^{-5}$. However, the very small spectral decay of this kernel required the computation of a very large number of eigenpairs, as shown in Table 3.2. This number of eigenpairs is one of the largest reported in the iterative eigenmethods literature, and the only one we are aware of that works with a dense matrix of size of more than a million. For comparison, the EigenExa code is a Petascale dense eigensolver and was recently reported to have diagonalized the first million size dense matrix on the K supercomputer in 3464 seconds using 663,552 cores and achieving 1.7 PFLOPs [56]. Instead, by using an iterative method we can compute 1.5% of the spectrum in 2666 seconds, on 8704 cores of Stampede 2[119], attaining around 100 TFLOPs for the entire PRIMME run.

More importantly, we do not expect regression problems to require the computation of so many eigenvalues. Machine Learning users would want to use a kernel that displays a fast spectral decay for their problem, requiring just a small number of eigenvalues that can be solved far more efficiently (as in Table 3.1). What the experiment shows, however, is that our software enables this investigation even in extreme cases in an efficient and robust way.

Looking closer at the timings of Table 3.2, we see that for every eigenvalue problem 404.4 seconds are spent reading the distances to create the kernel locally. Although this is a small part of this problem, it would dominate the execution time for kernels that only require 10-100 eigenpairs. In that case, it is advisable to increase the number of nodes to allow a copy of the distances to be stored in memory so that the matrix is recomputed for every new hyperparameter. The smaller memory footprint of the eigenvectors in that case will moderate the increase in the number of nodes.

PRIMME timings show the computation of a single eigenproblem to take less than an hour. This in itself is remarkable. Orthogonalization takes about 25% of the total execution time. This is expected as its complexity grows as $O(k^2N)$ where k is the number of eigenvalues. To contrast, orthogonalization for our stock data took less than 0.5% of total time. Randomized methods that avoid orthogonalization have been proposed [95] but their robust implementation is still under investigation and the potential benefits are limited. The AllReduce time, which is needed in orthogonalization and in PRIMME inner products, took a 416.8 seconds which is reasonable considering the 19,000 eigenvectors.

The last stage is the low rank regression and evaluation for each wanted rank. Notice that because there are 1000 classes in ImageNet, regression has to solve 1000 linear systems per rank. This is performed with level-2 BLAS and we update for each rank. If all 19000 ranks were needed to be evaluated, this stage would take more than 175000 seconds. However, by monitoring the accuracy that each rank produces, we can stop the evaluations early when evaluation scores cease to increase.

3.6 Model Accuracy

To show the merit of developing software that combines a high performance, large scale kernel regression optimization with a boosting optimizer, we apply it on the equity return forecasting problem and validate its effectiveness by comparing the obtained accuracy with some widely used machine learning baselines. The goal is not to compare computing performance but to show that making kernel methods computationally feasible adds a competitive tool to the arsenal of machine learning methods.

Universe	50	1000	3000
Train	35108	700253	2004030
Test	2848	56808	163652
Kernel	4.6GB	1.78TB	14.6TB

Table 3.3: The training/testing datasets sample sizes and estimated kernel memory footprint for different universes.

3.6.1 Methodology

In the equity return problem, we are usually interested in stocks within a specific universe (e.g., SP500 or Russell3000) and denote the number of stocks in the universe as n. We assume the equity market proceeds in periods. Let $\mathbf{y}_{i,t} \in \mathbf{R}$ be the return of stock i at the t-th period, and $\mathbf{y}_t = (y_{1,t}, \ldots, y_{n,t}) \in \mathbf{R}^d$. Our goal is to forecast $\mathbf{y}_t$ based on all information available up to period $t - 1$.

In our experiments, we use seven years of equity data from the US market. We use three consecutive years of data for training and the following three months for testing. For each test year, we examine three different universes. Each universe consists of Top N stocks in trading volume (during the training period) with $n \in \{50, 1000, 3000\}$. Table 3.3 shows the sample sizes and estimated kernel sizes of different universes. We retrain the model for each test year and each universe. We use past 1-day returns and past 3-months dollar volume as features and next 1-day returns as the response (i.e. each equity market period consists of one day). All returns are the "log-transform" of all open-to-open returns. Our

Universe	50				1000				3000			
Year	2015	2016	2017	2018	2015	2016	2017	2018	2015	2016	2017	2018
Lasso	0.0168	0.0471	0.0307	0.0171	0.0161	0.0160	0.0336	0.0136	0.0186	0.0213	0.0133	0.0140
Ridge	0.0104	0.0471	0.0065	0.0301	0.0044	0.0156	0.0301	0.0006	0.0152	0.0152	0.0115	0.0111
GBRT	0.0576	0.0703	0.0476	0.0588	0.0261	0.0180	**0.0432**	0.0285	0.0194	**0.0362**	0.0162	0.0135
MLP	0.0311	0.0267	0.0185	0.0356	0.0190	0.0067	0.0179	0.0117	0.0175	0.0309	0.0154	0.0176
LSTM	0.0258	0.0371	0.0236	0.0493	0.0138	0.0037	0.0104	0.0155	0.0162	0.0147	0.0143	0.0152
KPCR	0.0563	0.0602	0.0565	0.0673	0.0344	0.0183	0.0055	0.0253	0.0169	0.0274	0.0175	0.0183
B-KPCR	**0.0929**	**0.0944**	**0.1035**	**0.1001**	**0.0356**	**0.0249**	0.0069	**0.0296**	**0.0231**	0.0310	**0.0218**	**0.0220**

Table 3.4: A summary of accuracy results for equity return forecasting problem. Results are presented in testing correlations. Bold numbers denote the highest accuracy for each year. B-KPCR denotes the KPCR with boosting method.

baselines include linear regression (”Lasso” [122] and ”Ridge” [54]), GBRT [37], MLP [105] and LSTM [53]. We apply correlation as the metric because it is more suitable than MSE in our setting.

In addition, the equity return forecasting problem usually has very small signal-to-noise. For example, a model for predicting the next 1-day return can start profiting when its r^2 score is only 2×10^{-4} (i.e., 2 basis points). This means a boosting approach that aggregates forecasts from multiple models is needed.

3.6.2 Results

Table 3.4 shows the accuracy results of three universes. Firstly, linear regressions perform worst among benchmarks as expected. Secondly, deep learning models also do not achieve the best accuracy despite being more powerful than linear models. This is because, compared to other methods, deep learning models have complex structures with many more hyperparameters to tune (e.g. layer number of the model, the type and parameters of each layer, etc), while each hyperparameter is expensive to evaluate (re-train the model and test the accuracy on the dataset). Finally, KPCR consistently outperforms all other models on most datasets, and the boosting method is able to dramatically improve the accuracy of KPCR. This highlights the importance of our efficient KPCR solver because many KPCR instances with different hyperparameters need to be solved for the boosting method.

3.7 Conclusion

There is an important set of downstream applications that depend on the solution of the Kernel Principal Component Regression for datasets of size more than a million observations. Yet, current software packages cannot scale to this size.

In this work, we designed and implemented a software solution that includes a front-end pipeline that handles the hyperparameter optimization and check-pointing, and a high performance backbone based on an efficient block matrix-vector multiplication and a state-of-the-art eigensolver. All algorithms are implemented to achieve high single-node performance and to overlap as much communication as possible. The software can run even when the matrix cannot be stored in its entirety.

As a feasibility study we were able to apply KPCR for the first time on the entire ImageNet dataset with an adverse kernel that stress-tested the eigensolver capabilities. Experiments on an even larger asset pricing dataset showed that with proper hyperparameter optimization KPCR outperforms other models.

Chapter 4

ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories

4.1 Introduction

Memory technologies are advancing fast, and new memory devices that feature high-performance, high-density, or low-power are emerging [62, 63, 64]. Recently, 3D-stacked memories, such as Hybrid Memory Cube (HMC) [23] and HBM [63], and byte-addressable non-volatile memories (NVM) have become commercially available. These new memory devices, together with the conventional DRAM technology, make heterogeneous memory system (HMS) a feasible solution for building large-scale systems under the limited area, power, and cost budget.

A typical HMS consists of a *high-performance memory* and a *large-capacity memory*, where the high-performance memory has a smaller capacity, and yields higher memory bandwidth and/or lower memory latency than the large-capacity memory. Consider two popular HMS examples. The 2nd Gen Intel® Xeon® Scalable processor supports up to 6 TB Optane byte-addressable NVM and up to 384 GB DDR4 DRAM on a single machine [101]. Another example is the Intel Knights Landing (KNL) processor with up to 192 GB DRAM and 16 GB MCDRAM [118]. While DRAM is the high-performance memory compared to the Intel Optane NVM, providing three times bandwidth of NVM, it becomes the high-capacity memory on KNL, where MCDRAM provides nearly four times bandwidth of DRAM.

On HMS, data placement plays an essential role in performance optimization. There have been extensive efforts for tackling this challenge [31, 97, 36, 26, 82]. The general optimization strategy tries to place critical data onto the high-performance memory. Some specialized optimization also tries to utilize memory bandwidth on both memories concurrently, which is only feasible on architectures providing independent memory channels to each memory. State-of-the-art optimization techniques have explored coarse-grained solutions that place whole data structures onto the high-performance memory [36, 26, 82, 103, 134, 130, 97, 31]. However, these approaches are inefficient for graph applications—a more challenging class of applications that have massive data structures with skewed access patterns.

Graph applications play significant roles in a spectrum of fields ranging from bioinformatics, scientific computing, social network, to machine learning and data mining. The current solutions face two main challenges. First, whole data structure placement might move non-critical data regions with few reuses, e.g. the data associated with low-degree vertices in graph processing, to high-performance memory, resulting in a waste of scarce resource. For instance, applications running on servers need to share all resources, resulting in even smaller high-performance memory available to an application. Second, due to the data-driven behavior of graph applications, effective data placement largely depends on the feature of input data, i.e., the sparsity and the structure of a graph, and also the query at each run.

Relying on the application programmer to explicitly manage data placement is only feasible at a coarse granularity, i.e., changing the placement of a whole data structure. Even so, it is a tedious and error-prone process, especially for large-scale applications.

Moreover, a statically managed placement may not be portable when applications are running on a different system. Thus, an optimal decision may include feedback from the application behavior on the underlying hardware into consideration. These challenges require a dynamic solution and fine-grained data placement scheme to address.

This work proposes ATMem as a data placement optimization framework to tackle these challenges in graph applications. ATMem enables adaptive granularity in managing data placement on HMS with three novel designs. First, ATMem enables adaptive data chunk profiling for subsequent *partial* data structure placement. The ultimate goal is to achieve the maximum *performance gain per byte*, i.e., improving fast memory utilization by only placing critical data regions that yield the highest performance gains on it. This is especially meaningful for server machines with multiple applications competing for precious fast memory. Second, ATMem supports lightweight sampling-based profiling and more importantly, enhances the analysis of sampling results with a novel tree-based information patching procedure to promote prospective data chunks into the critical category. Third, ATMem supports high-bandwidth data migration between memories at the application level without changes to operating systems or hardware.

The main contributions of this work are as follows:

- It proposes a lightweight profiler that uses hardware sampling to identify access patterns to data chunks of adaptive granularity.
- It employs a local relative ranking strategy to select critical regions inside data structures based on sampling results.
- It employs a novel *m-ary tree-based* strategy to promote prospective data chunks into critical based on a global relative adaption.
- It designs a multi-stage multi-threaded migration strategy at the application level to enable high-bandwidth data migration and reduce TLB misses.
- We provide the implementation in a framework called ATMem and evaluate in five applications on two HMS hardware, including the state-of-the-art byte-addressable

NVM.

ATMem is evaluated in five graph applications on real NVM-DRAM and MCDRAM-DRAM hardware. Our evaluation results show that by selecting 5%-18% data onto high-performance memory, ATMem achieves an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM, compared to the baseline that places all data on the large-capacity memory. On NVM-DRAM, ATMem achieves performance comparable to a full-DRAM system with as low as 9%-54%. ATMem also enables 2.07×-5.32× faster data migration than the system service.

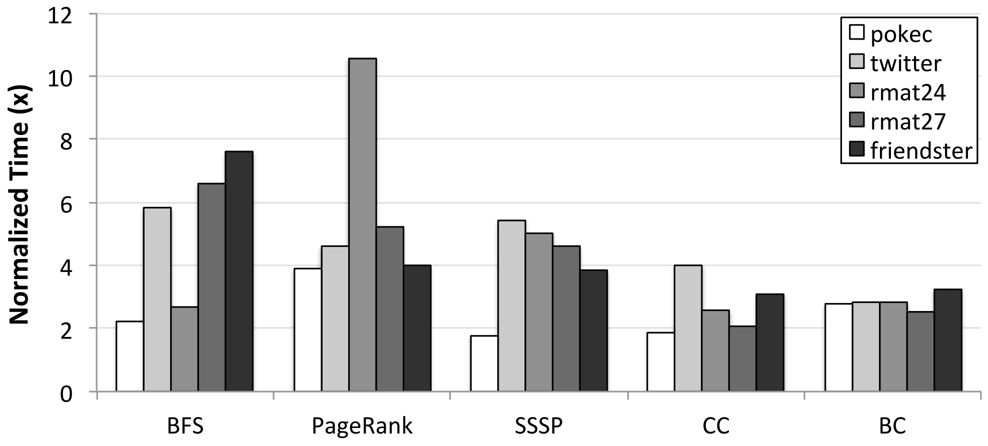
4.2 Background and Motivation

This section introduces the architecture of systems under consideration, the target workload, and the limitations of existing solutions. It also presents a preliminary study on real hardware (including the state-of-art Intel Optane byte-addressable NVM).

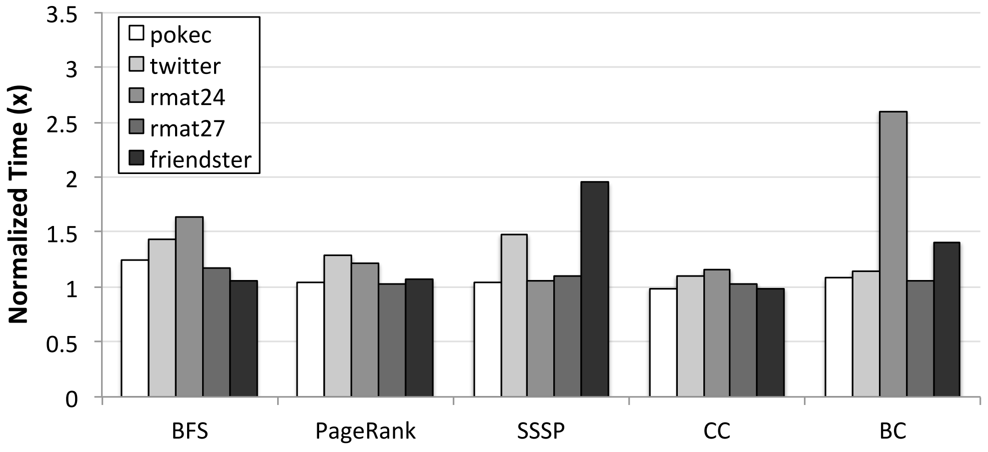
4.2.1 Heterogeneous Memory Systems

This work considers heterogeneous memory systems that place a small-capacity *high-performance memory* side-by-side to a low-performance *large-capacity memory*. The latest 2nd Gen Intel® Xeon® Scalable processor can work with up to 6 TB Optane byte-addressable NVM of 39 GB/s bandwidth and 384 GB DRAM of 104 GB/s bandwidth [101]. Another example is the Intel Knights Landing (KNL) processor that has 96 GB DRAM next to 16 GB MCDRAM of 400 GB/s bandwidth [118]. We conduct preliminary studies on real hardware and find that application performance on different memories can have a much larger gap than that predicted by emulators [34, 134, 102].

Figure 4.1a and 4.1b report the slowdown when data are placed on the large-capacity memory compared to that on high-performance memory. The Intel Optane NVM has three times latency and 38% bandwidth of DRAM [101]. However, application performance could slow down by up to 10× (Figure 4.1a). MCDRAM has a limited capacity and system



(a) Normalized execution time with data placement on the Intel Optane byte-addressable NVM as to that with data on DRAM.



(b) Normalized execution time with data placement on DRAM as to that with data preferably placed on MCDRAM.

Figure 4.1: Compare the performance of five applications on two HMS using different data placement for five datasets.

solutions like 'numactl -p' might choose less critical data onto high-performance memory, reducing performance improvement. These results highlight prospective benefits from fine-grained data placement that selects critical regions into high-performance memory.

Objective I: *our work identifies critical data chunks inside a data object and only migrates them onto high-performance memory for efficient memory utilization.*

4.2.2 Graph Applications

Graph applications, such as data analytic workload, often exhibit data-driven access pattern and have low data locality. Contemporary computer systems use highly optimized caches to keep frequently used data close to processing units. Such optimization, however, has shown its inadequacy for graph applications [98]. With low data reuse, the overhead of managing hardware cache might even hurt application performance. The opportunity in optimizing data placement stems from dense (hot) regions and sparse (cold) regions in data that drive accesses. Our work identifies these regions dynamically and manages them in the high-performance memory explicitly at the application level.

The application-level approach requires to address several challenges, specifically compared to system-level or architecture solutions. First, it lacks the full statistics of page accesses as the operating system have. Second, it cannot flexibly modify the hardware units on an existing platform. Thus, ATMem utilizes the widely available hardware counters to develop a sampling profiler for estimating dense regions in data. Unlike a system solution that usually operates at page-size granularity, ATMem adapts the size of data chunks, i.e., the basic unit of a data structure, to reduce metadata and migration overhead. Moreover, a sampling-based approach has to trade off overhead and accuracy. Even a high-frequency sampling approach cannot guarantee to capture all the information. ATMem proposes a *tree-based clustering* strategy to "patch up" information that is likely missed due to sampling.

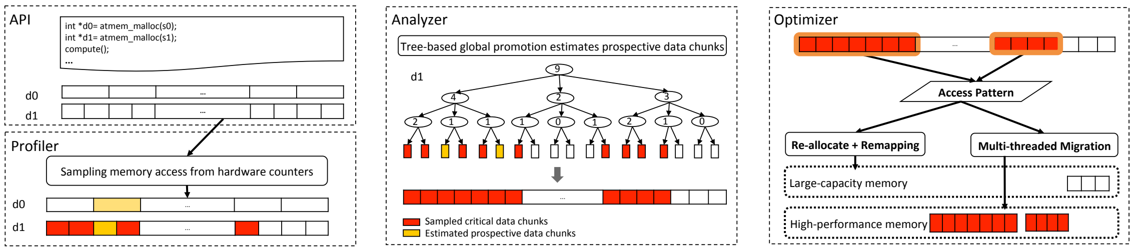
Objective II: *our work uses low-overhead sampling profiling and addresses possible information loss in samples to estimate critical data regions.*

4.2.3 Migration Mechanism

On state-of-the-art heterogeneous memory systems, different memories, e.g., NVM and DRAM, are exposed to CPU as separate non-uniform memory access (NUMA) nodes [101]. In this way, traditional system services for NUMA control, i.e., *mbind*, can be used for migrating data from one memory to another. As pointed out by previous works [87, 138], the current system service is inefficient for heterogeneous memory systems. The migration mechanism is often single-threaded, which cannot exploit the high bandwidth supported by the hardware. Also, the data movement procedure is long and blocking, with substantial overhead spent for enforcing correctness for system-wide reliability. An application-level approach has the opportunity to bypass some of these overhead given sufficient application knowledge. Another side-effect from the standard service is the increased TLB misses after migration. While previous works have proposed solutions in the operating systems or hardware, we develop a multi-stage multi-threaded migration strategy to tackle these challenges at the application level on an existing platform.

Objective III: *our work improves data migration between memories at application level without changes in hardware or operating systems.*

4.3 Overview of ATMem



```
int *d0= atmem_malloc(s0);
int *d1= atmem_malloc(s1);
compute();
...
```

API
d0
d1
Profiler
Sampling memory access from hardware counters
d0
d1
Analyzer
Tree-based global promotion estimates prospective data chunks
d1
Sampled critical data chunks
Estimated prospective data chunks
Optimizer
Access Pattern
Re-allocate + Remapping
Multi-threaded Migration
Large-capacity memory
High-performance memory

Figure 4.2: The overview of ATMem framework.

ATMem consists of three main components (as illustrated in Figure 4.2): a *profiler*, an *analyzer*, and an *optimizer*. **First**, ATMem profiler employs low-overhead hardware-counter based sampling to learn access patterns in an application. **Second**, based on the collected samples, ATMem analyzer identifies critical data chunks (of adaptive granularity) by a *global relative ranking* scheme. ATMem analyzer addresses one common challenge in sampling-based approaches, i.e., the loss in sampled information. The analyzer utilizes an *m-ary tree-based* strategy to "patch up" information to estimate potentially critical data chunks. **Finally**, ATMem optimizer performs a multi-stage multi-threaded migration to move both sampled and estimated critical data chunks onto high-performance memory. Section 4.4 introduces the design of these components in details.

4.4 Design of ATMem

This section describes the design of ATMem in identifying critical data regions of adaptive granularity, predicting prospective data regions, and migrating data regions at high bandwidth between memories.

4.4.1 Adaptive Data Chunks

The basic unit of data management in ATMem is called *data chunk*. Data chunk is an adaptive unit such that a data object (DO_i) is composed of N equal-sized data chunks ($DC_{ij}, j = 1..N$), while data chunks in different data objects can have different sizes. Figure 4.2 (Profiler panel) illustrates two chunk sizes in $d0$ and $d1$, respectively. ATMem adjusts the granularity of a data chunk DC_{ij} based on the size of the data object DO_i. This adaptive data chunk design has two main advantages. First, it breaks down a (potentially) large data object into finer-grained segments. By comparing the priority of data chunks inside a data object, ATMem can separate critical data chunks from non-critical data chunks. These critical data chunks correspond to dense (hot) regions of a data structure that has non-uniform access patterns, which is common in irregular applications. Second, ATMem can control the profiling overhead by managing the number of data chunks, i.e., coarsening the granularity of data chunks. Each collected sample in the profiling stage will be associated with a data chunk. Thus, changing the granularity of data chunks affects the metadata and profiling overhead directly.

4.4.2 Hybrid Local Selection

The first stage of ATMem analyzer employs a local relative ranking to select critical data chunks for each data object. These selected data chunks are called *sampled selection* to be distinguished from the estimated selection in a later stage. Equation 4.1 quantifies the metric of local priority (PR_{local}) for a data chunk DC_{ij} that represents the j-th data chunk in data object DO_i. ATMem uses the number of missed reads from the last-level cache LLC_{mr} as an indicator of priority and normalizes it to the size of a data chunk ($Size$). Note that the normalization is necessary for global relative ranking among different data objects in a later stage.

$$PR_{local}(DC_{ij}) = \frac{LLC_{mr}(DC_{ij})}{Size(DC_{ij})} \tag{4.1}$$

$$\theta(DO_i) = \max(P_n, \alpha \max PR, \min PR/Freq_{sample}) \tag{4.2}$$

$$CAT(DC_{ij}) = \begin{cases} 1, & \text{if } PR_{local}(DC_{ij}) \geq \theta \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

ATMem combines the conventional top-N selection with a derivative-based classification to select the threshold value of θ. In Equation 4.3, data chunks with priority score (PR) higher than the threshold θ has categorization (CAT) 1, i.e., critical. A top-N selection chooses a fixed ratio of data chunks that have the highest PR_{local} score in a data object, i.e., the n-th percentile P_n in Equation 4.2. However, a fixed selection is inefficient in two scenarios. First, a highly skewed access distribution has a high concentration in a small number of data chunks. In this case, the top $\frac{N}{2}\%$ data chunks have significantly higher priority than the next $\frac{N}{2}\%$ data chunks. Thus, selecting the second $\frac{N}{2}\%$ data chunks may not bring much improvement. On the contrary, in a relatively even distribution, the top $N\%$ data chunks may not have quantitatively significant difference compared to data chunks after them, i.e., more than $N\%$ data chunks should be selected. ATMem employs a derivative-based search, similar to a k-means clustering technique, to adjust the threshold value by quantifying the changes relative to the highest priority score ($\max PR$).

Additionally, ATMem includes a theoretical minimum priority for a given data chunk size and adjusts for the sampling rate denoted as $Freq_{sampling}$.

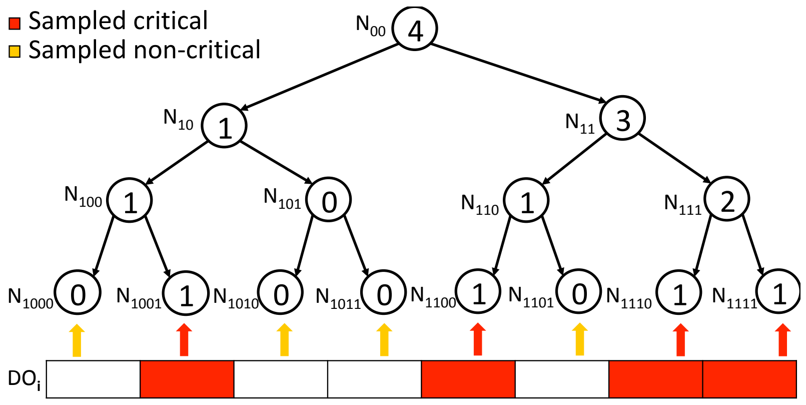
4.4.3 Tree-based Global Promotion

The second stage in ATMem analyzer considers the global view of all data objects and adapts the selection to reflect the relative importance of these data objects. ATMem constructs an m-ary tree for each data object to perform *estimated selection*. This procedure promotes data chunks that are not in sampled selection into *prospective critical*. ATMem adjusts the selection with two parameters, i.e., m and the threshold value of *tree ratio* (TR). This tree-based promotion helps "patch up" information that is likely missed due to sampling-based profiling. Similar approaches have shown effectiveness for prefetching data from CPU memory into GPU memory [39]. Additionally, the promotion can merge multiple discrete segments into a continuous one, which improves the efficiency of migrating data between memories. Figure 4.2 (the middle panel) illustrates an example of a ternary tree derived for data object $d1$. The leaf nodes in red represent critical data chunks from the sampled selection. In the remainder of this section, we introduce three main steps in the tree-based global promotion.

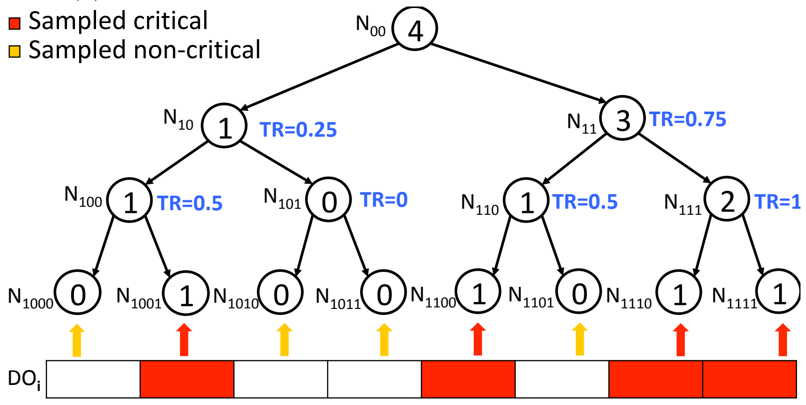
4.4.3.1 Tree Construction

ATMem analyzer uses the classification results in the first stage to construct an m-ary tree for each data object. A data chunk has value of either 1 (critical) or 0 (non-critical) from Equation 4.3. These data chunks correspond to the leaf nodes of the tree. Each leaf node has the value from its corresponding data chunk. Figure 4.3a illustrates an example tree constructed from the eight data chunks in DO_i. Each critical data chunk (in red) becomes a leaf node of value 1. From bottom-up, ATMem creates internal nodes of the tree. Each internal node carries value as the sum of its children nodes.

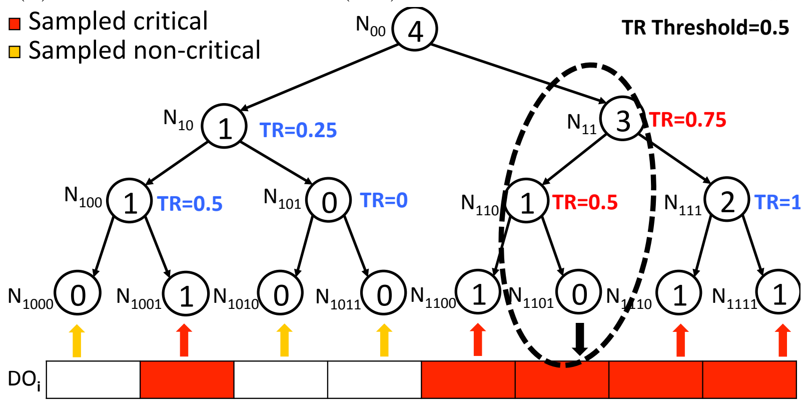
Tree ratio (TR) of an internal node is defined as the ratio between its value and the number of its descendant leaf nodes. In Figure 4.3b, node N_{11} has four leaf nodes and its



(a) Construct a m-ary tree from the data chunks in DO_i.



(b) Calculate the tree ratio (TR) of each internal node bottom-up.



(c) Promote prospective data chunks from top-down.

Figure 4.3: Three stages in a tree-based global promotion: construction, bottom-up calculation of tree ratio (TR), and top-down promotion.

TR is calculated as 3/4. Tree ratio is a metric that quantifies the likelihood of critical data chunks in a *range* of memory space. Here, the range of memory space depends on the level of a node. For instance, the root node N_0 covers the entire address space of DO_i, while the node N_{100} only covers the first quarter address space. ATMem adjusts m to control the range of memory space represented by an internal node as well as the sensitivity to a threshold value of tree ratio. For instance, a quad-tree can have more threshold values of tree ratio than a binary tree.

4.4.3.2 Global Adaptive TR Threshold

ATMem uses the tree ratio as an indicator for whether the sampled non-critical data chunks (yellow arrows in Figure 4.3) could still be important but not captured in the sampling. Also, a small gap in a large continuous address space can be "patched up" to improve data migration because launching multiple migrations would have higher overhead than a single migration. A naive design would use a fixed threshold value for tree ratio ($\theta(TR_i)$) such that if an internal node has TR value higher than the threshold value, all its non-critical children will be promoted to critical.

$$W(DO_i) = \frac{\sum_{j=1}^{N} PR_{local}(DC_{ij}) \cdot CAT(DC_{ij})}{\sum_{j=1}^{N} CAT(DC_{ij})} \tag{4.4}$$

$$\theta(TR_i)' = \epsilon + \frac{\theta(TR_i) \cdot (\max W - W(DO_i))}{\| \min W - \max W \|} \tag{4.5}$$

ATMem adjusts the threshold value for each data object based on its global relative ranking. The adapted threshold value ($\theta(TR_i)'$ in Equation 4.5) mitigates influence from different sampling frequencies, applications, data sets, and platforms. ATMem calculates the averaged priority of a data object in Equation 4.4 denoted as its weight (W). Weight quantifies the significance of selected data chunks, where a data structure of fewer critical data chunks with high priority has a higher weight than a data structure of more critical data chunks with low priority. From Equation 4.5, a large weight value would decrease the tree ratio threshold, causing the top-down promotion procedure (to be introduced next) to promote more non-critical data chunks. ATMem calculates the weight space as the gap between the minimum and maximum weight of all data structures. It also includes ϵ as a theoretical minimum threshold value that depends on the value of m. For instance, an octree would have $\epsilon = 0.125$ as a meaningful lower bound.

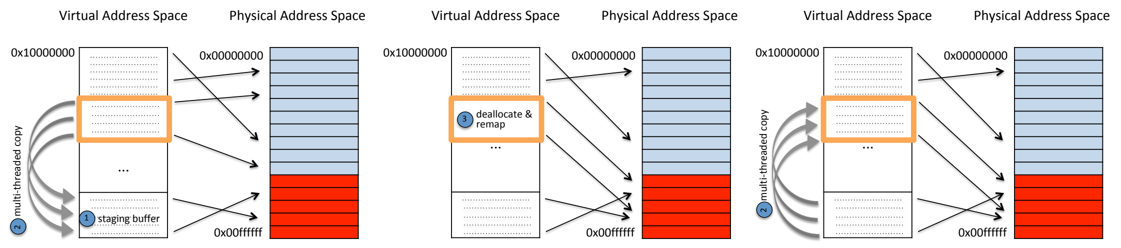
4.4.3.3 Top-down Promotion

ATMem uses the threshold value of tree ratio from Equation 4.5 to promote prospective data chunks into *estimated selection*. Each data object may be assigned a different threshold value after the global adaptive ranking. ATMem starts traversing from the root node and performs a breadth-first search to find an internal node with tree ratio higher than the threshold value. Starting from that node, ATMem tries to "patch up" data chunks on its descendant leaf nodes. For example, in Figure 4.3c, the data object has a threshold value of 0.5, and the node N_{11} has a tree ratio of 0.75. Next, ATMem identifies those N_{11}'s children nodes whose tree ratio are lower than the threshold value, i.e., the node N_{110}. ATMem promotes the right children node with zero tree ratio to be estimated critical. This top-down promotion procedure results in a single continuous region in the data object DO_i to be placed on high-performance memory.

4.4.4 Data Placement Optimization

ATMem uses the decision from the analyzer to optimize data placement at the application level. In particular, ATMem changes the physical memory of the selected data chunks to high-performance memory without changing the virtual memory address of the data object. This partial migration of a data object minimizes the modifications to the application source code.

Figure 4.4 illustrates three main steps of the multi-stage multi-threaded data migration approach. In this example, a data object has a virtual memory address from `0x10000000` to the low end of the yellow box. The system has two types of physical memories, as indicated in blue and red in the physical address space. Each segment in the address



(a) Staging (b) Remapping (c) Moving

Figure 4.4: Fast migration at application level using staging buffers and multi-threaded data copy.

space represents a physical page. ATMem analyzer has determined that the data chunks in the yellow box, denoted as a *source region*, should be placed on high-performance memory.

In the first step, ATMem uses multiple threads to copy the value in the source region to a staging buffer concurrently. The staging buffer is physically located on the target physical memory, as indicated by the mapping (black arrows Figure 4.4a) to red pages in physical space. After that, ATMem remaps the virtual address of the source region to point to (empty) physical pages on the target memory (Figure 4.4b). Note that no data movement occurs in this stage and the virtual address space of the data object remains intact. Finally, in Figure 4.4c, ATMem uses multiple threads to copy the stored value from the staging buffer to the yellow region. The whole procedure has data moved twice, i.e., one between two memories and one within the same memory. ATMem adjusts the concurrency for data copy to exploit memory bandwidth supported by the hardware.

4.5 Implementation and Optimization

We implement ATMem as a runtime library for general heterogeneous memory systems. ATMem uses precise address sampling for profiling and provides a set of API for registering data structures and initiating data migration.

4.5.1 Hardware Supported Sampling

ATMem profiler uses hardware counters for low-overhead profiling. In particular, ATMem relies on the precise address sampling capability supported by hardware to collect the memory addresses of data accesses and correlates them to data chunks [121]. Currently, ATMem is implemented on performance monitoring units (PMU) with processor event-based sampling (PEBS) on Intel processors [57]. ATMem can be easily extended to other processors with similar features, e.g., AMD processor [33].

ATMem automatically adjusts the sampling frequency of PMUs at runtime. Before enabling PMUs, ATMem combines the size and number of all data chunks and the number of application threads to adjust an empirical sampling rate on a given platform. This adaption avoids unnecessarily high sampling frequency while also ensures efficient information collection.

4.5.2 API

ATMem provides a minimal set of API (Listing 4.1) for easy adoption in existing applications. The main purpose of the interface is to inform ATMem runtime about the data objects so that ATMem can link the collected memory address to data objects. Upon registration using `atmem_malloc()`, ATMem runtime internally determines the granularity of the data chunks for that data structure. Code transformation that converts memory allocation routines from `malloc`-like functions to `atmem_alloc()` is also feasible. However, programmers could use application knowledge to improve this decision.

Listing 4.1: API for registering, profiling and optimization.

```
void *atmem_malloc (size_t);
void atmem_free (void *);
void atmem_profiling_start();
void atmem_profiling_stop();
void atmem_optimize();
```

ATMem profiler monitors PMU on cores and aggregated data for analysis for multi-

Table 4.1: Experiment Platform Specifications.

NVM-DRAM Testbed	
Model	Intel® Xeon® Platinum 8260L
Processor	2nd Gen Intel® Xeon® Scalable processor
Core	2.4 GHz, 3.9 GHz Turbo frequency
Cache	32 KB d-cache and 32 KB i-cache, 1 MB private L2, 35.75 MB shared L3
Memory	96 GB DDR4 DRAM and 768 GB Optane DC NVDIMM per socket
MCDRAM-DRAM Testbed	
Model	Intel® Xeon® Phi 7200
Processor	2nd Gen Intel® Xeon Phi™ processor
Core	1.1 GHz, 1.7 GHz Turbo frequency
Cache	32 KB d-cache and 32 KB i-cache, 512 KB private L2
Memory	16 GB MCDRAM and 96 GB DDR4 DRAM

threaded applications. One possible optimization is to monitor only a subset of PMUs, which is beyond the scope of this work. Currently, ATMem requires programmers to indicate when to start migrating data, i.e., `atmem_optimize()`. Future works on compiler optimization could automatically insert this function based on static analysis.

4.6 Experimental Setup

Our evaluation is performed on two real hardware testbeds. The first testbed is the 2nd Gen Intel® Xeon® Scalable processor platform that features the Intel Optane byte-addressable NVM and DDR4 DRAM. Table 4.1 summarizes the configurations. The Optane NVM is configured in App Direct mode and exposed to CPUs as a NUMA node. DRAM and NVM on the same socket share six memory channels that operate 2400 GT/s, i.e., a theoretical peak bandwidth of 115 GB/s. Our experiment uses 48 hardware threads and memories on one socket to eliminate the reported NUMA issues [101]. The second testbed is the Inter Knights Landing Xeon Phi processor (KNL) [118] that features 256 hardware threads, 16 GB high-bandwidth 3D MCDRAM and 96 GB DDR4 DRAM. MC-DRAM is configured in flat mode.

We use SIMD implementation of five irregular applications [104], i.e., breadth-first

Table 4.2: Characteristics of graph inputs.

Graphs	Number of Vertices	Number of Edges
pokec	1.6 M	30.6 M
rmat24	16.8 M	268.4 M
twitter	41.7 M	1.5 B
rmat27	134.2 M	2.1 B
friendster	68.3 M	2.1 B

search (`BFS`), single-source shortest path (`SSSP`), PageRank (`PR`), betweenness centrality (`BC`), and connected components (`CC`) for evaluation. `BFS`, `SSSP`, `PR`, `BC`, and `CC` use a mixture of five graphs, including `pokec`, `rMat24`, `twitter` (`twt`), `rMat27`, and `Friendster` (`friend`). Table 4.2 shows these graphs' vertices count and edges count, e.g., `Friendster` network dataset [1] contains 68.3M vertices and 2.1B edges.

Applications were compiled by the Intel C++ Compiler (`icc` 19.0.2.187) with `-O3` option and AVX512 flag. For each test, ATMem turns on hardware profiling in the first iteration and migrates data before the second iteration starts. The evaluation uses the benchmark run time from the second iteration as the optimized execution time. On both platforms, data to memory binding is controlled by 'numactl' in libnuma [70]. The experiments are repeated ten times and the average time is reported.

4.7 Evaluation

In this section, we first present the overall performance of ATMem in five graph applications using five data sets on two testbeds. Next, we perform a sensitivity test to evaluate the effectiveness of tree ratio in selecting data chunks. Finally, we compare the multi-stage multi-threaded migration strategy with the standard system service.

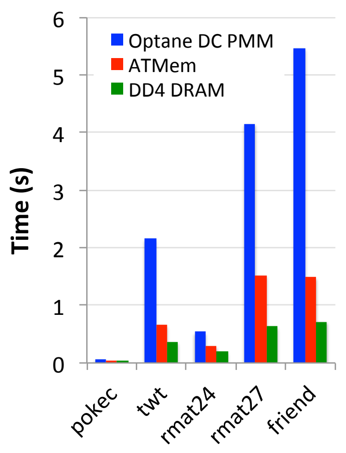
4.7.1 Overall Performance

This section evaluates the performance improvement from ATMem adaptive data placement. In particular, we compare the performance of applications on each testbed with the two reference performance. On the NVM-DRAM system, the baseline places data on the Intel Optane NVM (blue bars in Figure 4.5) while the ideal reference places all data in the DDR4 DRAM (green bars in Figure 4.5). On the MCDRAM-DRAM system, the baseline places all data in the DDR4 DRAM. We cannot have an ideal reference where all data is placed in MCDRAM due to its limited capacity and the large data sets in use. Thus, we use the MCDRAM preferred NUMA policy provided by libnuma, i.e., 'numactl -p MCDRAM' (denoted as MCDRAM-p) as the ideal reference.

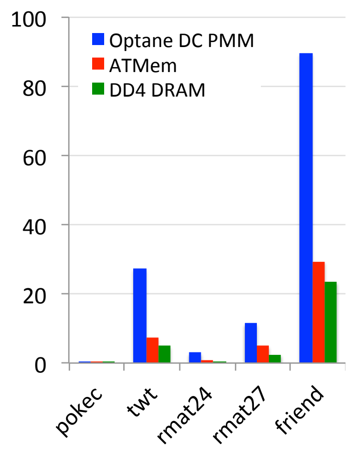
Table 4.3: ATMem performance on NVM-DRAM testbed compared to an all-DRAM ideal case

Slowdown	BFS	SSSP	PR	BC	CC
Min.	25%	26%	24%	9%	54%
Max.	2.4×	2.0×	3.0×	1.8×	2.0×

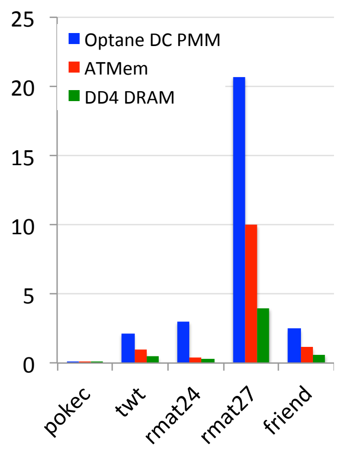
On the NVM-DRAM testbed, ATMem can effectively bridge the performance gap between NVM and DRAM with a low requirement on DRAM capacity. Figure 4.5 presents the execution time of applications on this NVM-DRAM testbed. This result shows that data placement by ATMem significantly reduces the execution time compared to the all-NVM baseline, reaching 1.25×-8.4× improvements—this is calculated from the first bar (blue) and second bar (red) in Figure 4.5. In Table 4.3, we compare ATMem with the ideal case where all data is placed in DRAM. ATMem can either achieve comparable performance or reduce the performance gap in Figure 4.1a significantly. For instance, `SSSP` with `Friendster` dataset using ATMem data placement is only 26% slower than placing all data in DRAM. Note that ATMem solution only places 12% data in DRAM (as shown in Figure 4.7). Figure 4.7 also reports the ratio of data that is placed by ATMem on high-performance memory for other applications. This ratio is calculated by the data



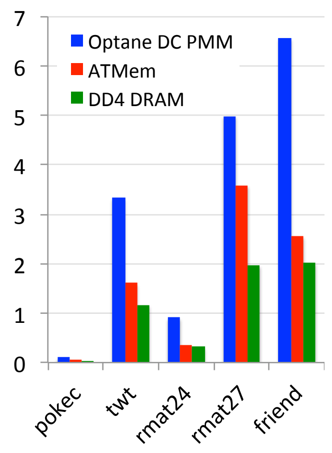
(a) BFS



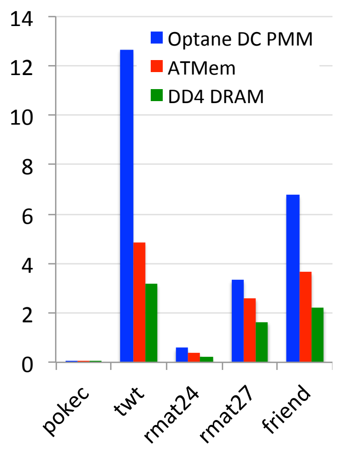
(b) SSSP



(c) PR



(d) BC

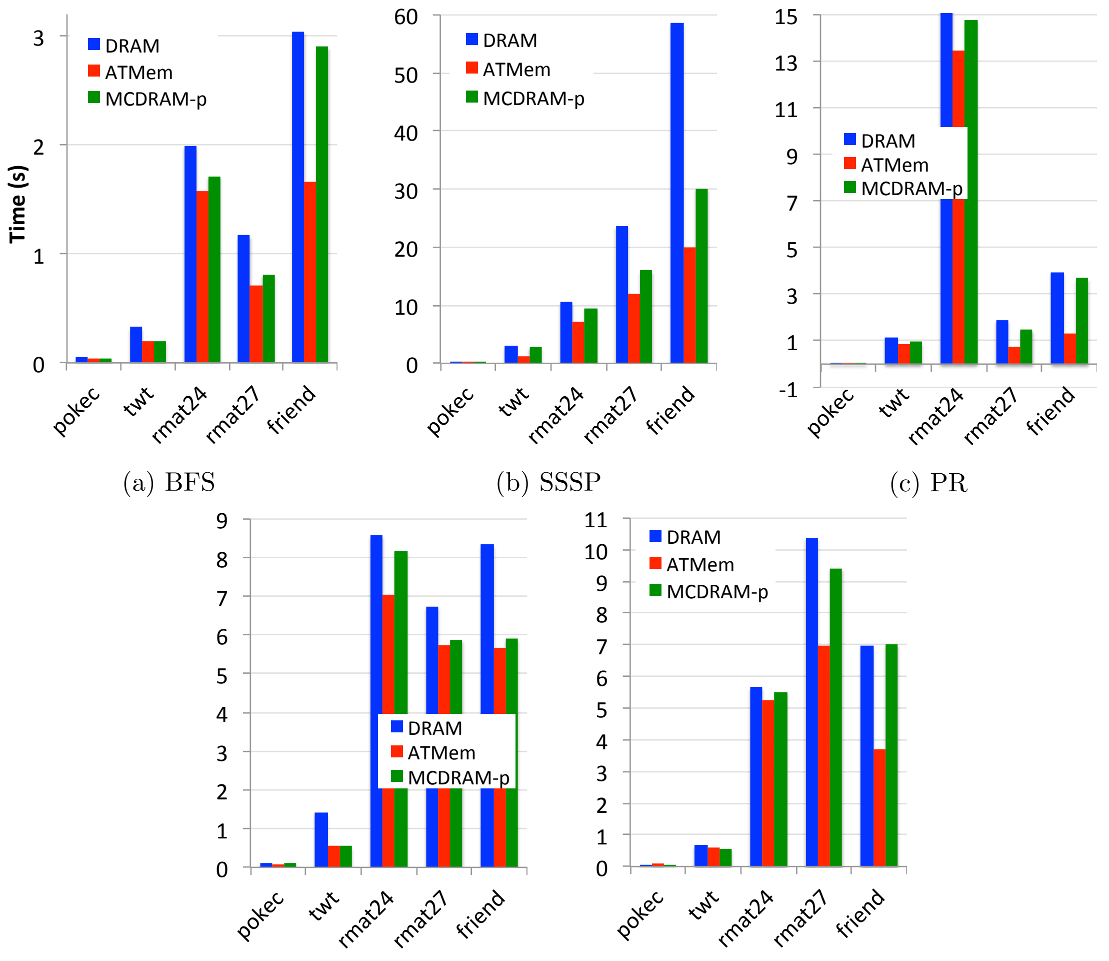


(e) CC

Figure 4.5: Execution time on NVM-DRAM testbed: NVM-only, NVM-DRAM with ATMem, and DRAM-only.

placed on the high-performance memory (DRAM in this case) over the total data size.

On the MCDRAM-DRAM testbed, a similar trend is preserved between the ATMem data placement and the baseline all-DRAM case, i.e., ATMem data placement can significantly outperform the baseline, achieving 1.1×-3× execution time speedup with only placing a small portion of data (3.8%-18.2%) on high-performance MCDRAM. An interesting result comes from the superior performance of ATMem compared to the ideal reference. In Figure 4.6, ATMem placement significantly reduces the execution time for large datasets like `Friendster` and `rMat27`, compared to MCDRAM-p option. For `Friendster`, ATMem



(a) BFS (b) SSSP (c) PR

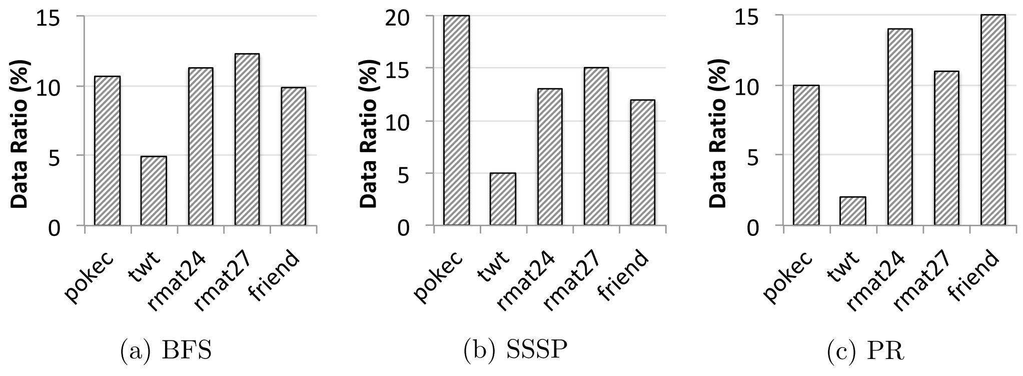
(d) BC (e) CC

Figure 4.6: Execution time on MCDRAM-DRAM testbed: DRAM-only, DRAM-MCDRAM with ATMem, and MCDRAM-p.

reaches up to 2.79× improvement in `BFS` with only 15% data in MCDRAM. Figure 4.8 reports the ratio of data that is placed by ATMem on high-performance MCDRAM for all applications.

4.7.2 Impact of Data Ratio

ATMem uses the tree ratio threshold value to tradeoff the data size on the high-performance (but small) memory with performance improvement. An optimal tradeoff would reach a data ratio beyond which performance improvement is not proportional to



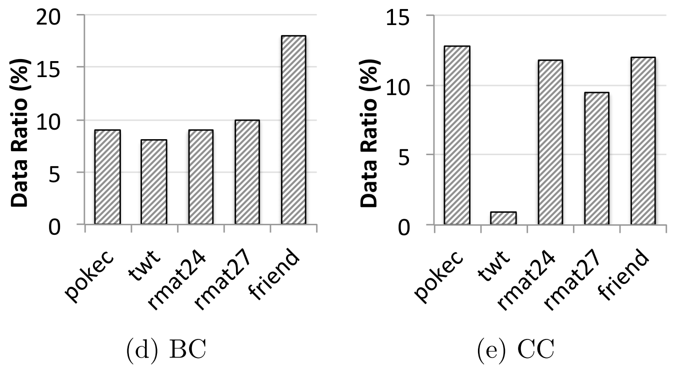


Figure 4.7: Data ratio on NVM-DRAM testbed: Data ratio is calculated by DRAM data size over total size.

the increased data size on high-performance memory. We evaluate the effectiveness of the ATMem tree ratio by manually sweeping ϵ values in Equation 4.5. Consequently, ATMem would place different data ratios on high-performance memory. Figure 4.9 and 4.10 report the performance sensitivity to data ratio on two testbeds using the `BFS` benchmark.

ATMem consistently reaches the optimal tradeoff between performance and data ratio in all tested data sets. In Figure 4.9 and 4.10, there exist optimal regions in each dataset, where most data points are gathered. On the left of this region, increasing data size could still bring significant performance improvement. Beyond this region, however, the performance only shows minimal changes, even when the data ratio is substantially increased. For instance, `Twitter` dataset on NVM-DRAM testbed (Figure 4.9d) stabilizes at approximately 15% data ratio. Further migrating data to DRAM only gains negligibly. Overall, the results indicate that ATMem can effectively detect the dense regions in graph

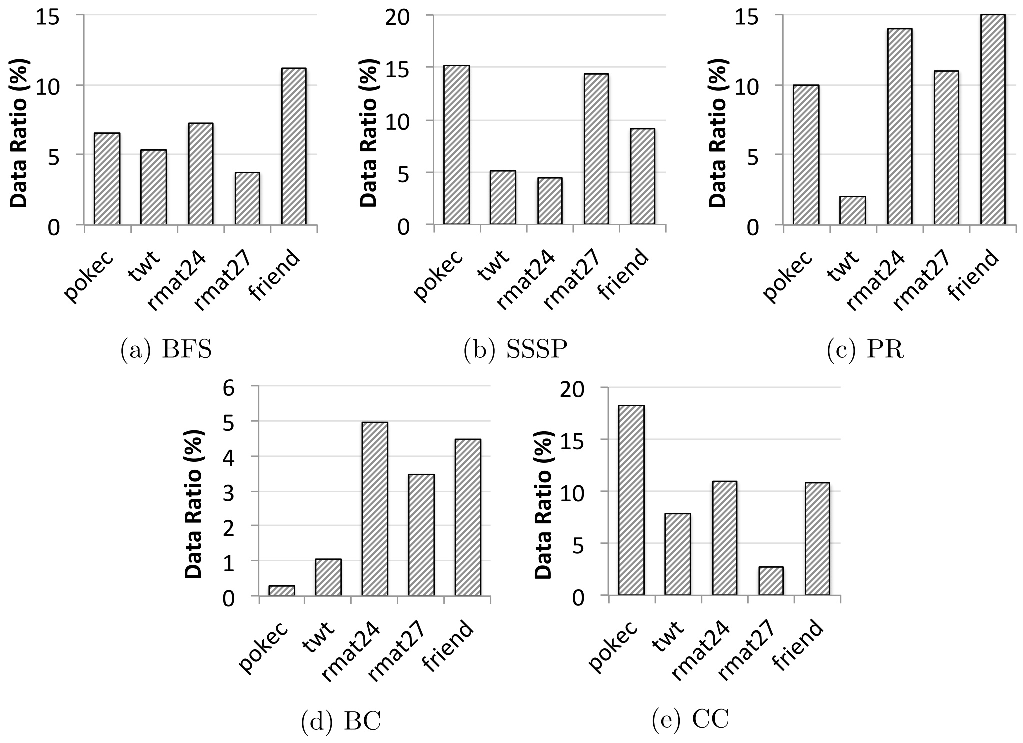


Figure 4.8: Data ratio on MCDRAM-DRAM testbed: Data ratio is calculated by MC-DRAM data size over total size.

applications to achieve near ideal performance using small memory capacity.

MCDRAM-DRAM testbed has limited capacity on the high-bandwidth memory, unable to accommodate all data in `rMat27`, `Twitter`, and `Friendster` in `BFS`. Thus, the maximum data ratio in Figure 4.10 is less than one. We notice that placing data size near the memory capacity, i.e., 16 GB, could sometimes lower performance. Instead, the AT-Mem identifies optimal regions much smaller than the capacity, avoiding this peformance degradation. The experimental results also indicate that the efficient detection and placement of dense regions in graph applications is essential for performance optimization on heterogeneous memory systems.

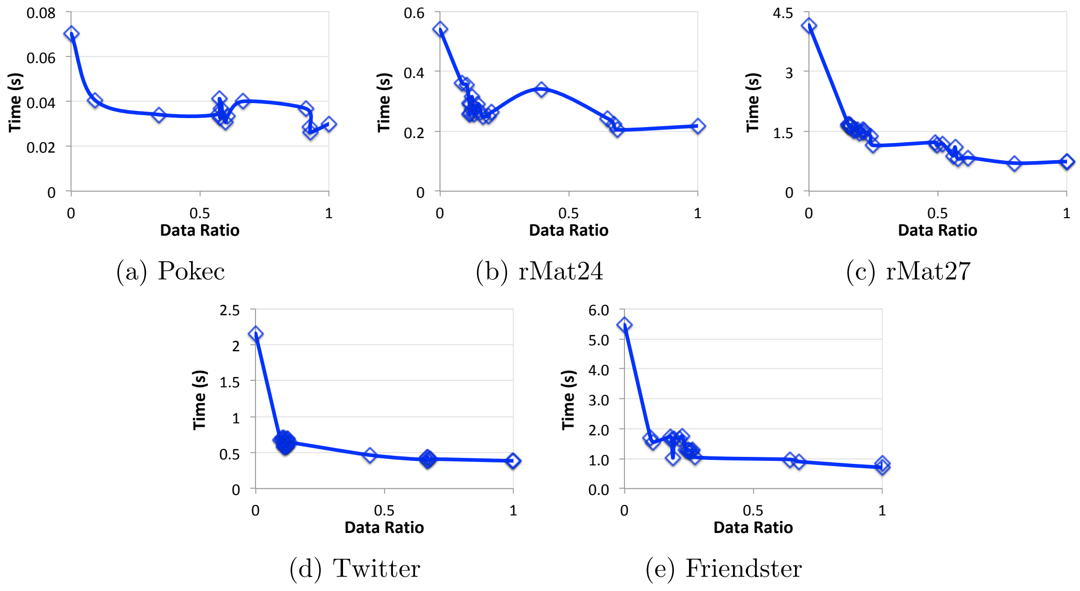


Figure 4.9: Data ratio impact on performance on NVM-DRAM testbed for `BFS`: x-axis is the data ratio placed in DRAM.

4.7.3 Data Migration

We evaluate the effectiveness of the multi-stage multi-threaded migration in ATMem by comparing it to the system service `mbind`. In this experiment, each benchmark has two versions of implementation using the two mechanisms, respectively. Table 4.4 reports the number of TLB misses after migration and the time spent in migration using `mbind` implementation, as normalized to that using our ATMem approach.

The results demonstrate that our ATMem approach dramatically reduces the number of TLB misses after data migration on both testbeds. The improvement in TLB misses on the NVM-DRAM testbed is considerably higher than that on MCDRAM-DRAM testbed. However, the data migration time on MCDRAM-DRAM shows higher speedup than that on the NVM-DRAM testbed. On NVM-DRAM testbed, ATMem reduces the data migration by 1.3×-2.7× (average 2.07×), as compared to `mbind`. On MCDRAM-DRAM testbed, ATMem manages to achieve 3.0×-8.2× (average 5.32×) improvement. The different bandwidth of source memories on the two testbeds likely causes this difference. Migration from NVM to DRAM is bottlenecked at the read bandwidth of the Intel Op-

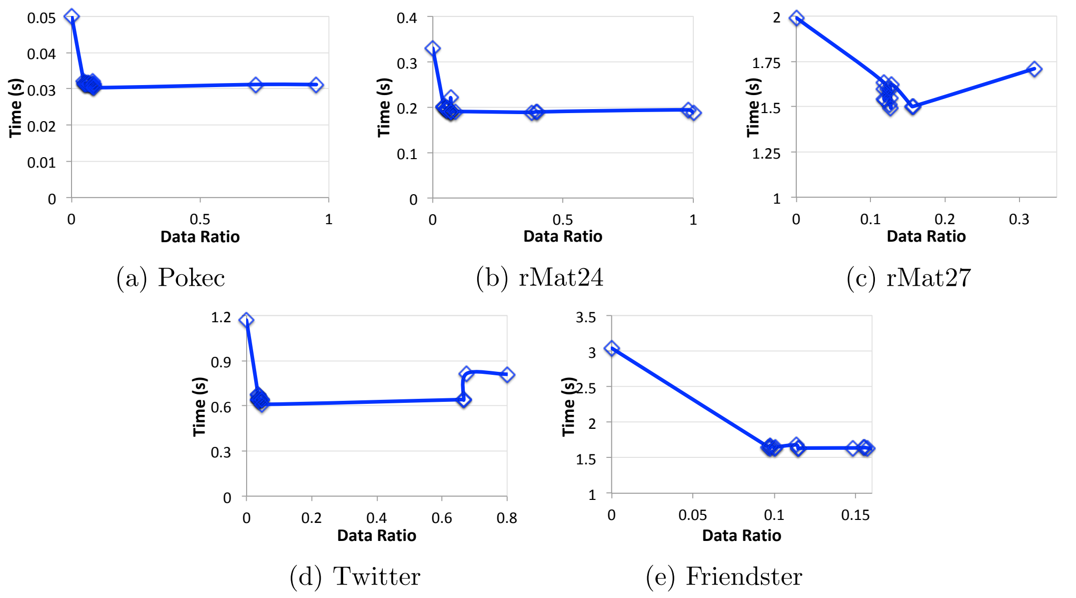


Figure 4.10: Data ratio impact on performance on MCDRAM-DRAM testbed for `BFS`: x-axis is the data ratio placed in MCDRAM.

tane NVM, while data migration from DRAM to MCDRAM can exploit the bandwidth of DRAM. The Intel Optane NVM read bandwidth is reported to be 39 GB/s [101] while DRAM on KNL platform can reach 90 GB/s bandwidth [118].

Table 4.4: Reduction in TLB Misses and migration time by the multi-stages multi-threaded approach compared to `mbind` in `PR`.

Dataset	**NVM-DRAM**		**MCDRAM-DRAM**	
	TLB misses	Time	TLB misses	Time
pokec	2.09×	1.32×	2.00×	8.26×
rmat24	73.62×	2.71×	2.53×	4.42×
rmat27	15.77×	2.66×	1.17×	5.71×
twitter	1.16×	1.94×	1.64×	3.08×
friendster	12.26×	1.72×	1.25×	5.16×
Avg.	20.98×	2.07×	1.72×	5.32×

4.7.4 Overhead Analysis

The overhead of ATMem comes from two sources: profiling and data movement. With the help of the hardware PMU, ATMem incurs minor overhead during profiling, i.e., less than 10% of the first iteration. The data movement overhead depends on the amount of data selected for migration.

The number of iterations required to amortize ATMem's overhead is decided by the kernel and the input data. In our experiments, most benchmarks can get enough benefits to compensate the overhead caused by ATMem within a few iterations. For example, data movement operation incurs 37% overhead for *SSSP* with *Friendster* dataset on HBM for the first (single) iteration. Since ATMem brings over 50% speedup for a single iteration in *SSSP*, the overhead is amortized after only one more iteration.

4.8 Related Work

Heterogeneous memory systems have been extensively studied recently. Before real hardware became available, many prior efforts used emulators and simulators [109, 79, 34, 134, 127, 88, 85]. Constraint by the gap between emulation and real commodity hardware, some previous findings may need to be revisited. For this consideration, ATMem is evaluated on two real hardware. The following works are related to ATMem.

Data placement. [130] employs a data-centric analysis and a differential analysis to profile and associate each data structure with varied latency and bandwidth configurations. [34] classifies the memory access pattern of each memory region into three classes using an Intel Pin tool. It aims to maximize the overall data placement benefit with a greedy algorithm. [116] provides guidance for data placement by collecting memory traces with Intel Pin-based offline profiling tool. Above works employ offline profiling. More recently, [134] proposes Tahoe, a run-time PMU based data placement tool. Tahoe also uses LLC miss as the main metric to identify the data placement benefit of each data structure. It targets the whole data structure placement. These works do not target graph applications and thus, not considering adaptive granularity or dynamics as ATMem.

Data movement. [87] introduces a new OS-service for asynchronous memory movement with hardware acceleration. Similar efforts in [66, 5, 138, 15] also intend to optimize memory movement at either operating system or architecture level. As system-level services, their solutions need to ensure performance reliability at the cost of extra overhead. While they target future operating system or architecture, ATMem leverages application knowledge by using an application-level mechanism to improve migration on the existing systems.

Data placement on GPUs. Modern GPUs also feature heterogeneous memories, i.e., high-bandwidth GDDR or HBM on the device and DRAM on the host. [116] introduces a Pin based offline profiling tool. [20] proposes a memory specification language to guide data placement on GPUs. Extending our approach on GPUs requires special consideration in CPU-GPU links and data coherence and GPU execution model.

4.9 Discussion

This section discusses some current limitations and possible generalization in the future work.

Limitations. First, ATMem currently focuses on the performance aspect. Our future work will extend the heuristic in data management to guarantee data consistency (particularly for NVM) when on demand. Second, some HMS architecture could support aggregated bandwidth from memories. For instance, KNL has independent memory channels to MCDRAM and DRAM respectively. In contrast, the Intel Optane NVM is sharing memory channels with DRAM. ATMem will continue enhance placement decisions to utilize both memory bandwidth when supported by the architecture. Third, ATMem migrates data during the iterations of graph execution. Using advanced compiler analysis to automatically insert ATMem API between iterations could overlap the data movement.

Generalization. Although ATMem is specifically presented as an HMS management framework for graph applications, it also works well for other irregular applications or even regular ones because its profiling and data migration mechanisms are generally applicable to any applications. We also evaluated ATMem on sparse matrix computations, such as SpMV, and it achieved similar results as the graph applications. Data accesses in regular applications are more uniformly distributed so that adjusting data chunks to equal size of the whole data structure results in the same data placement as exiting coarse-grained solutions.

4.10 Summary

Active development in new memory devices brings heterogeneous memory systems as a solution to address the scaling challenge in DRAM. Efficient data placement in graph applications on heterogeneous memory systems needs to leverage the advantage of each memory while avoiding their limitations. This work proposes ATMem, an adaptive-grained runtime framework that consists of a lightweight profiler based on hardware sampling, a novel analyzer using *m-ary tree-based* heuristics to classify and predict data regions, and a high-bandwidth migration mechanism at the application level.

ATMem is evaluated on two real heterogeneous memory systems including the latest Intel Optane NVM, with five graph applications. The experimental results show that ATMem can achieves an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM over the baseline by selecting merely 5%-18% data onto high-performance memory. ATMem also helps to bridge the gap between NVM and DRAM on the NVM-DRAM machine, achieving a comparable performance to the case that places all data in the high-performance DRAM. ATMem data migration also outperforms the system service with 2.07×-5.32× speedup.

Chapter 5

Conclusion and Future Research Directions

5.0.1 Summary of Dissertation Contributions

Under the rapid development of machine learning (ML), ML applications with larger scale and higher complexity arise, demanding more optimized ML systems. Compare to general-purpose optimization, domain-specific optimization is a more effective method for building advanced ML systems, because it makes specialized system optimizations for a particular type of ML application based on its characteristics. Following this direction, in this dissertation, we present three works to perform domain-specific ML system optimizations for three different types of ML applications.

Checkpoint Construction Optimization for SGD-based Applications. We present LC-Checkpoint, a lossy scheme to compress checkpoints of SGD-based Applications. LC-Checkpoint is a delta-encoding scheme that only tracks the information on the difference between two checkpoints. It obtains the most significant information and represents them in a suitable format with an exponent-based quantization and a priority promotion method. At last, it applies a Huffman coding to further compress the bits. This work is accepted to ICML'20 [21].

Parallel Software for Million-scale Exact Kernel Regression. We present a software solution for kernel regression applications to handle million-scale datasets. It integrates a state-of-the-art parallel eigenvalue iterative solver, an advance block-based matrix-vector multiplication routine, and a software pipeline that coordinates the hyper-parameter optimizer and the HPC back-end. The software is efficient and robust, which has been proved by testing on the ImageNet dataset and a large asset pricing dataset. This work is under-reviewing by an HPC conference.

Memory Data Placement Optimization for Graph-based Applications. We present ATMem—a runtime framework for adaptive granularity data placement optimization in graph applications. ATMem enables adaptive data chunk profiling for subsequent partial data structure placement, and identifies the critical regions inside data structures by applying a local relative ranking strategy and a tree-based promoting method. At last, it uses a multi-stage multi-threaded migration strategy at the application level to enable high-bandwidth data migration and reduce TLB misses. This work is accepted to CGO'20 [22].

5.0.2 Future Directions

Our future work targets to further bridge the gap between ML systems and ML applications by customizing the ML algorithms. In this dissertation, although we perform effective optimizations for multiple types of ML applications by leveraging the application's domain knowledge, domain-specific system optimization is not a comprehensive solution for ML applications. Combining the ML application information without modifying the ML algorithms limits us from further improving the ML systems. Our future work will adjust the ML algorithms for software systems/architectures, deeply bind the design of ML algorithms to the implementation of ML systems, to achieve optimal solutions for ML applications.

Bibliography

[1] Friendster network dataset – KONECT, April 2017.

[2] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[6] Sina Alemohammad, Zichao Wang, Randall Balestriero, and Richard Baraniuk. The recurrent neural tangent kernel. *arXiv preprint arXiv:2006.10246*, 2020.

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[8] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1992.

[9] Haim Avron, Kenneth L Clarkson, and David P Woodruff. Faster kernel ridge regression using sketching and preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1116–1138, 2017.

[10] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Software Environ. Tools 11, SIAM, Philadelphia, 2000.

[11] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software*, 36(3), 2009. http://trilinos.sandia.gov/packages/anasazi.

[12] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the*

solution of linear systems: Building blocks for iterative methods. SIAM, Philadelphia, PA, 1994.

[13] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. SIAM, 1997.

[15] Santiago Bock, Bruce R Childers, Rami Melhem, and Daniel Mossé. Concurrent migration of multiple pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 420–423. IEEE, 2016.

[16] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[17] Serhat S Bucak, Rong Jin, and Anil K Jain. Multiple kernel learning for visual object recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(7):1354–1369, 2013.

[18] Junyi Chai, Hao Zeng, Anming Li, and Eric WT Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[19] Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the gpu, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021.

[20] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100. IEEE Computer Society, 2014.

[21] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On efficient constructions of checkpoints. *arXiv preprint arXiv:2009.13003*, 2020.

[22] Yu Chen, Ivy B Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 293–304, 2020.

[23] HMC Consortium. Hybrid Memory Cube Specification 2.1. `http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf`, 2015. [Online; accessed 22-May-2018].

[24] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[25] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[26] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S Nikolopoulos, Bronis R De Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 250–259. ACM, 2008.

[27] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019.

[28] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 261–272, 2013.

[29] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[30] Lee H Dicker, Dean P Foster, and Daniel Hsu. Kernel ridge vs. principal component regression: Minimax bounds and the qualification of regularization operators. *Electronic Journal of Statistics*, 11(1):1022–1047, 2017.

[31] Thaleia Dimitra Doudali and Ada Gavrilovska. Mnemo: Boosting memory cost efficiency in hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 412–421. IEEE, 2019.

[32] Petros Drineas, Michael W Mahoney, and Nello Cristianini. On the nyström method for approximating a gram matrix for improved kernel-based learning. *journal of machine learning research*, 6(12), 2005.

[33] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007.

[34] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and

Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

[35] Xin Feng, Youni Jiang, Xuejiao Yang, Ming Du, and Xin Li. Computer vision algorithms and hardware implementations: A survey. *Integration*, 69:309–320, 2019.

[36] Vincent W Freeh, Nandini Kappiah, David K Lowenthal, and Tyler K Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *Journal of Parallel and Distributed Computing*, 68(9):1175–1185, 2008.

[37] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[38] Andreas Frommer and Peter Maass. Fast cg-based methods for tikhonov–phillips regularization. *SIAM Journal on Scientific Computing*, 20(5):1831–1850, 1999.

[39] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 224–235, New York, NY, USA, 2019. ACM.

[40] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*, 2018.

[41] Mustansar Ali Ghazanfar, Adam Prügel-Bennett, and Sandor Szedmak.

Kernel-mapping recommender system algorithms. *Information Sciences*, 208:81–104, 2012.

[42] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *information retrieval*, 4(2):133–151, 2001.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[44] Shihao Gu, Bryan Kelly, and Dacheng Xiu. Empirical asset pricing via machine learning. *The Review of Financial Studies*, 33(5):2223–2273, 2020.

[45] Meng-Hao Guo, Tian-Xing Xu, Jiang-Jiang Liu, Zheng-Ning Liu, Peng-Tao Jiang, Tai-Jiang Mu, Song-Hai Zhang, Ralph R Martin, Ming-Ming Cheng, and Shi-Min Hu. Attention mechanisms in computer vision: A survey. *Computational Visual Media*, pages 1–38, 2022.

[46] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems*, pages 1379–1387, 2016.

[47] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[48] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[49] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[50] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604, 2017.

[51] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[52] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005.

[53] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[54] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[55] Mingyi Hong, Zhi-Quan Luo, and Meisam Razaviyayn. Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. *SIAM Journal on Optimization*, 26(1):337–364, 2016.

[56] Toshiyuki Imamura, Susumu Yamada, and Masahiko Machida. Development of a high performance eigensolver on the petascale next generation supercomputer system. *Progress in Nuclear Science and Technology*, 2:643–650, 2011.

[57] Intel. Intel® 64 and ia-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, May 2019.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerat-

ing deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[59] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Ion Stoica, Raman Arora, et al. Communication-efficient distributed sgd with sketching. In *Advances in Neural Information Processing Systems*, pages 13144–13154, 2019.

[60] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31, 2018.

[61] V Nisha Jenipher and S Radhika. Svm kernel methods with data normalization for lung cancer survivability prediction application. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pages 1294–1299. IEEE, 2021.

[62] JEDEC JESD229-2. Wide I/O 2 (WideIO2), 2014. [Online; accessed 22-May-2018].

[63] JEDEC JESD235A. High bandwidth memory (HBM) DRAM. *JEDEC Solid State Technology Association*, Nov 2015.

[64] JEDEC JESD250. Graphics double data rate 6 (GDDR6) SGRAM standard. *JEDEC Solid State Technology Association*, Jul 2017.

[65] Taichi Joutou and Keiji Yanai. A food image recognition system with multiple kernel learning. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pages 285–288. IEEE, 2009.

[66] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos—os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534. IEEE, 2017.

[67] Shubhra Kanti Karmaker, Md Mahadi Hassan, Micah J Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. Automl to date and beyond: Challenges and opportunities. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[68] Shristi Shakya Khanal, PWC Prasad, Abeer Alsadoon, and Angelika Maag. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies*, 25(4):2635–2664, 2020.

[69] Daekyum Kim, Sang-Hun Kim, Taekyoung Kim, Brian Byunghyun Kang, Minhyuk Lee, Wookeun Park, Subyeong Ku, DongWook Kim, Junghan Kwon, Hochang Lee, et al. Review of machine learning methods in soft robotics. *Plos one*, 16(2):e0246102, 2021.

[70] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.

[71] Marius Kloft, Ulf Brefeld, Sören Sonnenburg, and Alexander Zien. Lp-norm multiple kernel learning. *The Journal of Machine Learning Research*, 12:953–997, 2011.

[72] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[74] Harold Kushner and G George Yin. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

[75] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited:

Near optimal parallel matrix-matrix multiplication. SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[76] Tze Leung Lai. Martingales in sequential analysis and time series, 1945–1985. *Electronic Journal for history of probability and statistics*, 5(1), 2009.

[77] Ivano Lauriola and Fabio Aiolli. Mklpy: a python-based framework for multiple kernel learning. *arXiv preprint arXiv:2007.09982*, 2020.

[78] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[79] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[80] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK User's guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia, PA, 1998.

[81] Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[82] Dong Li, Bronis R de Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[83] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th*

{USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pages 583–598, 2014.

[84] Xin Li, Mengyue Wang, and T-P Liang. A multi-theoretical kernel-based approach to social network-based recommendation. *Decision Support Systems*, 65:95–104, 2014.

[85] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. Enforcing crash consistency of evolving network analytics in non-volatile main memory systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 124–137. IEEE, 2019.

[86] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[87] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. *ACM SIGARCH Computer Architecture News*, 44(2):369–383, 2016.

[88] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[89] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*, 2012.

[90] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[91] William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016.

[92] Per-Gunnar Martinsson. Randomized methods for matrix computations. *The Mathematics of Data*, 25(4):187–231, 2019.

[93] Jeffrey C Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194, 1997.

[94] Ashin Mukherjee and Ji Zhu. Reduced rank ridge regression and its kernel extensions. *Statistical analysis and data mining: the ASA data science journal*, 4(6):612–622, 2011.

[95] Yuji Nakatsukasa and Joel A. Tropp. Fast & accurate randomized algorithms for linear systems and eigenvalue problems, 2021.

[96] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*, 2020.

[97] M Ben Olson, Tong Zhou, Michael R Jantz, Kshitij A Doshi, M Graham Lopez, and Oscar Hernandez. Membrain: Automated application guidance for hybrid memory systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2018.

[98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[99] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5456–5464, 2017.

[100] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[101] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[102] Ivy B Peng and Jeffrey S Vetter. Siena: exploring the design space of heterogeneous memory systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 427–440. IEEE, 2018.

[103] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, pages 82–91. ACM, 2017.

[104] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graph-Phi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures. In *2018 International Conference on Parallel Architecture and Compilation (PACT)*. ACM, 2018.

[105] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[106] Ivens Portugal, Paulo Alencar, and Donald Cowan. The use of machine learning algorithms in recommender systems: A systematic review. *Expert Systems with Applications*, 97:205–227, 2018.

[107] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 631–644, 2018.

[108] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric P Xing. Fault tolerance in iterative-convergent machine learning. *arXiv preprint arXiv:1810.07354*, 2018.

[109] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3), 2009.

[110] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20, 2007.

[111] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[112] Marc Rothmann and Mario Porrmann. A survey of domain-specific architectures for reinforcement learning. *IEEE Access*, 10:13753–13767, 2022.

[113] Nicholas Roy, Ingmar Posner, Tim Barfoot, Philippe Beaudoin, Yoshua Bengio, Jeannette Bohg, Oliver Brock, Isabelle Depatie, Dieter Fox, Dan Koditschek, et al. From machine learning to robotics: Challenges and opportunities for embodied intelligence. *arXiv preprint arXiv:2110.15245*, 2021.

[114] Bernhard Schölkopf, Alexander J Smola, Francis Bach, et al. *Learning with kernels: support vector machines, regularization, optimization, and beyond.* MIT press, 2002.

[115] Hyunseok Seo, Masoud Badiei Khuzani, Varun Vasudevan, Charles Huang, Hongyi Ren, Ruoxiu Xiao, Xiao Jia, and Lei Xing. Machine learning techniques for biomedical image segmentation: an overview of technical aspects and introduction to state-of-art applications. *Medical physics*, 47(5):e148–e167, 2020.

[116] Du Shen, Xu Liu, and Felix Xiaozhu Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 13–23. ACM, 2016.

[117] Thiago Silveira, Min Zhang, Xiao Lin, Yiqun Liu, and Shaoping Ma. How good your recommender system is? a survey on evaluations in recommendation. *International Journal of Machine Learning and Cybernetics*, 10(5):813–831, 2019.

[118] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[119] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, Susan Mehringer, Eric Wernert, H Tufo, D Panda, et al. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pages 1–8. 2017.

[120] Andreas Stathopoulos and James R. McCombs. Primme: Preconditioned iterative multimethod eigensolver—methods and software description. *ACM Trans. Math. Softw.*, 37(2), apr 2010.

[121] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. Toward efficient interactions between python and native libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1117–1128, 2021.

[122] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[123] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. Magma users' guide. *ICL, UTK (November 2009)*, 2011.

[124] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. Xsede: accelerating scientific discovery. *Computing in science & engineering*, 16(5):62–74, 2014.

[125] Jan Van Leeuwen. On the construction of huffman trees. In *ICALP*, pages 382–410, 1976.

[126] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022.

[127] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–362. ACM, 2019.

[128] Ke Alexander Wang, Geoff Pleiss, Jacob R Gardner, Stephen Tyree, Kilian Q. Weinberger, and Andre Gordon Wilson. Exact gaussian processes

on a million data points. *33rd Conference on Neural Information Processing Systems, NeurIPS*, 2019.

[129] Li Wang and Ji Zhu. Financial market forecasting using a two-step kernel learning method for the support vector regression. *Annals of Operations Research*, 174(1):103–120, 2010.

[130] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 263–273. ACM, 2018.

[131] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13, 2000.

[132] David P Woodruff et al. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

[133] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[134] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 31. IEEE Press, 2018.

[135] Qiong Wu, Christopher G Brinton, Zheng Zhang, Andrea Pizzoferrato, Zhenming Liu, and Mihai Cucuringu. Equity2vec: End-to-end deep learning

framework for cross-sectional asset pricing. In *Proceedings of the Second ACM International Conference on AI in Finance*, pages 1–9, 2021.

[136] Qiong Wu, Felix M Wong, Yanhua Li, Zhenming Liu, and Varun Kanade. Adaptive reduced rank regression. *Advances in Neural Information Processing Systems*, 33:4103–4114, 2020.

[137] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[138] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 331–345, New York, NY, USA, 2019. ACM.

[139] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[140] Chenyun Yu and Ka Chi Lam. Applying multiple kernel learning and support vector machine for solving the multicriteria and nonlinearity problems of traffic flow prediction. *Journal of Advanced Transportation*, 48(3):250–271, 2014.

[141] Yuchen Zhang, John Duchi, and Martin Wainwright. Divide and conquer kernel ridge regression. In *Conference on learning theory*, pages 592–617. PMLR, 2013.

[142] Chunlin Zhao, Chongxun Zheng, Min Zhao, Yaling Tu, and Jianping Liu. Multivariate autoregressive models and kernel learning algorithms for classifying driving mental fatigue based on electroencephalographic. *Expert Systems with Applications*, 38(3):1859–1865, 2011.

[143] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

VITA

Yu Chen

Yu Chen is a Ph.D. candidate in the Department of Computer Science at the College of William & Mary. He is co-advised by Dr. Bin Ren, Dr. Zhenming Liu, and Dr. Andreas Stathopoulos. His research lies in machine learning systems, with a focus on building profiling tools and performing system-algorithm co-design to optimize machine learning applications. His Ph.D. research appeared in CGO 2020, ICML 2020 and FSE 2021. Previously, he received his Bachelor of Software Engineering at Southeast University, China, in 2014. Prior to his Ph.D., he worked as a senior software engineer in the industry.