RICE UNIVERSITY

A Data and Platform-Aware Framework For Large-Scale Machine Learning

by

Azalia Mirhoseini

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Farnaz Koushanfar, Chair
Associate Professor of Electrical and Computer Engineering

Behnaam Aazhang
J.S. Abercrombie Professor of Electrical and Computer Engineering

Richard Baraniuk
Victor E. Cameron Professor of Electrical and Computer Engineering

Christopher Jermaine
Associate Professor of Computer Science

Houston, Texas
April, 2015
ABSTRACT

A Data and Platform-Aware Framework For Large-Scale Machine Learning

by

Azalia Mirhoseini

This thesis introduces a novel framework for execution of a broad class of iterative machine learning algorithms on massive and dense (non-sparse) datasets. Several classes of critical and fast-growing data, including image and video content, contain dense dependencies. Current pursuits are overwhelmed by the excessive computation, memory access, and inter-processor communication overhead incurred by processing dense data. On the one hand, solutions that employ data-aware processing techniques produce transformations that are oblivious to the overhead created on the underlying computing platform. On the other hand, solutions that leverage platform-aware approaches do not exploit the non-apparent data geometry.

My work is the first to develop a comprehensive data- and platform-aware solution that provably optimizes the cost (in terms of runtime, energy, power, and memory usage) of iterative learning analysis on dense data. My solution is founded on a novel tunable data transformation methodology that can be customized with respect to the underlying computing resources and constraints.

My key contributions include: (i) introducing a scalable and parametric data transformation methodology that leverages coarse-grained parallelism in the data to create versatile and tunable data representations, (ii) developing automated methods for quantifying platform-specific computing costs in distributed settings, (iii)
devising optimally-bounded partitioning and distributed flow scheduling techniques for running iterative updates on dense correlation matrices, (iv) devising methods that enable transforming and learning on streaming dense data, and (v) providing user-friendly open-source APIs that facilitate adoption of my solution on multiple platforms including (multi-core and many-core) CPUs and FPGAs.

Several learning algorithms such as regularized regression, cone optimization, and power iteration can be readily solved using my APIs. My solutions are evaluated on a number of learning applications including image classification, super-resolution, and denoising. I perform experiments on various real-world datasets with up to 5 billion non-zeros on a range of computing platforms including Intel i7 CPUs, Amazon EC2, IBM iDataPlex, and Xilinx Virtex-6 FPGAs. I demonstrate that my framework can achieve up to 2 orders of magnitude performance improvement in comparison with current state-of-the-art solutions.
I am incredibly grateful that I have had the opportunity to work with so many great people over the course of my graduate studies. I would like to acknowledge my advisor Prof. Farinaz Koushanfar, who has been a masterful mentor and a true friend. I have thoroughly enjoyed being her student, and can barely express my thankfulness at her inspiration and understanding. Her encouragement and criticism were instrumental in helping me to grow as a researcher.

This thesis would not have been possible without the advice, support, and collaboration of many people. I would like to thank my thesis committee, Profs. Farinaz Koushanfar, Behnaam Aazhang, Richard Baraniuk, and Christopher Jermaine for their valuable suggestions and inputs. Thanks to my fellow graduate students Eva, Ebrahim, and Bita for being great collaborators. I would also like to thank Prof. Miodrag Potkonjak for his contributions to my research in my early years as a graduate student.

My experience at Rice would not have been nearly as fun and rewarding without a number of great friends that I have met along the way. Thank you Clay, Adriana, Bita, Chaitra, Ebrahim, Eva, Mehrdad, and Sitara for being awesome friends and great supporters.

Finally, I must thank my wonderful family. I am greatly fortunate to have the most loving, supportive, and inspiring sister and brother. Most importantly, I am deeply grateful to my amazing parents for their unending love, support, and encouragement. I dedicate this thesis to my parents.
## Contents

Abstract .................................................. ii
List of Illustrations ................................... ix
List of Tables ............................................ xiv

1 Introduction ........................................... 1
   1.1 Target machine learning problems ................. 2
   1.2 Challenges in large scale and prior art .......... 3
       1.2.1 Data-aware solutions ....................... 4
       1.2.2 Platform-aware solutions ................. 5
   1.3 Our pursuit ....................................... 6
       1.3.1 Our distinguished approach ............... 7
       1.3.2 Broader impact and re-usability .......... 8
   1.4 Roadmap .......................................... 9
       1.4.1 Parametric data transformation .......... 9
       1.4.2 Platform-aware cost quantification and customization 11
       1.4.3 Streaming datasets ...................... 12
       1.4.4 Evaluations ................................ 13
       1.4.5 Summary .................................. 13

2 Global flow ............................................ 14
   2.1 Framework’s overview .......................... 16
       2.1.1 Motivational example .................... 18
   2.2 Target applications ............................. 19
   2.3 Target datasets .................................. 21
3 A versatile and tunable data transformation

3.1 Overview of Extensible Dictionary transformation (ExD)  23
3.2 Related work  23
3.3 ExD algorithm  26
  3.3.1 Creating the extensible dictionary  26
  3.3.2 Forming sparse representations  28
  3.3.3 Complexity analysis  30
  3.3.4 Sparsity guarantees for $C$  30
  3.3.5 Computational benefits of ExD-based transformation  31
  3.3.6 Impact of transformation error on learning accuracy  32
  3.3.7 Error tuning  33

4 Distributed mapping and platform-aware transformation tuning

4.1 Overview of platform-aware mapping and customization  35
4.2 Related work  35
4.3 Distributed execution flow and data partitioning  37
  4.3.1 Performance quantification  38
4.4 Automated Customization of ExD  41
4.5 Adaptation to graph-parallel abstractions  44
  4.5.1 Distributed graph partitioning  45
  4.5.2 Performance bounds  46

5 An online data transformation for streaming applications

5.1 Overview of streaming data transformation (SSketch)  48
5.2 Related Work  50
  5.2.1 Streaming Model  50
5.2.2 Hardware-accelerated OMP .................................................. 51
5.3 SSketch Algorithm ................................................................. 51
  5.3.1 Blocking SSketch algorithm ............................................... 55
5.4 SSketch hardware-accelerated implementation ............................. 56
5.5 Automated SSketch customization .......................................... 59
  5.5.1 Constraint-driven optimization ....................................... 60
  5.5.2 Design space exploration/ SSketch customization examples ... 63
5.6 Theoretical bounds on transformation error ............................... 66

6 Evaluations 68
  6.1 Evaluation setup ............................................................... 68
    6.1.1 Datasets ................................................................. 68
    6.1.2 Implementation and API .............................................. 69
    6.1.3 ExD overhead .......................................................... 70
    6.1.4 Verification of the performance model ............................ 71
    6.1.5 Estimating sparsity of $C$ from subsets of data during
        preprocessing ............................................................ 72
    6.1.6 Effect of transformation error on sparsity of $C$ ............... 73
  6.2 Applications ................................................................. 74
    6.2.1 $\ell_1$ minimization applications .................................. 74
    6.2.2 Power method applications ....................................... 79
  6.3 Graph-parallel abstractions ................................................ 82
  6.4 Apache Spark-based implementation ..................................... 84
  6.5 SSketch experimental evaluations ....................................... 85
    6.5.1 Hardware-accelerated settings and streaming results .......... 86

7 Summary 90
  7.1 Future directions .......................................................... 91
    7.1.1 Non-linear kernel matrices ....................................... 91
7.1.2 Privacy-preserving machine learning .......................... 92
7.1.3 Low-cost machine learning on mobile devices .............. 93

Bibliography 94
Illustrations

2.1 The overall flow of our framework. Dataset $A$ is transformed into two factors, a dictionary matrix $D$ and a sparse coefficient matrix $C$. The result is used to facilitates the iterative Gram matrix multiplication by replacing $G = A^T A$ with $C^T D^T D C$. The degree of freedom in creating $D$ is exploited to customize the transformation according to the properties of the pertinent distributed processing platform such as the relative cost of floating point operations and word communication.

3.1 The dense data matrix $A$ is transformed into a product of a dictionary matrix $D$ and a sparse coefficient matrix $C$.

3.2 A union of two 1-dimensional signal models. By selecting vectors $a$ and $b$ as the atoms of the dictionary, each data point can be approximated using only one atom.

4.1 Average number of non-zeros in each column of $C$, i.e., $\alpha(L)$ as a function of the number of columns in the dictionary, i.e., $L$. The bars show the variance for 100 trials of random sub-sampling to form $D$. The average transformation error $\|A - DC\|_F / \|A\|_F$ as a function of $L$.

4.2 Distributed graph-parallel mapping and scheduling of iterative updates on the transformed data.

5.1 High level block diagram of SSketch.
5.2 Schematic depiction of blocking SSketch algorithm. .......................... 56
5.3 High level diagram of SSketch API. The constraint-driven customization unit takes user-defined properties and hardware limitations as inputs and delivers output parameters for an optimized sketch computation. The input parameters may include runtime, power, and memory constraints and the outputs include sketching algorithmic parameters as well as guidelines for the hardware mapping. 58
5.4 Overview of SSketch automated design space exploration and customization. ................................................................. 59
5.5 Performance of SSketch’s FPGA-accelerated implementation. In both (b), and (c) we set $k = L$ to let $k$ be as large as the dictionary size and use $\epsilon$ as the stopping criteria in the OMP algorithm. ............. 60
5.6 Motivational examples for SSketch’s automated constraint-driven customization. Each of these figures demonstrates the average runtime required to process one sample as a function of the dictionary size. The user power constraint determines the maximum number of OMP kernels that can work in parallel. The dashed horizontal line, reflects the user runtime deadline and the purple star illustrates the SSketch’s automated customization output (optimal dictionary size). The crossing point of the dashed line and the corresponding runtime curve (solid curve in each figure) is the actual optimized point in each of these settings. ......................... 64
6.1 Preprocessing time overhead of ExD: The graphs show the runtime of applying ExD on different subsets of $A$. For a fixed dictionary size $L$, the runtime scales almost linearly with the cardinality of the data (as shown on the figures). The runtime is a function of both $L$ and $\alpha(L)$. 70
6.2 Verification of performance model: The graphs show estimated (top row) and actual (bottom row) performance of $C^T D^T D C x$ for various platforms. Our predicted runtime (in terms of number of floating point operations) closely follows the trend (in terms of milliseconds) in the actual evaluations.

6.3 Effective estimation of sparsity for tuning ExD: The graphs show the average number of non-zeros per column of $C$ (i.e., $\alpha(L)$) versus $L$ for different subsets of $A$. As the sizes of the subsets increase, $\alpha(L)$ converges to that of the full data $A$.

6.4 Effect of transformation error on the sparsity: The graphs show the average number of non-zeros per column of $C$ (i.e., $\alpha(L)$) versus different transformation errors (i.e., $\epsilon$). As $\epsilon$ increases, better sparsity is achieved. This is because for smaller $\epsilon$’s, more non-zero coefficients are added to $C$ so that the error tolerance criteria is met.

6.5 Sparse representation-based face recognition. On top, we display a test image; on the bottom, we display the sparse solution obtained with the original Gram (blue, solid) and approximate Gram with ExD for $\epsilon = 0.05$ (red, dash). We display four training images corresponding to non zero coefficients: large amplitude coefficients correspond to images in the correct class and smaller amplitude coefficients correspond to images in incorrect classes. The block of coefficients corresponding to images in the correct class are highlighted.

6.6 (a) For a range of decompositions (varying $\epsilon$) we show learning accuracy which measures the 2-norm between the solution obtained with the full and approximate Gram, (b) For a range of decompositions (varying $\epsilon$) we show sum of coefficients in the correct class, the minimum sum of coefficients required to correctly classify the test image is also shown.
6.7 Runtime of Power method on data from customized ExD, non-customized ExD ($L = L_{min}$), and baseline. ExD results in significant runtime reduction. Platform-aware customization of ExD generates meaningful further enhancements. ................................. 79

6.8 Runtime performance of ExD vs. transformation error ($\epsilon$) and number of processing nodes ($N_p$). Reported values are runtime for finding the first 100 largest eigenvalues using Power method. When larger $\epsilon$'s are allowed, ExD produces more efficient transformations that further improve runtime. ................................................................. 80

6.9 Cumulative error of the first computed 100 eigenvalues for different values of transformation error $\epsilon$. Smaller $\epsilon$'s result in more accurate results. However, in all the experiments, the cumulative error are still very low and significantly smaller than the transformation error. .... 81

6.10 Comparing matrix-based and graph-based computing models for different dictionary sizes and data sparsity levels. The results report runtime for (a) varying $L$ sizes (for constant total number of non-zeros in $C$), (b) varying relative density of $C$ (for constant $L$), and (c) varying number of processors (for constant $L$ and total number of non-zeros in $C$). ................................................................. 82

6.11 Comparison of MPI/C++ based and Spark based implementation’s average runtime per iteration for power method algorithm. All the experiments are done on the same platform with $N_P = 64$. ............. 84
6.12 Experimental evaluations of SSketch. \( \alpha \) and \( \epsilon \) represent the user-defined projection threshold, and the transformation error threshold respectively. \( L \) is the number of columns in the dictionary matrix, and \( m_b \) is the block-size. (a) Factorization error (\( X_{err} \)) vs. \( L \) with \( \alpha = 0.1 \), block-size = 200 and \( \epsilon = 0.01 \). (b) Factorization error (\( X_{err} \)) vs. block-size with \( \alpha = 0.1 \) and \( L = 128 \). (c) Compression-rate vs. block-size with \( \alpha = 0.1 \) and \( L = 128 \).

6.13 The internal structure of one OMP kernel. All computations are done in IEEE 754 single precision floating-point format. The \( D \) and \( Q \) matrices are stored in multiple smaller sized block memories and are filled by cyclic interleaving.
Tables

2.1 Matrices $\mathbf{D}_{\text{opt}}(1)$ and $\mathbf{D}_{\text{opt}}(2)$ denote the optimal dictionaries for transforming a dataset, if analysis on this data is to be done on Platforms 1 and 2 receptively. The two transformed datasets are tested on Platforms 1 and 2. The platform-optimized transformation results in meaningful improvement in runtime. ........................ 19

5.1 Power consumption on Virtex 6 for different number of OMP kernels. ........................ 62

6.1 FISTA runtime (s) to reach to a specific PSNR. ........................ 78
6.2 Reduction in Power method runtime and memory usage. ExD inputs are $N_P = 64$ and $\epsilon = 0.1$. ........................ 81
6.3 Virtex-6 resource utilization. ........................ 88
6.4 SSketch scalability. The total processing time is linear in terms of the number of processed samples. ........................ 89
Chapter 1

Introduction

The world is undergoing a massive data revolution. While many exciting new discoveries lie hidden within the data, current algorithms and computing architectures are being overwhelmed by the exponential growth of datasets. Over the last decade, significant strides have been made in parallel research domains to address the challenges of big data including subsampling and dimensionality-reducing transformations, resource-efficient algorithms for deployment on distributed or heterogeneous hardware, and hardware-based acceleration.

Despite these amazing advances, existing solutions have yet to simultaneously realize the true potential of modern data processing and computing capabilities at our disposal. On the one hand data-aware solutions produce transformations that are oblivious to the costs and constraints of the underlying computing platform. On the other hand, platform-aware solutions do not exploit the hidden data properties.

The goal of this research is to bridge these different domains; to provide highly efficient and scalable machine learning solutions that holistically take into account the properties of data geometry, learning algorithms, and the underlying computing fabrics. Our work is the first to develop comprehensive data- and platform-aware transformation and architecture mapping techniques that provably optimize the computing cost in terms of runtime, energy, and memory usage.

In the rest of this chapter we begin with discussing the machine learning and optimization problems that are the focus of this thesis. We highlight the challenges
that arise while solving these problems in large scale and review prior art. Next, we introduce our novel solution and describe its versatility and broader impact. Finally, we provide the organization and a brief summary of each of the chapters.

1.1 Target machine learning problems

In the modern world, an ever-increasing set of problems are solved through exploitation of data. In particular, machine learning and statistical approaches are central in tackling problems in artificial intelligence, medicine, computer vision, marketing, and finance. The abundance of data being generated has prompted scientists and engineers to focus on new large-scale algorithmic and architectural solutions that capture the complexity of modern data, and are capable of processing massive datasets in parallel and decentralized modes. Perhaps, what justifies the staggering amount of research on learning from large data, or what is commonly known as big data, is that even highly complicated problems may be solved by relatively simple models if trained on massive datasets [4].

Many modern learning algorithms are based on exploring the underlying patterns, correlations, and dependencies present across the signals in the dataset. Some prominent examples of such algorithms and their applications are linear or penalized regression, power iterations, belief propagation, and expectation maximization [25, 55, 74]. In all of these settings, solving the underlying objective function requires iterative updates of parameters of interest until convergence is achieved. Such iterative updates often require matrix multiplications that involve the data correlation or Gram matrix. While there are many variations of the iterative update algorithms (surveyed in [6]), the underlying complexity of all them arises from a projection step on the correlation matrix:
\[ x^{t+1} = \eta^t \text{prox}_f(Gx^t), \tag{1.1} \]

where \( A_{M \times N} \) is the data matrix, \( G = A^T A \) is the correlation matrix, \( x^t \) is the unknown solution vector that the problem is trying to solve at the \( t^{th} \) iteration, and \( \text{prox}_f(\cdot) \) is often a relatively low-complexity function that depends on the specifics of the learning application. Parameter \( \eta^t \) is known as the step-size or the learning rate and it controls the convergence speed.

A concrete example of learning algorithms that use Equation 1.1 is the ones that are posed in the framework of convex optimization:

\[
\text{minimize } f(x) \\
\text{subject to } y = Ax, \tag{1.2}
\]

where \( x \in \mathbb{R}^N \) is the solution, \( y \in \mathbb{R}^M \) is the input, and \( f : \mathbb{R}^N \rightarrow \mathbb{R} \cup \{+\infty\} \) is a closed convex function. For example if \( f(x) = |x|_1 \) the problem becomes an instance of the \( \ell_1 \) loss minimization problem. In this case \( f(x) = |x|_1, \text{prox}_f(\cdot) \) (in Equation 1.1) becomes a simple soft-thresholding function. In Section 2.2 we provide an in-depth discussion of the broad set of applications for Equation 1.1.

1.2 Challenges in large scale and prior art

In large-scale applications where matrix \( A \) is massive, computing each iterative update on the connectivity matrix becomes a very cumbersome process. This is because (i) matrix \( A \) is too large to fit onto the main memory of the platform, and/or (ii) the performance costs such as runtime, power, and energy usage for the updates of type \( Gx^t \) (in Equation 1.1) cannot be afforded.
Several classes of emerging large data sets, including the image and video content, contain dense (non-sparse) dependencies, i.e., a large number of non-zeros in the data correlation matrix \( G \). Not only are these data sets often too large to process on a single platform, but this dense data dependency requires a high amount of costly communication between the distributed platforms on each iterative updated.

Due to the wide applicability of Equation 1.1 in solving real-world data analysis applications, a large body of work has been dedicated to address the challenges that arise while solving this problem in large-scale scenarios. Next we will discuss two mainstream classes of research that have tackled this problem.

1.2.1 Data-aware solutions

**Dimensionality reduction methods.** Dimensionality reduction methods rely on geometrical properties of the data to reduce the computation cost. Despite the apparent dimension of data, in several real settings data lies on a lower dimensional space or a union of lower dimensional spaces. Many classic and contemporary works in data mining and machine learning have leveraged the existence of a lower-dimensional space to obtain a better understanding of the data patterns. For instance, a host of dimension reduction algorithms (a.k.a., *matrix factorization*) have been developed to exploit the low-dimensional structures for efficient computation. Prominent examples of such algorithms are principal component analysis [93], QR [50], LU [51], and Cholesky [52]. In all the above algorithms, data matrix \( A \) is factorized into a lower dimensional approximation matrix with orthogonal columns (the column-wise inner-products are zero) and a coefficient matrix.

Column subset selection (CSS) based approaches are another category of factorization methods that relax the orthogonality condition [92]. In these methods, a subset
of columns of matrix $A$ are selected to form a super set, then the other columns of $A$ are approximated as a superposition of that super set.

Other approaches are known as matrix sketching [83], and their variants [49, 42, 95]. All of these algorithms exploit the concept of matrix rank for dimensionality reduction. Once the dimensionality of the data is reduced, subsequent iterative learning algorithms can run more effectively on the lower dimensional structure.

**Stochastic first-order methods.** Stochastic first-order methods approach the iterative updates based on non-exact, unbiased gradient estimates; instead of updating the solution vector based upon the entire data matrix, only a subset (or a batch) of data is used in each iteration. The data-subset, denoted by $A_r$, is randomly constructed from a subset of rows of $A$. Note that in this case the size of the correlation matrix constructed from the subset remains the same, however, computing $A_r^T A_r x$ becomes more efficient [48, 35]. Different subsets of rows are selected at each iteration. These methods are commonly referred to as stochastic gradient descent (SGD) techniques.

Due to the inherently sequential nature of SGD, the algorithm does not readily benefit from being run in parallel or distributed system environments. A common choice to address this issue is asynchronous SGD (A-SGD) that operates without locking the solution vector. Updates to the variables can happen at any time by the independent processors [12, 40].

### 1.2.2 Platform-aware solutions

**High performance distributed mapping.** A large body of work has focused on developing methodologies for efficient mapping of various linear algebra and analysis algorithms onto distributed, heterogeneous, or reconfigurable architectures [39, 37, 38,
43, 13, 27]. While such methods effectively optimize the performance with hardware-aware customization, they do not holistically customize the representation of data (by revealing its subspace structure) for the pertinent platform. They instead adhere to data/correlation structure given by the application, which is ubiquitous regardless of the hardware platform.

**Distributed machine learning engines.** Distributed abstractions designate a model for execution flow of the iterative algorithms in a decentralized manner. Efficient data partitioning techniques to minimize inter-node communication is critical. Multiple successful distributed abstractions and frameworks have been proposed that model data dependency in a graph format, most notably Pregel [31] and GraphLab [30]. They use efficient graph partitioning techniques and a vertex-centric computation model, in which the user-defined iterative algorithms are executed on each vertex in parallel. Other projects such Apache Spark [94] provide distributed machine learning engines based on key-value tables that also efficiently represent many graph-parallel algorithms.

### 1.3 Our pursuit

This thesis proposes a novel framework that enables highly efficient execution of iterative learning algorithms on massive and densely correlated data through novel platform- and data-aware solutions.

We propose the first parametric data transformation algorithm as well as automated tuning and distributed mapping techniques that holistically take into account the underlying computing hardware’s costs and constraints. Two metrics characterize the system costs associated with designing an iterative algorithm on a (distributed) computing platform: (i) the number of arithmetic operations (flops), and (ii) the
amount of messages passed among the processing nodes (communications). Depending on the properties of the underlying computing platform (such as the underlying technology, hardware specifications, and number of computing nodes), the relative cost of these two metrics varies drastically. The constraints are dictated by the available memory, power, energy, and runtime constraints.

1.3.1 Our distinguished approach

In large scale applications, even the lower dimensional structures need to be processed in a distributed manner. However, all the existing dimensionality reduction techniques are oblivious to the cost and constraints imposed by the distributed computing platform while factorizing or creating a sketch of the data. For example, the discussed dimensionality reduction algorithms always return the same factorized components, irrespective of the underlying platform properties and configuration.

Our work demonstrates that we can significantly improve the overall performance of iterative learning approaches, specially in the case of large and dense data, by adaptive tuning of the transformation to the target compute platform.

Practical appeal of SGD methods remains limited as they require much larger number of iterations for convergence. This is due to the replacement of gradients with stochastic estimates [34]. Moreover, to avoid divergence from the solution, adaptive tuning of the learning rate $\eta$ becomes much more critical in SGD methods. Tuning $\eta$ is a challenging task and is the subject of several ongoing works [18, 47]. Our approach, unlike SGD, reduces the memory footprint via data transformation. In addition, since we operate on the entire dataset each iteration, our method does not suffer from high sensitivity to $\eta$.

An important property of our data transformation approach is to produce sparse
representations by revealing and leveraging the inter-dependency between data samples. As a result, our framework enables existing graph-parallel distributed engines such as GraphLab, that heavily rely on sparse data structures for performance optimization, to become applicable for densely correlated data. Furthermore, the performance of existing distributed engines is highly dependent on efficient partitioning of the dataset across the distributed processing nodes, but finding an efficient and balanced data and computation partition is a challenging problem. Several possible methods are proposed for partitioning the data, including spectral graph clustering [23] and heuristic multi-level recursive partitioning methods such as METIS [28]. However, these methods are only suited for sparse datasets. Efficient data partitioning is inherently impossible when data is densely connected. None of the existing distributed engines take advantage of dense data geometry for compression and efficiency.

1.3.2 Broader impact and re-usability

The broader goal of this thesis is to push large scale machine learning one step closer towards realizing the immense potential of big data, via the integrated exploitation of both intelligent data transformation/partitioning and platform-aware computing. To this end, we provide several APIs that enable machine learning practitioners and data scientists to take advantage of our highly efficient and versatile framework. Our user-friendly APIs can be readily modified to handle different problems that are in the general format of Equation 1.2. More specifically, we provide an entirely hand-coded C++/MPI based API for both data transformation and distributed mapping techniques for CPU-based clusters. We also provide Apache Spark and GraphLab based adaptations of the framework. The aforementioned distributed engines extend
the usability of our framework to massive distributed systems by providing fault-
tolerance.

While the majority of our solutions focus on multi-core and many-core distributed
platforms, we also investigate FPGA-based implementations. In particular, emerging
applications have an ever-growing data matrix, \( \mathbf{A} \), and can even be latency con-
strained; these scenarios may necessitate specialized high-throughput hardware that
can handle growing datasets. To achieve this goal, we carefully adapt the data trans-
formation approach with respect to the constraints of the aforementioned setting and
provide reusable high-level synthesis C-based APIs.

1.4 Roadmap

In Chapter 2, we provide the global flow and overview of our framework. In the
following we provide a brief summary of each of the technical chapters of this thesis.

1.4.1 Parametric data transformation

In chapter 3, we propose the idea of extensible dictionaries, the building block for our
parametric data transformation. Our transformation seeks to find matrices \( \mathbf{D} \) and \( \mathbf{C} \)
such that the following objective is minimized:

\[
\min \| \mathbf{C} \|_0 \quad s.t. \quad \| \mathbf{A} - \mathbf{DC} \|_F \leq \epsilon \| \mathbf{A} \|_F, \tag{1.3}
\]

where \( \mathbf{D} \) is an \( M \times L \) and \( \mathbf{C} \) is an \( L \times N \) matrix, \( L \leq N \) and \( \epsilon \) is the approximation
error, \( \| \mathbf{C} \|_0 \) measures the total number of non-zeros in \( \mathbf{C} \), and \( \| \cdot \|_F \) is the Frobenius
norm. We refer to matrix \( \mathbf{D} \) as a dictionary matrix. Columns of \( \mathbf{A} \), \( \mathbf{a}_i \)'s, are trans-
formed into a linear superposition of columns in the dictionary; that is, \( \mathbf{a}_i = \mathbf{Dc}_i \),
where \( c_i \)'s are the corresponding columns of \( C \).

The \( \ell_0 \) constraint imposes sparsity on elements of \( C \). We simultaneously exploit two inter-dependent approaches to produce sparse representations. The first approach is to create coarse-grained redundant structure in the dictionary. The other approach is to reveal the structural rank property of the data that can result in sparsity. More exactly, we exploit the fact that in many settings, the dense data might lie not on a single low rank space, but instead on a union of low-rank subspaces. Such datasets appear in many image and video processing applications [79, 78]. We identify and exploit the relationship between parameters \( L \), approximation error \( \epsilon \), and the achieved sparsity (or memory footprint) for customizing the transformation to the computing platform.

An important property of the transformation is its scalability. The dictionary matrix \( D \) is formed by sub-sampling columns of \( A \). Once matrix \( D \) is created, we use a matching pursuit-based algorithm to form matrix \( C \). Notably, each column of \( C \) can be constructed independently and thus in parallel.

Naturally, as we increase the factorization error (controlled by \( \epsilon \)) the accuracy of our learning algorithm (learning error) also increases. Oppositely, a larger \( \epsilon \) yields sparser decompositions. Thus, we are interested to determine the largest factorization error that we can afford to achieve a desired accuracy in the learning algorithm. As we will show through multiple examples, the answer to this question heavily depends on the application of interest. Previous theoretical works have established a connection between the total error in a factorization of a kernel (or correlation) matrix and the accuracy of certain popular learning algorithms, including kernel ridge regression and kernel SVM [9]. While our transformation allows the user to specify the error parameter \( \epsilon \), we also provide an alternative generic approach for tuning the
factorization error to achieve a specified learning accuracy.

### 1.4.2 Platform-aware cost quantification and customization

In Chapter 4, we propose an efficient distributed computational model to perform iterative updates (of the form of Equation 1.1) on the transformed data in a distributed fashion. Our data partitioning and execution flow takes advantage of both data and computation replication techniques. We demonstrate that our model can achieve the optimal bounds on communication provided in numerical computing literature [14]. To be able to customize the transformation to the platform, we define a performance metric which quantifies the computing cost (of our distributed model) in terms of number of floating point operations, number of communicated words, and memory usage. We then demonstrate how one can tune the transformation to optimize this metric.

We show that parameter $L$ governs the communication cost, creating a trade-off between the sparsity of the transformed data (which increases with $L$) and the communication overhead. We introduce an efficient method to predict the relationship between $L$ and the achieved sparsity. The proposed method is highly efficient and only requires access to a small subset of the original data. Given our prediction method, as well as the specifications of the pertinent hardware (e.g., memory capacity and relative cost of flop-operation and word-communication), we can tune the transformation to optimize the performance metric.

We provide APIs for both matrix-based and vertex-centric handling of the iterative update models on the transformed data. There are two versions of the matrix-based APIs. One uses a general Message Passing Interface (MPI) and the other one uses Apache Spark. Our vertex-centric API is written based on the GraphLab pro-
gramming model. We develop an efficient mapping of the iterative computations on the sparsified decomposed data within the constraints of the GraphLab distributed framework. We discuss the advantages of each of the APIs as well as their potential use-cases.

### 1.4.3 Streaming datasets

In Chapter 5, we extend our data transformation approach to streaming data scenarios. Several big data analysis applications, require real-time and online processing of streaming data. Examples include applications such as video surveillance and network traffic classification where storing the continuously generated data is superfluous and/or timely processing of data is vital [29].

The main concept is to adaptively update dictionary matrix $D$ as new data samples are being gathered. The single-pass property of the transformation incurs only a small memory usage, since only the newly generated sample and dictionary matrix have to be accessible at a time. Our *online* transformation is scalable and leverages coarse-grained parallelism existing in the dataset to significantly reduce costly memory interactions.

The low memory foot-print combined with the high-throughput requirement of our online transformation encourages hardware-accelerated FPGA-based implementation of our algorithm. The results show significant speedups in comparison to a single multi-core processing unit, highlighting the importance of incorporating hardware in large-scale streaming applications. A high-level-synthesis C-based API that facilitates the use of our proposed hardware/software co-design approach are provided.
1.4.4 Evaluations

In this Chapter 6, we provide extensive evaluations of our methodologies introduced in Chapters 3, 4, and 5. We test various learning algorithms including image classification, super-resolution. We investigate the accuracy of our cost quantification model by comparing it with real-world implementation results, and demonstrate that significant improvements can be achieved by our customizable transformation. We provide insights on the use-cases of our matrix-based and graph-based APIs. Finally, we show effectiveness of our hardware-accelerated streaming model for dynamically evolving data.

1.4.5 Summary

In Chapter 7, we provide a summary of our proposed solutions. We also discuss extension of our solution to non-linear kernel-based methods, privacy-preserving large-scale machine learning applications, and learning on mobile embedded devices.
Chapter 2

Global flow

Many modern learning algorithms attempt to make sense of the data by exploring the underlying patterns, correlations, and dependencies present across the signals. Some prominent examples of such algorithms and their applications are linear or penalized regression, power iterations, belief propagation, and expectation maximization. In all of these settings, solving the underlying objective function requires iterative updates of parameters of interest until convergence is achieved. Such iterative updates often require matrix multiplications that involve the data correlation matrix.

An increasingly important class of large scale data are those with dense (non-sparse) dependencies, i.e., a large number of non-zeros in the correlation matrix. Dense datasets appear in a large number essential applications including most image and video data. The dense dependencies severely degrade the performance of learning algorithms as they incur huge arithmetic operations. In scenarios where data is too large to fit on a single computing node and must be distributed, the dense structure can cause even more challenges due to the costly inter (processing) node communications.

Despite the presence of a dense correlation, in several real settings, the dense data lies on a lower dimensional space [79]. Several classic and contemporary works in data mining and machine learning have leveraged the existence of a lower-dimensional space to obtain a better understanding of the data patterns. For instance, a host of dimensionality reduction algorithms (a.k.a., matrix factorization) have been devel-
Figure 2.1: The overall flow of our framework. Dataset $A$ is transformed into two factors, a dictionary matrix $D$ and a sparse coefficient matrix $C$. The result is used to facilitate the iterative Gram matrix multiplication by replacing $G = A^T A$ with $C^T D^T D C$. The degree of freedom in creating $D$ is exploited to customize the transformation according to the properties of the pertinent distributed processing platform such as the relative cost of floating point operations and word communication.

Developed to exploit the low-dimensional structures for efficient computation. Prominent examples of such algorithms are singular value decomposition (SVD) [21], principle component analysis (PCA) [93], non-negative matrix factorization (NNMF) [16], factorizing by column subset selection (CSS) [92], matrix sketching [83], and their variants [49, 42, 95]. All these algorithms exploit the concept of matrix rank for dimensionality reduction and/or feature extraction. Once the dimensionality of the data is reduced, subsequent iterative learning algorithms can run more effectively on the lower dimensional structure.

This thesis proposes a novel distributed solution for efficient execution of a broad class of iterative learning algorithms on datasets with dense dependencies. Our overarching goal is to reduce the following critical metrics in high performance distributed computing systems: execution runtime, memory, power and energy usage, and communication overhead.
2.1 Framework’s overview

Figure 2.1 shows the overall flow of our framework. In Chapter 3, we introduce our tunable data transformation methodology. The transformation seeks to find a suitable dictionary matrix $D$ and a sparse coefficient matrix $C$ such that the following objective holds:

$$\min \|C\|_0 \ s.t. \ \|A - DC\|_F \leq \epsilon \|A\|_F,$$  

(2.1)

where $D$ is $M \times L$, $C$ is $L \times N$, and $L \ll N$. Parameter $\epsilon$ is a projection (transformation) error, $\|C\|_0$ is the total number of non-zeros in $C$, and $\| \cdot \|_F$ is the Frobenius norm.

The transformation is based on our novel idea of a Extensible Dictionary (ExD), a new scalable methodology that we suggest for data factorization. ExD provides a degree of freedom that we leverage to customize the mapping of the data to distributed platforms. The customization aims to optimize the performance cost metrics in terms of FLOPS and communications which translate to runtime and energy performance.

Our goal is to tune the transformation such that iterative processing on the factorized components (i.e., $(DC)^TDCx$) becomes much more efficient than iterative processing on the original data (i.e., $A^TAx$). The key idea is that parameter $L$ can be used to control the redundancy in the dictionary, which results in different levels of sparsity in the coefficient matrix $C$. In other words, by extending the dictionary size, we can achieve sparser $C$'s.

For a given dictionary size ($L$) and error ($\epsilon$), we provide an scalable algorithm to solve the data projection. Our algorithm exploits column subset selection techniques to create the dictionary, and a matching pursuit-based method [87] to sparsify $C$.

In Chapter 4, we propose an efficient distributed computing model to perform
iterative computations on $\mathbf{D} \mathbf{C}$. We demonstrate that our model can achieve the optimal numerical bounds on communication [14] while performing distributed matrix computations on the projected data. We show that parameter $L$ governs the communication cost as the number of communicated words increases linearly with $L$. Thus, there is a trade-off between the number of non-zeros (or computation and the memory footprint) of the factorized data and the communication overhead. We propose metrics to quantify the computing cost for the proposed distributed model, which directly characterize memory, runtime, and energy. Our publication on this research can be found at [33].

We next demonstrate our approach for tuning the ExD transformation to minimize the quantified computing costs. To do so, we introduce an efficient method to estimate the sparsity of $\mathbf{C}$ as a function of dictionary size. The proposed preprocessing method only requires access to a relatively small number of original data signals and thus it is applicable to massive datasets.

In some applications, such as video surveillance or Internet traffic data, the dataset $\mathbf{A}$ may dynamically grow over time. Let us assume $\mathbf{A} = \mathbf{D} \mathbf{C}$. Whenever a new set of columns are added to $\mathbf{A}$, we update the coefficient matrix $\mathbf{C}$ by applying ExD transformation on $\mathbf{A}_{\text{new}}$ to approximate $\mathbf{A}_{\text{new}} = \mathbf{D} \mathbf{C}_{\text{new}}$; where $\mathbf{A} = [\mathbf{A}, \mathbf{A}_{\text{new}}]$ and $\mathbf{C} = [\mathbf{C}, \mathbf{C}_{\text{new}}]$ represent the updated dataset and coefficient matrix. In some cases the newly added columns of $\mathbf{A}$ may not be expressed well by the space spanned by the current dictionary $\mathbf{D}$. For example, a new set of drastically different images can expand the space of the original dataset. In such cases, $\mathbf{D}$ should also be modified to include those new structures. In Chapter 5 we provide our methods for online data transformation. Our methods are embedded on FPGA to take advantage of its high throughput, an essential property in many online settings. Our publication on this
research can be found at [41].

### 2.1.1 Motivational example

Here we provide a motivational example that demonstrates the advantage of ExD for platform-specific customization and performance optimization. In this example, we customize the transformation for 2 different platform setups: Platform 1 is a single core and Platform 2 is a 64 core (8 processing node/8 cores each) implementation setup both on IBM iDataPlex server.

As mentioned earlier, many learning algorithms rely on iterative updates on the data correlation matrix to converge to a solution. An iterative step of the algorithm can be written in the following format:

\[ x^{t+1} = Gx^t, \]

where \( x^t \) denotes the vector of unknown variables at iteration \( t \). We apply the updates using the transformed correlation matrix \( x^{t+1} = (DC)^T(DC)x^t \).

In this example, matrix \( A \) is a dense dataset of cancer cell images (more information in Chapter 6) of size \( 1024 \times 111296 \). We set \( \epsilon \) to 5\%, and apply our optimization techniques to obtain the best dictionary for each platform. We denote the optimal dictionary size for Platforms 1 and 2 by \( L_{opt}(1) \) and \( L_{opt}(2) \). Table 2.1 shows the runtime performance of the iterative update algorithm. It can be seen that by using the optimal dictionary for Platform 1 (i.e., \( D = D_{opt}(1) \)) on that platform, the runtime for one iteration is 174.906ms. This runtime increases to 260.497ms if we instead use the optimal dictionary for Platform 2 (i.e., \( D = D_{opt}(2) \)) on Platform 1. Similar performance degradation happens when the optimal dictionary for Platform 1 is used on Platform 2.

To explain the runtime differences, we observe that the relative costs of computation and communication differs for the two platforms. While Platform 2 (with 64
cores) can perform arithmetics much faster than Platform 1 (using its many parallel cores), the communication cost of the iterative update algorithm across the 64 cores incurs a large overhead. Note that even before this tuning of $L$, both dictionaries improve the performance compared with prior art as they reduce the costly computations and communications.

Although the number of non-zeros in matrix $C$ (NNZ($C$) is almost twice when $D = D_{opt}(2)$ in comparison with the case where $D = D_{opt}(1)$ (Table 2.1), the transformation using $D = D_{opt}(2)$ is still more efficient for Platform 2. Notice that the dictionary $D = D_{opt}(2)$ has a smaller number of columns ($L$) that (as will be explained) results in a lower communication cost.

Table 2.1: Matrices $D_{opt}(1)$ and $D_{opt}(2)$ denote the optimal dictionaries for transforming a dataset, if analysis on this data is to be done on Platforms 1 and 2 receptively. The two transformed datasets are tested on Platforms 1 and 2. The platform-optimized transformation results in meaningful improvement in runtime.

<table>
<thead>
<tr>
<th></th>
<th>$D_{opt}(1)$</th>
<th>$D_{opt}(2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>4k</td>
<td>1k</td>
</tr>
<tr>
<td>NNZ($C$)</td>
<td>38675164</td>
<td>72137690</td>
</tr>
<tr>
<td>Runtime (Platform 1)</td>
<td><strong>174.906ms</strong></td>
<td><strong>260.497ms</strong></td>
</tr>
<tr>
<td>Runtime (Platform 2)</td>
<td>25.5396ms</td>
<td>16.8901ms</td>
</tr>
</tbody>
</table>

### 2.2 Target applications

To ground our framework in real-world problems, we now discuss two particular learning algorithms that are evaluated in this paper: (i) sparse approximation and
(ii) the power method for eigenvalue decomposition.

(i) Sparse approximation for image denoising, and classification. Sparse representation is used in a wide range of signal processing and machine learning applications, including denoising [8] and more recently in classification [45], and super-resolution application. For example, in an image denoising application, \( A \) is an image dictionary, \( y \) is a noisy image and \( Ax \) is a denoised approximation of \( y \). In a classification application, \( A \) is a labeled set and the sparse coefficient vector \( x \) is used to determine which class a test signal \( y \) belongs to. This is done by first measuring the sum of the coefficients in each class and then finding the class that has maximal coefficient energy. The objective function for the \( \ell_1 \)-minimization can be written as follows:

\[
\arg \min_x \|Ax - y\|_2 + \lambda \|x\|_1,
\]

where the solution \( x \) is a sparse representation of \( y \) with respect to \( A \) and \( \lambda \) is a regularization coefficient (increasing \( \lambda \) results in sparser solutions). This sparse approximation problem can be solved using the following iterative algorithm:

\[
x^{iter+1} = f(x^{iter} - \eta(Gx^{iter} - A^T y)),
\]

where \( f(.) \) is a low-complexity thresholding operation (e.g., a soft-thresholding operator [10]) to account for the term \( \lambda \|x\|_1 \) at each iteration and \( \eta \) is the step size.

(ii) Power method for eigenvalue decomposition. The power method is a simple and iterative algorithm that can be used to sequentially find the eigenvectors and eigenvalues of a matrix in descending order. Recall that an eigenvector \( x \) of a matrix \( A \) satisfies the following relationship \( Ax = \lambda x \), where \( \lambda \) is the eigenvalue associated with the eigenvector \( x \). To find an eigenvector of the symmetric matrix
\( \mathbf{G} = \mathbf{A}^T \mathbf{A} \), the power method utilized the following iterative update:

\[
x^{iter+1} = \frac{\mathbf{Gx}^{iter}}{\| \mathbf{Gx}^{iter} \|_2}.
\] (2.4)

Once the power method converges to an estimate of an eigenvector \( \mathbf{x} \), the contribution of this eigenvector is removed from \( \mathbf{A} \), and the power method is applied again to the residual to find the next eigenvector.

In both algorithms described above, the main cost of each iteration is due to the computation of \( \mathbf{Gx} \), as \( \mathbf{G} \) is dense, massive, and distributed onto multiple computing nodes.

**Other applications.** Our framework can be used for a broad class of optimization problems that are solved via iterative updates based upon the Gram matrix. For instance, projected gradient descent (PGD) methods which provide a generalization of standard gradient descent methods for certain classes of non-smooth objective functions, can also benefit from our solution. A large number of machine learning objective functions and penalized regression methods such as the LASSO or BPDN [8], and Ridge regression [25] can be solved via a PGD approach. In all these settings, our solution can efficiently handle the costly iterative computations based on the distributed Gram matrix.

### 2.3 Target datasets

While several current techniques and frameworks for running iterative update algorithms on big data rely on data/correlation sparsity to increase efficiency [22], their performance on densely correlated data heavily degrades. This is because of the massive number of arithmetics and communications incurred by the large number of non-zeros of dense data. Many dense datasets, however, benefit from a less apparent
property that is known as coarse-grained data parallelism. In other words, dense datasets may belong to a single or a union of low-rank subspaces [78].

A single low-rank subspace is a valid compact model for a homogeneous data that lives on one affine hyperplane. Contemporary massive collections of data, however, are often heterogeneous or clustered. Characterizing such heterogeneous data by a single (global) subspace model is insufficient as the local structure of the clustered subspaces would be masked by the global model. A union of low-rank subspaces is a valid approximation of several ensembles of data, including motion trajectory of point correspondence [86], collections of photos recorded from objects at various illumination conditions [88], structured sparse and block sparse signals [78], and electrical signals measured from the brain's motor cortex [84].

In this work we show that several important classes of visual datasets, including video frames, light field data, hyperspectral, and cancer cell images demonstrate union of low-rank property. We show that how one can exploit this property to form versatile projections of dense data.
Chapter 3

A versatile and tunable data transformation

3.1 Overview of Extensible Dictionary transformation (ExD)

In this chapter we present our tunable sparsity driven data transformation method and discuss its characteristics in detail. The goal is to transform a dense data matrix $A_{M \times N}$ into a product of a dictionary matrix $D_{M \times L}$ and a sparse coefficient matrix $C_{L \times N}$ such that the objective function in Equation 2.1 holds. Figure 3.1 presents a depiction of our transformation.

3.2 Related work

High-dimensional data often can be well modeled by the low rank structures that are present in the data. Extracting low dimensional structures not only reduces

Figure 3.1: The dense data matrix $A$ is transformed into a product of a dictionary matrix $D$ and a sparse coefficient matrix $C$. 
dimensionality, but also mitigates the effect of high-dimensional noise and improves the performance of learning and inference tasks [79].

**Singular value decomposition (SVD).** In settings where the column span of $A$ admits a low rank model, the SVD provides a powerful tool for forming low rank approximations. The best rank-$k$ approximation of $A$ is given by $A_k = U_k \Sigma_k V_k^T$, where $U_k \in \mathbb{R}^{m \times k}$ and $V_k \in \mathbb{R}^{n \times k}$ are the truncated singular vectors (first $k$ columns of $U$ and $V$) and $\Sigma_k \in \mathbb{R}^{k \times k}$ contains the first $k$ singular values of $A$ along its diagonal. The rank equals the number of non-zero singular values or $\text{rank}(A)$. The truncated SVD also provides the solution to principal components analysis (PCA), which seeks to find a $k$-dimensional subspace that best approximates $A$ in the least-squares sense.

The classic SVD and PCA are known as powerful sketching methods for forming low dimensional matrix approximations. However, computing SVD (or PCA) incurs $O(M^2N)$ complexity for factorizing an $M \times N$ matrix. This cost is prohibitive for big data. To address this limitation, a number of randomized algorithms for computing an “approximate SVD” [90] or an approximate matrix decomposition by selecting a random subset of columns in $A$ have been suggested [77]. Such CSS methodologies have been utilized for providing scalable and efficient (approximate) solutions to Gram matrix decomposition, least-squares regression and also spectral clustering [77, 82]. Recently, streaming data sketching methods have also been proposed [29]. This work also relies on running SVD on the sketch matrix which does not scale to larger sketch sizes.

**Sparse transformation.** The objective in Equation 2.1 has been already used in a host of data mining and machine learning applications such as dictionary learning and feature selection applications. Notable examples include K-SVD and Sparse PCA
However, those methods are computationally infeasible for large $N$ values as they incur $O(M^2N)$ complexity. Recent work have proposed scalable approaches that use column-subset-selection in solving Equation 2.1 [78, 79]. However, the goal in those approaches is to classify signals in $A$ by reducing the dimensionality of the search space (from dense $A$ to sparse $C$). To the best of our knowledge, our work is the first that exploits the degree of freedom in tuning the dictionary in order to improve system performance for subsequent distributed computations on the dense data.

**Column subset selection (CSS)-based matrix factorization** In contrast to methods such as PCA and its sparse variants, an alternative strategy for low rank matrix factorization is to form a decomposition based upon columns (or row) from the data. CSS-based solutions form an approximate matrix decomposition in which one factorized component is a subset of the columns of the data itself [77]. CSS-based approaches have been used to provide a scalable and efficient strategy for finding approximate solutions to least-squares regression [17], Gram matrix decomposition [77], and also in spectral clustering [82]. In all of the aforementioned settings, CSS is employed to select a few columns from $A$ and a low rank decomposition is obtained by projecting the data onto the subspace spanned by this set of columns. Our proposed data transformation approach leverages a CSS-based to form the left factor matrix $D$.

Note that the key property that distinguishes our ExD from existing data transformation and sketching methods is that it is customizable to the pertinent distributed computing platform. To the best of our knowledge, our framework is the first to proposes a transformation that can be tuned for platform-specific customization.
3.3 ExD algorithm

Here we present our customizable data transformation methodology. Our approach first selects a subset of columns from the dataset $A$ to form a dictionary matrix $D$. It then uses this subset of columns as a basis from which sparse representation of the remaining columns in the dataset can be created.

If we denote the transformation by $A = DC$, each column of $C$ is computed by finding the sparse approximation of the corresponding column of $A$ with respect to $D$. This sparse approximation problem can be solved by an efficient greedy routine called orthogonal matching pursuit (OMP) [42].

Our key idea is that we can leverage the degree of freedom in forming $D$ for tuning the transformation. By increasing the number of subsampled columns in $D$ (i.e., increasing $L$), we can vary the sparsity level of $C$. The sparsity is achieved due to the coarse-grained redundancy in the columns of the dictionary. We extensively use this property to tune ExD in order to optimize the performance in a distributed setting.

3.3.1 Creating the extensible dictionary

In order to ensure that the total approximation error in our transformation is sufficiently small (i.e., $\leq \epsilon$), we must ensure that the columns selected from $A$ to form $D$ well approximate the range of the original matrix. We present two methods for subsampling columns of $A$ in order to create $D$. The first method is based on a random selection and the second is based on an adaptive error-based selection.

Random selection In this approach, we randomly select columns of $A$ to create $D$. In this case, it is critical that we select enough columns to ensure that we meet the
transformation error criteria. Note that when \( L > M \), with a high probability, matrix \( \mathbf{D} \) becomes full rank and thus the OMP algorithm converges (meets the error criteria). Theoretical work in the domain of subspace sampling has attempted to find bounds for \( L \) with respect to the intrinsic rank of a matrix. Recent work by Mahoney et al. [20] proves that by sampling \( L \leq \Omega \left( \frac{k \log k}{(1-\delta)^2} \right) \) columns at random, a maximum (least-squares) error equal to \( \frac{1}{\delta} \) of the nuclear norm of the best \( K \)-dimensional approximation of the data (i.e., \( K \)-dimensional truncated SVD of \( \mathbf{A} \)) is guaranteed. In a union of subspace model \( K \leq \sum_{i=1}^{N_s} K_i \).

Unlike existing decomposition approaches based on subspace sampling that create matrix \( \mathbf{C} \) by projecting the data using \( \mathbf{C} = \mathbf{D}^+ \mathbf{A} \), our approach focuses on creating sparse representations in \( \mathbf{C} \). Our key observation is that by increasing the redundancy in \( \mathbf{D} \) (i.e., increasing \( L \)), one can vary the sparsity level of \( \mathbf{C} \). We extensively use this property to tune ExD in order to optimize the performance in a distributed setting.

**Adaptive selection**  For cases where the input \( L \) in Equation 2.1 is less than \( M \), a more effective approach to create \( \mathbf{D} \) is by selecting the columns adaptively as follows. First, we compute the projection error \( \mathbf{E} = \mathbf{A} - \mathbf{A}_S \mathbf{A}_S^+ \) based upon the current set of selected columns given by \( \mathbf{A}_S \). Second, we select a new batch of columns according to the following probability distribution:

\[
p(i) \propto \frac{\| \mathbf{A}_S \mathbf{A}_S^+ \mathbf{a}_i - \mathbf{a}_i \|_2}{\| \mathbf{a}_i \|_2},
\]

where \( p(i) \) equals the probability of selecting the \( i \)th column from \( \mathbf{A} \), given by \( \mathbf{a}_i \).

This subsampling approach can be used by either specifying the maximum number of columns to select and/or specifying the maximum amount of error in each column of \( \mathbf{A} \).

\*The pseudo-inverse is calculated as: \( \mathbf{D}^+ = (\mathbf{D}^T \mathbf{D})^{-1} \mathbf{D}^T \).
When the data is approximately low rank, there exists a large body of work that characterizes the performance of the adaptive column selection method used to form $D$. In particular, the work in [15] demonstrates an exponential decrease in the least-squares estimation error if adaptive sampling is used. More specifically, assume that at each iteration, we select a batch of $L_s > \frac{r}{\delta}$ samples from the columns of $A$ and let $L = tL_s$ denote the set of columns selected after $t$ iterations. Let $A_r$ denote the best rank $r$ approximation to $A$ and let $\tilde{A} = A_S A_S^T A$ denote the approximation of $A$ based upon the $L$ selected columns. Then according to [15], the difference between the expected value of the approximation error, i.e., $\|A - \tilde{A}\|_F^2$ and that of the best rank $r$ approximation $\|A - A_r\|_F^2$ decreases exponentially with rate $\delta^t$.

### 3.3.2 Forming sparse representations

After selecting a subset of $L$ columns $A_S \in \mathbb{R}^{M \times L}$, we normalize each column to produce a matrix with unit norm columns $D$. Now, to form the sparse matrix $C$, we find a sparse representation of the remaining columns in $A$ (i.e., $A_{-S}$) in terms of the normalized dictionary $D$. The problem is formally written as follows:

$$\forall \ a_i \in A_{-S} \ \text{solve:} \ \min_c \ \|c\|_0 \ \text{s.t.} \ \frac{\|a_i - Dc\|_2^2}{\|a\|_2} \leq \epsilon. \ (3.2)$$

We employ a matching pursuit-based solver called Batch OMP [42] to solve Equation 3.2. The solver allows us to enforce sparsity either by fixing the number of non-zeros in each column of $C$ (i.e., $\|c\|_0$) or by fixing the total amount of approximation error for each column (i.e., $\epsilon$).

In Algorithm 1, we outline our scalable method for transforming $A$. We demonstrate how we can scalably run the transformation on $N_P$ processor. We first create matrix $D$ which is a dictionary matrix formed by sub-sampling columns of $A$. Once
Algorithm 1: ExD Algorithm

**Input:** Normalized data matrix $A \in \mathbb{R}^{M \times N}$, error tolerance $\epsilon$, number of processors $N_P$, and number of columns to select $L$.

**Output:** A sparse dictionary matrix $C \in \mathbb{R}^{L \times N}$ and a dictionary matrix $D \in \mathbb{R}^{M \times L}$ such that $\| A - DC \|_F \leq \epsilon \| A \|_F$.

1. $D$ is created via either random or adaptive sampling.
2. $p_{id} = i$ loads $A_i = A(:, \frac{iN}{N_P} : \frac{(i+1)N}{N_P})$ and a copy of $D$.
3. $p_{id} = i$ applies OMP to solve $a_i = Dc_i$ for the tolerance error $\epsilon$:
   
   3.0. Initialize $r = a_i, \phi = \emptyset$

   while $\| r \|_2 < \epsilon \| a_i \|_2$ do
      
      3.1. $k = \text{argmax}_j |d_j.r|$  
      3.2. $\phi = (\phi, k)$  
      3.3. $y = D_\phi^+a_i$  
      3.4. $r = a_i - D_\phi y$

   end while

$D$ is created, Equation 2.1 becomes equivalent to a generic sparse approximation problem. Each column $c_i$ of $C$ is a sparse approximation of the column $a_i$ of $A$ with respect to $D$ and the user-specified approximation error $\epsilon$. Once $D$ is created, each column $c_i$ can be independently computed using the parallel processing nodes. When the data is sufficiently sampled such that the span of columns of $D$ is “close” to the span of columns of $A$ (i.e., $\| a_i - DD^+a_i \| \leq \epsilon$), OMP can successfully find $C$ such that the error tolerance criteria is met. Setting the error tolerance to zero ($\epsilon = 0$) guarantees achieving the same approximation error as least-squares approaches.

*Processor ID’s are denoted by $p_{id}$. This notation means processor with $p_{id} = i$ ($0 \leq i < N_P$) is
3.3.3 Complexity analysis

The low-overhead and scalability of ExD is crucial for its applicability to large-scale datasets. In Algorithm 1, the main computing task is executing the OMP sparse coding routine. In our implementations, we use Batch-OMP based on Cholesky factorization updates [42]. The upper bound on the complexity is $O(LMN + L^2NNZ(C))$, where $NNZ(C)$ is the number of non-zeros in $C$. As we show in our experiments for many datasets $NNZ(C) \ll LN$ can be achieved. Since each column of $C$ is computed independently, this algorithm is highly parallel. Let $N_P$ be the number of parallel processing cores. By replicating $D$ and a fraction of columns of $A$ in each node (i.e., $\frac{N}{N_P}$ columns), the complexity of Step 3 would reduce to $O(\frac{N}{N_P}(LM + L^2\frac{NNZ(C)}{N}))$.

3.3.4 Sparsity guarantees for $C$

Our approach is motivated by recent results for sparse subspace clustering [79], where the goal is to discover multiple low-dimensional subspaces present in a collection of data and then cluster signals according to their subspace membership. The key intuition is that a sparse representation of a data signal (column of $A$) ideally corresponds to a combination of a few other signals of $A$.

In particular, when columns of $A$ lie on a union of subspaces, if enough columns are selected that represent a particular subspace (cluster), then the representations of the remaining columns in $A$ are guaranteed to be sparse [78]. Formally, we say that a set of $N$ columns of $A = a_1, \ldots, a_N$ each of dimension $M$, live on a union of $N_s$ subspaces if $a_1, \ldots, a_N \subset \bigcup_{i=1}^{N_s} U_i$, where $U_i$ is a $K_i$-dimensional subspace of $\mathbb{R}^N$. In this case those columns of $C$ that lie on a $K_i$-dimensional subspace will admit a
$K_i$-sparse representation, i.e., at most $K_i$ non-zeros will be used to represent a signal within this subspace. This data modeling is quite general because it can be used to describe data living on a single subspace (low rank model) and/or multiple subspaces that might be corrupted with a few outlier columns [78].

In the following, we provide an example that shows how ExD benefits from the union-of-subspace properties of the data for sparsification. Figure 3.2 shows a dataset in $\mathbb{R}^2$ space. Each point is represented by two non-zero elements in the direction of $X$ and $Y$ axis. This dataset is full-rank, i.e., an SVD factorization finds two orthogonal basis to represent the data. Figure 3.2 shows the orthonormal basis that is found via SVD. However, one can observe that by choosing a more suitable basis for the data, i.e. the two vectors $a$ and $b$ on the figure, we can approximately represent each data point using only one non-zero element in the direction of either $a$ or $b$. Thus, while the 2D dataset is not inherently low-rank (as its rank is 2), it lies on a union of two lower dimensional (rank-1) subspaces. Our approach uses the union of lower-rank spaces by forming the dictionary columns, a.k.a., dictionary atoms from the sample data points, i.e., $a$ and $b$ in our example. When the data is much higher dimensional than our 2D example, there is a higher degree of freedom in the way atoms in $D$ can be selected.

3.3.5 Computational benefits of ExD-based transformation

ExD is a preprocessing step that results in computational benefits when the size of the dictionary is small (i.e., $L$ is small relative to $M$) and/or when matrix $C$ is sparse. In general, predicting the amount of savings in computation is a function of (i) the geometry of the data that results as discussed in Section 3.3.4, and (ii) the amount of accuracy required from the learning algorithm.
Figure 3.2: A union of two 1-dimensional signal models. By selecting vectors $a$ and $b$ as the atoms of the dictionary, each data point can be approximated using only one atom.

**Impact of increasing the transformation error $\epsilon$.** The transformation error of ExD is controlled by the parameter $\epsilon$ in Algorithm 1. In the case where we set $\epsilon = 0$, then we are guaranteed an exact data transformation. ExD can produce an exact and compact transformation when the data is exactly low rank. In practice, however, datasets are approximately low rank. In this case, we can introduce a small amount of error into the transformation by setting $\epsilon > 0$. By introducing some error into the transformation, we observe that both the number of selected columns in $D$ and the sparsity level of $C$ can be reduced further. In our evaluations we show how increasing the transformation error produces a more compact result.

### 3.3.6 Impact of transformation error on learning accuracy

In the previous section, we discussed the computational benefits associated with introducing some approximation error into the transformation. Naturally, as we increase this error (controlled by $\epsilon$) the learning error also increases. Thus, the key question is how much transformation error we can afford to achieve a certain degree of accuracy in learning. The answer to this question heavily depends on the specific learning
algorithm and the application of interest.

Previous theoretical work have established a connection between the total error in a factorization of a kernel (or Gram) matrix ($\epsilon$) and the accuracy of certain popular learning algorithms, including: kernel ridge regression and kernel SVM [9]. While for some learning algorithms, our framework can exploit the existing work in the literature to relate $\epsilon$ and the learning error which we denote by $\delta_L$, the aim of this section is to propose a generic approach for tuning the transformation error to achieve a specified learning accuracy. In the following we introduce an approach for iteratively remapping of the data to find a compact transformation that satisfies a pre-specified amount of learning error.

### 3.3.7 Error tuning

Given an already established relationship between the transformation error and a specific algorithm, a practitioner who uses our framework can easily specify the error parameter $\epsilon$ to achieve a particular learning accuracy. Alternatively, if a practitioner specifies a target accuracy for a learning algorithm, the transformation error $\epsilon$ can be tuned in order to achieve a particular learning error $\delta_L$.

Our strategy for guaranteeing that we have small $\delta_L$, is to solve the transformation for a particular $\epsilon$ and then approximate the accuracy of a learning algorithm $\delta_L$. Once $\epsilon$ is small enough then we can ensure that $\delta_L$ is small as well. Depending on the underlying computing resources available, ExD can be applied for multiple values of $\epsilon$ in parallel and the largest value of $\epsilon$ (most compact representation) that achieves a particular value of $\delta_L$ is selected.

When computing resources are constrained and thus running the algorithm for multiple values of $\epsilon$ in parallel is not possible, we use the bisection method. In
essence, the idea is to: (i) set the transformation error to predefined maximum value \( \epsilon^{\text{max}} \) (0.4 in our experiments) and evaluate \( \delta_L \), (ii) if \( \delta_L \) is below a target value then we stop, otherwise we decrease \( \epsilon \) by half. By exponentially decreasing \( \epsilon \), we are also guaranteed to decrease \( \delta_L \) exponentially, provided that there is a polynomial relationship between the two variables. We observe a polynomial relationship holds both in theory [9] and in practice. In Chapter 6, we supply empirical results which demonstrate the connection between the transformation error and learning accuracy for numerous datasets and algorithms of interest.
Chapter 4

Distributed mapping and platform-aware transformation tuning

4.1 Overview of platform-aware mapping and customization

In this chapter we provide our solutions for adaptively tuning ExD with respect to the properties of the underlying distributed platform. First, we introduce our approach for applying iterative updates on the transformed data in a distributed fashion, i.e., the execution phase shown as in Figure 2.1. We describe an execution flow for updates on the correlation matrix, i.e., $Gx = C^T(D^TD)Cx$, and introduce an efficient method for data partitioning. To achieve our end goal, which is to adaptively minimize the computing cost, we provide our performance quantification approach that measures the cost in terms of memory usage, number of flop operations, and number of communicated bytes across the computing nodes. Finally, we introduce our tuning techniques to customize the transformation in order to provably optimize the quantified cost.

4.2 Related work

A large body of work has focused on developing methodologies for efficient mapping of various linear algebra and analysis algorithms onto distributed, heterogeneous, or reconfigurable architectures [39, 37, 38, 43, 13, 27]. While such methods effectively optimize the performance with hardware-aware customization, they do not holistically
customize the representation of data (by revealing its subspace structure) for the pertinent platform. They instead adhere to data/correlation structure given by the application, which is ubiquitous regardless of the hardware platform.

A number of successful distributed abstractions for processing large datasets on clusters have been proposed. MapReduce is a widely popular programming model for processing large data sets with a distributed algorithm on a cluster [11]. A MapReduce program is composed of a Map step that performs filtering and sorting (such as counting the number of times different words have occurred in different pages of a website) and a lower complexity Reduce step that gives a summary operation (such as combining the partial results from the map step to find the overall count of the words). Despite its popularity for sorting and counting applications, it has been shown that abstracting and parallelizing iterative matrix computations using MapReduce does not yield a high performance [30]. In particular, the performance of the parallel computation can be severely degraded by the dense matrix dependencies.

Several new distributed abstractions have been proposed that model data dependency in a graph format, most notably Pregel [31] and GraphLab [30]. They use a vertex-centric computation model, in which the user-defined programs are executed on each vertex in parallel. However, the present distributed methodologies that focus on parallelizing learning computations and dependency graphs are built upon the sparse structure of the data dependency graph or Gram matrix [31], and GraphLab [30]. These methods achieve performance gain due to the low number of neighbors of the vertices; their performance rapidly degrades for higher vertex connectivity and dense graphs.

Apache Spark is another specialized distributed abstraction for iterative algorithms on large data [94] which supports both matrix and graph-based implemen-
tations. Spark facilitates distributed computing by hiding the complexities of the platform from the users.

In this work, we devise three implementations for distributed execution of the iterative updates on the transformed data. The first one is a customized fully hand-coded implementation that takes advantage of the standard Message Passing Interface (MPI) for inter-node communication. The other two approaches realize our developed distributed computing model on GraphLab and Spark abstraction. Note that our sparsity-inducing transformation, for the first time, enables graph-parallel abstractions such as GraphLab to become applicable to dense datasets. All our implementations are publicly available at [2].

4.3 Distributed execution flow and data partitioning

In this section, we first propose an efficient data partitioning and distributed computing model to perform iterative analysis on the projected data, i.e., \( C^T D^T D C x = A^T A x \). We next quantify the performance metrics based on the proposed distributed model.

Algorithm 2 outlines our proposed distributed computing model. Depending on whether \( L > M \) (Case 0) or \( L < M \) (Case 1), we propose two different approaches. In Case 0, we replicate matrix \( D \) in all the processors to reduce communication. However, doing so requires all the processors to do the redundant multiplication, i.e, \( D^T v^2 \) in Step 7. In Case 1 however, the computation corresponding to \( D^T v^2 \) is done only by processor 0. As discussed in Section 3.3, most contemporary datasets are in the regime that \( M, L \ll N \). Thus, \( D \) is a relatively small matrix that can easily fit into the memory of processor 0. Execution phase in Figure 2.1 shows Case 1 where \( L < M \) and \( D \) is store only in processor 0.
Algorithm 2: Distributing Gram Matrix Multiplication on DC

**Input:** Vector \( x_{N \times 1} \) and transformed data matrices \( D_{M \times L} \) and \( C_{L \times N} \).

**Output:** \( C^T D^T D C x \).

0.0 \( pid = i \) loads \( C_i = C(\cdot, iN_{NP} : (i+1)N_{NP}) \).

0.1 \( pid = i \) loads \( x_i = x(iN_{NP} : (i+1)N_{NP}) \).

1. \( pid = i \) computes \( v^1_i = C_i x_i \); \( v^1_i \) is an \( L \times 1 \) vector.

**Case 0: \( L > M \)**

2. \( pid = i \) loads \( D \).

3. \( pid = i \) computes \( v^2_i = D v^1_i \); \( v^2_i \) is an \( M \times 1 \) vector.

4. Vectors \( v^2_i \) will be reduced in \( pid = 0 \).

5. \( pid = 0 \) computes \( v^2 = \sum v^2_i \); \( v^2 = D C x \) is an \( M \times 1 \) vector.

6. \( pid = 0 \) broadcasts \( v^2 \) to all other processors.

7. \( pid = i \) computes \( C^T_i (D^T v^2) \).

**Case 1: \( L \leq M \)**

2. \( pid = 0 \) loads \( D \).

3. Vectors \( v^1_i \) will be reduced in \( pid = 0 \).

4. \( pid = 0 \) computes \( v^2 = D (\sum v^1_i) \); \( v^2 = D C x \) is an \( M \times 1 \) vector.

5. \( pid = 0 \) computes \( v^3 = D^T v^2 \); \( v^3 = D^T D C x \) is an \( L \times 1 \) vector.

6. \( pid = 0 \) broadcasts \( v^3 \) to all other processors.

7. \( pid = i \) computes \( C^T_i v^3 \).

4.3.1 Performance quantification

**Bounds on arithmetics.** The cost of arithmetics is directly dependent on the number of floating point operations for doing \( C^T D^T D C x \). The computations involving \( C \)
should be done in an efficient way to exploit its sparsity. This reduces the computations in Steps 1 and 7 (for both cases) to the number of non-zeros in \( C \) or \( \text{NNZ}(C) \).

The number of floating point operations (in serial) is \( 2(ML + \frac{\text{NNZ}(C)}{N_P}) \) multiplications and \( 2MN_P \) additions. Here, the cost of additions is negligible because in many cases we have \( N_P \ll L \).

**Bounds on communication.** The communication overhead of Algorithm 2 stems from the *reduce* and *broadcast* activities. In Case 0, Step 4, each processor sends a message containing \( M \) words to Processor 0, and in Step 6, Processor 0 sends a message containing \( M \) words back to other processors. In a similar fashion in Case 1, at Steps 3 and 5, \( L \) words are communicated. **The total number of words that are communicated simultaneously is** \( 2 \times \min(L,M) \).

We exploit the extensive work in applied numerical linear algebra to show that our computational model achieves the optimal communication. More exactly, Demmel et al.’s work on communication-optimal parallel recursive rectangular matrix multiplication directly applies to our target problem [14]. In that work it is shown that for multiplying \( Z = XY \) where dimensions of matrices \( X, Y, \) and \( Z \) belong to \( \{d_1, d_2, d_3\} \) such that \( (d_1 \leq d_2 \leq d_3) \), if \( 2\frac{d_3}{d_2} > N_P \) (which is the case in our framework when \( d_3 = N \)), the communication lower bound (number of communicated words) is \( \Omega(d_1d_2) \). Substituting the dimensions by those of matrices \( D, C, \) and \( x \) we get \( d_1 = 1 \) and \( d_2 = \min(M,L) \) which brings the number of transferred words to \( \min(M,L) \). Thus, our communication achieves the optimal (minimum) bound.
Runtime performance

The overall execution runtime is directly affected by the arithmetic and communication costs and is approximately proportional to:

$$ML + \frac{NNZ(C)}{N_P} + \min(M, L)N_PR_{time}^{b2f},$$

where $R_{time}^{b2f}$ is the word-per-FLOPs ratio that characterizes the memory bandwidth per unit of time performance.

The first two terms reflect the computational operations and the third term reflects the adjusted communication overhead. During the message passing phases in Algorithm 2, all processors are locked and no computation is done. Although a number of other factors affect the runtime such as memory hierarchy, cache size, size of shared memory (for the cores within a computing node), and geometry of the distributed nodes, in our experiments, we show that our approximation provides a reasonable estimation of the actual runtime.

Energy performance

Similarly, the overall execution energy is approximately proportional to:

$$ML + NNZ(C) + \min(M, L)N_P R_{energy}^{b2f},$$

where $R_{energy}^{b2f}$ is the word-per-FLOPs ratio that characterizes the memory bandwidth per unit of energy performance. The first two terms reflect the computational operations and the third term reflects the adjusted communication overhead.

Memory performance

The sparsifying effect of ExD transformation results in significant reductions in memory footprint. Memory usage decreases due to replacing the original dense matrix $A$
with the relatively lower dimensional dictionary matrix $D$ and the sparse coefficient matrix $C$. In both proposed implementations (Algorithm 2) the memory footprint per processing node is bounded by:

$$ML + \frac{\text{NNZ}(C) + N}{N_P}.$$  \hspace{1cm} (4.3)

### 4.4 Automated Customization of ExD

In this section, we present our approach for optimizing the performance of the iterative update algorithms. More exactly, we discuss how the extensible dictionaries proposed in Section 3.3 can be tuned to minimize the performance costs quantified in Section 4.3.

We tune ExD by finding an optimized $L$ as an input for Algorithm 1. The input must be such that the resulting $(L, \text{NNZ}(C))$ pair minimizes the performance costs quantified in Equations 4.1, 4.2, or 4.3. In the following, we propose a novel scalable method to tune ExD. Our method estimates $\text{NNZ}(C)$ as a function of $L$ with preprocessing only portions of the original massive data matrix $A$.

**Estimating $\text{NNZ}(C)$**. Here we provide an example dataset to show the relationship between $L$ and $\text{NNZ}(C)$. Figure 4.1 shows the normalized transformation error ($\|A - DC\|_F/\|A\|_F$) as a function of $L$. The figure also shows the average per-column number of non-zeros in $C$ as a function of $L$. We denote this function as:

$$\alpha(L, A, \epsilon) = \frac{\text{NNZ}(C)}{N}.$$  \hspace{1cm} (4.4)

The dataset is a collection of hyperspectral signals from [1] with $M = 203$, $N = 54129$. The search space for parameter $L$ is limited to $L \geq L_{\text{min}}$, where $L_{\text{min}}$ is the minimum number of columns that should be sampled such that the transformation error criteria is met ($\|A - DC\|_F < \epsilon\|A\|_F$). In this example $L_{\text{min}} \approx 175$. 

Figure 4.1: Average number of non-zeros in each column of \( C \), i.e., \( \alpha(L) \) as a function of the number of columns in the dictionary, i.e., \( L \). The bars show the variance for 100 trials of random sub-sampling to form \( D \). The average transformation error \( \|A - DC\|_F / \|A\|_F \) as a function of \( L \).

It can be seen that function \( \alpha(L, A, \varepsilon) \) is decreasing for \( L > L_{\text{min}} \). This is due to the fact that a larger \( L \) would result in a larger ensemble of signals in \( D \). This enables representing columns of \( A \) as a linear combination of fewer signals from \( D \). In an extreme case when \( L = N \) (or \( D = A \)), then the \( i^{th} \) column of \( A \) \( (a_i) \) can be transformed as \( a_i = De_i \), where \( e_i \) is a unit vector whose \( i^{th} \) entry is 1 and all of its other entries are 0. In this case \( \alpha(N, A, \varepsilon) = 1 \), however, the cost in Equation 4.1 becomes too large due to the large \( L \). When \( L < L_{\text{min}} \), the OMP method cannot reconstruct some of the \( a_i \) columns even if all the columns in \( D \) are selected in the reference set \( \phi \) (see Algorithm 1).

The bars on the graph shows the variation in \( \alpha(L, A, \varepsilon) \) for 100 different initial ensemble collection for \( D \). It can be seen that the variations is very negligible for a fixed \( L \) (dispersion index is less than 4% for this example).

**Estimating \( \alpha(L, A, \varepsilon) \) from smaller subsets of \( A \).** In order to optimize the performance cost (e.g., Equation 4.1), function \( \alpha(L, A, \varepsilon) \) needs to be characterized.
While a Brute Force approach (i.e., running Algorithm 1 for various $L$ values) would provide us with the best $L$, it is not an efficient nor a feasible (for massive $N$) solution. To provide an effective way to find $\alpha(L, A, \epsilon)$, we rely on the following two important observations.

First, let $A$ be a data with a union of subspace signal model. Let $A_s$ be a random subset of $A$ such that $|A_s|^* = n$, then for $n \to N$: $E[\alpha(L, A_s, \epsilon)] = E[\alpha(L, A, \epsilon)]$.

Second, recalling the formal definition of union-of-subspace data, we assume columns of $A$ are a collection of signals from $N_s$ subspaces where each subspace $U_i$ is $K_i$-dimensional (for $1 \leq i \leq N_s$). In this case if $n_i$ columns of $A$ lie on subspace $U_i$, then the coefficient matrix corresponding to those $n_i$ columns have at most $K_i n_i$ non-zeros [78, 79]. Based on our definition of the density measure, we have $\alpha(L, A, \epsilon) \leq \sum_{1 \leq i \leq N_s} K_i \frac{n_i}{N}$. Let us create $A_s$ by selecting $n$ columns at random from $A$. The expected number of columns in $A_s$ that belong to subspace $U_i$ is $\frac{n}{N} n_i$. If we apply Algorithm 1 to $A_s$, we get $A_S = D_s C_s$. Thus, the following expected upper bound is achieved for $\alpha(L, A_s, \epsilon) \leq \sum K_i \frac{n_i}{N}$ which is the same as the bound on $\alpha(L, A, \epsilon)$.

The above observations indicate that finding $\alpha(L, A_s, \epsilon)$ from a subset of $A$ yields a close estimation of $\alpha(L, A, \epsilon)$. One can run ExD for random subsets of $A$ denoted by $A_1, A_2, \ldots$, such that $|A_1| < |A_2| < \cdots < |A|$ until the discrepancy in $\alpha(L, A_i, \epsilon)$ reduces to a pre-specified threshold. In Section 6.1.5 we provide more in-depth analysis that verify the above observations.

* | · | is the cardinality or number of columns.
Figure 4.2: Distributed graph-parallel mapping and scheduling of iterative updates on the transformed data.

4.5 Adaptation to graph-parallel abstractions

In this section, we describe the adaptation of our distributed model to graph-parallel abstractions. In this model, as opposed to the previously discussed matrix-based approach, data is presented in graph format. All partitioning and distributed computations are customized for the data graph. A graph-based model can result in better partitioning (lower communication) and thus faster execution time in comparison to the matrix-based model. However, the performance gain is highly dependent on the transformed data. As we discuss in the next chapter, the sparsity and structure of non-zeros in $C$ is an important factor in the performance of our graph-parallel adaptation.

Figure 4.2 shows a schematic of the graph model. The decomposed data graph denoted by $G_A(S_X, S_P, S_R)$ is a 3-layer graph with vertex sets $S_X = \{X_i\}_{i=1}^n$ in the bottom layer, $S_P = \{P_i\}_{i=1}^l$ in the middle layer, and $S_R = \{R_1\}$ in the top layer. Each non-zero element in $C$, e.g., $C_{ij}$, is represented by an edge which connects $X_i$ to $P_j$. Each column of $D$, e.g., $D_i$, is represented by an edge which connects $P_i$ to $R_1$. Value of vertices in $S_X$ correspond to the elements of $x$. 
We use GraphLab Distributed API [30] to implement this model. While GraphLab is a highly optimized distributed engine for Graph-based computation on iterative data, we perform extensive customizations in order to adapt GraphLab to our factorized setting. We also force GraphLab to use our developed graph partitioning method as opposed to its automated partitioning schemes. The proposed partitioning is customized for our factorized data, a property that we take advantage in order to significantly improve the graph partitioning.

### 4.5.1 Distributed graph partitioning

In the graph-based model, we partition $G_A(S_X, S_P, S_R)$ with the aim of balancing the number of components assigned to each node and also minimizing the internode communications characterized by the edges. Since the edge distribution of $G_A$ is highly non-uniform ($l \ll n$), a vertex partitioning inevitably results in many undesirable edge-cuts across the computing nodes. Instead, we apply a vertex-cut method in which the goal is to partition graph edges evenly such that the number of vertices that are spanned across multiple partitions is minimized. As a results of edge partitioning, in our implementation, vertices may reside onto two or more computing nodes. In this case we assign one of the copies to be the master vertex and the others to be the replica vertices (these definitions are borrowed from GraphLab [22]). The replicas directly cause (expensive) inter-node communication costs.

Fig. 4.2 shows the graph-based distributed design. Our detailed edge partitioning method is as follows. (i) Distribute master of vertices $X_i \in S_X$ uniformly onto the available computing nodes such that vertex chunks of size $\frac{n}{N_P}$ are assigned to each node. (ii) Add the edges between vertices $X_i \in S_X$ and $P_j \in S_P$ to the node in which the corresponding master of $X_i$ resides. (iii) Add master of vertices $P_i \in S_P$ and
\( R_1 \in S_R \) to a central node. and (iv) Add the edges between the vertices \( P_i \in S_P \) and \( R_1 \in S_R \) to the central node.

The proposed edge partitioning algorithm is highly efficient in that it does not induce any replicas for vertices in \( S_X \) and \( S_R \). However from Step (ii) of the algorithm, replicas of vertices in \( S_P \) may exist in computing nodes other than the central node. At the beginning of an iteration, master vertices in \( S_P \) and their replicas perform vertex updates with respect to \( S_X \). The replicas send the updated values to their own master vertices in the central node. The master vertices in \( S_P \) reduce the received values (\( p = Cx \)). Then master vertex \( R_1 \) performs a vertex update (\( r = Dp - y \)). Next master vertices in \( S_P \) complete vertex updates with respect to \( S_R \) and broadcast the results to their own replicas (\( p = D^T r \)). Finally master vertices in \( S_X \) update themselves (\( x = C^T p \)). We integrate and implement the proposed customized partitioning and distributed computation flow with the distributed GraphLab API [22].

### 4.5.2 Performance bounds

We now provide bounds on the memory usage, computation, and communication required by our proposed graph-based model.

- **Memory usage**
  
  \[
  \# \text{edges} \propto \text{nnz}(C) + L. \\
  \# \text{vertices} \propto n + \sum_{1 \leq i \leq l} \text{rep}(P_i).
  \]

- **Computation** (per iteration)
  
  \[
  \# \text{additions} \propto 2(\text{nnz}(C) + ML) + \sum_{1 \leq i \leq L} \text{rep}(P_i). \\
  \# \text{multiplications} \propto 2(\text{nnz}(C) + ML).
  \]

- **Communication**
  
  \[
  \# \text{edge-cuts} \propto 2 \sum_{1 \leq i \leq l} \text{rep}(P_i).
  \]
Each of the computing nodes receive approximately $\frac{1}{N_P} (n + \sum_{1 \leq i \leq l} \text{rep}(P_i))$ vertices and $\frac{1}{N_P} \text{nnz}(C)$ edges. The central node has $l$ additional edges between the master vertices in $S_P$ and $R_1$. The computation cost is induced by vertex update operations. The communication overhead is incurred by the message passing across master and replica vertices in $S_P$.

From above discussions, it is clear that reducing number of replicas of $S_P$ reduces the communication overhead. The following are the bounds on the total number of replicas: $L \leq \sum_{1 \leq i \leq L} \text{rep}(P_i) \leq LN_P$. The inequalities hold since each $P_i$ is replicated at least once and at most $N_P$ times (one replica per computing node). Both $L$ and $N_P$ are much smaller than the size of the graph. Thus, our graph-based model readily provides efficient/balanced computation and reduced communication without using complicated and costly graph partitioning algorithms. The minimum communication is achieved when $C$ is block-diagonal.
Chapter 5

An online data transformation for streaming applications

5.1 Overview of streaming data transformation (SSketch)

Several big data analysis applications demand real-time and online processing of streaming data, as data can be read at most once [29]. This disruption of convention requires new transformation, a.k.a. sketching techniques that are amenable to the limited single-pass access to data. In this chapter, we propose a new sketching technique for efficient data transformation in streaming applications. Our approach efficiently maps the stream of input data to a corresponding lower dimensional signal model. We refer to our streaming data sketching method by SSketch. Similar to ExD, SSketch algorithm transforms the original (densely correlated) large matrix into two new matrices: (i) a dictionary matrix which includes a set of atoms subsampled from the input data, and (ii) a sparse coefficient matrix.

To take advantage of FPGA’s, we provide a high throughput and resource-optimized hardware-accelerated implementation of SSketch. We provide methodologies for customizing the transformation approach based upon the user-defined requirements and the hardware limitations. More precisely, we develop an automated optimization approach that can be used to compute the best transformation (with the least approximation error) under the given memory, power, and runtime constraints. As the stream of input data arrives, we adaptively learn from the incoming samples
and update the dictionary matrix $D$ and the coefficient matrix $C$.

We utilize the abundant hardware resources on current FPGAs to provide a scalable, floating-point implementation of the sparsifying routine OMP. One may speculate that GPUs may show a better acceleration performance than FPGAs. However, the performance of GPU accelerators is limited in our application because of two main reasons. First, for streaming applications, the memory hierarchy in GPUs increases the overhead in communication and thus reduces the throughput of the whole system. Second, in our framework the number of required operations to compute the sketch of each individual sample depends on the input data structure and may vary from one sample to the other. Thus, the GPU’s applicability is reduced due to its Single Instruction Multiple Data (SIMD) architecture.

The global flow of SSKetch transformation is presented in Figure 5.1. The algorithm takes the stream of a massive, dynamic dataset in the matrix form as its input and computes/updates a sketch matrix of the collection as its output. The constraint-driven customization unit takes user-defined properties and hardware limitations as inputs and delivers output parameters for an optimized sketch computation. The input parameters may include runtime, power, and memory constraints and the outputs include sketching algorithmic parameters as well as guidelines for the hardware.

Figure 5.1: High level block diagram of SSKetch.
mapping. Our sketch formation algorithm is devised to minimize the costly message passings to/from the memory and cores, thereby it reduces the communication delay and energy. All computations are done in IEEE 754 single precision floating-point format.

SSketch consists of two main components: (i) a dictionary learning unit, and (ii) a data sketching unit. As the stream of data comes in, the first component adaptively learns a dictionary as a subsample of input data such that the hybrid structure of data is well captured within the learned dictionary. Next, the data sketching unit solves a sparse approximation problem using the OMP algorithm to compute a block-sparse matrix. In the data sketching unit, the representation of each new arrival sample is computed based on the current values of the dictionary, and the result is sent back to the host. We provide supporting API’s to facilitate automation and adaptation of our proposed transformation for rapid prototyping of an arbitrary matrix-based data analysis algorithm.

5.2 Related Work

5.2.1 Streaming Model

Modern data matrices are often extremely large which require distribution of computation beyond a single core. Some prominent examples of such massive datasets include image acquisition, medical signals, recommendation systems, wireless communication, and Internet user’s activities [60]. The dimensionality of modern data collections renders usage of traditional algorithms infeasible. Therefore, matrix decomposition algorithms should be designed to be scalable and pass-efficient. In pass-efficient algorithms, data is read at most a constant number of times. Streaming-based
algorithm refers to a pass-efficient computational model that requires only one pass through the dataset. By taking advantage of a streaming model, the sketch of a collection can be obtained online which makes storing the original matrix superfluous [61].

5.2.2 Hardware-accelerated OMP

OMP has been shown to be very effective in inducing sparsity, although its complexity makes it costly for streaming applications. A number of implementations on GPU [56, 63, 64], ASICs [57], and FPGAs [65, 58, 66] are reported in the literature to speed up this complex reconstruction algorithm. FPGA implementation of OMP for problems of dimension $32 \times 128$ and $256 \times 1024$ are developed for signals with sparseness of 5 and 36 respectively [65], [58]. To the best of our knowledge, none of the previous implementations of OMP is devised for streaming applications with large and densely correlated data matrices. In addition, use of fixed-point format to compute and store the results limits their applicability for sketching purposes.

5.3 SSketch Algorithm

Many modern massive datasets are either low-rank or lie on a union of lower dimensional subspaces. This convenient property can be leveraged to efficiently map the data to an ensemble of lower dimensional data structures [33]. Here we pursue a similar transformation objective as in Equation 2.1. Despite ExD which is a “static” transformation and requires access to the full datasets $A$, SSketch receives columns of $A$ in a streaming manner. Our platform-aware matrix sketching algorithm is summarized in Algorithm alg:dv. SSketch algorithm, approximates matrix $A$ as a product of two other matrices $D_{M \times L}$ and $C_{L \times N}$ based on a streaming model.
Algorithm 3 SSketch algorithm

Inputs: Data matrix $A$, projection threshold $\alpha$, sparsity level $k$, transformation error $\epsilon$, and dictionary size $L$.

Output: Dictionary matrix $D$, and coefficient matrix $C$.

1: $D \leftarrow \text{empty}$
2: $j \leftarrow 0$
3: for $i = 1, \ldots, N$ do
4: \hspace{1em} $W(a_i) = \frac{\|D(D'D)^{-1}D'r_i - l_i\|_2}{\|a_i\|_2}$
5: \hspace{1em} if $W(a_i) > \alpha$ and $j < L$ then
6: \hspace{2em} $D_j = a_i / \sqrt{\|a_i\|_2}$
7: \hspace{2em} $C_{ij} = \sqrt{\|a_i\|_2}$
8: \hspace{2em} $j \leftarrow j + 1$
9: \hspace{1em} else
10: \hspace{2em} $C_i \leftarrow OMP(D, a_i, k, \epsilon)$
11: \hspace{1em} end if
12: end for
For each newly arriving sample, our algorithm first calculates a projection error, \( W(a_i) \), based on the current values of the dictionary matrix \( D \). Next, it compares the calculated error with a user-defined projection threshold \( \alpha \) and updates the sketch accordingly. Dictionary matrix \( D \) is updated based on each arriving data sample. OMP routine is used to compute the sparse matrix \( C \). OMP can be used, either by fixing the number of non-zeros in each column of \( C \) (sparsity level \( k \)) or by fixing the total amount of approximation error (error threshold \( \epsilon \)) for each sample. In our experiments, we consider Frobenius norm error \( (Xerr = \frac{\|A-VD\|_F}{\|A\|_F}) \), as the sketch accuracy metric.

As can be seen, SSketch algorithm requires only one pass through each arriving sample. This method only requires storing a single column of the input matrix \( A \) and the matrix \( D \) at a time. Note that the dictionary matrix \( D_{M \times L} \) is constructed by columns of data matrix \( A_{M \times N} \). The column space of \( D \) is contained in the column space of \( A \). Thus, \( rank(DD^+A) = rank(D) \leq L \leq M \). It simply implies that for over-complete datasets OMP computation is required for \( N - L \) columns and the overhead time of copying \( D \) is ignorable due to its small size compared to \( A \).

**OMP with QR Decomposition.** As we describe in Section 6.5, computational complexity of the projection step (line 4 of Algorithm 3) is small compared to the \( O(MNL^2) \) complexity of the OMP algorithm. Thus, the computational bottleneck of SSketch algorithm is OMP. To boost the computational performance, it is necessary to modify the OMP algorithm such that it maximally benefits from the available resources and incurs a scalable computational complexity.

Algorithm 4 demonstrates the pseudocode of OMP where \( \epsilon \) is the error threshold, and \( k \) is the target sparsity level. The Least Squares (LS) minimization step (line 5 of Algorithm 4) involves a variety of operations with complex data flows that
introduce an extra hardware complexity. However, proper use of factorization techniques like QR decomposition or Cholesky method within the OMP algorithm would reduce its hardware implementation complexity and make it well-suited for hardware accelerators [58, 68].

Algorithm 4 OMP algorithm

Inputs: Matrix D, measurement a_i, sparsity level k, threshold error ε.

Output: Support set Λ and k-dimensional coefficient vector c.

1: r ← a_i
2: Λ^0 ← ∅
3: for i = 1,...,k do
4:   Λ ← Λ ∪ argmax_j | < r_{i-1}, d_j > | Find best fitting column
5:   c_i ← argmin_ε || r_{i-1} - D_{Λ^i}c ||^2 Least Squares Optimization
6:   r_i ← r_{i-1} - D_{Λ^i}c_i Residual Update
7: end for

Thus, to efficiently solve the LS optimization problem in line 5 of Algorithm 4, we decide to use QR decomposition (Algorithm 5). QR decomposition returns an orthogonal matrix Q and an upper-triangular matrix R. It iteratively updates the decomposition by reusing the Q and R matrices from the last OMP iteration. In this approach, the residual (line 6 of Algorithm euclid) can be updated by r^i ← r^{i-1}Q^i(Q^i)^t r^{i-1}. The final solution is calculated by performing back substitution to solve the inversion of the matrix R in c^k = R^{-1}Q^t a_i.

Assuming that matrix A is of size M × N and D is of size M × L, then the complexity of the OMPQR is \( O(MNL^2) \). This complexity is linear in terms of \( M \) and \( N \) since in many settings \( L \) is much smaller in compared to \( M \) and \( N \). The
Algorithm 5 Incremental QR decomposition by modified Gram-Schmidt

Inputs: New column $D_{A^s}$, last iteration $Q^{s-1}$, $R^{s-1}$.

Output: $Q^s$ and $R^s$.

1: $R^s \leftarrow \begin{pmatrix} R^{s-1} & 0 \\ 0 & 0 \end{pmatrix}$

2: $\xi^s \leftarrow D_{A^s}$

3: for $j = 1, \ldots, s-1$ do

4: $R_{js}^s \leftarrow (Q^{s-1})_j H \xi^s$

5: $\xi^s \leftarrow ξ^s - R_{js}^s Q_j^{s-1}$

6: end for

7: $R_{ss}^s \leftarrow \sqrt{\|ξ^s\|^2}$

8: $Q^s \leftarrow [Q^{s-1}, \frac{ξ^s}{R_{ss}^s}]$

linear complexity enables SSketch algorithm to readily scale up for processing a large amount of data based on a streaming model.

5.3.1 Blocking SSketch algorithm

Let $A = \begin{bmatrix} A_1; A_2; A_3 \end{bmatrix}$ be a matrix consisting of rows $A_1$, $A_2$, and $A_3$ that are stacked on the top of one another. Our key observation is that if we obtain the sketch of each block independently and combine the resulting sketches (blocking SSketch algorithm) as illustrated in Figure 5.2, then the combined sketch can be as good as sketching $A$ directly (nonblocking SSketch algorithm) in terms of error-performance trade-off. This property can be generalized to any number of partitions of $A$. We leverage this convenient property to increase the performance of our proposed framework for sketching massive datasets based on a streaming model. In blocking SSketch algorithm, the
data matrix \( \mathbf{A} \) is divided into more manageable sized blocks such that there exist enough block RAMs on FPGA to store the corresponding \( \mathbf{D} \) and a single column of that block. The blocking SSketch algorithm achieves a significant bandwidth saving, shorter load/store duration, less communication traffic between kernels, and a fixed memory requirement on FPGA. The methodology also provides the capability of factorizing massive, dense datasets in an online streaming model.

Independent analysis of each block is especially attractive if the data is distributed across multiple machines. In such settings, each platform can independently compute a local sketch. These sketches can then be combined to obtain the sketch of the original collection. Given a fixed memory budget for the matrix \( \mathbf{D} \), as it is presented in Section 6.5, blocking SSketch algorithm results in a more accurate approximation compared with nonblocking SSketch algorithm. The blocking SSketch algorithm computations are done on smaller segments of data which confers a higher system performance. The achieved higher accuracy is at the cost of a larger number of non-zeros in \( \mathbf{C} \). Note that as our evaluation results imply, designers can reduce number of non-zeros in the computed block-sparse matrix by increasing the error threshold \( \epsilon \) in SSketch algorithm.

![Figure 5.2 : Schematic depiction of blocking SSketch algorithm.](image)

### 5.4 SSketch hardware-accelerated implementation

In this section, we discuss the details of our hardware-accelerated implementation. Our implementation incorporates multiple OMP kernels and a control unit to effi-
ciently compute the block-sparse matrix $\mathbf{C}$. As the stream of data arrives, the control unit looks for availability of OMP kernels and assigns the newly arriving sample to an idle kernel for further processing. The control unit also has the responsibility of reading out the outputs and sending back the results to the host. Our APIs provide designers with a user-friendly interface for rapid prototyping of arbitrary matrix-based data analysis algorithms and realizing streaming applications on FPGAs, Figure 5.3. A constraint-driven customization unit takes user-defined properties and hardware limitations as inputs and customizes the framework for an optimized sketch computation. The input parameters of the constraint-driven customization unit may include runtime, power, and memory constraints and its outputs include sketching algorithmic parameters as well as guidelines for the hardware mapping. Note that the algorithmic parameters of SSketch including the projection threshold $\alpha$, error threshold $\epsilon$, dictionary size $L$, target block-sparsity level $k$, and block-size $m_b$, can also be easily changed through the API. The SSketch API is open source and is freely available on our website [2].

In hardware implementation of OMP, we utilize several techniques to reduce the iteration interval of two successive operations and exploit the parallelism within the algorithm. We observe that the OMP algorithm includes multiple vector multiplications which result in frequent appearance of for-loops. We use a tree-based reduction module by implementing a tree-based adder to accelerate the dot product and norm computation steps that appear frequently in the OMP routine. By means of the reduction module, we are able to reduce the iteration interval and handle more operations simultaneously. As such, SSketch requires multiple concurrent loads and stores from a particular RAM. To cope with the concurrency, instead of having a large block RAM for matrices $\mathbf{D}$ and $\mathbf{Q}$, we use multiple smaller sized block memo-
Figure 5.3: High level diagram of SSketch API. The constraint-driven customization unit takes user-defined properties and hardware limitations as inputs and delivers output parameters for an optimized sketch computation. The input parameters may include runtime, power, and memory constraints and the outputs include sketching algorithmic parameters as well as guidelines for the hardware mapping.

We fill these block RAMs by cyclic interleaving. Thus, we can perform a faster computation by accessing multiple successive elements of the matrices and removing the dependency in the for-loops.

Using the block RAM is desirable in FPGA implementations because of its fast access time. The number of block RAMs on one FPGA is limited, so it is important to reduce the amount of utilized block memories. We reduce block RAM utilization in our realization by a factor of 2 compared to the naive implementation. This reduction is a consequence of our observation that none of the columns of matrix $D$ would be selected twice during one call of the OMP algorithm. Thus, for computing line 4 of Algorithm 4 we only use the indices of $D$ that are not selected during the previous OMP iterations. We instead use the memory space that was originally assigned to the selected columns of $D$ to store the matrix $Q$. By doing so we reduce the block
RAM utilization, which allows us to employ more OMP kernels in parallel.

5.5 Automated SSketch customization

In this section, we demonstrate how SSketch automatically customizes itself based upon a set of user/platform constraints including runtime, power, and memory to deliver the most accurate transformation while maximizing the sparsity, Figure 5.4.

Figure 5.4 : Overview of SSketch automated design space exploration and customization.

To handle massive, dynamic datasets in streaming applications, SSketch can automatically customize itself corresponding to different set of user/platform constraints. As we demonstrate in Section 6.5, there is a trade-off between accuracy of the sketch matrix and the performance which can be carefully leveraged to improve the sketch computation’s efficiency.

Note that each dictionary sub-matrix is constructed from the columns of its corresponding data sub-matrix, Figure 5.2. The column space of the dictionary is contained in that of the data matrix, thus, the rank of each dictionary sub-block is less than $m_b$, where $m_b$ is the block size. Therefore, in blocking SSketch algorithm, each dictionary sub-block can at most consist of $m_b$ independent samples. Such property automatically ensures that the memory constraint is satisfied. However, in practice
one might need to customize the architecture such that not only memory constraint is addressed but also runtime and power constraints are taken into consideration.

Our key observation is that by increasing the dictionary size, $L$, the redundancy in the dictionary is increased. Thus, $D$ can better capture the hybrid structure of the input data, which results in a higher sparsity level in the coefficient matrix $C$ and typically higher sketch accuracy, Figure 6.12. To carefully leverage this trade-off between performance and accuracy, SSketch learns/builds the maximum possible dictionary size while a set of runtime, power, and memory constraints are met.

5.5.1 Constraint-driven optimization

(a) Runtime per signal in blocking SSketch algorithm vs. the sparsity level, $k$, with $L = 128$, block-size = 256, $\epsilon = 0.001$.  
(b) Runtime per signal in blocking SSketch algorithm vs. the dictionary size, $L$, with $k = L$, block-size = 256, and $\epsilon = 0.1$.  
(c) Runtime per signal in blocking SSketch algorithm vs. number of OMP kernels while $k = L = 128$, block-size = 256, and $\epsilon = 0.1$.

Figure 5.5 : Performance of SSketch’s FPGA-accelerated implementation. In both (b), and (c) we set $k = L$ to let $k$ be as large as the dictionary size and use $\epsilon$ as the stopping criteria in the OMP algorithm.

Memory constraint on the computing platforms is one of the main limitations in big data regime. Blocking SSketch computes the sketch of dynamic data collections by
breaking up the data into more manageable blocks according to the memory budget. In SSketch framework, the memory requirement can be approximated by \(((Lm_k + m_b) \times (N_k + 1) + N_kL^2) \times 4\) bytes, where \(N_K\) is the number of OMP kernels, \(m_b\) is the block-size in blocking SSketch algorithm, and \(L\) is the number of samples in the dictionary matrix.

Total delay of SSketch \((T_{SSketch})\) can be expressed as:

\[
T_{SSketch} \approx T_{\text{dictionary learning}} + T_{\text{Communication Overhead}} + T_{\text{FPGA Computation}}
\]

\[
\approx \beta_0 MN + \beta_1 MN(L + L^2) + \beta_2 \frac{MN(kL + k^2)}{N_K},
\]

where \(\beta_i\)s are constant coefficients that characterize the runtime requirement per unit of floating point operation. The latter term in (5.1) represents the runtime cost of computing the block-sparse matrix \(C\), which is the dominant factor in SSketch’s total delay. Figure 5.5 demonstrates the average runtime of SSketch as a function of sparsity level \((k)\), dictionary size \((L)\), and number of OMP kernels \((N_K)\). As Figure 5.5 shows, the average processing time for a sample in SSketch framework is almost independent of the total number of samples. This convenient property enables SSketch to readily scale up for analyzing any number of input samples.

According to the OMP algorithm, for each newly arriving sample we expect \((M(kL + k^2))\) operations for updating the block-sparse matrix \(C\). As it is illustrated in Figure 5.5a, and 5.5b our hardware-accelerated implementation of SSketch follows the same trend as predicted by the OMP algorithm complexity. Note that none of the columns of matrix \(D\) would be selected twice during one call of the OMP algorithm. In our hardware-accelerated realization of SSketch, we compute line 4 of Algorithm 4 by searching among those columns of the dictionary that have not been selected.
during the previous OMP iterations. Our hardware implementation approach results in a milder slope for larger $k$s in Figure 5.5a. For example this property can be seen for $100 \leq k \leq 128$ versus $80 \leq k \leq 100$ in Figure 5.5a.

Figure 5.5c illustrates the average runtime of SSketch as a function of $N_K$, where $N_K$ is the number of OMP kernels. The kernels are used in parallel to compute the block-sparse matrix $C$. As it is illustrated, the average runtime of SSketch is proportional to $1/N_K$.

Table 5.1 reflects the total power consumption of our hardware-accelerated implementation of SSketch on Xilinx Virtex-6-XC6VLX240T FPGA ML605 Evaluation Kit. As we can see in Table 5.1, the total power consumption of SSketch is a linear function of the number of OMP kernels that are employed in data sketching unit.

Table 5.1 : Power consumption on Virtex 6 for different number of OMP kernels.

<table>
<thead>
<tr>
<th>Number of Kernels ($N_K$)</th>
<th>Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.358w</td>
</tr>
<tr>
<td>2</td>
<td>0.447w</td>
</tr>
<tr>
<td>3</td>
<td>0.546w</td>
</tr>
<tr>
<td>4</td>
<td>0.634w</td>
</tr>
</tbody>
</table>

As we mentioned earlier, SSketch provides users with a domain-specific architecture that scalably performs data analysis on an ensemble of lower dimensional structures rather than the original big matrix without a significant loss. It adaptively learns the hybrid structure of the streaming input data to effectively improve the iterative matrix computations performance. The SSketch’s optimization can be expressed as:

$$\minimize_{L, m_b, N_K} \text{(approximation error)}, \quad (5.2)$$
subject to: \[ L \leq m_b, \]
\[ m_b \leq m, \]
\[ N_K \in \mathbb{N}, \]
\[ \beta_2 MNL^2/N_K \leq T_u, \]
\[ P_{SSketch} \leq P_u, \]
\[ (L + 1)(N_K + 1)m_b + N_KL^2 \leq M_u, \]

where \( T_u, P_u, M_u \) are a set of user defined parameters which imply the user/platform constraints in terms of runtime, power, and memory respectively.

SSketch approximates the solution of (5.2) using the Karush-Kuhn-Tucker (KKT) conditions. To efficiently capture the hybrid structure of streaming data collections, SSketch automatically tunes its framework according to the application and sets its algorithmic parameters including block-size \( m_b \) in blocking SSketch algorithm, dictionary size \( L \), as well as the number of OMP kernels \( N_K \). To facilitate automation, we provide a solver for our optimization approach. The solver gets the constraints from the user as input and uses Mathematica computational software program to solve the optimization.

5.5.2 Design space exploration/ SSketch customization examples

We provide three clarifying examples to demonstrate the advantage of SSketch’s automated tuning approach for user/platform specific customization and performance optimization. Here, we adapt SSketch’s automated customization approach to tune the framework for three different sets of user/platform constraints. We also compare the approximation results with the actual optimal values. In all these examples, we see that our approximations for SSketch algorithmic parameters are very close to the
Figure 5.6: Motivational examples for SS sketch’s automated constraint-driven customization. Each of these figures demonstrates the average runtime required to process one sample as a function of the dictionary size. The user power constraint determines the maximum number of OMP kernels that can work in parallel. The dashed horizontal line, reflects the user runtime deadline and the purple star illustrates the SS sketch’s automated customization output (optimal dictionary size). The crossing point of the dashed line and the corresponding runtime curve (solid curve in each figure) is the actual optimized point in each of these settings.

real optimized values which implies the accuracy of SS sketch automated constraint-driven customization. Our target platform is a Virtex-6-XC6VLX240T FPGA ML605 Evaluation Kit with 1872kB available block RAM memory [73].

Example (i): First, consider a situation where a user wants to compute the sketch matrix of a dynamic data collection with a size of \( M = 256, \) and \( N = 5000 \) within a runtime budget of 30s. In this case, the memory and power are only limited by the target platform; in other words there are no user constraints in terms of memory or power. Since no power constraint exists, \( N_K \) (the number of OMP kernels) is set at its maximum value, which is determined by the memory budget. With the Virtex-6-XC6VLX240T board, 4 OMP kernels fit on an FPGA with 1872kB of available block
RAM memory (Table 6.3). Figure 5.6 demonstrates the average runtime for a sample sketch update as a function of dictionary size in our real-world evaluation on FPGA. As Figure 5.6a illustrates, $L = 107$ is the maximum size dictionary one can use to compute the sketch of data with the aforementioned set of user/platform constraints. On the other hand, solving our constraint-driven optimization in (5.2) results in $l \approx 98$ (purple star-marked point in Figure 5.6a). Thus, our parametric optimization outputs efficiently model the real-world hardware-accelerated implementation.

Example (ii): Second, consider a situation where a user wants to compute the sketch matrix of a dynamic data collection with a size of $M = 256$, and $N = 5000$ within a runtime budget of $25\text{s}$ and a power budget of $0.6\text{ watts}$. In this case, the memory is limited by the target platform. Due to the power constraint, one can make use of 3 OMP kernels to compute the sketch (Table 5.1). As Figure 5.6b illustrates, $L = 80$ is the maximum size dictionary one can use to compute the sketch of data with the aforementioned set of user/platform constraints. On the other hand, solving our constraint-driven optimization in (5.2) results in $l \approx 78$ (purple star-marked point in Figure 5.6b).

Example (iii): Third, consider a situation where a user wants to compute the sketch matrix of a dynamic data collection with a size of $M = 512$, and $N = 2500$ within a runtime budget of $90\text{s}$ and a power budget of $0.4\text{ watts}$. In this case, the memory is limited to the 1872kB block RAM memory available with the target platform. Due to the power constraint, one can only make use of 1 OMP kernel to compute the sketch (Table 5.1). As Figure 5.6c illustrates, $L = 95$ is the maximum size dictionary one can use to compute the sketch of data with the aforementioned set of user/platform constraints. On the other hand, solving our constraint-driven optimization in (5.2) results in $L \approx 85$ (purple star-marked point in Figure 5.6c).
The above examples further demonstrate the applicability and efficiency of our constraint-driven optimization approach proposed in this section.

5.6 Theoretical bounds on transformation error

In SSAnna methodology, the dictionary matrix $D$ is constructed such that the column space of $D$ is contained in the column space of the data matrix $A$. To bound the reconstruction error using SSAnna, we propose Theorem 1.

**Theorem 1**

For blocking SSAnna algorithm, the reconstruction error of a massive, dynamic input data $A$ is

$$
\|A - DC\|_F^2 \leq \max(\alpha, \epsilon).
$$

*Proof:* Let $a_{ij}$ represent the $i$th segment of the $j$th column of input data matrix $A$. $S_i$ denotes the set of $L$ column indices that have been selected from the $i$th sub-block of $A$ to construct the corresponding sub-block of dictionary matrix $D$, and $T_i$ denotes the remaining $N - L$ columns of the same block. In blocking SSAnna algorithm, for each newly arriving sample, if $W(a_{ij}) \geq \alpha$, then it is added to the dictionary matrix $D$, Figure 5.2. The reconstruction error for the added $a_{ij}$ is exactly zero via SSAnna’s methodology. For the remaining part of the input data, the greedy OMP routine is used to compute sparse approximation of the sample.

In our methodology, when the dictionary size $L$ is set to be large enough (i.e. $l = m_b$), then the set of $L$ samples that are linearly independent will span the ambient dimension of the corresponding data block $\mathbb{R}^{m_b}$, which results in exact decomposition, i.e., $\|a_{ij} - D_i D_i^+ a_{ij}\|_F = 0$. Note that, OMP routine does not stop unless either the reconstruction error $(\|a_{ij} - D_i a_{ij}\|_F)$ reaches a value less than or equal to $\epsilon$, or unless all the column samples in $D_i$ are used for reconstructing $a_{ij}$, where $D_i$ is the corresponding dictionary sub-block for the input segments $a_i$. In the latter
case, the normalized reconstruction error is less than or equal to $\alpha$ according to the SSketch algorithm. Therefore, for each $a_{ij}$ we have:

$$\frac{\|a_{ij} - D, c_{ij}\|_F^2}{\|a_{ij}\|_F^2} \leq max(\alpha, \epsilon)$$

$$\|a_{ij} - D, c_{ij}\|_F^2 \leq max(\alpha, \epsilon)\|a_{ij}\|_F^2. \tag{5.3}$$

Summing up (5.3) over all blocks for an input sample $a_j$ results in:

$$\sum_{i=1}^{m_b} \|a_{ij} - D, c_{ij}\|_F^2 \leq \sum_{i=1}^{m_b} max(\alpha, \epsilon)\|a_{ij}\|_F^2$$

$$\|a_j - Dc_j\|_F^2 \leq max(\alpha, \epsilon)\|a_j\|_F^2. \tag{5.4}$$

Equation (5.4) is a result of the blocking structure of the dictionary matrix $D$ (Figure 5.2). Finally, the overall reconstruction error can be presented by summing up (5.4) over all $N$ input samples.

$$\sum_{j=1}^{n} \|a_j - Dc_j\|_F^2 \leq \sum_{j=1}^{n} max(\alpha, \epsilon)\|a_j\|_F^2$$

$$\|A - DC\|_F^2 \leq max(\alpha, \epsilon)\|A\|_F^2 \blacksquare \tag{5.5}$$
Chapter 6

Evaluations

In this Section, we perform extensive evaluations on the proposed customizable transformation (Chapter 3) and distributed platform-aware tuning and mapping techniques (Chapter 4). We study the sparsity inducing capability and scalability of ExD, the connection between approximation error as a result of ExD and learning accuracy, and performance improvement in terms of timing, energy, and memory performance for multiple iterative learning algorithms.

6.1 Evaluation setup

6.1.1 Datasets

Our evaluations are done on the following datasets. Dataset 1 which is referred by Salinas, is a collection of hyperspectral images and its size is $203 \times 54129$ (87.9MB). The data has been collected by a 224-band AVIRIS sensor over Salinas Valley, California [1]*. Dataset 2 which is referred by Cancer Cells, is a medical dataset collected in MD-Anderson cancer center and its size is $1024 \times 111296$ (911.7MB). It consists of cancer tumor morphologies collected from different patients over time. Dataset 3 which is referred by Light Field, is a collection of light field images and its size is $18496 \times 272320$ (40.3GB). It consists of $8 \times 8$-pixel patches of images taken from a $17 \times 17$ array of light field cameras. Each image in the array is captured from a slightly

---

*21 frequency bands were omitted from the data due to invalid values.
different viewpoint. The dataset is collected from Stanford Computer Graphics Laboratory Archive [3]. Dataset 4 which is referred by VideoDict contains patches of multiple frames and its size is $1764 \times 100000$ (1.4GB) [24]. Dataset 5 which is referred by face image consists of 631 images of 10 faces of different subjects under varying illumination conditions. Each image is $48 \times 84$ pixels, which produces a dataset of size $4032 \times 631$ (20.3MB) [19].

6.1.2 Implementation and API

We use three different computing platform. Platform 1 is an 8-core CPU (Intel Core™i7 processor) with 12GB of RAM is used. Platform 2 is a cluster of up to 2304 cores in 192 Westmere nodes (12 processor cores per node) at 48GB of RAM per node (on IBM iDataPlex). Platform 3 is a cluster of 16 m3.large nodes (machines) on Amazon EC2. Each node has 16 cores (two Intel Xeon processors) at 7.5GB of RAM per node.

We implement the tunable ExD transformation and the distributed iterative model on the transformed data in C++ using the standard message passing system (MPI). We use Eigen library for linear algebra computation. Our API takes the following user inputs: dataset $A$, transformation error $\epsilon$, and the learning algorithm as an iterative update function on Gram matrix. We use a routine to experimentally measure the platform-specific relative cost of arithmetic vs. communication ($R_{b2f}^{time}$). Our codes are available for download at [2]. We also implement the distributed mapping for iterative updates on the factorized data on Apache Spark [46], a key value based distribute abstraction that supports fault tolerance.

We adapt a graph-parallel implementation using GraphLab abstraction [22]. GraphLab enables vertex-update-based computations. We replace GraphLab’s in-
ternal data partitioning core with our own optimally bounded partitioning method described in Chapter 4. Note that the GraphLab framework is designed to accelerate distributed learning for sparse graphs and is not suited for processing the original dense data.

Our SSketch implementations are done on Xilinx Virtex-6-XC6VLX240T FPGA ML605 Evaluation Kit.

6.1.3 ExD overhead

![Graphs showing runtime overhead of ExD](attachment:Graph.png)

Figure 6.1: Preprocessing time overhead of ExD: The graphs show the runtime of applying ExD on different subsets of $A$. For a fixed dictionary size $L$, the runtime scales almost linearly with the cardinality of the data (as shown on the figures). The runtime is a function of both $L$ and $\alpha(L)$.

Figure 6.1 shows the total preprocessing time overhead of ExD for various cardinalities of the input data and dictionary sizes during preprocessing. The computations are done on 64 cores (8 nodes each with 8 cores). For the same $L$, the runtime almost linearly scales with the cardinality of the input data. This is due to the parallel nature of ExD algorithm. For larger $\alpha(L)$ values, the runtime increases due to the overhead of calculating each non-zero coefficient. On the other hand, as discussed in
Section 3.3.3, for larger $L$’s (even though $\alpha(L)$ decreases) the OMP execution time becomes longer. The mentioned observations explain the variations in the behavior of ExD runtime. The overhead of ExD is amortized for large data because it enables efficient processing of a massive dataset by using preprocessing on a relatively small data subsample.

6.1.4 Verification of the performance model

Figure 6.2: Verification of performance model: The graphs show estimated (top row) and actual (bottom row) performance of $C^T D^T D C x$ for various platforms. Our predicted runtime (in terms of number of floating point operations) closely follows the trend (in terms of milliseconds) in the actual evaluations.

Figure 6.2 compares our estimation of the runtime (quantified in Equation 4.1) and the actual runtime for one iteration of $G x \simeq C^T D^T D C x$ on the transformed data. The actual runtime measures are averaged over 100 iterations. For predicting the cost, we use $NNZ(C) = N\alpha(L)$. The tests are done on various number of
distributed processing cores \( (N_P)'s \). We estimate \( \alpha(L) \) by running ExD on 10\% of the data, Figure 6.3. In our platform, \( R_{02f(N_P)} < 1 \) for various \( N_P \)'s and is zero for \( N_P = 1 \). As can be observed, the actual performance is very similar to our estimated performance. This property enables us to tune ExD (by finding the optimal dictionary size: \( L_{opt} \)) using our proposed performance model.

### 6.1.5 Estimating sparsity of \( C \) from subsets of data during preprocessing

![Graphs showing average number of non-zeros per column of \( C \) versus \( L \) for different subsets of \( A \).](image)

Figure 6.3: Effective estimation of sparsity for tuning ExD: The graphs show the average number of non-zeros per column of \( C \) (i.e., \( \alpha(L) \)) versus \( L \) for different subsets of \( A \). As the sizes of the subsets increase, \( \alpha(L) \) converges to that of the full data \( A \).

Figure 6.3 shows the average number of non-zeros per column of \( C \) (denoted by \( \alpha(L) \)\(^*\)) as a result of applying ExD on different datasets. The transformation error \( \epsilon \) is set to 0.1. The first observation is that, as expected, when the size of the dictionary (parameter \( L \)) increases, \( C \) becomes sparser. This is due to the increased redundancy in the dictionary. The second observation is that even by applying ExD on the subsets of the data, i.e., \( A_1 \subset A_2 \subset A_3 \subset A_4 \subset A_5 \subset A \), a reasonable estimation of \( \alpha(L) \) can be achieved. For example for \( L = 1000 \) the error of estimating \( \alpha(L) \), i.e.,

\(^*\)Since \( A \) and \( \epsilon \) are fixed, we abbreviate \( \alpha(L, A, \epsilon) \) with \( \alpha(L) \).
\[
\frac{\alpha(L,A) - \alpha(L,A_3)}{|\alpha(L,A)|}\]

is less than 14% in all datasets. In this case, \(A_3\) has only 10% of the columns in \(A\). Thus, we can tune \(\alpha(L)\) more efficiently by applying ExD on only a subset of the data during the preprocessing phase.

### 6.1.6 Effect of transformation error on sparsity of \(C\)

![Figure 6.4: Effect of transformation error on the sparsity](image)

Figure 6.4: Effect of transformation error on the sparsity: The graphs show the average number of non-zeros per column of \(C\) (i.e., \(\alpha(L)\)) versus different transformation errors (i.e., \(\epsilon\)). As \(\epsilon\) increases, better sparsity is achieved. This is because for smaller \(\epsilon\)'s, more non-zero coefficients are added to \(C\) so that the error tolerance criteria is met.

Figure 6.4 shows the relationship between \(\alpha(L)\) and transformation error \(\epsilon\). As expected, larger \(\epsilon\) values increase the sparsity (smaller \(\alpha(L)\)'s are attained). Similar to our example in Section 4.4, we observe a varying trend in \(\alpha(L)\) as \(L\) increases (first increasing to a peak and then decreasing). The reason is that when the dictionary size is not large-enough (i.e., \(L < L_{\text{min}}\)) so that ExD can meet the transformation error criteria, the columns of \(C\) are fully dense. After enough columns are added to the dictionary such that the error criteria is met, \(\alpha(L)\) starts to decrease (as \(L\) increases). When the coarse-grained parallelism exists, as it is the case in our test datasets, \(\alpha(L)\) quickly becomes much smaller than \(M\) (number of non-zeros per column of \(A\)).
6.2 Applications

In this section, we provide multiple machine learning applications based on two correlation matrix-based iterative update methods: $\ell_1$ minimization and power method.

6.2.1 $\ell_1$ minimization applications

To evaluate the performance of SSketch for sparse approximation, we use the *fast iterative shrinkage-thresholding algorithm (FISTA)* [5] to solve the $\ell_1$-minimization problem in Equation 2.2. We study the utility of SSketch for two applications of the sparse representation problem discussed in Section 2.2. The applications include, sparse representation-based classification for face recognition and image denoising.

**Sparse representation-based classification for face recognition**

To employ sparse approximation for classification, our aim is to use a collection of labeled images (training set) as our dictionary $A$ and then form a sparse representation of a test image $y$ in terms of $A$. Upon finding a sparse coefficient vector $x$ which provides a sparse approximation of $y \approx Ax$, we can then determine which test signals (columns of $A$ or non zeros in $x$) are selected to represent the test signal $y$. Based upon the class of the selected columns, we then make a decision about which class the test signal lies in. One easy way to do this is to simply sum the absolute value of the coefficients in $x$ in a certain class and then find the class that has the maximum sum.

In Figure 6.5, we provide a demonstration of sparse representation-based classification for face recognition. We show the test image of interest on top and the corresponding sparse coefficient vector obtained by solving Equation 2.2 with FISTA for $\lambda = 1$. We solve FISTA with the full Gram matrix $A^T A$ and approximate Gram
Figure 6.5: Sparse representation-based face recognition. On top, we display a test image; on the bottom, we display the sparse solution obtained with the original Gram (blue, solid) and approximate Gram with ExD for $\epsilon = 0.05$ (red, dash). We display four training images corresponding to non zero coefficients: large amplitude coefficients correspond to images in the correct class and smaller amplitude coefficients correspond to images in incorrect classes. The block of coefficients corresponding to images in the correct class are highlighted.
provided by ExD, where the decomposition error $\epsilon = 0.05$ ($L = 62$).

Figure 6.6: (a) For a range of decompositions (varying $\epsilon$) we show learning accuracy which measures the 2-norm between the solution obtained with the full and approximate Gram, (b) For a range of decompositions (varying $\epsilon$) we show sum of coefficients in the correct class, the minimum sum of coefficients required to correctly classify the test image is also shown.

To understand the connection between the factorization error and learning accuracy for face recognition, we solve Equation 2.2 using FISTA for two different regularization parameters $\lambda = \{0, 1\}$, where $\lambda = 0$ corresponds to the least-squares solution and $\lambda = 1$ produces sparse solutions. We vary the decomposition error $\epsilon = \{0.4, 0.2, 0.1, 0.05\}$ and solve FISTA for 30 different test images (after removing them from training) for each of these decompositions. For each decomposition, we calculate the: learning accuracy by measuring the $\ell_2$-norm between the solution obtained with the full and approximate Gram (Figure 6.6a), the sum of coefficients in the correct class (Figure 6.6b). In Figure 6.6b, we also display the minimum sum of
coefficients required to correctly classify the test image. These results suggest that while learning accuracy might be relatively large for high amounts of decomposition error, correct classification is possible even for large amounts of decomposition error (i.e., correct classification occurs for all test images when $\epsilon < 0.2$).

**Light field image denoising**

We evaluate ExD’s performance gain in solving the Light Field data reconstruction problem. A light field is a multi-dimensional array of images. Each image is captured from a slightly different viewpoint. Combining the images enables creating new views and representing observer positions not present in the original array. Most currently available devices trade image resolution for the ability to capture different views of a light field. As a result, the final image resolution is reduced by orders of magnitude compared to the sensor resolution. To address this challenge, $\ell_1$-minimization (Equation 2.2) is employed to find a sparse representation of a light field image patch with respect to an *over-complete light field dictionary* consisting of a large number of light field image patches collected from many scenes [32].

We first apply ExD for decomposing the dictionary corresponding to the Light Field (dataset. Two different $L$ values (240 and 1000) are used in Algorithm 1. The decomposition error $\epsilon$ is set to 0.1. We then evaluate FISTA on the decomposed data for matrix-based models. We also implement a tailored distributed MPI-based model to evaluate FISTA on the original datasets ($A$) using regular dense matrix representations. In the following, we refer to this model as *baseline*.

Peak Signal to Noise Ratio (PSNR) is the ratio between the maximum possible power of a signal and the power of the corrupting noise. PSNR is most commonly used to measure the quality of reconstruction of noisy images and is defined
as $10 \log_{10}(\frac{\text{MAX}}{\sqrt{\text{MSE}}})$ (dB). $\text{MAX}$ is the maximum pixel value of the original image patch and $\text{MSE}$ is the mean square reconstruction error defined as $\frac{|y - \hat{y}|^2}{m}$. In our experiments $\text{MAX} = 0.0255$. Generally, higher PSNRs indicate the better quality of reconstruction. Typical recommended PSNR values in image noise reduction application are 30 dB and higher [44, 7].

Table 6.1 shows the total runtime of FISTA to achieve different PSNRs. In all the experiments, a batch of 10 noisy input patches is used as the input. The norm of the noise is 0.3 of the norm of the input vector (PSNR=21.14). It can be observed that by running FISTA on the decomposed data, the same PSNR is reached much faster in comparison with running FISTA on $\text{A}$. We observe that the runtime to achieve the same PSNR is orders of magnitude faster compared to the baseline. For instance, if our desired PSNR is 30.0 dB, running FISTA on the decomposed takes 13.9s ($l=240$) and 162s ($l=1000$) while it takes 1050s for the baseline. However, as expected, the baseline ($\text{A}$) reaches higher PSNRs in comparison with those achieved from running FISTA on the decomposed data. For example, while the baseline’s highest achieved PSNR at convergence is 48.5 dB, it is 31.3 dB and 38.5 dB for $l = 240$ and $l = 1000$ respectively.

<table>
<thead>
<tr>
<th>PSNR (dB)</th>
<th>$L = 240$</th>
<th>$L = 1000$</th>
<th>baseline (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>4.2</td>
<td>72</td>
<td>487</td>
</tr>
<tr>
<td>30</td>
<td>13.9</td>
<td>162</td>
<td>1050</td>
</tr>
<tr>
<td>35</td>
<td>-</td>
<td>356</td>
<td>2051</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>-</td>
<td>3171</td>
</tr>
</tbody>
</table>

Table 6.1: FISTA runtime (s) to reach to a specific PSNR.
Figure 6.7: Runtime of Power method on data from customized ExD, non-customized ExD ($L = L_{\text{min}}$), and baseline. ExD results in significant runtime reduction. Platform-aware customization of ExD generates meaningful further enhancements.

### 6.2.2 Power method applications

**Impact of ExD on application runtime performance.** Figure 6.7 shows the runtime performance of Power method for finding the first 100 eigenvalues of different datasets. The platform consists of 64 processing core, i.e., $N_P = 64$. The baseline method is the case where the Gram matrix $G$ is computed using $G = A^T A$. The customized method is the case where $G$ is computed using $G = (DC)^T DC$, where $D$ and $C$ are the results of ExD after tuning the dictionary size for the platform, i.e., $L = L_{\text{opt}}$. The non-customized method is the case where $G$ is computed using $G = (DC)^T DC$, where $D$ and $C$ are the results of ExD without tuning the dictionary size, i.e., $L = L_{\text{min}}$. The $\epsilon$ in both latter cases is set to 0.1.

Significant performance improvement is achieved when data is preprocessed by applying ExD. The importance of tuning ExD, however, is clear due to the meaningful additional improvement that is achieved over the non-customized method. Table 6.2 outlines the overall runtime improvement of both customized and non-customized ExD over the baseline method.

**Impact of ExD on memory and energy performance.** Table 6.2 shows the
Figure 6.8: Runtime performance of ExD vs. transformation error ($\epsilon$) and number of processing nodes ($N_p$). Reported values are runtime for finding the first 100 largest eigenvalues using Power method. When larger $\epsilon$’s are allowed, ExD produces more efficient transformations that further improve runtime.

improvement in memory usage (i.e., memory footprint of $CD$ vs. $A$). ExD is tuned for $N_P = 64$ and $\epsilon = 0.1$. The reductions are significant for all the datasets. In particular, the Light Field dataset achieves more than $70 \times$ memory reduction compared to baseline. The different achieved reductions signify the data-dependent nature of ExD. The energy usage highly depends on the number of floating-point operations (Section 4.3). While, we do not report the exact energy usage reduction, the reduction in memory (and thus the number of floating point operations) guarantees improvement in energy as well.

Figure 6.8 studies the impact of transformation error ($\epsilon$) and number of processors ($N_P$) on the runtime of power method. The Power method is executed until the first 100 eigenvalues of the Cancer Cell dataset is calculated. Figure 6.9 demonstrates the cumulative error of the calculated eigenvalues found while using ExD relative to those found while using the original data. We define cumulative error as: $\frac{\|A_k\|_F - \|\tilde{A}_k\|_F}{\|A_k\|_F} = \frac{\sum_{i=1}^{k} \sigma_i^2 - \sum_{i=1}^{k} \tilde{\sigma}_i^2}{\sum_{i=1}^{k} \sigma_i^2}$, where $A_k$ and $\tilde{A}_k$ are the derived k-rank approximations, and $\sigma_i$ and
Table 6.2: Reduction in Power method runtime and memory usage. ExD inputs are $N_P = 64$ and $\epsilon = 0.1$.

<table>
<thead>
<tr>
<th>Power method</th>
<th>Runtime (Non-customized)</th>
<th>Runtime (Customized)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salina</td>
<td>1.8×</td>
<td>7.1×</td>
<td>4.6×</td>
</tr>
<tr>
<td>Cancer Cells</td>
<td>3.5×</td>
<td>9.7×</td>
<td>3.5×</td>
</tr>
<tr>
<td>Light Field</td>
<td>20.3×</td>
<td>82.1×</td>
<td>70.9×</td>
</tr>
</tbody>
</table>

Figure 6.9: Cumulative error of the first computed 100 eigenvalues for different values of transformation error $\epsilon$. Smaller $\epsilon$’s result in more accurate results. However, in all the experiments, the cumulative error are still very low and significantly smaller than the transformation error.

$\tilde{\sigma}_i$ are the derived eigenvalues as a result of running Power method on the original and the transformed data. We observe that even though by increasing $\epsilon$ the accuracy of the computed eigenvalues decreases, the overall error is still very low even when $\epsilon$ is as high as 0.1. On the other hand, a higher $\epsilon$ results in significant runtime performance improvement.

Our evaluations of the Power method application not only demonstrate that significant speed ups can be achieved if ExD is used, it also verifies that ExD preserves
Figure 6.10: Comparing matrix-based and graph-based computing models for different dictionary sizes and data sparsity levels. The results report runtime for (a) varying $L$ sizes (for constant total number of non-zeros in $C$), (b) varying relative density of $C$ (for constant $L$), and (c) varying number of processors (for constant $L$ and total number of non-zeros in $C$).

the structure of data, i.e., the singular values can be recovered with minimal error compared to the original data.

6.3 Graph-parallel abstractions

These experiments provide insights on the use-case of each model. Depending on the structure of the transformed data and the specifications of the platforms an appropriate model should be selected.

An important outcome of ExD is producing sparse representations in the data. We take advantage of this property to enable graph-parallel abstractions applicable to the data that was originally dense. Graph-parallel engines heavily exploit sparse data dependencies to produce efficient partitioning and execution flow scheduling in iterative learning applications. We have discussed our solution for adaptation of graph-parallel abstraction GraphLab on the transformed data in Chapter 4. Here,
we aim to demonstrate the potential advantages of graph-parallel approaches in comparison with regular matrix-based commutations.

To evaluate how the performance is dependent on the structure of the transformed data, we generate multiple synthetic factorized datasets (D and C pairs). In all the experiments, an iterative update on the correlation matrix is applied on a random input vector y. The experiments are done on Platform 2. In all the figures, the runtime results for the dense matrix-based implementation (i.e., regular deployment of the decomposed matrices without using compressive column storage format) are also provided to demonstrate the efficiency achieved by exploiting sparsity in C through graph-based and sparse matrix-based models.

Figure 6.10a shows the performance for different (block-diagonal) C’s with fixed number of non-zeros (set to 100M). As L increases, the density-level of C decreases. As can be seen, the graph-parallel design’s performance is more robust as L increases. However, the matrix-based model’s performance degrades for larger L’s. This observation is consistent with our cost quantification and performance bounds. Recall that the matrix-based computing model is more sensitivity to L.

Figure 6.10b shows the performance for a fixed L = 500 on block-diagonal matrices C for varying densities of C. As density increases, the performance decreases in both models. However, the performance degradation in graph-based model is worse due to the overhead of representing a large number of edges. Lastly, Figure 6.10c shows the scaling performance of the models for various number of processors. When the number of processors are less than 12, the computations are done on a single node. Thus, the reverse scaling behavior when the number of processors increases from 8 to 16 is due to the high overhead of the inter-node communication cost. For comparison purposes, the scaling performance of the baseline data, a dense M = 1000 by N = 10M dataset
(before decomposition) is provided. It can be seen that as the number of processors increases, the performance gap between different methods shrinks. However, even with a large number of processors ($\geq 100$), the decomposed models perform up to 2 orders of magnitude better than the baseline.

### 6.4 Apache Spark-based implementation

Here we compare our hand-coded MPI/C++ and Apache Spark implementation. Figure 6.11 shows the average runtime per iteration for power method on a cluster of 64 processors on IBM iDataPlex server. As can be seen, the hand-coded version can be more than 2 orders of magnitudes faster than the Spark version. In general, generic solutions such as Spark are not guaranteed to be fast, instead they provide easy and reusable programming frameworks that operate on very large datasets on a distributed computing platform. Users only have to deal with writing the functions of the algorithm in the given key-value based programming model. Spark on the other hand controls the distributed cluster, manages data partitioning and data transfers between the various parts of the system, and provides fault tolerance.

![Figure 6.11](image)

Figure 6.11: Comparison of MPI/C++ based and Spark based implementation’s average runtime per iteration for power method algorithm. All the experiments are done on the same platform with $N_P = 64$. 

6.5 SSketch experimental evaluations

Figure 6.12: Experimental evaluations of SSketch. \( \alpha \) and \( \epsilon \) represent the user-defined projection threshold, and the transformation error threshold respectively. \( L \) is the number of columns in the dictionary matrix, and \( m_b \) is the block-size. (a) Factorization error (\( X_{err} \)) vs. \( L \) with \( \alpha = 0.1 \), block-size = 200 and \( \epsilon = 0.01 \). (b) Factorization error (\( X_{err} \)) vs. block-size with \( \alpha = 0.1 \) and \( L = 128 \). (c) Compression-rate vs. block-size with \( \alpha = 0.1 \) and \( L = 128 \).

For evaluating SSketch algorithm accuracy, we run our experiments on a subset of light field data consisting of 2500 samples each of which constructed of 25 8 × 8 patches. The overall size becomes 1600 × 2500. In SSketch, the number of selected columns “\( L \)” for the dictionary has a direct effect on the convergence error rate as well as speed. Factorization with larger \( l \) achieves a smaller convergence error but decreases the performance due to the increase in computation. Figure 6.12a reports the results of applying both nonblocking and blocking SSketch on the data. We define the approximation error for the input data and its corresponding Gram matrix as \( X_{err} = \frac{\|A - \tilde{A}\|_F}{\|A\|_F} \), where \( \tilde{A} = DC \).

Given a fixed memory budget for the matrix \( D \), Figure 6.12a illustrates that the blocking SSketch results in a more accurate sketch compared with the nonblocking
one. Number of rows in each block of input data (block-size) has a direct effect on SS-ketch’s performance. Figures 6.12b and 6.12c demonstrate the effect of block-size on the transformation approximation error as well as matrix compression-rate, where the compression-rate is defined as $\frac{\text{NNZ}(D) + \text{NNZ}(C)}{\text{NNZ}(A)}$. In this setting, the higher accuracy of blocking SSketch is at the cost of a larger number of non-zeros in the block-sparse matrix C. As illustrated in Figures 6.12 (b) and (c), the number of non-zeros in the matrix C can be readily decreased by increasing the SSketch error threshold $\epsilon$. In SSketch algorithm, there is a trade-off between the sketch accuracy and the number of non-zeros in the block-sparse matrix C. The optimal performance of SSketch methodology is determined based on the error threshold and the user/platform constraints.

6.5.1 Hardware-accelerated settings and streaming results

![Diagram]

Figure 6.13: The internal structure of one OMP kernel. All computations are done in IEEE 754 single precision floating-point format. The D and Q matrices are stored in multiple smaller sized block memories and are filled by cyclic interleaving.

We use Xilinx Virtex-6-XC6VLX240T FPGA ML605 Evaluation Kit as our hardware platform. An Intel core i7-2600K processor with SSE4 architecture running on
the Windows OS is utilized as our general purpose processing unit hosting the FPGA. In this work, we employ Xilinx standard IP cores for single precision floating-point operations. We use Xilinx ISE 14.6 to synthesize, place, route, and program the FPGA.

We exploit Eigen library to implement the dictionary learning and CPU implementation of SSketch (used for comparison purposes) on the general-purpose Intel processor. The Eigen library exploits Intel Stream SIMD Extension (SSE) instructions on Intel processors to enhance the performance of intensive matrix computations. Using Eigen library pushes the performance of our software implementations to the limits of the general purpose processor.

Figure 6.13 demonstrates the internal structure of one OMP kernel. In this figure, the signals that end with “in” are inputs from the controller module and “ap-idle” is the controlling signal for each OMP kernel. We keep track of the selected indices of \( D \) in the “Supp” RAMs. The “MAX” unit finds the index of the column of \( D \) that has the maximum correlation with the residual. Finally, the back substitution in OMP algorithm is computed in “BS Mult” unit.

Table 6.3 shows Virtex-6 resource utilization for our heterogeneous architecture. SSketch includes four OMP kernels plus an Ethernet interface. For factorizing matrix \( A_{M \times N} \), there is no specific limitation on the size \( n \) due to the streaming nature of SSketch. However, the FPGA block RAM’s size is limited. To fit into the RAM, we decide to divide input matrix \( A \) to blocks of size \( m_b \times N \) where \( m_b \) and \( K \) are set to be less than 256. Note that these parameters are changeable in SSketch API. So, if a designer decides to choose a higher \( m_b \) or \( K \) for any reasons, she can easily modify the parameters according to the application.

To corroborate the scalability and practicability of our framework, we use synthetic
Table 6.3: Virtex-6 resource utilization.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Registers</td>
<td>50888</td>
<td>301440</td>
<td>16%</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>81585</td>
<td>150720</td>
<td>54%</td>
</tr>
<tr>
<td>RAM B36E1</td>
<td>382</td>
<td>416</td>
<td>91%</td>
</tr>
<tr>
<td>DSP 48E1s</td>
<td>356</td>
<td>768</td>
<td>46%</td>
</tr>
</tbody>
</table>

data with dense (non-sparse) correlations of different sizes as well as a hyperspectral image dataset. The runtime of for the different-sized synthetic datasets is reported in Table 6.4, where the total delay of SSketch ($T_{SSketch}$) is defined in (5.1).

As it is shown in Table 6.4, the total delay of SSketch is a linear function of the number of processed samples, which experimentally confirms the scalability of our proposed architecture. According to Table 6.4, the whole system including the dictionary learning process takes 21.029s to process 5K samples where each of them has 256 elements. 4872 of these samples pass through OMP kernels and each of them requires 89 iterations on average to complete the process. In this experiment, an average throughput of 169Mbps is achieved by SSketch.

For both hyperspectral dataset of size $204 \times 54129$ and synthetic dense data, our HW/SW co-design approach (with target sparsity level ($k$) of 128) achieves up to 200 folds speedup compared to the software-only realization on a 3.40 GHz CPU. The average overhead delay for communicating between the processor (host) and accelerator contributes less than 4% to the total delay.
Table 6.4: SSketch scalability. The total processing time is linear in terms of the number of processed samples.

<table>
<thead>
<tr>
<th>Size of $N$</th>
<th>Timing ($L = 128$)</th>
<th>Timing ($L = 64$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 1K$</td>
<td>3.635s</td>
<td>2.31s</td>
</tr>
<tr>
<td>$N = 5K$</td>
<td>21.029s</td>
<td>12.01s</td>
</tr>
<tr>
<td>$N = 10K$</td>
<td>43.446s</td>
<td>24.32s</td>
</tr>
<tr>
<td>$N = 20K$</td>
<td>90.761s</td>
<td>48.52s</td>
</tr>
</tbody>
</table>
Chapter 7

Summary

This thesis proposes an end-to-end solution for efficient execution of iterative learning algorithms on big and dense data. We introduce the novel idea of extensible dictionaries which enables a parametric data transformation approach. Our transformation uses coarse-level parallelism in the data to produce versatile dictionary and sparse coefficient matrices that represent the original dense data. In order to adaptively optimize the performance, we develop methods for quantifying the cost of iterative updates on a pertinent computing platform. We then provide automated techniques for tuning the transformation based on the cost model. We demonstrate that our customizable transformation can achieve significantly more improvements over existing data transformation methods which are oblivious to the pertinent computing platform. Extensive evaluations on image classification, super resolution, and denoising applications demonstrate that we can achieve up to 2 orders of magnitude improvement in execution runtime, energy, and memory footprint. We provide user-friendly APIs that can readily be used for several learning algorithms such as regularized regression, cone optimization, and power iteration.

We also provide online data transformation methods for streaming data such those generated by surveillance cameras and network traffic. Our online transformation requires only a single-pass access to data. To provide a high-throughput solution, we take advantage of hardware acceleration.

We carefully adapt the data transformation approach with respect to the costs
and constraints of the aforementioned computing hardwares and provide high-level synthesis C-based APIs that enable data transformation on FPGAs. We propose a model that creates the most accurate transformation out of the streaming input data for a given set of user-defined properties and hardware limitations (such as memory, power, and runtime constraints). We propose constraint-driven optimization methodologies which automatically tunes the online transformation with respect to the user-defined and hardware specifications.

7.1 Future directions

The research contained in this volume opens new interesting directions that we discuss in the following.

7.1.1 Non-linear kernel matrices

Kernel or similarity matrices are essential for many state-of-the-art machine learning applications such as classification and clustering. Kernel methods transform datasets into a high-dimensional space where the classes of data become linearly separable. This is done with the help of a kernel function which measures pairwise similarities between points of data. For example, for a dataset with $N$ samples, a kernel matrix is an $N \times N$ matrix whose entries are defined as follows: $G(i, j) = K(a_i, a_j)$, where $K$ is a linear or non-linear kernel function.

While the focus of this thesis has been on the widely used linear kernel functions, i.e., $G = A^TA$, a natural next step is to adopt our customizable and adaptive solutions for non-linear kernels. Several factorization methods tackle the challenges of large-scale kernel analysis by devising dimensionality reduction methods. A kernel can be approximated as $G = W^T W$, where $W$ is a lower dimensional matrix of
size $K \times N$. Once the factorization is complete, learning algorithms are run on the approximate components. We can apply the same principles proposed in this work to tune the transformation and find matrices $D$ and $C$ such that the following objective is minimized.

$$\min \|C\|_0 \ s.t. \ \|W - DC\|_F \leq \epsilon \|W\|_F. \quad (7.1)$$

Kernel SVMs and Kernel PCA are two examples of iterative approaches that can benefit from our solution [26, 36].

### 7.1.2 Privacy-preserving machine learning

Privacy-preserving machine learning is an emerging problem, due in part to the increased reliance on the cloud for large-scale analysis. For example, consider a scenario where a clinic wants to identify risk factors for a disease using a massive database of medical records that have been gathered overtime. An interesting property of our distributed solution is that only a central processing node has access to the dictionary matrix $D$ and input $y$. All other nodes only have access to partial columns of matrix $C$ and the corresponding elements of solution vector $x$. The above property suggests that, in a privacy preserving application, a client who does not want to reveal its database (matrix $A$) and/or input query (vector $y$) to others can take the role of the central processing node in our solution. We can even increase anonymity by shuffling the rows of $C$ before sharing them. While further analysis (and possible modifications) are required to ensure the privacy, our adaptive solution might indeed enable secure yet highly efficient large-scale machine learning.
7.1.3 Low-cost machine learning on mobile devices

Despite the presence of powerful multi-core processors and graphics processing units (GPUs) in today’s mobile devices, many mobile applications rely on the cloud for performing analysis. The revelation of abundant of personal data and information to the cloud raises major security and privacy concerns. While the focus of my research has been on large-scale applications, similar principles can be employed to enable highly efficient computing on mobile embedded devices at scales that have not been possible before. We are currently developing a CPU/GPU co-design of our cost-efficient solutions on a mobile development kit (NVIDIA Jetson TK1). Our end-goal is to facilitate runtime, energy, and memory-efficient deployment of a host of iterative update algorithms, including Kernel-SVM and $\ell_1$, to enable local processing of in-demand learning applications such as augmented reality.
Bibliography


[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale


[74] R. Tibshirani, Regression shrinkage and selection via the lasso, JSTOR, 1996.


