AHEAD: Automated Framework for Hardware Accelerated Iterative Data Analysis

Ebrahim M. Songhori, Azalia Mirhoseini, Xuyang Lu, Farinaz Koushanfar
Rice University
Houston Texas, USA
{ebrahim, azalia, xl27, farinaz}@rice.edu

***Abstract*—This paper introduces AHEAD, a novel domain-specific framework for automated (hardware-based) acceleration of massive data analysis applications with a dense (non-sparse) correlation matrix. Due to non-scalability of matrix inversion, often iterative computation is used for converging to a solution. AHEAD addresses two sets of domain-specific matrix computation challenges. First, the I/O and memory bandwidth constraints which limit the performance of hardware accelerators. Second, the hardness of handling large data because of the complexity of the known matrix transformations and the inseparability of non-sparse correlations. The inseparability problem translates to an increased communication cost with the accelerators. To optimize the performance within these limits, AHEAD learns the dependency structure of the domain data and suggests a scalable matrix transformation. The transformation minimizes the memory access required for matrix computing within an error threshold and thus, optimizes the mapping of domain data to the available (bandwidth constrained) accelerator resources. To facilitate automation, AHEAD also provides an Application Programming Interface (API) so users can customize the framework to an arbitrary iterative analysis algorithm and hardware mapping. Proof-of-concept implementation of AHEAD is performed on the widely used compressive sensing and general ℓ_1 regularized least squares solvers. On a massive light field imaging data set with 4.6B non-zeros, AHEAD attains up to 320x iteration speed improvement using reconfigurable hardware accelerators compared with the conventional solver and about 4x improvement compared to our transformed matrix solver on a general purpose processor (without hardware acceleration).**

***Keywords*-Iterative Solver; Gram Matrix; Least Squares; FPGAs; Sparse Approximation; FISTA; HLS; Dense Matrix; API**

I. Introduction

One of the grand challenges of our time is finding efficient and effective methods to extract relevant information from increasingly massive and complex data. The core of several important data analysis algorithms is the iterative computation on the correlation matrix (a.k.a., *Gram matrix*) whose elements are the Hamiltonian inner products of data vectors. Examples of Gram matrix usage include kernel-based learning and classification (e.g., spectral clustering, nonlinear dimensionality reduction), several regression and regularized least squares methods (e.g., ridge regression [1], and sparse recovery [2]).

When the dataset is sparse, partitioning the data and then using the conventional content analysis algorithms can efficiently perform the required operations. It is also well documented that customized hardware acceleration and domain-specific computing can bring a significant efficiency for computing partitioned data analysis kernels [3]. Previous studies have already exploited hardware acceleration and the data sparsity to perform more efficient data partitioning, analysis, and computations, e.g., [4]. However, when the data is massive and dense (non-sparse), the currently known partitioning methods have a limited capability for handling the Gram matrix iterative updating in a timely and cost-effective way.

When performing dense and big matrix acceleration in hardware, the limiting factor is communications to and from the memory and I/O (i.e., the I/O and memory bandwidths). To address this limitation, algorithmic solutions have been proposed to trade-off communication with computation [5]. For example, communication-avoiding iterative solver [6] and communication-avoiding QR decomposition [7] replace memory communication with redundant computations. However, the existing literature have only focused on optimizing specific algorithms used in linear algebra kernels, e.g., QR decompositions. They did not address domain-specific data mining or signal processing algorithms where the properties of the domain data can guide partitioning and efficient computations.

This paper proposes Automated HardwarE Accelerated Data analysis (AHEAD), a novel automated framework for efficient analysis of big and dense datasets. For each application domain, AHEAD devises an application-specific architecture based on preliminary learning of the data properties pertinent to the task. A key concept used to quantify the matrix dependency structure is the *rank*, i.e., the size of the largest collection of linearly independent matrix columns or rows. AHEAD exploits the fact that even dense datasets in big data analysis often admit a much lower rank structure, despite their apparently massive and non-sparse Gram matrix form. AHEAD operates based on sampling and learning of the data properties at the limits of the matrix rank. This learned data structure is then used by AHEAD for building a domain-specific architecture that scalably performs iterative analysis of the data. An accompanying API aids automation and adaptation of AHEAD to any arbitrary iterative matrix analysis task. More detailed about this work can be found in [8].

Motivating Application: A wide range of machine learning methods, computer vision computations, and graph processing

This research was supported in part by an Office of Naval Research grant (ONR N00014-11-1-0885) to the ACES lab at Rice University.

operate on an *overcomplete* linear equation system. This system can be shown as $\mathbf{Ax} = \mathbf{y}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$ is an overcomplete matrix ($m \ll n$), $\mathbf{y} \in \mathbb{R}^{m \times 1}$ is an input vector, and $\mathbf{x} \in \mathbb{R}^{n \times 1}$ is a solution vector. Computing a direct solution to this problem involves inversion of the Gram matrix ($\mathbf{G} = \mathbf{A}^t\mathbf{A}$), which is utterly impractical for massively large matrices because of its super-linear complexity ($> \mathcal{O}(n^2)$).

Iterative gradient descent is an alternative approach for solving the overcomplete linear system; the approach reduces the required resources while allowing additional constraints such as sparsity on the solution. This approach is typically based upon the following iterative update in each step:

$$\mathbf{x}_i = f(\mathbf{x}_{i-1}, \mathbf{G}\mathbf{x}_{i-1}), \tag{1}$$

where $\mathbf{x}_i$ is the estimated solution in iteration i, and f is a linear kernel function. Various approaches including iterative ridge regression, LASSO [1], power methods for PCA [9], PageRank [10], and high resolution image reconstruction [11] exploit a similar iterative paradigm.

The *matrix-vector multiplication (MxV)* operation $\mathbf{G}\mathbf{x}_{i-1}$ in (1) carries almost all of the computations and memory interactions of the iterative solver. Our observation is that learning the structure of the domain data dependencies in $\mathbf{A}$ may significantly reduce the amount of the MxV computations. Our learning method approximates $\mathbf{A}$ by forming a factorization into two matrices $\mathbf{D} \in \mathbb{R}^{m \times l}$ and $\mathbf{V} \in \mathbb{R}^{l \times n}$, i.e., $\mathbf{A} \approx \mathbf{DV}$. The factorization is performed such that it minimizes the computation and memory interactions required for MxV such that the approximation error is less than a threshold. Thus, the original single MxV (on $\mathbf{G}$) can be replaced with 4 consecutive MxVs which are $\mathbf{V}^t$, $\mathbf{D}^t$, $\mathbf{D}$, and $\mathbf{V}$ with much less computations and memory interactions.

Contributions: Our contributions are as follows.

- We propose AHEAD, a novel domain-specific framework for automated hardware acceleration of massive data analysis applications with a dense correlation matrix.
- We introduce a set of new transformation methods to learn the pertinent dependencies in the underlying data. The transformations minimize the memory access required for matrix computing within an error threshold.
- We suggest using the new transformations for optimized mapping of domain-data to the available (communication constrained) accelerator resources.
- We design a high-level-description API for AHEAD such that it allows user to easily customize the framework to an arbitrary iterative algorithm.
- We integrate the AHEAD API within a commercial high level synthesis (HLS) tool to automate the hardware-based acceleration from user's high level program.
- Proof-of-concept implementation of AHEAD is performed on a widely used ℓ_1 regularized least squares algorithm for an image denoising application with a dense and massive dataset.

II. Preliminaries and Related Work

This section provides our notations, relevant background materials of our domain learning methods, and related work on accelerating matrix computation on hardware.

A. Notation and Basics

We write vectors in bold lowercase script, $\mathbf{x}$, and matrices in bold uppercase script, $\mathbf{A}$. Let $\mathbf{A}^t$ denote the transpose of $\mathbf{A}$. We use $\|\mathbf{x}\|_p = (\sum_{j=1}^{n} |\mathbf{x}(j)|^p)^{1/p}$ as the p-norm of a vector where $p \geq 1$. The ℓ_0-norm of a vector $\|x\|_0$ is defined as the number of non-zero coefficients in x. The Frobenius norm of matrix $\mathbf{A}$ is defined by $\|\mathbf{A}\|_F = \sqrt{(\sum_{i,j} |\mathbf{A}(i,j)|^2)}$. In this work, we target dense overcomplete matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ where $m \ll n$. We suppose the dataset $\mathbf{A}$ is large enough to cover all the structures in the domain, thus, it does not require additional updates.

B. Low Rank Approximation

In a setting where a dataset has a strong dependent structure, truncated singular value decomposition (SVD) provides the best low rank approximation in terms of least square error. SVD also provides the solution for principle component analysis (PCA) which seeks to find the approximate k-dimensional subspace of $\mathbf{A}$ [12]. A set of alternating methods have been proposed which target finding approximations that produce a sparse representation. Sparse PCA (SPCA) [13] and K-SVD [14] are the most widely known methods in this category. These two methods use SVD as a starting point and then continue with iterative updates on the factorized matrices to reach the desired sparsity and approximation error. The high complexity to reach the exact decomposition makes these methods unscalable for massive data.

Recently, a number of randomized methods have been proposed which aim to compute approximate factorization using a random subset of columns of $\mathbf{A}$. Column Subset Selection (CSS) method has been used in approximate least square regression [15] and Gram matrix decomposition [16]. Recent work by Dyer et al. proposed a CSS-based method to decompose a low rank dataset using orthogonal matching pursuit (OMP) [17]. This work inspired us to use CSS-based methods for scalable learning of the dependencies in the domain data. We tailor our learning methods for hardware based acceleration.

C. Hardware Acceleration of Matrix Computation

It is known that a significant performance efficiency can be gained by using customized hardware acceleration [3]. MxV operation is one of the fundamental kernels in many intense scientific computations. Various work has been dedicated to implement hardware accelerated MxV, e.g., [18], [19]. For sparse matrices, the poor locality of the coefficients reduces the performance of general-purpose processor for MxV which rely on cache hierarchy. This makes using customized hardware platform with direct access to all memory hierarchy level a sound option for high performance sparse MxV. A large set

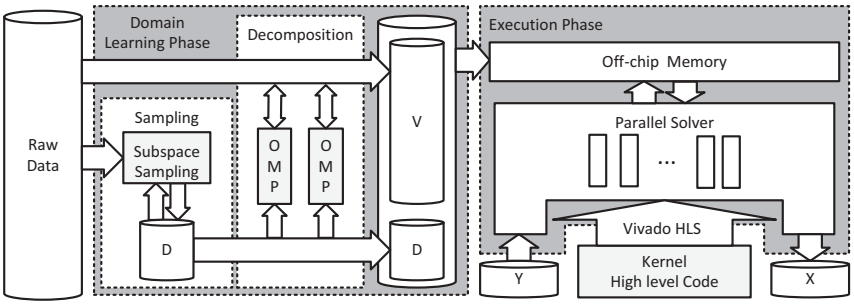


Fig. 1: Global flow of AHEAD consists of learning and execution phases. The dependencies inside the data are learned using a CSS-based method and a sparse decomposition. The resulting learned data is stored in memory. Parallel solver can thus separately operates on input vectors. After finding the solution vector using an iterative method, the solution is sent to the host.

of work have been proposed to improve the performance of sparse MxV using FPGA platform e.g., [20], [21].

These improvements have made accelerating full flow of scientific algorithms feasible on hardware platforms, e.g., linear algebra solver [22] and SVD decomposition [23], [24]. A few work on sparse linear-algebra algorithms has also been done using hardware acceleration such as [25] which implemented a linear regression solver $\mathbf{Ax} = \mathbf{y}$ on FPGA for sparse dataset $\mathbf{A}$. In the present work, we propose a framework for general iterative solver on *large and dense* matrices. We use a domain-specific approach to leverage dependencies within the data to improve the performance of the accelerator.

III. AHEAD Overview

Fig. 1 demonstrates the global overview of AHEAD. Steps for accelerating an iterative solver are as follows:

1) AHEAD learns the dependencies in the data domain by factorizing the dataset matrix ($\mathbf{A} \approx \mathbf{DV}$) based on user's parameters and underlying platform's constraints in an offline phase (Section IV).
2) AHEAD stores the learned data into an off-chip memory on the hardware accelerator platform.
3) Using an HLS tool, AHEAD compiles user's high level description of the iterative solver in AHEAD API and then embeds it into the solver accelerator design (Section V).
4) In the execution phase, first, the parallel solvers receive input vectors from the host. Next, they start the iterative updates until the convergence. Then, they send the solution back to the host and wait for new inputs.

IV. Domain Learning

A wide range of big datasets which have been generated by real systems, e.g., digital images, medical signals, and Internet users' activities, have been shown to admit a low rank structure. In this work, we use an offline learning phase in order to exploit low rank structure of data. This can be done by a low rank approximation like $\mathbf{A} \approx \mathbf{DV}$. After this approximation, one can repeatedly solve the linear equation problem ($\mathbf{DV}x = y$) using $\mathbf{V}$ and $\mathbf{D}$ for different inputs y with lower computation and storage requirements. For most applications, the dataset can be large enough to cover all the structures of the application domain such that the learning can be done once and offline. The rest of this section is a brief description of the set of CSS-based methods for the low rank approximation in AHEAD.

A. Sparse Matrix Decomposition

Factorizing a matrix $\mathbf{A}$ while enforcing sparsity on the factorized component $\mathbf{V}$ can be found in many applications like feature selection, data classification, and dictionary learning problems in machine learning. This problem can be represented by the equation below:

$$\underset{\mathbf{D},\mathbf{V}}{\text{minimize}} \ \|\mathbf{V}\|_0 \ \text{ subject to } \ \|\mathbf{A} - \mathbf{DV}\|_F \leq \delta, \quad (2)$$

where δ is a learning parameter. A few well-known examples of algorithms tackling this problem, are SPCA and K-SVD which have a super-linear complexity. We have been motivated by recent work [17] which uses a greedy CSS approach to solve *exact feature selection* problem. We now describe our three proposed methods for the learning phase in AHEAD. For each method there exists a different trade-off between sparsity levels and approximation error.

1) Random Sparse Matrix Decomposition (rSMD): In rSMD method, matrix $\mathbf{D}$ consists of randomly sub-sampled columns of $\mathbf{A}$. In order to enforce sparsity on representation matrix $\mathbf{V}$, rSMD uses Orthogonal Matching Pursuit (OMP) which is greedy sparse approximation. OMP is used locally to determine sparse representation of each column of $\mathbf{A}$ in the space spanned by $\mathbf{D}$.

2) Adaptive Sparse Matrix Decomposition (aSMD): Although the randomized subspace selection reduces the complexity in rSMD, it may decrease the robustness of approximation. aSMD is an extension of rSMD which, unlike rSMD, gradually selects column of $\mathbf{A}$ and adds them to $\mathbf{D}$. At each step, aSMD calculates *projection errors* for all columns and selects new ones based on the distribution of the errors. This projection error, $W(\cdot)$, is defined as:

$$W(\mathbf{A}_i) = \frac{\|\mathbf{D}(\mathbf{D}^t\mathbf{D})^{-1}\mathbf{D}^t\mathbf{A}_i - \mathbf{A}_i\|_2}{\|\mathbf{A}_i\|_2}, \quad (3)$$

Algorithm 1 : **sSMD**

Input: $\mathbf{A}$, number of selected column l, selection error tolerance α, and OMP error tolerance ϵ.

Output: $\mathbf{D}$ and sparse $\mathbf{V}$ where $\mathbf{A} \approx \mathbf{DV}$.

$\mathbf{D} \leftarrow \emptyset$, $j \leftarrow 0$
for $i = 1$ **to** n **do**
  if $W(\mathbf{A}_i) > \alpha$ **and** $j < l$ **then**
    $D_j \leftarrow \frac{\mathbf{A}_i}{\sqrt{\|A_i\|_2}}$, $V_{ij} \leftarrow \sqrt{\|A_i\|_2}$, $j \leftarrow j + 1$
  else
    $V_i \leftarrow omp(\mathbf{D}, \mathbf{A}_i, \epsilon)$
  end if
end for

where $\mathbf{A}_i$ is the ith column of matrix $\mathbf{A}$, $\mathbf{D}$ is the current selected columns, and $\mathbf{D}(\mathbf{D}^t\mathbf{D})^{-1}\mathbf{D}^t$ is a projection matrix into the space spanned by $\mathbf{D}$. aSMD selects the columns of $\mathbf{A}$ in Δ steps. Δ is a learning parameter specified by the user.

3) Serial Sparse Matrix Decomposition (sSMD): Both rSMD and aSMD require access to the entire data for sampling. Due to the massive size of $\mathbf{A}$, their memory footprints can be unbearable for embedded devices. sSMD method can locally decide about adding each column of $\mathbf{A}$ to $\mathbf{D}$. Thus, this method only requires storing a single column and $\mathbf{D}$ matrix at a time. sSMD, like aSMD, uses the weight function defined in (3) to sample columns of $\mathbf{A}$. Algorithm 1 demonstrates the pseudocode of sSMD where ϵ is threshold error in OMP method and α is projection threshold error.

V. Execution Phase and Hardware Design

We use a combination of HLS and hardware description language (HDL) design to implement AHEAD's hardware iterative solver. The reason for adopting an HLS tool is to allow the user to employ high-level programming in AHEAD API. In this section, we provide a detailed description of our hardware architecture.

A. Solver Kernel

As a result of the domain learning phase, the dataset $\mathbf{A}$ is factorized to two matrices $\mathbf{D}$ and $\mathbf{V}$. Thus, the iterative update in (1) changes accordingly from two MxVs: $\mathbf{y} = \mathbf{Ax}$ and $\mathbf{x} = \mathbf{A}^t(\mathbf{y}' - \mathbf{y})$ to four MxVs: $\mathbf{p} = \mathbf{Vx}$, $\mathbf{y} = \mathbf{Dp}$, $\mathbf{p} = \mathbf{D}^t(\mathbf{y}' - \mathbf{y})$, and $\mathbf{x} = \mathbf{V}^t\mathbf{p}$. In both cases, the updates are followed by the linear function f.

Due to the massive size of both matrices $\mathbf{D}$ and $\mathbf{V}$, they must be stored in an off-chip memory. In order to push the performance to the limits of the platform, our hardware iterative solvers, *Solver Kernels*, are designed to maximize the memory bandwidth utilization for reading these two matrices. Solver Kernels use burst-mode memory reads to leverage all the available memory bandwidth. We use compressed sparse row format to store sparse matrix $\mathbf{V}$. This makes burst-mode reads available even for the sparse $\mathbf{V}$. Fig. 2a shows the overall data-path of the Solver Kernel. The linear function module f is given by user in AHEAD API using high level C/C++ description. AHEAD synthesizes it by an HLS tool and embeds it in the solver implementation.

B. Parallel Solver

The pattern of accesses to the off-chip memory is independent of the input vector $\mathbf{y}$ for the Solver Kernel. This indicates that a coarse-grain parallel architecture can be the most efficient way to maximize the utilization of the hardware while retaining the memory bandwidth utilization. In this architecture, memory read data is broadcast to all the Solver Kernels. Fig. 2b shows the overall data-path of the *Parallel Solver* design.

VI. Evaluation

A. Experiment Setup

We use Xilinx Virtex-6 FPGA ML605 Evaluation Kit as a reconfigurable hardware accelerator and a system with Linux operating system and Intel core i7 processor as our general purpose processor platform.

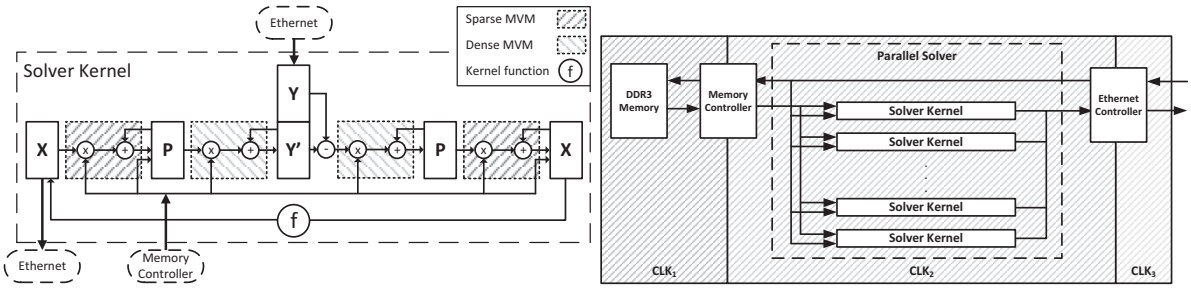
B. Case Study: Light Field Denoising using FISTA

In our evaluation, we operate on the domain data from *light field* imaging dataset extract from Stanford Light Field Archive [26]. Light field data consists of images from an array of cameras taken from slightly different viewpoints [27]. We use a light field dataset with $256k$ high resolution light field patches. Each patch is extracted from a 17×17 array of 8×8 windows. Thus, the final dataset is a dense $18k \times 256k$ matrix which has 4.6 billion non-zero coefficients.

As an application, we consider solving light field denoising problem with ℓ_1 regularized least-squares [28]. Several iterative methods have been proposed to tackle this problem, e.g., gradient descent, Jacobi, ISTA, and FISTA. In this work, we consider FISTA or A Fast Iterative Shrinkage-Thresholding Algorithm to solve the denoising problem because it is known to be a high performance solver with a rapid convergence rate [11].

C. FPGA Implementation

We use Xilinx Vivado HLS to translate C/C++ iterative update description of FISTA inside the AHEAD API to Verilog code. The resulting system is successfully placed and routed on three different clock domains on Virtex-6 FPGA ($CLK_1 = 400MHz$, $CLK_2 = 200MHz$, and $CLK_3 = 125MHz$). Table I shows the FPGA resource utilization for implementing FISTA using AHEAD system with 1, 3, and 6 Solver Kernels. The result includes the infrastructure overheads such as memory and Ethernet controller. Here, the number of Solver Kernels in the system is limited by the available block memories in Virtex-6 FPGA.



(a) (b)

Fig. 2: (a) Solver Kernel receives $\mathbf{y}$ from the host and starts with a random solution $\mathbf{x}$. In each iteration, it computes $\mathbf{p} = \mathbf{V}\mathbf{x}$, $\mathbf{y}' = \mathbf{D}\mathbf{p}$, $\mathbf{p} = \mathbf{D}^t(\mathbf{y}' - \mathbf{y})$, and $\mathbf{x} = f(\mathbf{x}, \mathbf{V}^t\mathbf{p})$ where f is linear function based on user's solver algorithm. (b) Each input $\mathbf{y}$ is assigned to a Solver Kernel to increase the utilization. Regardless of input $\mathbf{y}$, the memory accesses to $\mathbf{D}$ and $\mathbf{V}$ matrices have the same pattern. Thus, the coefficients of these matrices are broadcasted to all the Solvers Kernel.

TABLE I: Virtex-6 resource usage (utilization) versus number of Solver Kernels.

# of Solver Kernels	1	3	6	Available
Slice Registers	15k(5%)	25k(8%)	40k(13%)	301k
Slice LUTs	13k(9%)	25k(17%)	44k(29%)	151k
DSP48E1s	40(5%)	120(15%)	240(31%)	768
RAMB36E1	70(17%)	182(44%)	350(84%)	416

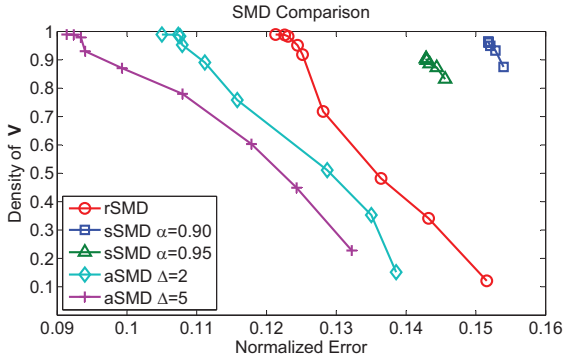


Fig. 3: Density of $\mathbf{V}$ versus the normalized approximation error is shown. Tolerating larger error in learning allows the decomposed matrix to be sparser in general. In sSMD, the degree of freedom is limited due to greedy local CSS.

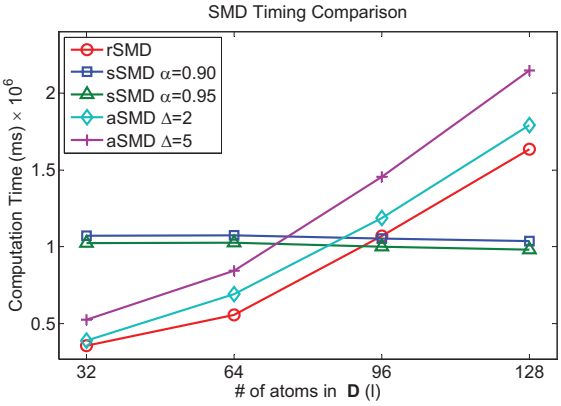


Fig. 4: Computation time of the proposed factorization method grows with increase of the learning factor l. The timing performance of sSMD remains steady as l increases.

D. Domain Learning Results

The methods in the learning phase are implemented in C++ using Eigen library and executed on the host Intel processor. This phase is required to be executed only once at the beginning to generate $\mathbf{D}$ and $\mathbf{V}$. A user may choose among the three available approximation methods: rSMD, aSMD, and sSMD and their parameters.

Fig. 3 shows the relation between error and sparsity of the learned data for rSMD, aSMD, and sSMD. In order to trade-off between error and sparsity, one can alter error threshold, ϵ, to achieve the desired sparsity. As it appears, more steps (Δ) in aSMD and larger selection error threshold (α) in sSMD result in smaller errors and sparser representation matrices.

Fig. 4 shows the computation time for different learning methods versus l. As it appears, performance of sSMD remains steady for large l, while computation times for the other two methods increase with l. As expected, larger Δ results in slower learning process in aSMD. Furthermore, α in sSMD does not have a significant impact on its timing performance.

E. Execution Phase Results

We realize the FISTA algorithm for the learned data on a multi-core general purpose processor in order to examine the benefit of our hardware acceleration; we call this implementation CPU Factorized (CF). In addition, we implement conventional FISTA without learning using the original dataset ($\mathbf{A}^t\mathbf{A}$) on a multi-core CPU; we call this implementation CPU Conventional (CC). We call the realization of FISTA using AHEAD API, FPGA Factorized (FF).

We leverage Eigen and OpenMP libraries to realize FISTA in a parallel manner for the last two implementations. The Eigen library exploits Intel Stream SIMD Extension (SSE) instructions on Intel processors to enhance the performance of intensive matrix computation. Using Eigen library pushes the performance of these two implementations to the limits of

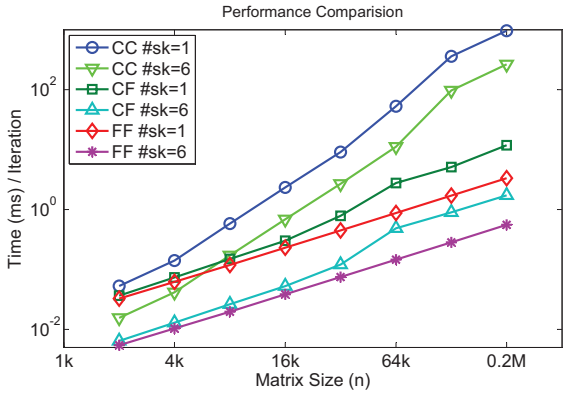


Fig. 5: Time per iteration for solver implementations with different number of Solver Kernels versus size of data is shown. CC method directly uses $\mathbf{A}$, while CF and FF leverage the learned decomposed matrices ($\mathbf{D}$ and $\mathbf{V}$).

general purpose processors. In all the experiments on both FPGA and CPU, IEEE 754 single precision floating point format is used for computation and storing data.

We use time per iteration as a metric to compare the speed of solvers. Fig. 5 illustrates the performances of CC, CF, and FF with 1 and 6 Solver Kernels versus dataset size n. In this experiment, we use rSMD with $l = 64$ and $\epsilon = 0.1$ on the data. By increasing n, the factorized methods dramatically outperform the CC solvers. For example with using 1 Solver Kernel and $n = 256k$, updates for CC is almost $80\times$ slower than that for CF. CF is also $4\times$ slower than the FF at this point. The $80\times$ gain arises from minimization of costly memory and I/O interactions which happened in the domain learning phase. The $4\times$ gain is caused by use of a customized and domain-specific hardware accelerator instead of a general purpose processor. Parallelizing 6 Solvers Kernel results in almost linear ($6\times$) improvement in FF, while for CF and CC, it is almost $4.2\times$ and $3.6\times$ respectively. The FF linear scalability is a direct result of customized design in hardware.

VII. Conclusion

In this paper, AHEAD an automated domain-specific framework for iterative algorithms using hardware accelerators is presented. AHEAD utilizes an offline dependency learning to factorize the dataset into set of matrices with lower dependencies. The learning is used to optimize the hardware acceleration by reducing the computations and memory interations. An integerated API within Vivado HLS tool is proposed so users can implement arbitrary iterative analysis algorithms and easily map them to the undelying hardware accelerators. As a case study, we implement a widely used iterative ℓ_1 least squares solver on a massive imaging dataset for denosing application. Our experiments show that the proposed system using Xilinx Virtex-6 FPGA effectively improves the performance of the iterative method compared to conventional methods and software implementations.

References

[1] R. Tibshirani, "Regression shrinkage and selection via the lasso," *JSTOR*, 1996.

[2] S. S. Chen, D. L. Donoho, and M. A. Saunders, "Atomic decomposition by basis pursuit," *SIAM*, vol. 20, no. 1, pp. 33–61, 1998.

[3] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," *IEEE Design and Test of Computers*, 2011.

[4] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA based sparse matrix vector multiplication using commodity dram memory," in *FPL*, 2007.

[5] A. Rafique, N. Kapre, and G. Constantinides, "Application composition and communication optimization in iterative solvers using FPGAs," in *FCCM*, 2013.

[6] M. Hoemmen, "Communication-avoiding krylov subspace methods," Ph.D. dissertation, University of California, 2010.

[7] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding qr decomposition for gpus," in *IPDPS*. IEEE, 2011, pp. 48–58.

[8] E. M. Songhori, "ShuFFLE: Automated framework for hardware accelerated iterative big data analysis," Master's thesis, Rice University, 2014.

[9] M. Journée, Y. Nesterov, P. Richtárik, and R. Sepulchre, "Generalized power method for sparse principal component analysis," *JMLR*, 2010.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[11] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM*, 2009.

[12] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2005.

[13] H. Zou, T. Hastie, and R. Tibshirani, "Sparse principal component analysis," *JCGS*, vol. 15, no. 2, pp. 265–286, 2006.

[14] M. Aharon, M. Elad, and A. Bruckstein, "k -svd: An algorithm for designing overcomplete dictionaries for sparse representation," *Signal Processing, IEEE Transactions on*, 2006.

[15] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, "Clustering large graphs via the singular value decomposition," *Machine learning*, vol. 56, no. 1-3, pp. 9–33, 2004.

[16] P. Drineas and M. W. Mahoney, "On the nyström method for approximating a gram matrix for improved kernel-based learning," *JMLR*, 2005.

[17] E. L. Dyer, A. C. Sankaranarayanan, and R. G. Baraniuk, "Greedy feature selection for subspace clustering," *J. Mach. Learn. Res.*, 2013.

[18] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on fpgas," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004.

[19] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 2005.

[20] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007.

[21] M. DeLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for fpgas," in *FPGA*. ACM, 2005.

[22] J. Gonzalez and R. C. Núñez, "Lapackrc: Fast linear algebra kernels/solvers for fpga accelerators," in *Journal of Physics: Conference Series*. IOP Publishing, 2009.

[23] W. Ma, M. Kaye, D. Luke, and R. Doraiswami, "An fpga-based singular value decomposition processor," in *Electrical and Computer Engineering, 2006. CCECE'06. Canadian Conference on*. IEEE, 2006, pp. 1047–1050.

[24] J. R. Cavallaro and F. T. Luk, "Cordic arithmetic for an svd processor," *Journal of parallel and distributed computing*, 1988.

[25] H. ElGindy and Y.-L. Shue, "On sparse matrix-vector multiplication with fpga-based system," in *FCCM*. IEEE, 2002.

[26] "The light field archive," lightfield.stanford.edu, 2008.

[27] K. Marwah, G. Wetzstein, Y. Bando, and R. Raskar, "Compressive light field photography using overcomplete dictionaries and optimized projections," *ACM TG*, 2013.

[28] B. Wilburn, "High-performance imaging using arrays of inexpensive cameras," *PhD Dissertation, Stanford University*, 2005.