Hardware Accelerator for Analytics of Sparse Data

Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, Debbie Marr
Intel Corporation
Hillsboro, OR USA

Abstract — **Rapid growth of Internet led to web applications that produce large unstructured sparse datasets (e.g., texts, ratings). Machine learning (ML) algorithms are the basis for many important analytics workloads that extract knowledge from these datasets. This paper characterizes such workloads on a high-end server for real-world datasets and shows that a set of sparse matrix operations dominates runtime. Further, they run inefficiently due to low compute-per-byte and challenging thread scaling behavior. As such, we propose a hardware accelerator to perform these operations with extreme efficiency. Simulations and RTL synthesis to 14nm ASIC demonstrate significant performance and performance/Watt improvements over conventional processors, with only a small area overhead.**

Keywords—Hardware accelerator, analytics, machine learning

I. INTRODUCTION

The proliferation of Internet has led to large scale web applications, such as social networks, online shopping, etc. These applications produce a tremendous amount of datasets that are unstructured and sparse in nature. Machine learning (ML) algorithms are the basis for many data analytics workloads that process such datasets and extract knowledge from them for various purposes (e.g., categorize news articles, identify spam emails, recommend items to users).

These workloads are typically deployed on large-scale servers in data centers, which demand extreme energy efficiency in addition to high performance. To this end, recent systems (e.g., IBM® PowerEN, Oracle® SPARC M7) integrate hardware accelerators alongside general purpose processors to deliver significant execution efficiency for specific application domains (e.g., edge network, database).

This paper investigates such an accelerator-based system for the aforementioned domain of sparse data analytics based on ML. First, we characterized analytics workloads that use popular ML algorithms to process real-world datasets on an Intel® Xeon server. We found that runtimes are dominated by a set of sparse matrix operations, some of which are different than the typical sparse matrix dense vector multiplication found in high-performance computing (i.e., sparse matrix sparse vector multiplication, scaling sparse matrix elements and updating a dense vector).

Furthermore, we found that these operations are bandwidth dependent. Since the input data is sparse, these operations perform only a small amount of operation for every data they access (low compute-per-byte). Efficiently accessing a large sparse data is the key to performance of these operations. However, our characterization study found that software implementations of these operations were not able to saturate the system memory bandwidth.

To address this issue, we propose a hardware accelerator that dramatically improves the execution efficiency of these operations. Unlike existing accelerators that only support a specific matrix operation [12,13,19,21], a specific machine learning algorithm [14,15,21], or operate only on dense datasets [7-11], our accelerator utilizes a unified design to efficiently support all matrix and vector operations that are critical for important ML algorithms in analytics. This makes our accelerator widely applicable to the analytics domain.

An evaluation study utilizing cycle-accurate simulator and RTL synthesis to a state-of-the-art 14nm ASIC demonstrates that our accelerator can improve performance over multi-threaded software implementations on a high-end server, consumes much less energy than a general-purpose processor, and requires only a small amount of chip area. To our knowledge, ours is the only unified accelerator we are aware of that efficiently supports all sparse matrix operations used by a diverse set of analytics applications.

The rest of the paper is organized as follows. Next section provides background on machine learning for analytics of sparse data. Section III presents our characterization study on an Intel® Xeon server system. Section IV details the important matrix compute patterns we found in the characterization study. Section V presents the proposed hardware accelerator. Our evaluation study of the accelerator is provided in Section VI. Section VII and VIII offer related work and concluding remarks, respectively.

II. MACHINE LEARNING FOR ANALYTICS

Machine learning (ML) is used in analytics to *create a model* from a known set of data samples (i.e., training/learning phase), which is then used to *make predictions* on new data samples (i.e., testing/prediction/scoring phase). This paper focuses on training, since it is an iterative time consuming process with inter-iteration dependencies that is difficult to scale out. For example, in [17], URLs labeled with their reputation (malicious or normal site) were used to train a model that predicts reputation of new URLs.

A sample consists of features that describe the characteristics of the data along with a label for that data. In our example, a feature could be whether there is an unusual text in the URL, such as a .com that is not a top-level domain (TLD), e.g., www.ebay.com.phishy.biz instead of www.ebay.com. And a label would be the known classification of the URL reputation.

The set of data samples can be naturally organized as a matrix, where each row represents a sample and each column represents a feature. An ML algorithm may need to operate on

978-3-9815370-6-2/DATE16/©2016 EDAA

a particular feature (i.e., a matrix column) or on a particular sample (i.e., a matrix row). Therefore, both row-oriented as well as column-oriented matrix formats are desirable.

While there can be many features in a sample, not all may present. In our example, normal URLs have .com as its TLD, so the feature indicating the presence of .com in the middle of the URL would be zero. This leads to sparsity in datasets. In practice, there are many sparse real world datasets [18].

While we only describe so far a particular example in analyzing URL reputation, there are many uses of ML for analytics. The next section describes our characterizations of a range of ML workloads for analytics.

III. Workloads Characterization

A. Workloads Under Study

We studied several categories of workloads that are widely used in the training (or learning) phase of analytics pipeline.

Classification. Classification is the problem of identifying to which of a set of categories a new (unknown) sample belongs, based on known training data samples. We studied kernelized support vector machines (K-SVM), linear support vector machine (L-SVM) and logistic regression (LogReg) algorithms using LibSVM [1] and LibLinear [2] packages.

Regression. Classifiers and regression engines have the same compute pattern. The main difference is the optimization problem being solved. Classifiers require integer solutions whereas regression models could have real numbered solutions for the class labels. We used LibSVM and LibLinear software packages for regression workload analysis.

Recommendation Engine. Recommender systems are widely used to predict the rating that a user would give to an item. For our work, we studied a sparse linear method (SLIM) that implements a set of top-N recommendation method [4] and alternating least squares (ALS) engine which models a collaborative filtering method. We used SLIM [4] and libfm software packages [5].

Clustering. Clustering models aim to partition samples into clusters where each sample belongs to a cluster with the closest distance. Clustering methods are popular unsupervised learning models and for our analysis we studied the K-means algorithm using SofiaML software package [6].

B. Datasets

Table I lists the datasets used in our study from [1,4,16,17]. The datasets we use are real-world, big (the entire dataset would not fit in on-chip memory), and sparse (too large and inefficient to store and process as dense data due to very large number of zero values).

E2006 dataset contains the term frequencies of unigrams and bigrams of 10-K reports from thousands of publicly traded U.S. companies. RCV is a corpus of newswire stories from Reuters. Webspam unigram is spam corpus that contains both positive (spam) and negative (non-spam) samples. The Gamevideo dataset contains feature and class labels for game videos uploaded on YouTube. URL is a malicious URL detection dataset from the example in Section II. MovieLens is a dataset for recommendation engines and consists of movie ratings given by users.

TABLE I. Datasets under study. They are large, sparse, and high-dimensional. K=1000 and M=1 Million in the table. % non-zeros is the number of non-zero elements in the matrix divided by the total number of elements in the matrix.

Name	# of samples	Avg. sample length	% non-zeros	Size on disk
E2006	16K	1242	0.83%	485 MB
RCV	677K	74	0.15%	1.2GB
Webspam	245K	86	33.38%	268MB
Gamevideo	97K	221	22%	225MB
URL	1.6M	117	0.003%	1.5GB
MovieLens	70K users, 10K movies, 10M ratings			253MB

C. Characterizations

Our workload characterization is done on an Intel® Xeon server (E5-2679 v2) consisting of 12 cores, 30 MB shared L3, 2.7 GHz nominal core frequency, 128 GB DDR3 memory and 60 GB/s maximum memory bandwidth. We use Intel® VTune and gprof tools for hotspot characterization.

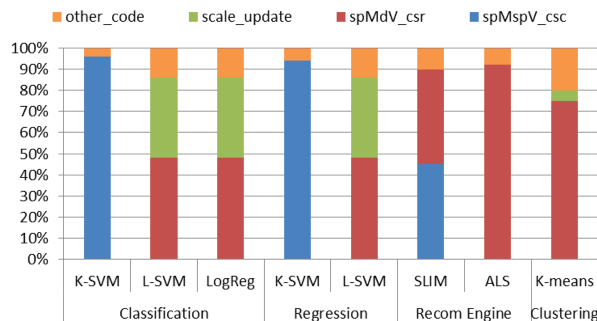


Fig. 1. Runtime breakdown of workloads under study. Majority of time is spent on matrix compute patterns. Each bar represents an average across all of the datasets we studied.

Figure 1 summarizes the workload characteristic. We found that over 80% of runtime is spent on sparse matrix compute patterns. They are (1) sparse matrix dense vector multiplication on a row-oriented data (spMdV_csr), (2) multiplication of a sparse matrix against a sparse vector on a column-oriented data (spMspV_csc), and (3) a scale and update (scale_update) operation where sparse matrix elements are multiplied against scaling factors and used to update a dense vector. Since these compute patterns dominate runtime, we focus on optimizing and accelerating them in the rest of the paper. We detail each of these patterns in the next section.

IV. Important Matrix Compute Patterns and Their Performance Considerations

The runtime-dominating compute patterns we found in our characterization study are variations of matrix multiplication against a vector in row-oriented and column-oriented fashion.

They work on well-known matrix formats: compressed sparse row (CSR) and compressed sparse column (CSC). Figure 2(a) depicts an example of a multiplication between a sparse matrix A against a vector x to produce a vector y. Figure 2(b) and 2(c) illustrate the CSR and CSC representation of matrix A. Figure 2(d), 2(e), and 2(f) show the pseudo code of each compute pattern, which we describe in detail next.

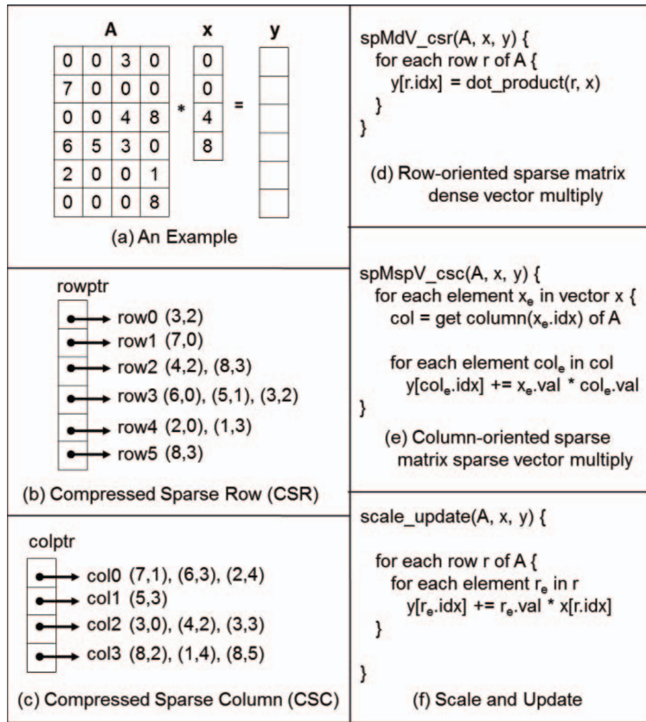


Fig. 2. The compute patterns and sparse matrix formats.

A. Row-Oriented Sparse Matrix Dense Vector Multiplication (spMdV_csr)

This is a well-known compute pattern that is important in many application domains such as high-performance computing. Here, for each row of matrix A, a dot product of that row against vector x is performed, and the result is stored in the y vector element pointed to by the row index.

This computation is used in an ML algorithm that performs analysis across a set of samples (i.e., rows of the matrix). It is used in technique such as "mini-batch" [6]. There are also cases where ML algorithms perform only a dot product of a sparse vector against a dense vector (i.e., an iteration of the spMdV_csr loop), such as in the stochastic variants of learning algorithms [20].

A known factor that can affect performance on this computation is the need to randomly access sparse x vector elements in the dot product computation. For a conventional server system, when the x vector is large, this would result in irregular accesses (gather) to memory or last level cache.

A possible way to address this is to divide matrix A into column blocks and the x vector into multiple subsets (each corresponds to an A matrix column block). The block size can be chosen so that x vector subset can fit on chip. Hence, random accesses to it can be localized on-chip.

B. Column-Oriented Sparse Matrix Sparse Vector Multiplication (spMspV_csc)

This pattern that multiplies a sparse matrix against *a sparse vector* is not as well-known as spMdV_csr. However, we do find it important in the ML algorithms we studied. It is used when an algorithm works on a set of features, which are represented as matrix columns in the dataset (hence, the need for column-oriented matrix accesses).

In this compute pattern, each column of the matrix A is read and multiplied against the corresponding non-zero element of vector x. The result is used to update partial dot products that are kept at the y vector. After all the columns associated with non-zero x vector elements have been processed, y vector will contain the final dot products.

While accesses to matrix A is regular (i.e., stream in columns of A), the accesses to the y vector to update the partial dot products is irregular. The y element to access depends on the row index of the A vector element being processed.

To address this, the matrix A can be divided into row blocks. Consequently, the vector y can be divided into subsets corresponding to these blocks. This way, when processing a matrix row block, it only needs to irregularly access (gather/scatter) its y vector subset. By choosing the block size properly, y vector subset can be kept on-chip.

C. Scale and Update (scale_update)

This pattern is typically used by ML algorithms to apply scaling factors to each sample in the matrix and reduced them into a set of weights, each corresponding to a feature (i.e., a column in A).

Here, x vector contains the scaling factors. For each row of matrix A (in CSR format), the scaling factors for that row are read from the x vector, and then applied to each element of A in that row. The result is used to update the element of y vector. After all rows have been processed, the y vector contains the reduced weights.

Similar to prior compute patterns, the irregular accesses to the y vector could affect performance when y is large. Dividing matrix A into column blocks and y vector into multiple subsets corresponding to these blocks can help localize the irregular accesses within each y subset.

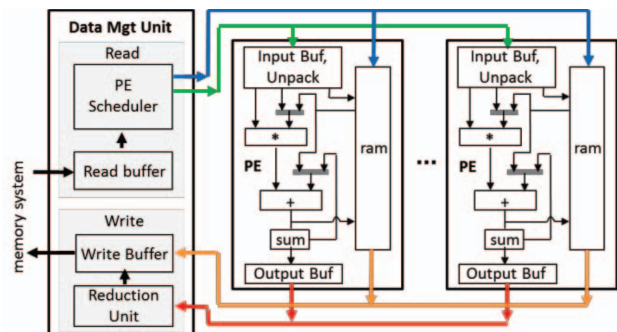


Fig. 3. The architecture of the proposed accelerator. It contains a data management unit (DMU) that accesses data via an external interface to the memory system and delivers data into array of processing elements (PEs).

V. HARDWARE ACCELERATOR DESIGN

A. System Overview

We developed a hardware accelerator that can efficiently perform the compute patterns discussed in the previous section. The accelerator is a hardware IP block that can be integrated with general purpose processors, similar to those found in existing accelerator-based solutions (e.g., IBM® PowerEN, Oracle® M7). It independently accesses memory through interconnect shared with the processors to perform the compute patterns. It supports any arbitrarily large matrix datasets that reside in off-chip memory.

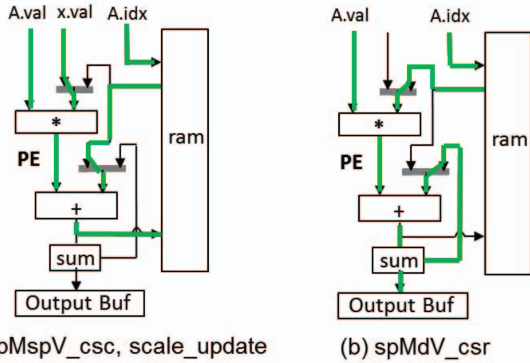


Fig. 4. A processing element (PE) can be configured to support the set of compute patterns identified in the characterization study.

B. Accelerator Architecture

The accelerator contains a data management unit (DMU) and a set of processing elements (PEs), as in Figure 3. It supports the matrix blocking schemes discussed in the previous section (i.e., row and column blocking) to support any arbitrarily large matrix data. The accelerator is designed to process a block of matrix data. Each block is further divided into sub-blocks processed in parallel by PEs.

The DMU reads the matrix rows or columns from memory into its read buffer, which is then dynamically distributed by the PE scheduler across PEs for processing. It also writes results to memory.

Each PE is responsible for processing a matrix sub-block. A PE contains an on-chip RAM to store the vector that needs to be accessed randomly (i.e., a subset of x or y vector, as described in previous section). It also contains a floating point multiply-accumulate (FMA) unit, unpack logic to extract matrix elements from input data, and a sum register to keep the accumulated FMA results.

The accelerator achieves extreme efficiencies because (1) it places irregularly accessed (gather/scatter) data in on-chip PE RAMs, (2) it utilizes hardware PE scheduler to ensure PEs are well utilized, and (3) unlike with general purpose processor, the accelerator consists of only the hardware resources that are essential for sparse matrix operations. In overall, the accelerator efficiently converts the available memory bandwidth provided to it into performance.

Scaling of performance can be done by employing more PEs in an accelerator block to process multiple matrix sub-blocks in parallel, and/or employing more accelerator blocks (each has a set of PEs) to process multiple matrix blocks in parallel. In our evaluation, we consider a combination of these. The number of PEs and/or accelerator blocks should be tuned to match the memory bandwidth.

C. Accelerator Operations

The accelerator can be programmed through a software library (similar to Intel® Math Kernel Library). Such library prepares the matrix data in memory, sets control registers in the accelerator with information about the computation (e.g., computation type, memory pointer to matrix data), and starts the accelerator. Then, the accelerator independently accesses matrix data in memory, performs the computation, and writes the results back to memory for the software to consume.

The accelerator handles the different compute patterns by setting its PEs to the proper datapath configuration, as depicted in Figure 4. The accelerator operation to perform each compute pattern is detailed below.

For spMspV_csc, the initial y vector subset is loaded in to PE's RAM by the DMU. It then reads x vector elements from memory. For each x element, the DMU streams the elements of the corresponding matrix column from memory and supplies them to the PE. Each matrix element contains a value (A.val) and an index (A.idx). A.idx points to the y element to read from PE's RAM. The DMU also provides the x vector element (x.val) that is multiplied against A.val by the FMA unit. The result is used to update the y element in the PE's RAM pointed to by A.idx. Note that even though not used by our workloads, the accelerator also supports column-wise multiplication against a *dense* x vector (spMdV_csc) by processing *all* matrix columns instead of only a subset (since x is dense).

The scale_update operation is similar to the spMspV_csc, except that the DMU reads the rows of an A matrix represented in a CSR format instead of a CSC format.

For the spMdV_csr, the x vector subset is loaded in to the PE's RAM. DMU streams in matrix row elements (i.e., {A.val,A.idx} pairs) from memory. A.idx is used to read the appropriate x vector element from RAM, which is multiplied against A.val by the FMA. Results are accumulated into the sum register. The sum register is written to the output buffer each time a PE sees a marker indicating an end of a row, which is supplied by the DMU. In this way, each PE produces a sum for the row sub-block it is responsible for. To produce the final sum for the row, the sub-block sums produced by all the PEs are added together by the Reduction Unit in DMU (see Figure 3). The final sums are written to the output buffer, which DMU then writes to memory.

VI. EVALUATION

A. Selecting Accelerator Design Parameters

We first use our in-house cycle-accurate simulator to explore the design space of our accelerator. We vary the number of PEs and size of buffers in the accelerator to pinpoint an accelerator design instance to study further. We observe that

as we add more PEs, the accelerator can indeed consume more bandwidth and performance scales accordingly.

Since we want to place our accelerator alongside the processor core and tap into the core's interface to the memory system, we need to target our accelerator design to match the available bandwidth provided by that interface.

We envision integrating the accelerator into the server system we used in our characterization study. We measured how much memory bandwidth a processor can consume in our system using Zsmith benchmarking tool. We found that a processor can consume up to ~15GB/s in practice. Furthermore, the aggregate memory bandwidth available to the 12 cores in the system is ~60GB/s.

In our design exploration, we found that an accelerator block with 4 PEs can saturate ~15GB/s. Therefore we selected this configuration to study further. We also consider placing four accelerator blocks in the system to match the ~60GB/s aggregate system memory bandwidth available.

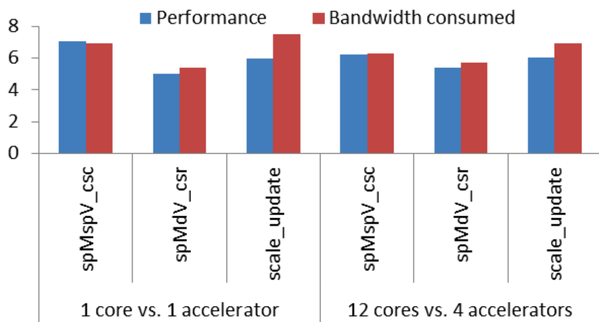


Fig. 5. Average improvements in performance and bandwidth consumed achieved by the proposed accelerator. Comparisons between a processor core and a single accelerator as well as between 12 processor cores and 4 accelerators are provided. The proposed accelerator consumes bandwidth more efficiently, resulting in improved performance over software on conventional processors.

B. Performance of Compute Patterns

Next, we study the performance of our accelerator over software implementations. Figure 5 shows the average improvements in performance and bandwidth consumed across datasets for 1 accelerator block (with 4 PEs) and 4 accelerator blocks (16 PEs total) against single and multithreaded software implementations (with up to 16 threads).

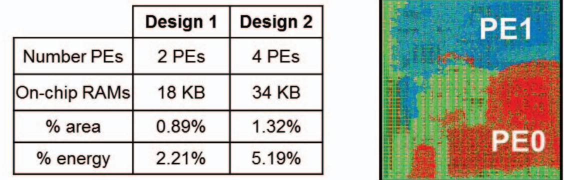
As the figure shows, our accelerator performs better than software. This is because, for these bandwidth bound matrix operations, our accelerator can saturate the available memory bandwidth more efficiently. The figure shows the bandwidth consumed by the accelerator(s) relative to the processor core(s), which is in line with the performance speedups.

Further study on the software implementation using Intel® VTune reveals that a single loop iteration to process a non-zero matrix element takes multiple instructions. Even with an out-of-order execution, the core still spends multiple cycles on average to process a matrix element. The accelerator, however, can process one or more matrix elements per cycle since the PEs are kept busy by the data management unit designed for these matrix operations.

C. Accelerator Area and Energy

We implemented accelerator designs with 2 PEs and 4 PEs in RTL and synthesized them using 14nm Intel process. Both designs meet the 1GHz frequency target. We also further placed and routed the 2-PE design. The results are shown in Figure 6. Both designs are estimated to consume only small area and energy relative to a Xeon processor core.

The four accelerator blocks (each with 4 PEs) in the system we studied in the previous sub-section consume only 0.44% area and 1.73% energy relative to the 12 cores in the system. In other words, a single 4-PE accelerator block is 19x more energy efficient than a single processor core, and four 4-PE accelerator blocks are 57x more energy efficient than 12 cores, while needing only a small die area.



	Design 1	Design 2
Number PEs	2 PEs	4 PEs
On-chip RAMs	18 KB	34 KB
% area	0.89%	1.32%
% energy	2.21%	5.19%

(a) Accelerator designs evaluated (b) Design 1 placed & routed

Fig. 6. Synthesis and place and route results. In 6(a), the % of area and % of energy numbers are relative to a single processor core.

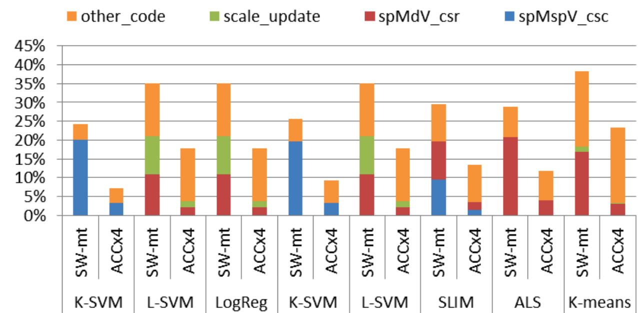


Fig. 7. Runtime breakdown of the workloads under study for multithreaded software implementation (SW-mt) and with four accelerator blocks (ACCx4). The breakdown is shown as a percentage of a single-threaded software runtime. E.g., a 50% means that the workload runs at half the time of the single-threaded software implementation, or a 2x speedup.

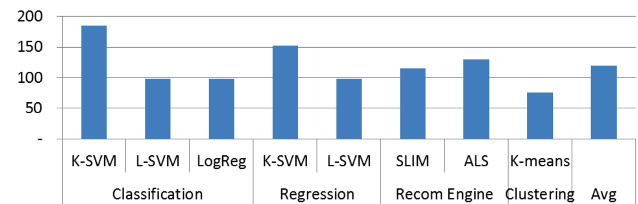


Fig. 8. Performance/Watt improvements achieved by 4 accelerator blocks over multithreaded software implementation. On average, accelerator achieves 117x better performance/Watt.

D. Application-Level Performance

Finally, we study the performance of the entire workloads. Figure 7 shows the execution time for the multithreaded software implementation (SW-mt) and 4 accelerator blocks (ACCx4). The time is shown as the fraction of runtime of the

single-threaded software implementation presented in our characterization study.

Since the critical compute patterns are now optimized, the serial code (other_code) becomes more dominant (i.e., Amdahl's law). On average, scaling up the software implementation to multiple threads leads to 3.2x speedups. The ACCx4 approach delivers 2.2x speedups over multithreaded software and 7.2x speedups over single threaded software.

Figure 8 depicts the performance/Watt improvements of the 4 accelerator blocks over the multithreaded software implementation. The improvements in performance coupled with the extreme energy efficiency of the accelerator lead to 117x improvements in performance/Watt on average.

In cases where serial code dominates (other_code in Figure 7), the performance/Watt improvements are not as great since the serial code still has to be executed by the processor cores.

TABLE II. Comparison against existing accelerators. The proposed accelerator provides coverage for all sparse matrix operations needed by the machine learning analytics workloads we studied.

Accelerator	spMdV_csr	spMdV_csc*	spMspV_csc	Scale Update
[13],[19],etc	✔	✘	✘	✘
[12][21]	✘	✔	✔	✘
This work	✔	✔	✔	✔

* spMdV_csc compute pattern is not used by any of the workloads we studied, but supported by the proposed accelerator as described in section V.

VII. Related Work

There are existing accelerators for sparse matrix operations (e.g. [12,13,19,21]), as summarized in Table II. However, they target only the sparse matrix dense vector multiplication commonly found in high-performance computing. Most process the matrix in row-oriented fashion (e.g., [13,19]). Only [12][21] propose column-oriented sparse matrix multiplication, but they do not support other constructs found in our analytics workloads. There are also accelerators for specific machine learning algorithms, such as K-means [14] and support vector machine [15][21]. Our work shows that a variety of sparse matrix operations dominate runtimes in sparse analytics applications. Our accelerator is the only unified solution we know of that supports all of these matrix operations.

There are a few proposals for accelerators in analytics and machine learning applications [7,8,9]. Most of these works are on accelerators for convolutional neural networks targeting dense image datasets [7,9]. Recently, authors in [10] have proposed an accelerator that supports a set of machine learning algorithms. However, the accelerator still works only on dense datasets. The work in [11] proposed accelerators specialized for linear algebra, though the authors focus on dense datasets. In contrast, our study focuses on analytics where the data is sparse. Unlike with dense data, sparse data leads to much lower compute/byte and more irregular control and data movements, which necessitates different design considerations.

Finally, some of the aforesaid works target FPGAs, while here we target ASIC accelerator(s) alongside processor core(s).

VIII. Summary and Conclusions

We have investigated workloads for analytics of sparse data that rely in machine learning algorithms. Characterization on a high-end server system using large real-world datasets shows that runtime is dominated by a set of matrix operations. We propose a hardware accelerator to do these matrix operations with extreme efficiency. Evaluations using simulations and RTL synthesis to a 14nm ASIC technology demonstrate that our accelerator improves performance and performance/Watt significantly while incurring only a small area overhead.

References

[1] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," ACM Trans. on Intelligent Systems and Technology, 2011.

[2] R.-E. Fan, K.-W. Chang, et. al., "Liblinear: A library for large linear classification," Journal of Machine Learning Research, 2008.

[3] C.-J. Lin, R. C. Weng, and S. S. Keerthi, "Trust region newton method for logistic regression," Journal of Machine Learning Research, 2008.

[4] X. Ning and G. Karypis, "Sparse linear methods with side information for top-n recommendations," Conf. on Recommender Systems, 2012

[5] S. Rendle, "Factorization machines with libFM," ACM Trans. on Intelligent Systems and Technology, 2012.

[6] D. Sculley, "Web-scale k-means clustering," ACM International Conference on World Wide Web, 2010.

[7] T. Chen, Z. Du, N. Sun, et. al, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," Architectural Support for Programming Languages and Operating Systems, 2014.

[8] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," Design, Automation & Test in Europe Conference & Exhibition, 2015.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," International Symposium on Microarchitecture, 2014.

[10] D. Liu, T. Chen, S. Liu, J. Zhou, et. al., "Pudiannao: A polyvalent machine learning accelerator," Arch. Support for Programming Languages and Operating Systems, 2015.

[11] A. Pedram, A. Gerstlauer, and R. Geijn, "A high-performance, low-power linear algebra core," Application-Specific Systems, Architectures and Processors, 2011.

[12] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs," Int. Symp. on Field-programmable Gate Arrays , 2014

[13] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," Int. Symp. on Field-programmable Gate Arrays, 2005.

[14] F. Winterstein, S. Bayliss, G. A. Constantinides, "FPGA-based K-Means Clustering Using Tree-Based Data Structures," Field Programmable Logic and Applications (FPL), 2013.

[15] M. Papadonikolakis, C-S Bouganis, "A Heterogeneous FPGA Architecture for Support Vector Machine Training," Field Programmable Custom Computing Machines, 2010.

[16] University of California Irvine, Machine Learning Repository, https://archive.ics.uci.edu/ml

[17] J. Ma, L. K. Saul, S. Savage, G. M. Voelker, "Identifying Suspicous URLs: An Application of Large-Scale Online Learning," Int. Conference on Machine Learning, 2009.

[18] H. B. McMahan, G. Holt, D. Sculley, et al., "Ad click prediction: a view from the trenches," Knowledge Discovery and Data Mining, 2013.

[19] Y. Zhang, Y. Shalabi, R. Jain, et al., "FPGA vs. GPU for sparse matrix vector multiply," Int. Conf. on Field-Programmable Tech (FPT), 2009.

[20] L. A., Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," Computational Statistics, 2010.

[21] E. Nurvitadhi, A. Mishra, and D. Marr, "A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine", Compilers, Architectures, and Synthesis of Embedded Systems, 2015.