ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation

Kaan Kara[†] Ken Eguro[‡] Ce Zhang[†] Gustavo Alonso[†]

[†]Systems Group, Department of Computer Science
ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

[‡]Microsoft Research, Redmond, USA
lastname@microsoft.com

ABSTRACT

The ability to perform machine learning (ML) tasks in a database management system (DBMS) provides the data analyst with a powerful tool. Unfortunately, integration of ML into a DBMS is challenging for reasons varying from differences in execution model to data layout requirements. In this paper, we assume a column-store main-memory DBMS, optimized for online analytical processing, as our initial system. On this system, we explore the integration of coordinate-descent based methods working natively on columnar format to train generalized linear models. We use a cache-efficient, partitioned stochastic coordinate descent algorithm providing linear throughput scalability with the number of cores while preserving convergence quality, up to 14 cores in our experiments.

Existing column oriented DBMS rely on compression and even encryption to store data in memory. When those features are considered, the performance of a CPU based solution suffers. Thus, in the paper we also show how to exploit hardware acceleration as part of a hybrid CPU+FPGA system to provide on-the-fly data transformation combined with an FPGA-based coordinate-descent engine. The resulting system is a column-store DBMS with its important features preserved (e.g., data compression) that offers high performance machine learning capabilities.

PVLDB Reference Format:
Kaan Kara, Ken Eguro, Ce Zhang, Gustavo Alonso. ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation. *PVLDB*, 12(4): 348-361, 2018.
DOI: https://doi.org/10.14778/3297753.3297756

1. INTRODUCTION

Integrating advanced analytics such as machine learning into a database management system and taking advantage of hardware accelerators for data processing are two important directions in which databases are evolving. Online analytical processing engines (OLAP) are the natural place to implement these advanced capabilities, being the type of data processing system with rich functionality and the ability to deal with large amounts of data. Yet, the problem of efficiently integrating ML with OLAP is not trivial, especially if the functionality of the latter needs to be preserved. In this paper, we take a step forward in understanding the question: *How can we integrate ML into a column-store DBMS without disrupting either DBMS efficiency or ML quality and performance?*

Proceedings of the VLDB Endowment, Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297756

Landscape of Previous Research. This paper is built upon previous research in the following three areas.

In-DBMS Machine Learning. Integrating machine learning into DBMS is an ongoing effort in both academia and industry. Current prominent systems include MADlib [18], SimSQL [10], SAP HANA PAL [14] and various products from Oracle [55], Impala [28], and LogicBlox [6]. The combination of ML and DBMS is attractive because businesses have massive amounts of data residing in their existing DBMS. Furthermore, relational operators can be used to pre-process and denormalize a complex schema conveniently before executing ML tasks [31].

Column-Store for Analytical Workloads. Column-stores are the standard solution for OLAP workloads [20, 32, 7]. When combined with main memory processing and techniques like compression [4], they become highly efficient in processing large amounts of data. However, it is not clear whether the column-store format is suitable for most ML algorithms, notably stochastic gradient descent (SGD), which access all attributes of a tuple at the same time; processing tuples by row. In fact, existing in-DBMS ML systems tend to work on row stores [18, 10, 6].

New Hardware for Data Processing. Emerging new hardware—FPGA, GPU—has the potential to improve processing efficiency in a DBMS through specialization of commonly used sub-operators such as hashing [26], partitioning [27], sorting [52] or advanced analytics [37]. Specialized hardware is also becoming available in data centers: Microsoft uses FPGAs [42] to accelerate Bing queries [41] and neural network inference [11]; Baidu uses FPGA instances in the cloud to process SQL; Intel develops hybrid CPU+FPGA platforms enabling tight integration of specialized hardware next to the CPU [39]. In most of these systems, efficient I/O is required to move data to the accelerator; otherwise the performance advantages disappear behind the cost of data movement. Column-stores are a better fit to achieve this purpose as column oriented processing makes it easier to move data from memory while still being able to exploit the available parallelism in the hardware—especially when the columns are compressed, thereby increasing I/O efficiency.

Scope. In this paper, we focus on a specific problem: *Given a column-store database, how can we efficiently support training generalized linear models (GLMs)?* We provide a solution by first choosing an established ML training algorithm that natively accesses data column-wise: stochastic coordinate descent (SCD) [48]. This algorithm, however, is not efficient due to lack of cache locality and the complexity of model management during the learning process. Therefore, we apply a partitioned SCD algorithm, inspired by recent work of Jaggi et al. [22] to improve cache-locality and performance. Both the standard and the partitioned SCD work on raw data values. Yet, data in column-stores is usually compressed (often twice, through dictionary compression and run-length or delta encoding) and also encrypted [5]. When adding the necessary steps to deal with compression and encryption, we observe that the overall performance on a CPU suffers. To overcome this, we use an FPGA to do on-the-fly data transformation. We choose an FPGA as accelerator because its architectural flexibility enables us to put vastly different processing tasks, such as data transformation (decompression, decryption) and ML training (SCD) into a dataflow pipeline, resulting in high throughput for all of them. The contributions of the paper are as follows[1]:

- We employ a partitioned stochastic coordinate descent algorithm (pSCD) to achieve cache-efficient training natively on column-stores, leading to higher performance both on a multicore CPU and an FPGA.
- We analyze the performance of pSCD against standard SCD and SGD on both row and column-stores by performing an in-depth experimental evaluation on a wide range of data sets, showing the effectiveness of pSCD for real-world ML tasks.
- We provide a detailed implementation of an FPGA-based training engine able to perform SCD and pSCD. We analyze memory access and computational complexity for both algorithms. We validate our design on a CPU+FPGA platform, showing that it saturates the memory bandwidth available to the FPGA.
- We extend the FPGA-based training with data transformation steps. We utilize those steps to show that performing delta-encoding decompression and/or AES-256 decryption on-the-fly comes at practically no performance reduction for the SCD/pSCD engine on the FPGA, whereas on the CPU it leads to significant decrease in the training throughput.

2. BACKGROUND

In this section, we explain the advantages of stochastic coordinate descent (SCD) against the popularly implemented stochastic gradient descent (SGD) algorithm when the underlying data is organized as a column-store. We provide the necessary background on SCD and also introduce our target CPU+FPGA platform.

Problem Definition. We aim at solving optimization problems of the form:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left(\frac{1}{m} \sum_{i=1}^{m} J(\langle \mathbf{x}, \mathbf{a}_i \rangle, b_i) \right) + \lambda \|\mathbf{x}\|_1 \quad (1)$$

$$J = \begin{cases} \frac{1}{2}(\langle \mathbf{x}, \mathbf{a}_i \rangle - b_i)^2 & \text{for Lasso} \\ -b_i \log(h_{\mathbf{x}}(\mathbf{a}_i)) - (1-b_i)\log(1-h_{\mathbf{x}}(\mathbf{a}_i)) & \text{for Logreg} \end{cases} \quad (2)$$

• $h_{\mathbf{x}}(\mathbf{a}_i) = 1/(1+\exp(-\langle \mathbf{x}, \mathbf{a}_i \rangle))$ is the sigmoid function.

where $(\mathbf{a}_1, b_1), ..., (\mathbf{a}_m, b_m) \in ([-1,1]^n \times [0,1])$ is a set of samples and $J : \mathbb{R}^n \times \mathbb{R} \to [0, \infty)$ is a non-negative convex loss function. λ is the regularization parameter adjusting the strength of regularization, that tries to prevent overfitting to the training data by limiting how much the norm of the model $\mathbf{x}$ can grow We use ℓ_1 regularization, turning the optimization problem to Lasso [56] for quadratic loss and logistic regression (Logreg) for cross-entropy loss, as shown in Equation (2). For denoting a column, we use the notation $\mathbf{a}_{:,j} \in [-1,1]^m$, indicating the jth column/feature of the sample matrix $[\mathbf{a}_1; \mathbf{a}_2; ...; \mathbf{a}_m]$, where each $\mathbf{a}_i$ is a row.

[1]https://github.com/fpgasystems/ColumnML

Algorithm 1: Stochastic Coordinate Descent

```
Initialize:
· $\mathbf{x} = 0$, $\mathbf{z} = 0$, step size $\alpha$
· $S(\mathbf{z}) = \begin{cases} \mathbf{z} & \text{for Lasso} \\ 1/(1+\exp(-\mathbf{z})) & \text{for Logreg} \end{cases}$
· $T(x_j, g_j) = \begin{cases} \alpha g_j + \alpha\lambda & x_j - \alpha g_j > \alpha\lambda \\ \alpha g_j - \alpha\lambda & x_j - \alpha g_j < -\alpha\lambda \\ x_j & \text{else (to set } x_j = 0) \end{cases}$
for epoch = 1, 2, ... do
    — randomly without replacement
    for j = shuffle(1, ..., n) do
        $g_j = \frac{1}{m}(S(\mathbf{z}) - \mathbf{b}) \cdot \mathbf{a}_{:,j}$ — partial gradient computation
        $\mu = T(x_j, g_j)$ — thresholding due to regularization
        $x_j = x_j - \mu$ — coordinate update
        $\mathbf{z} = \mathbf{z} - \mu\mathbf{a}_{:,j}$ — inner-product vector update
```

2.1 SGD on Column-Stores

For all gradient-based optimization methods, the knowledge of the inner product $\langle \mathbf{x}, \mathbf{a}_i \rangle$ is required to compute a gradient estimate. Any gradient-descent algorithm, e.g., most notably SGD, computes this inner product at each iteration of the algorithm. This implies that a sample $\mathbf{a}_i$ has to be accessed *completely* at each iteration requiring all features in $\mathbf{a}_i \in \mathbb{R}^n$ to be read in a row-wise fashion.

It is still possible to perform SGD on a column-store, but to make reading from distant addresses in the memory efficient, one has to read *in batches* due to row-buffer locality of DRAMs [24]: Reading multiple cachelines from one column, storing them in the cache, reading from the next column, and so on. This is practically an on-the-fly column-store to row-store conversion that is disadvantageous: It requires cache space proportional to the number of features in the data set and a large batch size in order to read as sequentially as possible [59]. We show the disadvantage of working on a column-store for SGD empirically in Section 4.

Coordinate-descent based algorithms eliminate this problem by enabling a way of accessing the samples *one feature at a time*, which natively corresponds to column-wise access.

2.2 Stochastic Coordinate Descent

Shalev-Shwartz et al. [48] introduce SCD and provide a bound on the runtime for convergence. The core idea in SCD is to maintain a vector $\mathbf{z} \in \mathbb{R}^m, z_i = \langle \mathbf{x}, \mathbf{a}_i \rangle$ containing the results of the inner-products between the model and the samples. It is then possible to always apply the change that was done to one coordinate of the model (x_j) also to the inner-product vector $\mathbf{z}$, as shown in Algorithm 1. This maintains up-to-date inner products between all samples and the current model in vector $\mathbf{z}$. As a result, an update to the model requires accessing only one coordinate—one feature—for all samples, which equals to a column-wise access pattern.

We perform SCD by randomly (without replacement) selecting a feature at each iteration. As we show in Algorithm 1, the model $\mathbf{x}$ is initialized to 0 and therefore, the inner-product vector $\mathbf{z}$ also starts at 0. An epoch corresponds to processing the entire data set: Each column is accessed completely to compute the partial gradient either for Lasso or Logreg. Then, the corresponding coordinate of the model is updated with the partial gradient. Finally, an inner-product vector update takes place to keep the inner products in $\mathbf{z}$ up-to-date, the last step in Algorithm 1. All steps of the algorithm access samples column-wise. We perform multiple epochs until convergence for the optimization problem is observed, by evaluating the loss function.

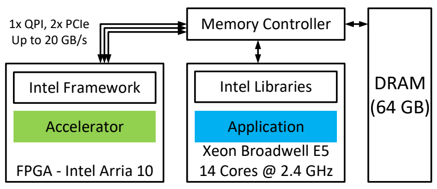


Figure 1: The target platform: Intel Xeon+FPGA v2.

2.3 Target Platform: Intel Xeon+FPGA

Our target platform is the second generation Intel Xeon+FPGA[2], which combines an Intel Broadwell with 14 cores at 2.4 GHz nominal frequency and an Arria 10 FPGA in the same package (Figure 1). The FPGA has cache-coherent access to the main memory (64 GB) of the CPU, via 1 QPI and 2 PCIe links. The read bandwidth when utilizing all 3 memory links is approximately 17 GB/s and the aggregate read-write bandwidth is approximately 20 GB/s.

We implement FPGA accelerators using VHDL, against a 64 B cacheline granular read/write-request based interface. This interface works with physical addressing, so address translation has to take place on the FPGA. On the software side, Intel libraries provide a memory management API to allocate pinned memory that is accessible by both the CPU and the FPGA. The shared memory region is allocated using 4 MB pages and a page-table on the FPGA is populated with physical addresses of the page offsets before an application is started.

Evaluation Setup. We perform all our experiments on the Intel Xeon+FPGA machine. We use *gcc 5.4* and compile the binaries with "*-O3 -march=native*". We set the frequency governor of the CPU to *performance*, enabling it to run at 3.2 GHz peak frequency during program execution.

2.4 System Overview

Our implementation is based on DoppioDB [51], a system providing FPGA-acceleration inside a popular column-store database, MonetDB [20]. The high level diagram in Figure 2 shows how we perform training and inference using user defined functions (UDF). A table name and hyperparameters are given as arguments to a UDF, in which we implement our CPU and FPGA based algorithms. The model produced by the training is stored as a database internal data structure in DoppioDB—similar to a database index. Inference can be performed also with UDFs if a model to a corresponding table has been trained beforehand. MonetDB uses compression by default only on strings, so we implement numeric compression and also encryption as part of the UDFs for test purposes.

3. CACHE-CONSCIOUS SCD

3.1 Overview

Although column-wise access of SCD suits column-store DBMS well, it has one drawback compared to SGD: The intermediate state that needs to be kept is the inner-product vector $\mathbf{z} \in \mathbb{R}^m$ which may be much larger than the model $\mathbf{x} \in \mathbb{R}^n$ itself, since the number of samples is usually much larger than the number of features ($m \gg n$). We use a partitioned version of SCD (pSCD), inspired by the CoCoA algorithm introduced by Jaggi et al. [22]. CoCoA aims at reducing the communication frequency when performing

[2]Results in this publication were generated using pre-production hardware and software donated to us by Intel, and may not reflect the performance of production or future systems.

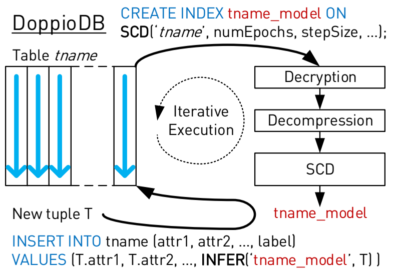

DoppioDB
CREATE INDEX tname_model ON
SCD('tname', numEpochs, stepSize, ...);
Table tname
Iterative Execution
Decryption
Decompression
SCD
tname_model
New tuple T
INSERT INTO tname (attr1, attr2, ..., label)
VALUES (T.attr1, T.attr2, ..., INFER('tname_model', T))

Figure 2: System overview showing the training and inference procedures in DoppioDB.

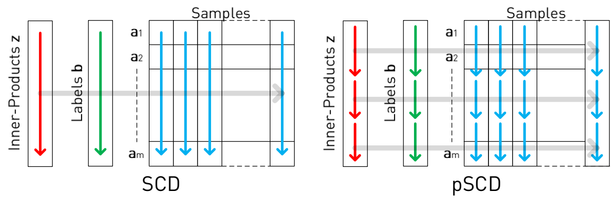


Figure 3: A simplified representation of the data access patterns of SCD and pSCD. The crucial advantage of pSCD is that while processing the samples, only partition-sized portions of the inner-product and label vectors need to be accessed.

distributed dual coordinate ascent. Our main goal with pSCD is to reduce the amount of intermediate state that is kept. As a result, first, the memory access complexity can be reduced because of cache-locality, and second, the algorithm becomes trivially parallelizable, with only infrequent need for synchronization.

Method. The difference in pSCD is mainly that it processes all features in a partition, before moving onto the next one, thus requiring to keep only a partition-sized portion of the inner-product vector $\mathbf{z}$ and label vector $\mathbf{b}$, as depicted in Figure 3. However, when we split the gradient updates this way, the partial gradient that we compute is valid only for the current partition. Therefore, we can't apply the coordinate update step to a global model $\mathbf{x}$; we have to keep a separate model $\mathbf{x}[k]$ for each partition and apply updates locally, as in Algorithm 2. This is equivalent to training K separate models when a dataset is divided into K subsets. To achieve a common model, we perform model averaging and update the entire inner-product vector every P epochs, the so-called global inner-product update (last part in Algorithm 2). Notice that all steps in the algorithm still access data column-wise, even though a column is not scanned in its entirety as in SCD, but in large partitions/subsets.

Convergence Rate. Jaggi et al. [22] perform a theoretical analysis on the convergence rate of partitioned coordinate methods. They show that the convergence rate of partitioned methods is equal to non-partitioned block-coordinate descent methods [46], if the individual optimization problem for each partition is solved to optimality. Another way of analyzing the convergence properties of pSCD is from an update staleness perspective, which suits having the frequency of model aggregation as a tuning parameter: The convergence rate is affected mainly because we are introducing staleness to the algorithm by updating K models independently, then aggregating those every P epochs. The update scheme is similar to *stale synchronous parallel* [19, 62]. Using a similar approach, we assume K (given by the number of partitions) independent workers

Algorithm 2: Partitioned SCD

```
Initialize:
· $\mathbf{x}[K] = 0$, $\mathbf{z} = 0$, step size $\alpha$
· $S(\mathbf{z})$ and $T(x_j, g_j)$ as in Algorithm 1
· partition size $M$, number of partitions $K = m/M$
· inner-product update period $P$
for epoch = 1, 2, ... do
    for k = 0, ..., K-1 (each partition) do                                  ⎫
        — randomly without replacement                                        ⎪
        for j = shuffle(1, ..., n) do                                         ⎪
            $subset = kM + 1, ..., kM + M$                                    ⎪
            — partial gradient computation                                    ⎪
            $g_j = (S(\mathbf{z}_{subset}) - \mathbf{b}_{subset}) \cdot \mathbf{a}_{subset,j}$   ⎬ (1)
            — thresholding due to regularization                              ⎪
            $\mu = T(x[k]_j, g_j)$                                            ⎪
            — coordinate update                                               ⎪
            $x[k]_j = x[k]_j - \mu$                                           ⎪
            — inner-product vector update                                     ⎪
            $\mathbf{z}_{subset} = \mathbf{z}_{subset} - \mu\mathbf{a}_{subset,j}$   ⎭
    — global inner-product update with the averaged model                    ⎫
    if epoch mod P then                                                       ⎪
        $\bar{\mathbf{x}} = (\mathbf{x}[0] + ... + \mathbf{x}[K-1])/K$        ⎪
        $\mathbf{z} = 0$                                                      ⎬ (2)
        for k = 0, ..., K-1 (each partition) do                               ⎪
            $subset = kM + 1, ..., kM + M$                                    ⎪
            for j= 1, ..., n do                                               ⎪
                $\mathbf{z}_{subset} = \mathbf{z}_{subset} + \bar{x}_j\mathbf{a}_{subset,j}$   ⎭
```

and $S = nP$ as our deterministic staleness value, where n is the number of features and P is the global inner-product update period. Ho et al. [19] show that the noisy model state due to staleness is at most $KS = KnP$ updates away from the reference state and this factor is proportional to the convergence rate. In conclusion, the convergence rate of pSCD is closer to standard SCD for smaller K, n, or P. In practice, we can adjust the staleness by selecting a smaller global inner-product update period P, thus resulting in a better convergence rate, as we show in the next section.

Runtime overhead. Performing a global inner-product update ((2) in Algorithm 2) has the same memory access cost as the main part ((1) in Algorithm 2). If the main part (1) has a runtime of T_{main}, the runtime for one pSCD epoch can be approximated by $T_{epoch} = T_{main} \times (1 + 1/P)$. If a relatively high P value delivers good convergence, the overhead by part (2) becomes minimal, because (2) needs to be executed infrequently.

Limitation. One limitation of pSCD in a DBMS is that it assumes the input data is *shuffled*—when the data is ordered, the convergence of the pSCD approach could be slower. In our CPU and FPGA based implementations, we support both pSCD and standard SCD, the latter is scan order resilient. The user may use both depending on the assumptions on the underlying data. The study of scan order for stochastic first order methods is a challenging problem open for decades. Recent theoretical studies on this topic [17, 49, 44] do not completely eliminate data shuffling.

Evaluation. To evaluate the overall efficiency of pSCD, we need to compare both its convergence rate and data processing rate to those of standard SCD. Compared to standard SCD, the convergence rate, termed *statistical efficiency*, is expected to be less for pSCD due to its staleness. However, the data processing rate, termed *hardware efficiency*, is expected to be higher for pSCD due to its cache-locality. In the following sections, our goal is to show that, for pSCD, the advantage offered by hardware efficiency is larger than the disadvantage in its statistical efficiency. Thus, pSCD can lead to an overall shorter training time.

Table 1: Data sets used in the evaluation.

Name	# Samples	# Features	Size	Type
IM	332,800	2,048	2,726 MB	classification
AEA	32,769	126	16,5 MB	classification
KDD1	391,088	2,399	2,188 MB	classification
KDD2	131,329	2,330	1,224 MB	classification
SYN1	33,554,432	16	2,147 MB	regression
SYN2	2,097,152	256	2,147 MB	regression

Table 2: Training quality results comparing SCD and pSCD, with varying inner-product update period P. For pSCD results, we use red for worse and green for better results compared to SCD.

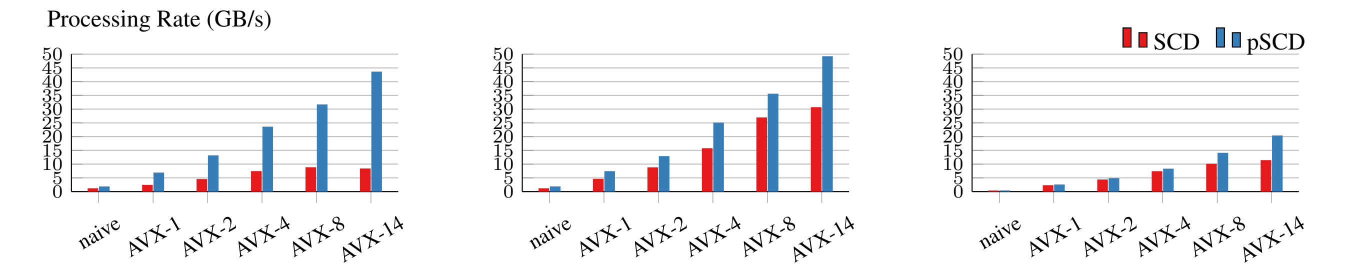
Configuration	Validation Metric	SCD	pSCD P=∞	pSCD P=100	pSCD P=10
Data set: IM Epoch = 200 Train Size = 266k Test Size = 66k Partition Size = 16k	**Log Loss**	0.10154	0.10575 +4.15%	0.10380 +2.23%	0.10191 +0.36%
	Test Accuracy	96.17%	96.071% -0.10%	96.109% -0.06%	96.196% +0.03%
Data set: AEA Epochs = 5k Train Size = 32k Test Size = 59k Partition Size = 16k	**Log Loss**	0.13531	0.25927 +91.61%	0.14972 +10.65%	0.13947 +3.07%
	Test AUC	0.91029	0.86880 -4.56%	0.90891 -0.15%	0.91013 -0.02%
Data set: KDD1 Epochs = 1k Train Size = 391k Test Size = 45k Partition Size = 16k	**Log Loss**	0.24672	0.24712 +0.16%	0.24701 +0.12%	0.24698 +0.11%
	Test AUC	0.62430	0.62369 -0.10%	0.62274 -0.25%	0.62226 -0.33%
Data set: KDD2 Epochs = 1k Train Size = 131k Test Size = 45k Partition Size = 16k	**Log Loss**	0.32285	0.32294 +0.03%	0.32286 +0.003%	0.32285 0%
	Test AUC	0.61145	0.61144 -0.002%	0.61145 0%	0.61139 0.01%

3.2 Statistical Efficiency

In this section, we evaluate the statistical efficiency—convergence behavior—of pSCD against standard SCD, using data sets from real-world use cases (Table 1). We are interested in comparing the convergence rate and the resulting model quality when using either SCD or pSCD. Due to the large staleness introduced by pSCD, our expectation is to observe a deviation in the final loss achieved after N epochs when comparing the two algorithms. Our goal here is to show that, in practice, the deviation is very limited and can further be adjusted by tuning the global inner-product update period P.

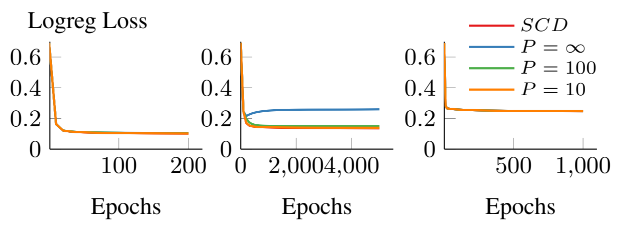
Data sets. We select a variety of data sets which have relational properties from *Kaggle* competitions, using MLBench from Liu et al. [36] as a guideline. We also create a data set of our own to have a large-scale real-world classification task: We run *InceptionV3* [54] neural network on ImageNet (ILSVRC 2012 contest) to extract 2048 features from images, resulting in the IM data set.

- **IM**: Half the samples contain features extracted from cat images (classes 281, 282, 283, 284) and the other half from dog images (classes 153, 230, 235, 238), for classifying cats and dogs.
- **AEA**: Winning features [1] for Amazon employee resource access/denial prediction competition.
- **KDD1 and KDD2**: Winning features [2] for the KDD Cup 2014 competition, predicting excitement about projects.
- **SYN1 and SYN2**: Synthetically generated data sets having varying number of samples and features with uniform random noise, for throughput measurement purposes.



(a) Lasso, SYN1 **(b)** Lasso, SYN2 **(c)** Logreg, IM

Figure 4: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. Partition size: 16384. For pSCD $P = 10$.



(a) IM. **(b)** AEA. **(c)** KDD1.

Figure 5: Convergence of the Logreg loss for three data sets trained with either SCD or pSCD with varying P.

Methodology. For each data set, we select a certain configuration and perform first SCD, then pSCD with varying inner-product update periods, P. Step size is held constant at 4. We evaluate the convergence both by the optimization objective (loss function) and a test score: for IM we split the data set into 80/20 training/test sets; for AEA, KDD1 and KDD2 we perform inference on the test sets given in their respective *Kaggle* competitions and submit the predictions to *Kaggle* to obtain an area under curve (AUC) score. The scores we obtain are ranked in the top 50 in *Kaggle* for the respective data sets.

Analysis. The results are in Table 2: We present the deviation from the SCD results as percentages. As expected, the negative impact by the staleness of pSCD shows itself in both the loss function evaluation and the test scores. However, for all data sets, the negative effect is very limited. Furthermore, it can be decreased by using a lower global inner-product update period P: We observe loss and test score values closer to the ones obtained with SCD when using lower P ($= 10$), which means a global inner-product update is performed more frequently. Performing the global update frequently turns out to be especially important for AEA data set. The reason for this is that it has a low number of features (32k), leading to just two partitions (each sized 16k). When those two partitions have substantially different distributions, frequent global updates are needed to share information regarding optimization steps the two partitions take separately during pSCD. The fact that even relatively infrequent global updates lead to very satisfactory results in Table 2 shows the usefulness of pSCD to train generalized linear models, for real-world problems. In Figure 5, we plot the Logreg loss over the number of epochs performed for 3 data sets. Apart from AEA, all curves are overlapped, making a visual distinction not possible. For AEA, we observe that for $P = 10$, a total visual overlap happens. From the results in this section, we see that not only the final loss with pSCD is very close to SCD, but the *empirical convergence rate* is also very similar. Thus, we conclude the statistical efficiency of pSCD is very close to that of SCD.

3.3 Hardware Efficiency

Previously we showed that epochs of SCD and pSCD are *statistically* very similar. Now our goal is to show that an epoch of pSCD can be performed faster on a multi-core CPU, compared to SCD; thus leading to an overall shorter training time. We are interested in the sample processing rate reported in GB/s, calculated by dividing the total size of all samples in a data set (number of samples $\times$ number of features $\times$ 4 Bytes) by the time required for one epoch.

CPU performance. On the CPU we have multi-threaded implementations of both SCD and pSCD, using Intel intrinsics to take full advantage of the AVX (256-bits) and fused multiply-add (FMA) instructions. We parallelize standard SCD by distributing the partial gradient computation to multiple threads and we synchronize before the coordinate update step for each feature. Parallelizing pSCD is much simpler, since each thread can independently work on its own partitions without any need for synchronization, which is only needed when performing a global inner-product update. In Figure 4, we report numbers using synthetic and IM data sets, covering a range of dimensionality and sample count properties. The advantage of using intrinsics and multi-core parallelism (AVX-N, denoting N-threaded CPU implementation) is clearly visible, compared to a naive single-threaded implementation. For Lasso with AVX-14 doing pSCD, the processing rate reaches the memory bandwidth of the CPU (Figure 4b). This is thanks to the well parallelizable and cache-local processing nature of pSCD. The performance difference between SCD and pSCD is most noticeable for SYN1, in Figure 4a: The large number of samples in that data set leads to a large inner-product vector ($\sim$135 MB) that cannot be kept in the CPU's cache. SCD needs to read the inner-product vector from main memory during gradient computation and inner-product update, leading to lower compute efficiency. For data sets where the inner-product vector fits the last level cache, for example with SYN2 ($\sim$ 8 MB), this effect is less detrimental, however still visible (Figure 4b). Doing Logreg on the CPU has larger overhead due to having to compute the sigmoid function during gradient computation, therefore the throughput is substantially reduced for IM (Figure 4c).

Conclusion. We have shown that pSCD has better hardware efficiency than SCD on a multi-core CPU. Although this is already useful when the target platform is a multi-core CPU, the advantage of pSCD over SCD will be more pronounced for an FPGA implementation, as it will be discussed in Section 6.

4. EMPIRICAL COMPARISON TO SGD

In this section we empirically compare the coordinate descent based methods (SCD and pSCD) with stochastic gradient descent

Table 3: Algorithms used in comparison analysis. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics.

Name	Minibatch Size	Step Size	Impl.	Storage
SGD-tf	512	0.1	default	row-store
SGD-row	1	0.01	AVX-1	row-store
SGD-col-1	1	0.01	AVX-1	column-store
SGD-col-8	8	0.1	AVX-1	column-store
SGD-col-64	64	0.5	AVX-1	column-store
SGD-col-512	512	0.9	AVX-1	column-store
Name	**Partition Size**	**Step Size**	**Impl.**	**Storage**
SCD	-	4	AVX-1	column-store
pSCD	16384	4	AVX-1	column-store

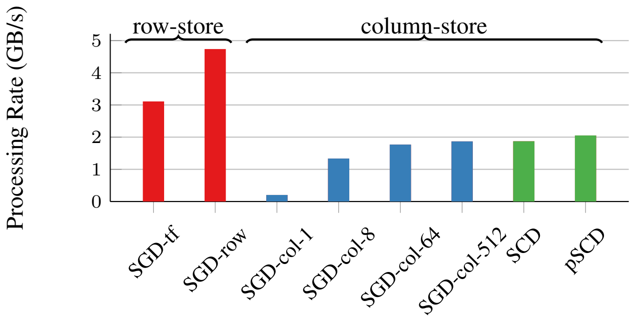


Figure 6: Throughput of algorithms while running Logreg on IM. For pSCD $P = 10$.

(SGD) regarding convergence speed. Our goal is to understand under which circumstances SCD/pSCD is preferable over SGD, focusing on the assumption of columnar storage.

Algorithms. The algorithms used in this analysis are given in Table 3. For SGD, we use the term *minibatch* to refer to how many samples are used to compute a partial gradient. As a baseline (SGD-tf), we use the standard SGD optimizer in Tensorflow v1.11 to perform logistic regression (Logreg). The performance of SGD-tf for a minibatch size of 1 is quite low (we assume due to the overhead of calling its C++ back-end frequently), so we use a minibatch size of 512 for SGD-tf. The other algorithms are our own AVX-optimized CPU implementations. SGD-row performs standard SGD with a minibatch size of 1 on row-store data. Although its minibatch size is much lower than what we used in Tensorflow, it achieves a higher throughput due to being a native implementation and therefore provides a good baseline for SGD performance on row-stores. SGD-col-1 to SGD-col-512 perform *SGD on column-store data*, with varying minibatch sizes. SCD and pSCD are coordinate-descent based methods as described previously, working on column-store data.

Methodology. We use single-threaded CPU implementations for each algorithm to obtain a fair comparison. We explain the reason for doing a single threaded analysis in the following. SGD when training linear models is not straightforward to parallelize because of its iterative nature: In each iteration the model under training is updated and the next iteration requires the up-to-date model, creating a tight dependency. There are many studies about modifying SGD by relaxing certain constraints, such as allowing asynchronous updates to enable better thread parallelism [45, 12, 38]. However, since the efficiency of the algorithm depending on the storage layout is an orthogonal issue to how it is modified for better thread parallelism, using standard SGD with one thread serves the purposes of our analysis. Although SCD/pSCD is straightforward to parallelize without any modifications to the algorithm, to remain fair regarding how much compute resources are used, we use also one threaded implementations for SCD/pSCD in this analysis. All algorithms are implemented using AVX (256-bits) instructions to take full advantage of one core. For all algorithms, we tune the step size by sweeping over a range and then use the one that delivers the best final loss. For SGD algorithms, the step size is diminished at each epoch (step size/epoch number).

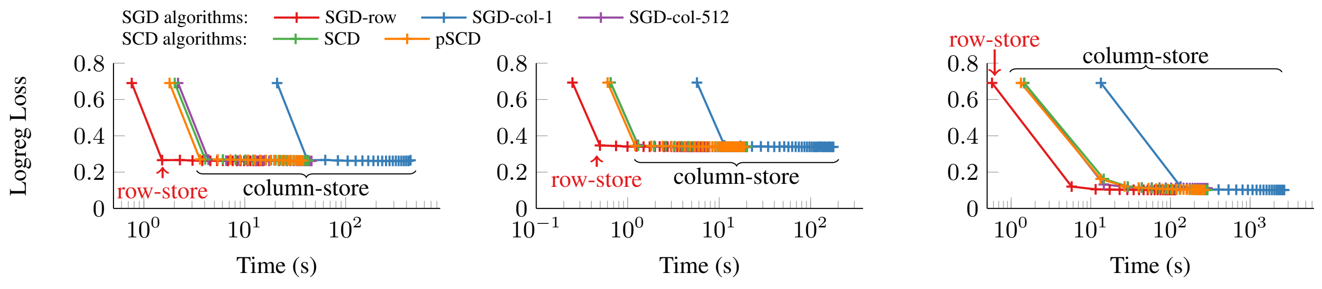
Analysis of SGD performance. Figure 6 shows the processing rate of all algorithms when running Logreg on IM data set. Higher bars indicate that the algorithm can execute epochs faster, corresponding to better *hardware efficiency*. When the underlying storage layout is a row-store, SGD is the clear winner in hardware efficiency. However, when the same algorithm is run on a column-store (SGD-col-1), the throughput drops more than 20x. This is because, regular SGD—having a minibatch size of 1—is very inefficient when gathering individual values of a row from different memory locations. Moreover, with a minibatch size of 1, any means of achieving instruction level parallelism is also blocked due to a tight data dependency, where each iteration depends on the previous iteration's result. This observation is consistent with previous work, analyzing SGD performance on row vs. column stores [59].

Still, there is a way to improve the hardware efficiency of SGD on column-stores: Increasing the minibatch size. This eliminates the tight data dependency, allowing unrolling the innermost loop of the algorithm and the usage of SIMD parallelism on minibatch sized columns. Furthermore, since we read minibatch size chunks from the memory contiguously, we also increase memory access efficiency. As we observe in Figure 6, larger minibatch variants of SGD (SGD-col-8 ... SGD-col-512) have a much higher processing rate, nearly reaching SCD/pSCD on columns-stores.

Hardware and statistical efficiency combined. To get a complete picture of algorithm efficiency, we evaluate hardware and statistical efficiency combined. While the hardware efficiency tells us how fast an algorithm can complete a training epoch (corresponding to the results in Figure 6), the statistical efficiency tells us how much optimization progress the algorithm makes in that epoch.

In Figure 7 we show the convergence curves obtained by running different algorithms on different data sets, plotted over time. Thus we can observe which algorithm reaches a lower optimization objective faster. While the statistical efficiency for SGD-row and SGD-col-1 are—by definition—exactly the same, the difference in hardware efficiency discussed earlier leads to SGD-row's much faster convergence, as observed for all data sets. The statistical efficiency of SGD-col-512 can remain practically unaffected by increasing the initial step size (Table 3) and it offers a good alternative to SCD in terms of hardware efficiency, if the underlying data format is column-store. The statistical efficiency of coordinate descent methods is slightly lower than SGD variants for IM data set (Figure 7c), having a large number of samples to number of features ratio; nevertheless SCD/pSCD reaches as good a solution after the same number of epochs. The difference in convergence rates is explained well in theory [9] that favors SGD for data sets with large number of samples to number of features ratio.

FPGA-related considerations. Large-minibatch-SGD gives a comparable hardware efficiency to pSCD even when working with a column-store, on a CPU. We discuss here if large-minibatch-SGD would be also a viable alternative to pSCD for FPGA acceleration. We consider FPGA as a target platform, because our final goal is to efficiently combine on-the-fly data transformation with a training algorithm. When implementing large-minibatch-SGD on FPGA, following difficulties regarding resource consumption may arise:



(a) Convergence over time for KDD1. **(b)** Convergence over time for KDD2. **(c)** Convergence over time for IM.

Figure 7: Convergence of the Logreg loss using different training algorithms on three data sets, plotted over time to observe the combined effect of hardware and statistical efficiencies. Partition size: 16384. For pSCD $P = 10$.

(1) For efficient usage of the limited memory bandwidth, the model needs to be kept in FPGA on-chip memory (BRAM). Otherwise, half the memory bandwidth will be used for reading the model. This problem does not exist in pSCD, as only one coordinate of the model needs to be updated per each partition, thus leading to less BRAM usage.

(2) For SGD, a large minibatch size is required to saturate the memory bandwidth when reading individual columns from a column-store. Empirical studies on our platform show that around 32 cache-lines (4KB) need to be read from subsequent addresses to saturate the bandwidth [51], leading to a minibatch size of 512. Besides potentially lowering the statistical efficiency [9], larger minibatch sizes would also increase the BRAM usage as discussed next.

(3) In SGD, a row is required twice during an update, once in the initial dot product and once in the partial gradient computation. To avoid wasting memory bandwidth, an entire minibatch of all rows need to be buffered during an update, leading to large BRAM usage. For instance, to run KDD1 with 2399 features and a batch size of 512, we need 4.9 MB of BRAM which is around 70% of our target FPGA. Such high resource usage makes meeting timing constraints for an FPGA design more difficult, potentially requiring reducing the clock frequency. This problem does not exist in pSCD, since only one column is needed per update and therefore resource usage does not scale with dimensionality.

With these arguments, large-minibatch-SGD seems to be less suitable than pSCD for an FPGA implementation when working on column-stores. However, when working on row-stores, efficient linear model training designs have been proposed on FPGAs [25] as well, potentially reaching the same hardware efficiency as pSCD.

Conclusion. When the underlying storage format is row-store, the clear choice of algorithm is SGD, providing both high hardware and statistical efficiency. This conclusion is also inline with existing in-DBMS machine learning solutions [18, 10, 6], which work on row-store format and use SGD for training. When we are dealing with column-stores, both large-minibatch-SGD and SCD/pSCD emerge as good candidate algorithms. The main advantage of pSCD over large-minibatch-SGD is its suitability for an FPGA implementation when processing column-store data. This leads to an efficient and scalable FPGA implementation, which in turn can be combined with other necessary data processing modules such as decompression and decryption, as we discuss in Section 6.

5. NON-DISRUPTIVE INTEGRATION

A primary limitation of integrating ML algorithms into a DBMS is that the two systems usually have different assumptions about the underlying data. In previous sections, we described the inconsistency between the nature of SGD and the storage layout in column-stores. We offered pSCD as a solution, enabling both statistical and hardware efficient generalized linear model training on a column-store, with a CPU. However, DBMS usually store and manage data in ways that are inherently unfriendly to machine learning algorithms. For instance, column-stores are very suitable to being compressed efficiently thanks to the fact that items of the same type are stored next to each other. DBMS take advantage of this and store tables compressed. Sensitive applications also might choose to keep the data encrypted at all times. However, ML algorithms by default cannot work directly on compressed or encrypted data.

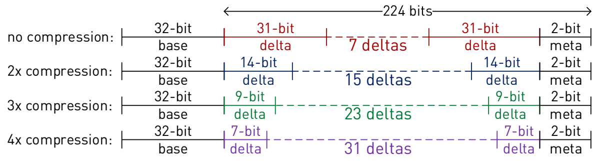


Figure 8: An illustration of the block-based delta encoding scheme.

The naive solution of creating a separate copy of data properly formatted for ML purposes is usually less than ideal. First, it creates maintenance problems and requires additional storage by creating two copies. Second, it defeats the benefits of certain transformations: Keeping a decompressed copy defeats the purpose of efficient storage and keeping a decrypted copy defeats the security objective. Therefore, non-disruptive integration requires on-the-fly data transformation from its in-storage state to that required by ML only when it is needed. We consider two such data transformations: delta-encoding decompression and AES decryption.

Block-based Delta-Encoding. The encoding tries to find small delta series in a column and packs the base and the deltas in 256-bit wide blocks, ergo block-based. During decompression, the blocks can be processed independently, thus allowing a high degree of parallelism. Choosing 256-bit wide blocks provides a nice trade-off between compression capability vs. ease of decompression: A larger block would increase compression capability, while decreasing the ease of decompression due to less independence.

Depending on the largest delta found in a series, a compressed block can have the following number of values (Figure 8): 8, 16, 24 or 32. The first value in a 256-bit block is always the base, which takes 32 bits. The rest of the block is used to store deltas: 7, 15, 23 or 31, taking 31, 14, 9 or 7 bits, respectively. At the end of each block, 2 bits are reserved for meta data storing the information about the compression rate applied to the current block.

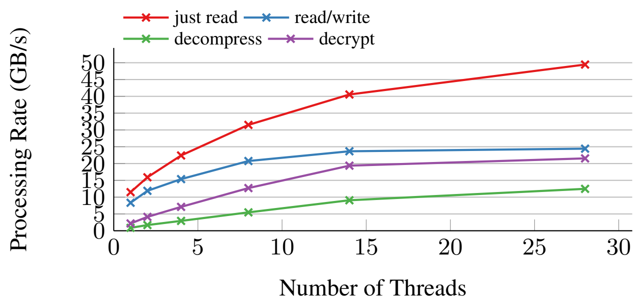


Figure 9: CPU scaling of read bandwidth, read/write bandwidth, decryption and decompression rates.

Algorithm 3: Steps in AES-256 decryption

```
AESDEC (in[127:0], out[127:0], last):
    temp = InvShiftRows(in)
    temp = InvSubBytes(temp)
    if last then
        out = temp
    else
        out = InvMixColumns(temp)

AESDEC-256 (in[127:0], out[127:0])
    AESDEC (in, temp1, 0)
    ... — multiple executions of AESDEC
    AESDEC (temp12, temp13, 0)
    AESDEC (temp13, out, 1)
```

Decompression Performance. We perform a micro-benchmark on our target CPU to analyze decompression rate and how it scales with the number of threads used. In Figure 9, we show the decompression rate in comparison to both read bandwidth and simultaneous read/write bandwidth on the target CPU. We observe that decompression is compute bound; as a result, the decompression rate scales linearly with the number of threads until the core count on the target CPU—14 cores—is reached. Therefore, we conclude that on-the-fly decompression on the CPU is expected to lead to substantial throughput reduction if performed before SCD.

AES-256 CBC Decryption. We use the Advanced Encryption Standard (AES) with 256-bit key and "Cipher Block Chaining" (CBC) as the block cipher mode—this determines the block-size for data that has to be encrypted and decrypted together. We select the block size of CBC to be the partition size of our ML algorithms. On the target Xeon CPU, AES encryption/decryption is supported by specialized intrinsic instructions which we use to implement AES-256 in CBC mode.

Decryption Performance. The decryption steps are shown in Algorithm 3, of which AESDEC is an intrinsic instruction providing 1.78 cycles/byte performance according to the Intel AES intrinsics manual [3]. This is consistent with our micro-benchmark in Figure 9, considering a peak clock frequency of 3.2 GHz. Although scaling better than decompression, decryption is still compute bound, even when using all cores available.

On-The-Fly Data Transformation on CPU. In Figure 10 we show a timing breakdown of individual parts when performing pSCD with on-the-fly data transformation. As expected, data transformation times dominate on the CPU. Decompression on-the-fly is more costly than decryption on-the-fly; when data is both compressed and encrypted, the decompression time dominates. When we perform both encryption and compression, we first compress the data in order to benefit from inherent numerical properties for a better compression rate and then encrypt the compressed data. This has the side effect of decryption being shorter when performed on

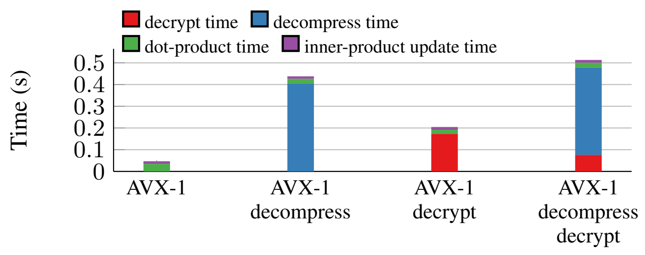


Figure 10: CPU breakdown analysis for pSCD with on-the-fly data transformation. Data: 1 Million samples, 90 features. Partition size: 8192. For pSCD $P = 10$.

compressed data, simply because less data has to be decrypted, as we see in the rightmost column in Figure 10. The breakdown experiment confirms that on-the-fly data transformation when present indeed dominates the runtime of ML training on a CPU.

6. SPECIALIZED HARDWARE

Since data transformation is so costly on the CPU, but is also required for a seamless integration of ML into a DBMS, we offer a specialized hardware solution. Our goal is to develop a data processing pipeline performing both data transformation tasks and machine learning. FPGAs excel at pipeline parallelism and due to their micro-architectural flexibility they also offer acceleration for vastly different compute tasks. Therefore, they are a suitable target platform for providing ML with on-the-fly data reconstruction. In this section we provide an in-depth description of an FPGA-based *SCD Engine* and the implementation of two data transformation tasks on the FPGA: decompression and decryption.

6.1 FPGA-based SCD Engine

The *SCD Engine* is designed to be runtime configurable regarding many parameters: Data properties (number of samples and features), partition size, global inner-product update period, and whether to perform SCD or pSCD. We can also dynamically choose which data transformation slots (e.g., decompression and decryption) to activate. Figure 11 shows a high-level diagram with the essential blocks of the *SCD Engine*.

6.1.1 Fetch Engine

A *Fetch Engine* is responsible for generating read requests with correct addresses to the memory; to read the inner-products, the labels, and the samples—in partitions. The read responses (64 B cachelines) arrive out-of-order from the memory links, so the *Fetch Engine* performs reordering of cachelines according to their transaction IDs, using a large enough (256 lines) internal reorder buffer.

Address Calculation. Since we perform address translation on the FPGA, the *SCD Engine* can work on a virtual address space. At the start of each epoch, the offsets for the inner-products, the labels, and the feature columns are read by the *Fetch Engine* and stored internally for calculating final addresses.

The essential part of final address calculation is partition offset determination, for which there are two modes of operation: (1) If the data is stored regularly—if the offsets for partitions can be computed—, then the partition offsets are calculated on the engine and the final address can be determined with those. (2) If the data is stored irregularly—for instance in the compressed case each partition might occupy a different size in memory—, then the partition offsets are stored in the memory and they first have to be read by the *Fetch Engine* before the final addresses can be determined.

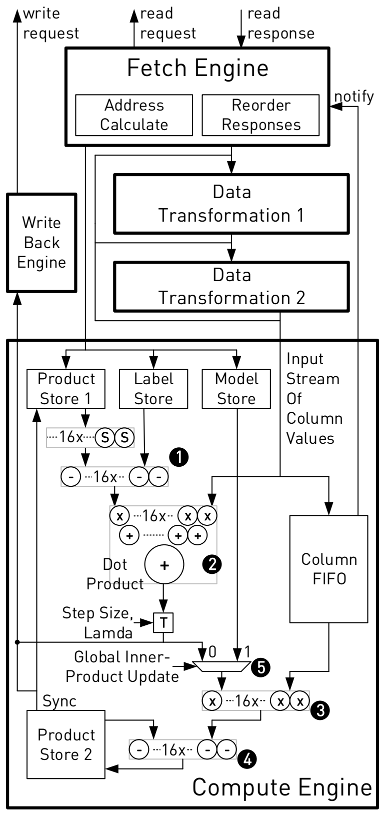


Figure 11: A high-level diagram showing the parts of the *SCD engine* on the FPGA.

In the latter mode, where the partition offsets have to be read from the memory, the *Fetch Engine* has to wait as long as the memory latency before it can start requesting the partition. This causes a substantial reduction in read throughput, if only one *SCD Engine* is employed. However, when multiple *SCD Engines* are employed, as discussed later in Section 6.1.4, there are enough requests generated to hide this effect, as we show later in our evaluation. The effect is similar to that observed in join algorithms where a large number of threads can help hide memory latencies [8].

Fetch Frequency. The *Fetch Engine* is allowed to request one complete partition without any control by the *Compute Engine*, which is further down in the pipeline. However, in order to start requesting the next partition, a notification has to come from the *Compute Engine*, as shown in Figure 11. This notification is generated once an entire partition is received by the *Compute Engine*. The longer the latency of the data transformation, the more the *Fetch Engine* has to wait until it can start requesting a new partition. The throughput reduction caused by this usually remains small, even for high latency operations such AES-256 decryption, since the read time for the partition sizes we are using (e.g., 64 KB) is much larger than internal data transformation latencies.

SCD vs. pSCD. The read pattern of the *Fetch Engine* depends on whether the FPGA is supposed to perform SCD or pSCD. In the pSCD case, a partition from the inner-products and labels are fetched and stored in the *Compute Engine*, then the *Fetch Engine* proceeds to reading the corresponding partitions from all feature columns. Since the inner-products and the labels are read only once per partition in the case of pSCD, the memory access complexity of one epoch is proportional to nm, where n is the number of features and m is the number of samples. In the SCD case, the access complexity is larger, because the intermediate state that needs to be kept is much larger than in pSCD: For *each column*, the labels are read once, the inner-products and all feature columns are read twice (once for the gradient computation, once for the inner-product update), and the inner-products are written once, resulting in a memory access complexity proportional to $6nm$.

Global Inner-Product Update. In pSCD, we perform a global inner-product update at every P epochs, with an aggregated model. The *Fetch Engine* enables this operation by reading the aggregated model from the memory and then scanning all columns in the partition pattern, just as in normal mode of operation. Labels and current inner-products do not have to be read, as they are not required in this operation. The memory access complexity of this operation is equal to performing one epoch of pSCD, nm, where n is the number of features and m is the number of samples.

6.1.2 *Compute Engine*

The *Compute Engine* is responsible for calculating the gradient of the current partition and for updating the local inner-product. It can also perform a global inner-product update with an averaged model kept in *Model Store*, a step required in pSCD. It is designed to consume one 64 B cacheline, with 16 single-precision floating-point values at every clock cycle, resulting in a processing rate of 12.8 GB/s when clocked at 200 MHz.

Compute Pipeline. In the following, we give a high-level description of the compute pipeline shown in Figure 11. Here, we first explain the mode of operation, when *Global Inner-Product Update* (Figure 11) is not performed. Partition-sized inner-products and labels are kept in *Product Store 1* and *Label Store*. When a feature column is fed into the *Compute Engine*, first the differences between inner-products (for Lasso without and for Logreg with *sigmoid S*) and labels are calculated (❶), then the dot product between the differences and the column values is calculated (❷). In the meantime, the column is written into the partition-sized *Column FIFO*, waiting there to be used during the inner-product update. Once the dot product is ready, it is multiplied with the step size and regularization term is applied as in the thresholding function in Algorithm 1. The resulting value is the model update for the current coordinate, or feature. This value is written back to the memory and also used internally in the *Compute Engine* to update the local inner-product. The inner-product update takes place by multiplying the column values from *Column FIFO* with the model update (❸) and subtracting the values from the local inner-product (❹) residing in *Product Store 2*. Notice that we keep two inner-product stores in the *Compute Engine*. The reason is that we use single-port read/write Block-RAM (BRAM) resources on the FPGA to implement local storage. As a result, at every clock cycle data from only a single address can be read from a BRAM. Since we need to read the inner-product both at the input for the dot product and during the inner-product update, we keep two replicas of the inner-product and update both of them at the same time.

Global Inner-Product Update. In the second mode of operation, when *Global Inner-Product Update* (Figure 11) is active, the aggregated model is kept in the *Model Store*. The columns are read just as in normal mode of operation but instead of performing a dot product, only a scalar-vector multiplication (❹) is performed between the values in the model store (❺) and values from the *Column FIFO*. Mathematically, the goal of this computation is to calculate the dot products between the aggregated model and all samples in the data set, to obtain an up-to-date inner-product vector.

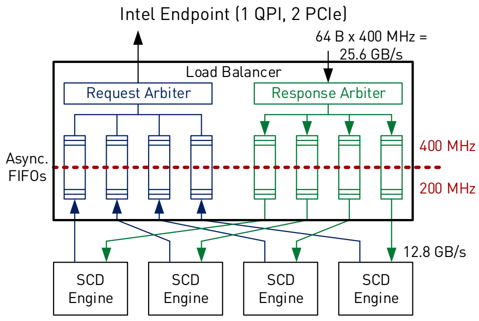


Figure 12: Load balancing to employ multiple *SCD Engines* on the FPGA.

6.1.3 Write Back Engine

The *Write Back Engine* is responsible for writing the model updates and updated inner-products back to the memory. Since the memory interface is granular as a 64 B cacheline, writes also have to be issued in this granularity. For inner-product writes, this happens naturally, because the interface to *Product Stores* are 64 B wide. For model updates, however, the *Write Back Engine* concatenates 16 of them to have a complete cacheline before requesting a write to the memory. That means model updates are written for every 16 columns processed. If the number of columns modulo 16 is larger than 0, the end of processing for the final column triggers the write back with a padded cacheline.

SCD vs. pSCD. For the pSCD case, the updated inner-products are written back after *all* columns are processed, resulting in a write complexity proportional to m, the number of samples. For the SCD case, the updated inner-products have to be written back after each processed column, since local storage is not large enough, resulting in a write complexity proportional to mn.

6.1.4 Employing Multiple SCD Engines

One *SCD Engine* can potentially process data close to its maximum processing rate (64 B $\times$ 200 MHz = 12.8 GB/s), if a very large partition size is selected. A large partition size means that the *Fetch Engine* can request more data at a time without waiting. However, in our evaluation, we observe that even using a relatively large partition (64 KB) leads to around 9 GB/s sample processing rate for the *SCD Engine*, due to the high memory access latency ($\sim$250 ns). Furthermore, when operating in dynamic partition offset reading mode, the high memory latency hurts performance even more, as explained in Section 6.1.1, around 5 GB/s sample processing rate. To overcome this limitation and to saturate the available memory bandwidth for the FPGA ($\sim$17 GB/s), we put multiple *SCD Engines* to work. The engines can process samples completely independently due to the well parallelizable nature of pSCD.

To work with multiple *SCD Engines*, we need a load balancer, as in Figure 12. The load balancer talks to the Intel Endpoint at 400 MHz, enabling a peak data processing rate of 25.6 GB/s. There are asynchronous FIFOs in the load balancer, responsible for the necessary clock domain crossing: 4 FIFOs for requests, 4 FIFOs for responses. The reason for keeping *SCD Engines* clocked at 200 MHz is to make meeting the timing requirements for the FPGA much easier; also, 4 *SCD Engines* at 200 MHz provide an aggregate processing rate already enough to saturate the memory bandwidth. During runtime, we can select how many engines we want to use. We use this to benchmark how overall throughput can be improved when multiple engines are employed, as discussed during the evaluation. With 4 *SCD Engines*, the memory latency can be hidden well and enough requests are generated to saturate the memory bandwidth.

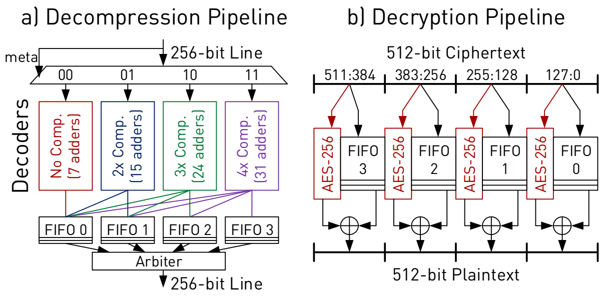


Figure 13: Decompression and decryption pipelines on the FPGA.

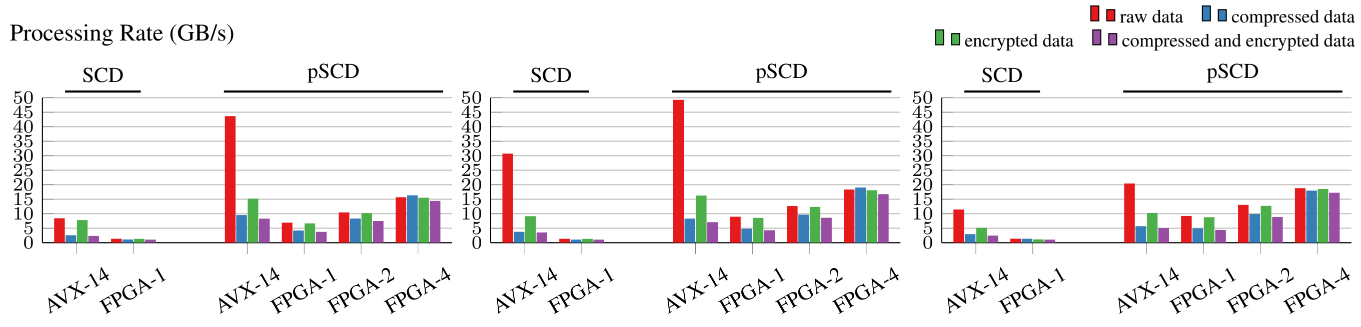
Regarding load balancing, we assign partitions as equally as possible among *SCD Engines*. Each *Engine* reads and writes data from its own partitions in the memory. Thus, the load balancer is passive and merely arbitrates access to the memory port: read/write requests from each Engine are tracked and replies are forwarded accordingly. The load balancer has to keep track of as many cacheline requests as *Engines* can produce in-flight, that is 256 cachelines.

6.2 On-The-Fly Data Transformation

The advantage of the architectural flexibility of an FPGA shows itself most when data processing can be done as a pipeline. Deep pipelines on FPGAs turn the problem of achieving high throughput into a question of *whether there are enough resources*, instead of counting cycles as done for a CPU. As much as the available resources on the FPGA allow, data processing modules can be put in a pipeline, resulting in an overall throughput determined by the slowest module. If all modules process data at the same rate, then the number of modules in the pipeline does not affect the throughput, it only affects resource consumption. Thanks to pipeline parallelism, when we put data transformation slots in front of the *Compute Engine*, we enable on-the-fly data transformation at no throughput reduction—if the data transformation happens at the same rate as the *Compute Engine*, that is 64 B/cycle.

Delta-Encoding Decompression FPGAs excel at compression and decompression tasks [47, 53, 16], mainly because they inherently can access and manipulate data in a bitwise manner. We implement block-based delta decompression on the FPGA using spatial parallelism to handle multiple values in the same cycle, using multiple adders. Determining the FPGA-based decompression performance is straightforward by analyzing the design, shown in Figure 13. A 256-bit input line is fed to its corresponding decoder, depending on its meta bits. In the case of 4x compression, the pipeline has to produce 4 times as many lines as it consumes. Therefore, its consumption rate in the worst case is 1 line per 4 cycles. We put two such decompression pipelines when doing decompression in front of the *Compute Engine* in order to handle 512-bit cachelines. In the worst case, the theoretical rate of decompression is 4x less than the rate of the *Compute Engine*. In practice we do not observe any performance reduction related to this, mainly because one *SCD Engine* does not read data as frequently to saturate memory bandwidth in dynamic partition offset reading mode (Section 6.1.1).

AES-256 CBC Decryption When performing decryption on the FPGA, we eliminate the performance disadvantages of the CPU. Since FPGA-based designs are specialized and use a restricted memory interface, having decrypted values only on the FPGA might also



(a) Lasso, SYN1 **(b)** Lasso, SYN2 **(c)** Logreg, IM

Figure 14: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. FPGA-N denotes using N *SCD Engines* simultaneously. Partition size: 16384. For pSCD $P = 10$.

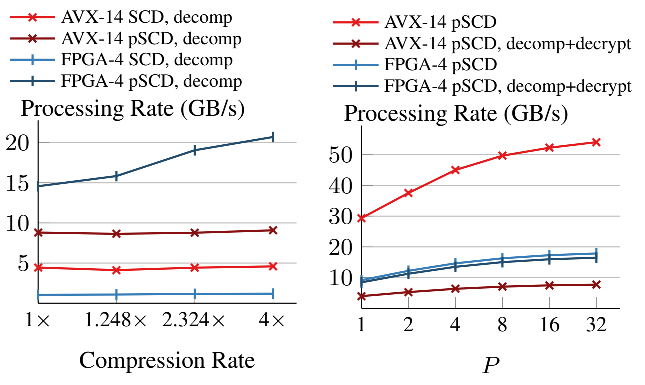
provide a security advantage, although recent work [43] has shown side-channel attacks are possible in multi-tenant FPGA settings. In Figure 13 we show the pipelined design for performing AES-256 decryption in CBC mode. Each *AES-256* block is implemented as a deep pipeline on the FPGA, performing the transformations shown in Algorithm 3. We implement the arithmetic functions in the Galois field using look-up tables on the FPGA. The pipelined design in Figure 13 is able to decrypt data at 12.8 GB/s. When put in front of the *SCD Engine*, it causes no reduction in throughput.

6.3 Evaluation with FPGA

In this section we aim to show the processing rate of the FPGA for pSCD, focusing on the capability to perform on-the-fly data transformation (decryption/decompression) without performance reduction, thus offering better overall hardware efficiency for pSCD than a multi-core CPU. In Figure 14, we present the performance for the CPU and FPGA designs performing SCD and pSCD with on-the-fly delta-encoding decompression and AES256-CBC decryption. We report the processing rate in GB/s, calculated by dividing the total size of all samples in a data set (number of samples $\times$ number of features $\times$ 4 Bytes) by the time required for one epoch.

FPGA performance. The reduced memory access complexity of pSCD helps the FPGA more than it helps the CPU , simply because the FPGA has a lower memory bandwidth ($\sim$17 GB/s) compared to the CPU ($\sim$50 GB/s). As a result, we observe a relatively low performance for the FPGA performing standard SCD compared to the CPU (Figure 14). When we perform pSCD, the FPGA processes samples at a much higher rate, due to the better intermediate state locality provided by partition-based processing. With 4 *SCD Engines* operating in parallel (FPGA-4), the FPGA processes samples at the maximum available memory bandwidth of $\sim$17 GB/s, for relatively high dimensional data sets, SYN2 and IM. For SYN1, the processing rate for samples is around 15 GB/s, lower compared to the other ones, because SYN1 has less columns; therefore more inner-product updates happen per column processed.

Data transformation on the FPGA. The advantage of the FPGA shows itself most when performing on-the-fly data transformation. When doing decompression, the FPGA-1 performance is lower because partition offsets have to be fetched from the memory dynamically, as explained in Section 6.1.1. However, the latency caused by this dynamic offset fetching can be hidden when more *SCD Engines* are employed: For FPGA-4, we see that processing compressed data might even increase performance on the FPGA (Figures 14a and 14b) because in total less data has to be read from the memory. Performing just decryption on-the-fly is shown to come at no performance reduction. Performing both decompression and decryption causes a slight decrease in the processing rate, around 8%, because of the circuit latency introduced by these operations: The *Fetch Engine* has to wait slightly longer until it can request a new partition. The effect of this latency could be hidden by using a larger batch size than we used for these experiments (64 KB), which comes at a higher on-chip storage consumption cost.



(a) Compression rate analysis. For pSCD $P = 10$.

(b) Global inner-product update period P analysis.

Figure 15: Sample processing rate shown at different compression rates and increasing global inner-product update periods P on the multi-core CPU and FPGA. Lasso, SYN2. Partition size: 16384.

Effect of the compression rate. We show the effect of the cost of decompression on the sample processing rate on the CPU and the FPGA in Figure 15a. Higher compression rate means higher cost of decompression, since more data is produced through delta addition (Section 6.2). Against intuition, sample processing rate increases with higher compression rate (=higher cost) for pSCD on the FPGA. This shows that reading less data—thanks to higher compression rate—is more beneficial than the increasing decompression cost on the FPGA. Furthermore, it shows that the *Compute Engines* are able to keep up with higher processing rates than the memory bandwidth, which is the bottleneck of FPGA-based pSCD when compression rate is 1$\times$. The CPU also reads less data with a higher compression rate, but higher decompression cost amortizes this leading to a constant processing rate.

Effect of global inner-product update period P. In Figure 15b, we show the effect of the global inner-product update period for pSCD. Performing a global inner-product update takes a similar amount of time as performing one pSCD epoch without the global update for Lasso, as analyzed in Section 3. We see this behavior in the experiment, a higher P resulting in a higher processing rate both on the CPU and the FPGA. As shown in Section 3.2, high P values such as 10 still lead to high quality training, so that the overhead by the global inner-product update can be kept minimal.

About inference performance. Since the FPGA is already memory bandwidth bound performing training with pSCD, inference on the FPGA would not be any faster than the rates we observe in Figure 14. For the CPU, significant throughput increase is also not expected when doing only inference, because just the inner-product update time is eliminated and that is only a small portion of the total runtime, as we showed previously in Figure 10.

Conclusion. We have showed the advantage of using an FPGA design, if the data needs to be decompressed or decrypted before a machine learning task. However, if the data exists in a non-transformed state, a multi-core CPU offers a faster solution. It is important to mention that the FPGA design becomes memory bandwidth bound and future FPGA-based architectures offering more memory bandwidth could turn this trade-off more in favor of FPGA-based solutions, even if the data does not need to be transformed.

7. RELATED WORK

ML in DBMS. As ML-based data analytics becomes commonplace, integrating ML functionality into a DBMS is ever more important and therefore a very active research field [55, 14, 15, 18, 29, 31, 30, 33]. The SAP HANA [14] Predictive Analysis Library enables a wide range of ML algorithms in the form of SQLScript procedures. Both stochastic gradient descent and cyclical coordinate descent (similar to standard SCD in this work) are given as possible training algorithms, however details about the implementations are not disclosed. Zhang et al. [59] provide an in-depth study on which ML algorithms to use depending on the storage layout, focusing on a NUMA-based CPU system, rather than the cache-conscious or specialized hardware approaches we take in this work. Most of the remaining work in this area focuses on row-store databases: Feng el al. [15] consider *gradient descent* based methods, noting its similar data access pattern with SQL aggregation and integrate it into a row-store DBMS. Hellerstein et al. [18] propose MADlib enabling SQL-based ML with user-defined-functions (UDF) primarily on a row-store database (PostgreSQL); similar to how we use MonetDB [20] UDFs, its main difference being columnar storage. To our knowledge, our focus on performing ML natively on column-store DBMS via coordinate-descent methods and our approach in using specialized hardware to work on transformed data with high performance is unique among existing work.

Stochastic Coordinate Descent. Shalev-Shwartz et al. [48] introduce stochastic coordinate update at each iteration and provide a theoretical convergence analysis with tight convergence bounds. Zhao et al. [61] use a minibatch-based coordinate descent approach, where they, for each epoch, first compute an exact gradient and then use that during minibatch-wise access to adjust partial gradients. Our approach with pSCD is similar to this, as we perform global inner-product updates every P epochs, similar to computing the exact gradient; however, our approach is different in that K models can be updated independently, thus allowing staled updates and enabling parallel processing with infrequent synchronization. Jaggi et al. [22] introduce CoCoA, a communication efficient distributed dual coordinate ascent algorithm. Our approach is similar in that it also partitions a dual variable (the inner-product vector) into disjunct pieces and performs local optimization in a distributed way followed by model averaging. While they analyze the local optimization and subsequent aggregation independently, we analyze the effect of the frequency of model averaging from a staleness point of view, similar to Ho et al. [19] and keep the number of coordinate descent steps the same, regardless of how many aggregation steps are performed. Liu et al. [35] perform SCD asynchronously to enable better parallelism and show near linear speed-ups in distributed settings, however not eliminating cache inefficiencies. Recently, coordinate-based methods have also been considered for deep neural network (DNN) training [60].

Specialized Hardware in DBMS. Database acceleration with specialized hardware, in the form of FPGAs, GPUs and ASICs, has been a very relevant topic in recent literature [26, 27, 50, 58, 52, 34]. Oracle's Sparc M7 processor [34] contains so-called database analytic accelerator (DAX) engines on silicon, performing predicate evaluation and decompression on-the-fly, the latter similar to our approaches in this work. Fang et al. [13] also target on-the-fly data transformation with an ASIC design, focusing on offloading extract-transform-load workloads from the CPU, albeit not combining it with other computation directly on specialized hardware, as we do in this work with decompression/decryption and SCD/pSCD. Istvan et al. [21] implement a persistent key-value store, similar to a noSQL database, on an FPGA-based system, using its on-the-fly data transformation capabilities. However, since we are interested in maintaining OLAP capability while providing efficient ML in this work, noSQL databases are not a direct option.

Specialized Hardware for ML. Due to the data and compute intensive nature of ML, specialized hardware designed for these algorithms has a large potential to accelerate both training [23, 27, 37] and inference [57, 40]. Similar to this work, Kara et al. [27] accelerate generalized linear model training on an FPGA; however they use SGD on row-stores and focus on optimizing the architecture for quantized input data, whereas we focus on coordinate-descent based methods and performing on-the-fly data transformation.

8. CONCLUSION

In this paper we have focused on generalized model training on column-stores using coordinate descent based methods. We use partition based stochastic coordinate descent (pSCD), that improves the memory access complexity of SCD, leading to better training performance both on the CPU and FPGA. We show the staleness of pSCD for model updates can be fine tuned, leading to high quality convergence. On the systems side, we presented an FPGA-based system capable of performing various compute intensive data transformation tasks—decompression and decryption—in a pipeline before an SCD engine, performing either SCD or pSCD on the FPGA. We compared our FPGA-based system to an AVX-optimized multi-core CPU implementation, showing that the multi-core CPU is faster on raw data. However, once it has to perform on-the-fly data transformation, its performance is reduced significantly; whereas the FPGA sustains high throughput even when it performs decompression/decryption, due to pipeline parallelism. We also compared pSCD to more popular SGD and discussed under which circumstances the choice of pSCD over SGD makes sense.

The resulting system shows the advantages of using specialized hardware for dataflow processing and machine learning, in a column-store database setting. Our future work includes query optimization to decide when to perform SCD either on the CPU or on the FPGA, depending on which type of data transformation is needed.

9. ACKNOWLEDGMENTS

We thank Intel for their generous donation of the Xeon+FPGA, Version 2 platform used in this work. This work has been partially funded by the Microsoft Swiss Joint Research Center (JRC).

10. REFERENCES

[1] `https://github.com/owenzhang/Kaggle-AmazonChallenge2013`.

[2] `https://www.datarobot.com/blog/datarobot-the-2014-kdd-cup`.

[3] `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption\-standard-new-instructions-set-paper.pdf`.

[4] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[5] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal Security with Cipherbase. In *CIDR*. Citeseer, 2013.

[6] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM, 2015.

[7] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 583–598. ACM, 2016.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[9] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[10] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 637–648. ACM, 2013.

[11] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[12] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pages 2674–2682, 2015.

[13] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–68. ACM, 2017.

[14] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[15] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336. ACM, 2012.

[16] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 52–59. IEEE, 2015.

[17] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo. Why random reshuffling beats stochastic gradient descent. *arXiv preprint arXiv:1510.08560*, 2015.

[18] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[19] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[20] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering*, 40, 2012.

[21] Z. István, D. Sidler, and G. Alonso. Caribou: intelligent distributed storage. *PVLDB*, 10(11):1202–1213, 2017.

[22] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan. Communication-efficient distributed dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 3068–3076, 2014.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[24] M. Kandemir, H. Zhao, X. Tang, and M. Karakoy. Memory row reuse distance and its role in optimizing application performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 137–149. ACM, 2015.

[25] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 160–167. IEEE, 2017.

[26] K. Kara and G. Alonso. Fast and robust hashing for database operators. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE, 2016.

[27] K. Kara, J. Giceva, and G. Alonso. FPGA-Based Data Partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 433–445. ACM, 2017.

[28] M. Kornacker et al. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[29] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A Distributed Machine-learning System. In *Cidr*, volume 1, pages 2–1, 2013.

[30] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722. ACM, 2017.

[31] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International*

Conference on Management of Data, pages 1969–1984. ACM, 2015.

[32] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-time analytical processing with SQL server. *PVLDB*, 8(12):1740–1751, 2015.

[33] B.-E. Laure, B. Angela, and M. Tova. Machine Learning to Data Management: A Round Trip. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1735–1738. IEEE, 2018.

[34] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, et al. 4.2 A 20nm 32-Core 64MB L3 cache SPARC M7 processor. In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, pages 1–3. IEEE, 2015.

[35] J. Liu and S. J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

[36] Y. Liu, H. Zhang, L. Zeng, W. Wu, and C. Zhang. MLBench: How Good Are Machine Learning Clouds for Binary Classification Tasks on Structured Data? *PVLDB*, 11(10):1220–1232, 2018.

[37] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB*, 11(11):1317–1331, 2018.

[38] C. Noel and S. Osindero. Dogwild!-distributed hogwild for cpu & gpu. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[39] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, et al. A reconfigurable computing system based on a cache-coherent fabric. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 80–85. IEEE, 2011.

[40] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–8. IEEE, 2017.

[41] A. Putnam. Large-scale reconfigurable computing in a Microsoft datacenter. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–38. IEEE, 2014.

[42] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[43] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier. FPGA side channel attacks without physical access. In *International Symposium on Field-Programmable Custom Computing Machines*, pages paper–116, 2018.

[44] B. Recht and C. Ré. Toward a noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. In *Conference on Learning Theory*, pages 11–1, 2012.

[45] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[46] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1-2):433–484, 2016.

[47] S. Rigler, W. Bishop, and A. Kennings. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 1235–1238. IEEE, 2007.

[48] S. Shalev-Shwartz and A. Tewari. Stochastic methods for l1-regularized loss minimization. *Journal of Machine Learning Research*, 12(Jun):1865–1892, 2011.

[49] O. Shamir. Without-replacement sampling for stochastic gradient methods. In *Advances in Neural Information Processing Systems*, pages 46–54, 2016.

[50] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 403–415. ACM, 2017.

[51] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso. doppioDB: A hardware accelerated database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1659–1662. ACM, 2017.

[52] E. Stehle and H.-A. Jacobsen. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 417–432. ACM, 2017.

[53] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 411–420. ACM, 2012.

[54] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[55] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, et al. Oracle data mining. In *Data mining and knowledge discovery handbook*, pages 1315–1329. Springer, 2005.

[56] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[57] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[58] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with GPUs. *PVLDB*, 7(11):1011–1022, 2014.

[59] C. Zhang and C. Ré. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.

[60] Z. Zhang and M. Brand. Convergent block coordinate descent for training tikhonov regularized deep neural networks. In *Advances in Neural Information Processing Systems*, pages 1719–1728, 2017.

[61] T. Zhao, M. Yu, Y. Wang, R. Arora, and H. Liu. Accelerated mini-batch randomized block coordinate descent method. In *Advances in neural information processing systems*, pages 3329–3337, 2014.

[62] F. Zhou and G. Cong. On the convergence properties of a K-step averaging stochastic gradient descent algorithm for nonconvex optimization. *arXiv preprint arXiv:1708.01012*, 2017.