Tensors: An abstraction for general data processing

Dimitrios Koutsoukos[1], Supun Nakandala[2], Konstantinos Karanasos[3], Karla Saur[3], Gustavo Alonso[1], Matteo Interlandi[3]

[1]`{dkoutsou,alonso}@inf.ethz.ch` ETH Zurich
[2]`snakanda@eng.ucsd.edu` UCSD
[3]`{<name>.<surname>}@microsoft.com` Microsoft

ABSTRACT

Deep Learning (DL) has created a growing demand for simpler ways to develop complex models and efficient ways to execute them. Thus, a significant effort has gone into frameworks like PyTorch or TensorFlow to support a variety of DL models and run efficiently and seamlessly over heterogeneous and distributed hardware. Since these frameworks will continue improving given the predominance of DL workloads, it is natural to ask what else can be done with them. This is not a trivial question since these frameworks are based on the efficient implementation of tensors, which are well adapted to DL but, in principle, to nothing else. In this paper we explore to what extent Tensor Computation Runtimes (TCRs) can support non-ML data processing applications, so that other use cases can take advantage of the investments made on TCRs. In particular, we are interested in graph processing and relational operators, two use cases very different from ML, in high demand, and complement quite well what TCRs can do today. Building on Hummingbird, a recent platform converting traditional machine learning algorithms to tensor computations, we explore how to map selected graph processing and relational operator algorithms into tensor computations. Our vision is supported by the results: our code often outperforms custom-built C++ and CUDA kernels, while massively reducing the development effort, taking advantage of the cross-platform compilation capabilities of TCRs.

PVLDB Reference Format:
Dimitrios Koutsoukos, Supun Nakandala, Karla Saur, Konstantinos Karanasos, Gustavo Alonso, Matteo Interlandi . Tensors: An abstraction for general data processing . PVLDB, 14(10): 1797 - 1804, 2021.
doi:10.14778/3467861.3467869

1 INTRODUCTION

Applications such as large-scale data analytics and machine learning (ML) are driving an unprecedented demand for computing capacity. ML model training alone is responsible for a 10× yearly increase in demand for computing capacity [35]. Informal data suggest that it costs as much as several millions of dollars to train a complex model from scratch [11]. This huge compute demand has been partially met by distributed computing, leveraging data center and cloud infrastructures to speed up data processing [4, 70].

Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3467861.3467869

To further boost processing efficiency, but to also reduce the number of machines required [58], specialized hardware and hardware acceleration are gaining momentum. The former refers to hardware tailored to the task at hand (typically tensor computations), while the latter to the use of standard components such as GPUs or FPGAs to replace CPUs, as their architecture is more suitable to tensor computations. However, specialized hardware increases the complexity for the programmer and often requires careful optimizations to improve performance. Thus, in parallel to these developments, a significant effort and investment is being made on software frameworks such as PyTorch [64], Tensorflow [4], or TVM [38] that simplify the development, management, deployment, and optimization of Deep Learning (DL) models. In such tools, DL models are created as a composition of tensor operations. This *tensor abstraction* is the basis for automatic optimizations [28, 39, 48, 53, 73, 79] and enables the compilation of the same code over heterogeneous hardware, e.g., CPUs, GPUs, TPUs [49], IPUs [15], FPGAs [29, 60], etc.

The investments in Tensor Computing Runtimes (TCRs) is likely to continue and specialized hardware tailored to tensor computations will evolve accordingly. It thus makes sense to consider whether TCRs can be used to support computations other than just DL—this would allow additional classes of computation to benefit from the continuous advances in TCRs. The challenge behind the vision of making tensors a general purpose abstraction for data processing lies in the tensor abstraction itself: the close relation between tensors and DL algorithms is what has made these frameworks so efficient. But is this abstraction, which is so far dedicated to DL, a good match for other use cases too?

Recent work has mapped traditional ML algorithms (e.g., decision trees) to the tensor model [61]. The resulting system, Hummingbird, is the starting point for the overarching vision we present in this paper: to investigate whether such mappings exist and can be beneficial for conventional data processing tasks as well. In particular, we explore how to map selected graph processing and relational operator algorithms to tensor computations. We use PyTorch and TVM to compile these implementations into code executable on CPUs and GPUs. Despite leveraging a high-level abstraction and using the same source code regardless of the underlying hardware, the resulting machine code often outperforms custom-built C++ and CUDA kernels.

These initial results are very encouraging and support the idea that the tensor abstraction and existing TCRs can be a good fit for data processing tasks well beyond ML. However, the results in this paper are just a promising first step and more work is needed to map the wealth of existing data processing algorithms to tensor computations in a way that the result is more efficient than state-of-the-art. To facilitate further exploration of this exciting direction, we plan to open-source the code used in this paper and contribute it to Hummingbird,[1] which is part of the PyTorch ecosystem.[2]

2 BACKGROUND

We first provide background on tensors and common tensor operations. Then we present Hummingbird [61], an initial attempt at exploiting the tensor abstraction for traditional ML algorithms.

2.1 The Tensor Abstraction

DL models are a family of ML models based on artificial neurons [43]. Frameworks used to implement DL models include TensorFlow [4], PyTorch [64], CNTK [2], MXNet [3], and ONNX [20]. DL models are expressed in terms of *operations over tensors*. DL frameworks such as TensorFlow and PyTorch provide hundreds of tensor operators. A new class of compilers for tensor programs has recently emerged. Systems like Halide [67], XLA [28] and TVM [38] employ the tensor abstraction to generate highly optimized code targeting heterogeneous hardware. While such frameworks and compilers have been mostly used in the computer vision and DL domains, in this paper we explore their potential to serve as generic compilers and runtimes for hardware accelerators. In the remainder of the paper we refer to compilers (e.g., TVM) and DL frameworks as Tensor Computing Runtimes (TCRs).

2.2 Tensor Operators

TCRs offer a rich set of operators over tensors. To help less familiar readers, we provide a quick overview of the operators commonly found in TCRs and the categories that they belong to (for the sake of explanation, we will use the PyTorch API here and in the following sections, but similar operators can be found in other TCRs).

Transformation operators. This category involves operations for selecting one or more elements of a tensor (using the square bracket notation), slicing a row/column (slice, index_select), concatenating two tensors (cat), sorting (sort), and reorganizing the elements of a tensor (gather, scatter).

Reduction operators. This category contains operations for calculating aggregates such as sum or mean, as well as the size of a tensor. It also includes more complex operations like scatter_add and histograms (bincount, histc).

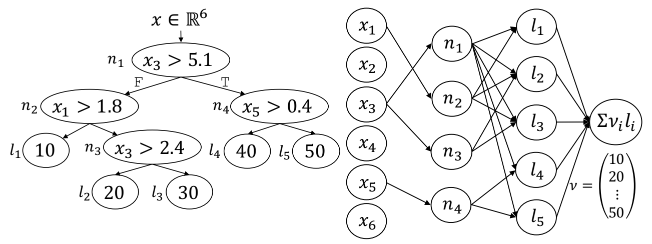
Arithmetic operators. These operators implement basic arithmetic operations between tensors or between scalars and tensors (add, mul, div, sub), matrix-vector (mv) or matrix-matrix (mm) multiplications, and cross product (cross).

Logical operators. Finally, logical operations contain element-wise comparisons between tensors (=, >, <), as well as operations such as where implementing conditional statements.

2.3 Hummingbird: Traditional ML to Tensors

The first step in shaping the vision outlined in this paper was Hummingbird [61], which enables inference of traditional ML algorithms (i.e., non-DL ML algorithms, such as linear regression, decision tree, one-hot encoding) on TCRs. Unlike DL, traditional

[1]https://github.com/microsoft/hummingbird
[2]https://pytorch.org/ecosystem



(a) Decision tree. **(b) Neural Network conversion.**

Figure 1: Translation of a decision tree (left) to an equivalent Neural Network (right) using Hummingbird.

ML algorithms and featurizers do not follow a common pattern as the tensor abstraction. Instead, they are often implemented using imperative programming in Python (e.g., scikit-learn [65]), .NET (ML.NET [31]) or Java (eg., H2O [1]). Expressing them efficiently using tensor computations is not trivial. We briefly explain the key intuition behind Hummingbird with an example [61].

Decision tree inference involves traversing a tree by comparing input features to the inner nodes to define the path from the root to a leaf node. This are *sparse* computations as only a small part of the tree is activated each time. For example, the decision tree of Figure 1a is composed of four inner nodes over three input features. Starting from the root node n_1, feature x_3 is compared with 5.1, and based on the result, either the left (n_2) or the right (n_4) child are selected. The process continues until a leaf node is reached.

Hummingbird supports three different strategies for translating a decision tree to tensor computations, and it picks the most promising one using a mix of heuristics and structural information of the input tree. Figure 1b depicts one of these strategies, whereby the decision tree of Figure 1a is translated into a neural network composed of two hidden layers: the first containing all the internal nodes, and the second containing all the leaf nodes. At inference time, the input features are multiplied with an adjacency matrix that encodes which feature is used by each internal node. The output is then compared against the thresholds, thereby returning the active internal nodes. Finally, another matrix multiplication and comparison are used to check which (unique) leaf node is active and to return the final leaf value.

A characteristic of the translation is that *all* internal and leaf nodes are evaluated at inference time. In other words, the computation becomes *dense*. Despite the *redundancy* introduced, the approach is implemented using two matrix multiplication operations that can be efficiently executed by TCRs both on CPU (e.g., using BLAS libraries such as MKL [16]) and GPU. This strategy is able to deliver state-of-the-art performance when the input trees are not too deep. For example, in [61] we show up to 3× better performance than the XGBoost [37] custom implementation both on CPU and GPU, as well as better performance than RAPIDS FIL [5] just by using PyTorch, without having to implement any of the custom operations that these systems implement on CPU and GPU.

2.4 Hardware Setup

For the experiments we use an Azure NC6 v2 machine with 112 GB of RAM, an Intel Xeon CPU E5-2690 v4 @ 2.6GHz (6 virtual cores), and an NVIDIA P100 GPU. The machine runs Ubuntu 18.04

with PyTorch 1.7.0, TVM 0.8-dev, and CUDA 10.2. We run TVM with `opt_level` 3. TVM models are compiled automatically from the PyTorch models. We report averages of 5 runs.

3 GRAPHS TO TENSORS

Existing efforts such as GraphBLAS [14] have already studied how to express graph algorithms using linear algebra. These approaches rely on the fact that graphs can be represented using adjacency or incidence matrices. Therefore, one could use such linear-algebra implementations to directly execute graph algorithms over TCRs. However, such implementations turn out to be suboptimal when executed over TCRs, because they rely on sparse representations of the graph, while TCRs are not efficient for sparse computations. Hence, novel implementations are required.

As an example, below we show how to efficiently execute PageRank [63], one of the most common graph algorithms, over TCRs. Several other graph algorithms (e.g., affinity propagation, all-pairs shortest paths, connected components) perform operations similar to PageRank, i.e., iterations and aggregations over sparse graphs. Therefore, we believe that TCRs can also be used for such algorithms, yielding similar performance benefits. Additionally, one can also express graph algorithms using more traditional approaches, such as BFS and DFS, over TCRs. Although the necessary data structures (e.g. queues, stacks) may not exist directly in TCRs, they can be implemented using TCRs' available operations.

3.1 PageRank

PageRank is used to rank web pages or, in general, influential nodes on a graph. It is based on the idea that the important nodes are more likely to receive links from other nodes. The algorithm receives as inputs a matrix M with dimensions $N \times N$, where N is the number of nodes that need to be ranked. Each entry $M_{(i,j)}$ contains a number, specifying a probability transition from j to i, such that for all j, $\sum_i M_{(i,j)} = 1$. A damping factor d represents the probability that a person will continue clicking on links. The output of the algorithm is a vector v with dimensions $N \times 1$, which contains the rank v_i of each page. The vector v is normalized and its entries sum up to 1.

The algorithm consists of two initialization steps and a loop that runs for a specific number of iterations n_{iter} or until the ranks v in two consecutive iterations change less than a predefined threshold. In the initialization steps the vector v is filled with random values and a matrix $\hat{M}$ is calculated using the following equation for each element: $\hat{M}_{(i,j)} = d \cdot M_{(i,j)} + \frac{1}{N} \cdot (1 - d)$. The core step of the algorithm is to multiply the matrix $\hat{M}$ with the vector v and assign the result to v for the next iteration.

3.2 Tensor Implementation

Implementing the algorithm on the tensor abstraction may look straightforward. Since its core is a matrix-vector multiplication, an operation that is heavily used and optimized in TCRs, PageRank should run efficiently on them. However, web or social graphs are extremely sparse and the number of connections is significantly smaller than the quadratic number of connections that a full graph would have. Therefore, by naïvely implementing the algorithm over dense tensors, we cannot execute PageRank even for a graph with $100K$ nodes (orders of magnitude smaller than real world examples) because it will require the allocation of terabytes of main memory. A first solution to this problem is to use the sparse modules that TCRs frameworks offer. Nevertheless, as the majority of calculations in DL models involves dense operations, these modules are highly experimental and not optimized. This is confirmed by our experimental results in Section 3.3.

In this algorithm, the matrix-vector multiplication is the bottleneck. To see why, let's assume that $d = 1$ and therefore $\hat{M} = M$. We also assume that M is an adjacency matrix, i.e. it contains only 0s and 1s. Without these assumptions, $\hat{M}$ is an affine transformation of our simplified matrix and we will re-introduce them after we have explained how the algorithm works. Note that M cannot be represented explicitly for very large inputs because it is a sparse matrix. Therefore we represent M as two 1-dimensional tensors which contain the *source* and *target* nodes for every edge.

Since M contains only 0s and 1s, its multiplication with vector v consists of the following steps: (1) for every node i *gather* the values of v for the target nodes to which i is connected (that's where $M(i,*)$=1); (2) *sum* them and store the result in a copy of v that contains the ranks of the next iteration, as follows:

$$v_{new}[\text{i}] \mathrel{+}= v_{old}[\text{M[i, *] != 0}]$$

The steps of gathering indices and adding values together is implemented in TCRs as a single scatter_add(dim, Index, Input) operator, where Input is the input tensor (v_{old}[target] in our case) and Index contains the indexes to gather (source). We have dim=0, since our input tensors (source and v) are 1-d. How scatter_add works within a single PageRank iteration is shown in Figure 2, illustrating how the scatter_add operation "hides" the loop over every node i, by executing all the steps in one single operation. Since we used two 1-d tensors that contain the source and target nodes for every edge, and the $\hat{M}$ contains an affine transformation of the adjacency matrix, the final form of the function is:

$$v_{new}\texttt{.scatter\_add(0, source, } d \cdot v_{old}\texttt{[target] + } \tfrac{1}{N} \cdot (1-d))$$

To keep the semantics of the algorithm intact and because M is a probability matrix where for all j, $\sum_i M_{(i,j)} = 1$, we have to normalize v_{new} at the end of every iteration by dividing each entry with the number of ingoing connections. Although the scatter-gather approach is well-known in distributed frameworks [56], to the best of our knowledge we are the first to apply it to PageRank with the tensor abstraction.

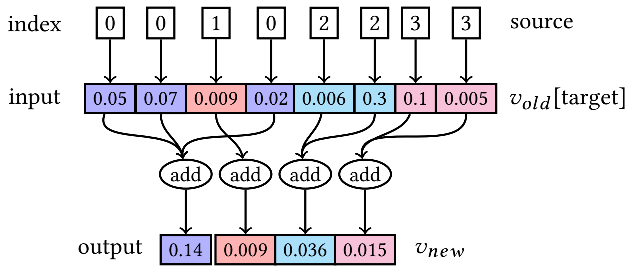


Figure 2: Schematic representation of one iteration of PageRank using scatter_add (before normalization). The source nodes are used to index the values contained in v_{old}[target]. The resulting sums are stored into v_{new}.

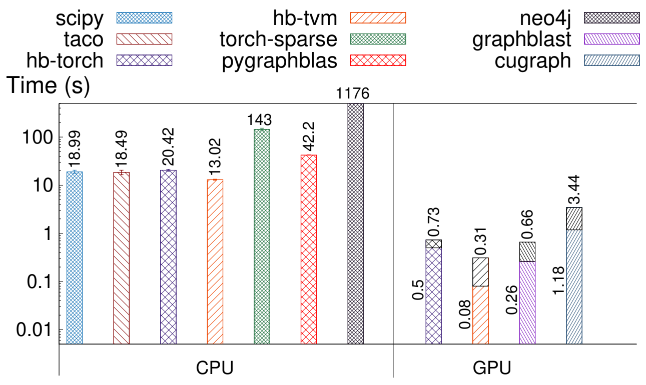


Figure 3: PageRank runtime comparison on CPU and GPU. For the GPU we report the computation time as well as the total time including data transfer (stacked gray boxes). For `cugraph`, the data transfer time includes the time to read the data from disk.

3.3 Performance Analysis

To evaluate our algorithm we use the LiveJournal Social Network,[3] consisting of 4,847,571 nodes and 68,993,773 edges. This dataset is often used to benchmark libraries performing graph computations [72, 76]. We compare the scatter_add approach implemented on PyTorch and TVM (denoted `hb-torch`, and `hb-tvm`, respectively), both on CPU and GPU. For the CPU experiments, we use the following baselines: 2 specialized libraries able to efficiently handle sparse linear algebra operations (`scipy` [25] and `taco` [53]); a PyTorch implementation using sparse matrix-vector multiplications (denoted `torch-sparse`); `pygraphblas` (a Python wrapper around GraphBLAS [14], a specialized library that expresses graph operations as linear algebra blocks on top of BLAS [6]); and `neo4j` [17] (a graph database). Besides Neo4j that is implemented in Java, all other CPU baselines are implemented in C/C++ with Python bindings. For the GPU experiments we further compare against `graphblast` [76] (a GPU implementation of GraphBlas); and `cugraph` [12] (a library developed by NVIDIA and implementing GPU accelerated graph algorithms).

We run 50 iterations of PageRank, which are enough for the algorithm to converge across all frameworks. The running time does not include the construction of the prerequisites of the algorithm (e.g. the construction of the sparse adjacency matrix M from the original indices), because this procedure and the optimal format for the sparse matrix are different across frameworks.

CPU Results. As shown in Figure 2, our PyTorch implementation is on par with `scipy` and `taco`. By adding TVM on top of our PyTorch implementation, we gain an additional 45% performance, as TVM unrolls the loops and fuses the different operators into one, and we outperform the aforementioned baselines by almost 30%. The Pytorch and TVM implementations are more than 2 and 3× faster, respectively, than `pygraphblas`. Finally, if we compare our implementations with `neo4j` and `torch-sparse`, we outperform both by orders of magnitude.

GPU Results. In this setting, if we take into account only the computation time, our PyTorch implementation is 2× slower than `graphblast` and 2.3× faster than `cugraph`. However, TVM improves the performance and it is 5× faster than the PyTorch implementation, making the overall runtime 3.2× faster than `graphblast` and 15× faster than `cugraph`. If we take into account both the transfer and the computation time, PyTorch is 10% slower than `graphblast` and 3× faster than `cugraph`. With TVM, Hummingbird is 2× faster than `graphblast` and 10× faster than `cugraph`. The scatter_add operation is non-deterministic in GPUs for floating point numbers due to the use of atomic add instructions which do not enforce a deterministic ordering. However this problem does not arise for integers (i.e., in our PageRank implementation).

[3]https://snap.stanford.edu/data/soc-LiveJournal1.html

4 RELATIONAL TO TENSORS

Among the operators composing relational database engines, selections, projections, and simple aggregates (e.g., COUNT(*) but also SUM, AVG, MAX, MIN) are already naïvely expressible in TCRs operators. As an example of a non-trivial relational operation over tensors, below we describe the implementation of *cardinality calculation*. Beyond this, we also have initial implementations of projection, selection, primary-foreign key join, aggregation (with and without group by clauses), CASE and IN statements. With these operators, we can currently run a handful of the TPC-H queries and we are working towards supporting the whole TPC-H benchmark.

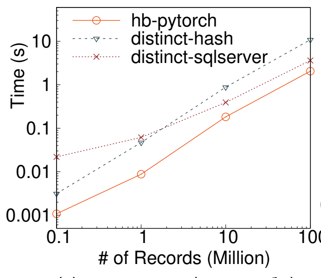
4.1 Cardinality Calculation

Knowing the cardinality of operators is crucial in query optimization [8] and to answer DISTINCT queries. TCRs can also calculate histograms on the input data, which can be used by the query optimizer to decide the right type of scan or join operators [27] or to distribute the data evenly across nodes on a distributed join [34]. We thus show how to calculate the output cardinality of relational operators like DISTINCT or JOIN using tensor computations.

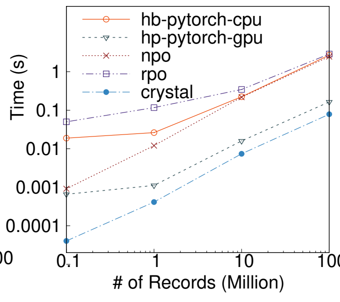
4.2 Tensor Implementation

Our cardinality estimation implementation is based on the bincount(T) operator, which counts the frequency of each value in the input one-dimensional tensor T, with the assumption that T contains only non-negative integers. The result of this operation is a frequency histogram over the input values. Specifically, the result is another 1-d tensor O of length equal to the maximum element present in T, where O[i] contains the number of times the number i appears in T. Using bincount, we can calculate the output cardinality of a DISTINCT operator by counting the number of elements that have a value > 1 in O. In case the input has negative numbers, we can apply tuple renaming, e.g., map the negative numbers to positive ones by using hashing or by adding a positive constant.

To estimate the output cardinality of a join if the keys are unique, we simply concatenate the inner and outer relation keys and call directly the function in the concatenated 1-d tensor. Every element > 1 is a match, therefore we only need to count the number of elements satisfying this condition. If the keys are not unique, we slightly modify the algorithm. We first call bincount function on the keys of the individual relations S, R. For the result of these calls, namely B_S, B_R, we calculate l_{min} = min(size(B_S), size(B_R)), since the matching keys are only in this range. We truncate B_S, B_R to length l_{min}. Finally, to cover all the possible pairs of keys with



(a) DISTINCT (CPU only).



(b) JOIN (CPU and GPU)

Figure 4: Cardinality estimation comparison.

more than one match in case the keys are not unique, we multiply B_S, B_R and sum the entries in the resulting one-dimensional tensor. Although our algorithm is very simple, it is sub-optimal when the keys do not come from a dense domain. We can overcome this obstacle similar to the DISTINCT case, e.g. by tuple renaming.

4.3 Performance Analysis

To evaluate the performance of our cardinality calculation, we compare against an handwritten C++ program that calculates the cardinality of DISTINCT using an optimized hashmap (denoted with `distinct-hash` in Figure 4a). We choose this baseline as hashing is one of the main techniques used to calculate the distinct number of elements in DBMSs [13]. We additionally preallocate the memory for the hashmap and we do not report this as part of the runtime. Finally, we also compare against an instance of the latest edition of SQLServer to qualify our performance against a real-world DBMS. For SQLServer, we create a table that we fill with unique keys with the desired cardinality and we run the query SELECT COUNT(DISTINCT(ID)) FROM TABLE.

For the JOIN output cardinality, we compare against CPU and GPU baselines that compute the full join. In order to have a fair comparison, we modify these latter baselines such that they return only the number of matching tuples (e.g., the cardinalities) instead of performing materialization. For the CPU baseline, we use the hash-join implementations provided by Balkesen et. al [33] that, to our knowledge, are the state-of-the-art on CPU. We run all of the algorithms contained in the paper (hardware oblivious join, hardware conscious join, parallel-radix hash join histogram-based, parallel-radix hash join histogram-based optimized), but for the sake of conciseness, we report only the numbers for the hardware oblivious join (denoted by `npo`) that is the fastest among all implementations, and the hardware conscious join (denoted by `rpo`) which is the fastest of all the parallel-radix implementations. Finally, for the GPU experiments, we compare against crystal [71], a recent library specifically tailored for relational operators on GPU.

DISTINCT Results. For the DISTINCT cardinality estimation the input consists of a vector of integers where each element occurs only once. We run the C++ baseline and the PyTorch implementation on a single thread, while we let the SQLServer query run over all the available virtual cores (6). We report the results in Figure 4a. As we observe, our implementation (denoted with `hb-pytorch`) is 2–4× faster than the C++ implementation, because of the vectorized processing that PyTorch offers, compared to the scan loop of the C++ program. Compared to SQLServer, our implemenation is almost 2-20× faster. The difference is larger for small inputs, as a DBMS has a larger overhead to distribute and parallelize the plan.

JOIN Results. For the join cardinality estimation, we use as input `{key, value}` tuples where each field is a 32-bit integer. We generate the inner and the outer relation such as all the keys are distinct and there is a 1-on-1 match for the inner and the outer side.

We use 6 threads both for the PyTorch implementation and the baselines. For small inputs, the hardware-oblivious implementation outperforms our implementation (denoted `hb-pytorch-cpu`), but the difference gets smaller as we increase the number of elements. On the other hand, the PyTorch implementation is from 50% to 5× faster than the hardware-conscious implementation, but the runtime is almost the same when the input is more or equal than 10 million elements for each relation. However, we observe that our implementation does not fully utilize the CPU, and therefore there is plenty of room for improvement.

For the GPU experiments, our implementation is 2-16× slower than `crystal`, but the difference is decreasing as the input grows. The difference comes from the fact that `crystal` leverages its optimized CUDA implementation, while PyTorch show some constant overhead (due to the Python interpreter) for small sizes.

Remarks. Our join cardinality estimation outperforms or is comparable to full join implementations performing counting (i.e., they do not execute materialization). The observed speedup shows the promise of TCRs in simple relational calculations. We are actively working towards supporting full joins with materialization and other relational operators. Finally, compiling the code with TVM did not give us additional performance but, as for scatter_add, support for the bincount function was only recently introduced.

5 RELATED WORK

Much as the MapReduce [40] abstraction helped democratizing distributed computing, the tensor abstraction could play a similar role for hardware accelerators. Similarly to how Apache Spark [78] proved that is possible to run ML [59], graph [42], and relational [32] workloads end-to-end on the MapReduce abstraction, in this paper we test the expressivity of the tensor abstraction, as well as the efficiency of TCRs for different classes of computations.

Hardware-portable languages [21] and compilers [18, 57] allow targeting both CPU and accelerators. However, they require programmers to write custom code in low-level languages, whereas our goal is to leverage already available optimized CPU and accelerator kernels exposed through the tensor abstraction of TCRs. Dandelion [69], Hocelot [44], and Voodoo [66] share the same goals, although they provide custom kernels or they use OpenCL to target the different hardware with a single implementation. More recently, offload annotations [77] suggests annotating functions implemented in high-level languages (i.e., Python) with a corresponding function from an accelerator library. This approach works well for specific domains where efficient kernels are already available (e.g., Numpy), while it is not clear how the approach will work in the generic case, i.e., how to map a tree-traversal algorithm.

In the database domain, in the past few years several efforts [46, 55, 75] have suggested to map tensor algebra into relational algebra for leveraging database optimizer over linear algebra expressions. We deem this approach complimentary, and we plan to explore in the future the opportunities brought by the unified representation. Tensor execution of relational operators has natural commonalities with vectorized execution over columnar data [36]. Indeed tensor operators are mapped to vectorized instruction sets on CPU (when available). Interestingly, we find that TCRs (and the tensor abstraction) is flexible enough to support both vectorized and compiled execution [62]; the latter implemented through the TVM compilation stack for example. In both cases, no low-level details on vectorized execution or LLVM compilation are required, which we think makes easier the exploration of novel (hybrid) strategies.

It is well known that graph algorithms can be mapped over linear algebra operations [14]. To our knowledge, we are however the first showing that graph algorithms can be run efficiently over TCRs. As for relational operators, TCRs are flexible enough to allow switching between a linear algebra-based implementation to an imperative one, as we describe in Section 4.

Finally, several proposals exist on how to run relational operators on hardware accelerators (e.g., GPU [71], FPGA [41] and even TPU [45]). The same applies for graph algorithms and traditional ML [37, 76]. In this work we claim that similar, or even better, performance can be reached by targeting the tensor abstraction and using TCRs, rather than using low-level languages and primitives (e.g., CUDA). In recent years, several companies have proposed to accelerate specific workloads or systems, e.g., Spark-RAPIDS [26], BlazingSQL [7], OmnisciDB [19]. Compared to these, we think that the tensor abstraction will allow us to be hardware and library agnostic, while enabling similar performance.

6 CHALLENGES AND OPPORTUNITIES

In this paper we investigate whether the tensor abstraction can be used for programming general computations that can be run end-to-end on hardware accelerators. Our experience [61] and preliminary experiments show that good performance is indeed achievable, thanks to the massive investments poured into TCRs, as well as the related optimizations [28, 39, 48, 53, 73, 79].

TCR coverage landscape. TCRs have an excellent coverage of dense arithmetic operations as well as operations manipulating statically sized tensors. Additionally, sorting and simple aggregation is very efficient, and we therefore expect sort-based grouping and sort-merge joins to have good performance. However, there are also "pain points" that hinder TCRs from executing efficiently and easily operations common in other classes of algorithms. For example, as we saw in the experimental evaluation of PageRank, sparse operations are very inefficient compared to other libraries (e.g. scipy). However, the current trend towards Graph Neural Networks (GNNs) and sparsity in DNNs in general [30] has increased the interest of the community towards this important topic. As GNNs are getting more popular, we expect better TCR coverage of sparse data operations in the near future. Additionally, there is limited coverage of group aggregates (except for very basic cases such as `scatter_add`, but the community is trying to fix this limitation [23]) and pipelining operators to avoid intermediate materialization of results (although TVM is improving this thanks to its ability to do operation fusion). There is almost no support in regards to other data structures (e.g., hash maps), non-numeric data types (e.g., strings), and inputs of irregular shape and size. These features are specifically required for supporting, for example, compression or text data types. Finally, we find that the tensor abstraction shines when one needs to do bulk data transformations, while it lacks flexibility and performance for fine-grained operations. For example, predication [68] is hard to express using the tensor API.

Challenges. While our initial results are promising, there are several limitations that need to be addressed. The two bigger challenges are (1) compilation time, (2) data movement between CPU and memory. The database community is actively working on improving compilation performance for SQL queries [54], and similarly the systems community is working on Just-In-Time compilation approaches for DL models [47]. Driven by these ideas, we could devise a scheme where interpreted code is executed first while generating machine code on-the-fly. Regarding data movements, our goal is to compile computations end-to-end such that data movement is minimized. Other techniques such as pipelining data access with compute, paging, and batching can be used both at conversion and runtime to improve data movement performance.

Hardware trends. In this paper, we focused on hardware acceleration over GPUs, given their widespread adoption and availability. An emerging trend in hardware acceleration is dataflow-based architectures both in academia [74] and in industry by companies such as SambaNova [24], Graphcore [15], or Cerebras [9]. Interestingly, those approaches also expose TCRs as a programming interface [10, 22, 60]. We are planning to test our approach on this hardware for relational and graph workloads.

Compiled vs vectorized execution. Main memory query rocessors [78] and databases [36, 62] are increasingly getting compute-bound rather than memory-bound. This is because compilation [62] and vectorization [36] approaches achieve even better hardware utilization. TCRs are built from the ground up for compute-bound workloads (i.e., DL). Additionally, the high-level tensor abstraction on top of TCRs considerably simplifies the process of implementing new algorithms, since there is no need to understand LLVM internals, or to write GPU kernels. We therefore believe that using TCRs as a component of main memory query processors will enable a new wave of systems and innovative algorithms. For instance, any vectorized algorithm (or in general any algorithm over columnar data) can be implemented on top of TCRs. Since TCRs also allow compilation and operator fusion, an exciting direction is the exploration of vectorized vs compiled implementations [51], and when to pick one or the other based on, e.g., the hardware.

Cross-optimizations. Apache Spark [78] has shown that Data Frames can be used to express relational, graph, and ML computations, although cross-optimizations between them is limited. SystemML [52] and others [75] have considered using relational algebra (and database-style optimizers) for logically (co-) optimizing linear and relational algebra. For example in Raven [50] we explored how to cross-optimize relational and traditional ML, beyond linear algebra (e.g., decision trees). With the approach presented in this paper, a new level of cross-optimizations will be possible since all computations are expressed using the same (tensor) abstraction.

ACKNOWLEDGEMENTS

We thank Carlo Curino and Masahiro Masuda for their discussions and suggestions on early draft of this work.

REFERENCES

[1] 2015. H2O Algorithms Roadmap. https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/flow/images/H2O-Algorithms-Road-Map.pdf.

[2] 2018. CNTK. https://docs.microsoft.com/en-us/cognitive-toolkit/.

[3] 2018. MXNet. https://mxnet.apache.org/.

[4] 2018. TensorFlow. https://www.tensorflow.org.

[5] 2019. RAPIDS Forest Inference Library. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows -per-second-19558890bc35.

[6] 2020. Blas. http://www.netlib.org/blas/.

[7] 2020. BlazingSQL. https://blazingsql.com/.

[8] 2020. Cardinality Estimation in SQLServer. https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver15.

[9] 2020. Cerebras. https://cerebras.net/.

[10] 2020. Cerebras Software. https://cerebras.net/product/#software.

[11] 2020. Cost of training GPT-3. https://bit.ly/361o7I0.

[12] 2020. cugraph. https://github.com/rapidsai/cugraph.

[13] 2020. Distinct using hashing. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver15.

[14] 2020. GraphBlas. https://people.engr.tamu.edu/davis/GraphBLAS.html.

[15] 2020. GraphCore. https://www.graphcore.ai/.

[16] 2020. Intel MKL. https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines.html.

[17] 2020. Neo4j. https://neo4j.com/.

[18] 2020. Numba. https://numba.pydata.org/numba-doc/latest/index.html.

[19] 2020. OmnisciDB. https://www.omnisci.com/.

[20] 2020. ONNX. https://github.com/onnx/onnx/blob/master/docs/Operators.md.

[21] 2020. OpenCL. https://www.khronos.org/opencl/.

[22] 2020. PyTorch Release for IPU. https://medium.com/pytorch/graphcore-announces-production-release-of-pytorch-for-ipu-f1a846de1a2f.

[23] 2020. PyTorch scatter with different reduce modes. https://github.com/pytorch/pytorch/issues/22378.

[24] 2020. Sambanova: Massive Models for Everyone. https://sambanova.ai/.

[25] 2020. Scipy. https://github.com/scipy/scipy/.

[26] 2020. Spark-RAPIDS. https://nvidia.github.io/spark-rapids/.

[27] 2020. Statistics in SQLServer. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15.

[28] 2020. Tensorflow XLA. https://www.tensorflow.org/xla.

[29] 2020. Xilinx Vitis Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[30] 2021. The future of sparsity in deep neural networks. https://www.sigarch.org/the-future-of-sparsity-in-deep-neural-networks/.

[31] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. 2019. Machine Learning at Microsoft with ML.NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2448–2458. https://doi.org/10.1145/3292500.3330667

[32] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[33] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[34] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528.

[35] OpenAI Blog. 2018. AI and Compute. openai.com/blog/ai-and-compute/.

[36] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. www.cidrdb.org, 225–237. http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05

[37] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[38] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[39] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[40] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[41] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *VLDB J.* 29, 1 (2020), 33–59. https://doi.org/10.1007/s00778-019-00581-w

[42] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

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

[44] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[45] Pedro Holanda and Hannes Mühleisen. 2019. Relational Queries with a Tensor Processing Unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 3 pages. https://doi.org/10.1145/3329785.3329932

[46] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB. *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond - BeyondMR'17* (2017). https://doi.org/10.1145/3070607.3070608

[47] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dongjin Shin, and Byung-Gon Chun. 2019. JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 453–468. https://www.usenix.org/conference/nsdi19/presentation/jeong

[48] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[49] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 http://arxiv.org/abs/1704.04760

[50] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[51] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[52] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[53] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[54] A. Kohn, V. Leis, and T. Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208. https://doi.org/10.1109/ICDE.2018.00027

[55] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. *A Relational Approach to the Compilation of Sparse Matrix Programs.* Technical Report. USA.

[56] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 393–394.

[57] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv e-prints*, Article arXiv:2002.11054 (Feb. 2020), arXiv:2002.11054 pages. arXiv:2002.11054 [cs.PL]

[58] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[59] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[60] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. arXiv:1807.04188 [cs.LG]

[61] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 899–917. https://www.usenix.org/conference/osdi20/presentation/nakandala

[62] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[63] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[64] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[65] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[66] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1707–1718. https://doi.org/10.14778/3007328.3007336

[67] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[68] Kenneth A. Ross. 2004. Selection Conditions in Main Memory. *ACM Trans. Database Syst.* 29, 1 (March 2004), 132–161. https://doi.org/10.1145/974750.974755

[69] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 49–68. https://doi.org/10.1145/2517349.2522715

[70] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG]

[71] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[72] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[73] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[74] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: accelerating machine learning from relational data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 309–321.

[75] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

[76] Carl Yang, Aydin Buluc, and John D. Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. arXiv:1908.01407 [cs.DC]

[77] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 293–306. https://www.usenix.org/conference/atc20/presentation/yuan

[78] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[79] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng