Tensor Relational Algebra for Distributed Machine Learning System Design

Binhang Yuan[†], Dimitrije Jankov[†], Jia Zou[‡], Yuxin Tang[†], Daniel Bourgeois[†], Chris Jermaine[†]
[†]Rice University; [‡]Arizona State University
[†]{by8, dj16, yuxin.tang, dcb10, cmj4}@rice.edu; [‡]jia.zou@asu.edu

ABSTRACT

We consider the question: what is the abstraction that should be implemented by the computational engine of a machine learning system? Current machine learning systems typically push whole tensors through a series of compute kernels such as matrix multiplications or activation functions, where each kernel runs on an AI accelerator (ASIC) such as a GPU. This implementation abstraction provides little built-in support for ML systems to scale past a single machine, or for handling large models with matrices or tensors that do not easily fit into the RAM of an ASIC. In this paper, we present an alternative implementation abstraction called the *tensor relational algebra* (TRA). The TRA is a set-based algebra based on the relational algebra. Expressions in the TRA operate over binary tensor relations, where keys are multi-dimensional arrays and values are tensors. The TRA is easily executed with high efficiency in a parallel or distributed environment, and amenable to automatic optimization. Our empirical study shows that the optimized TRA-based back-end can significantly outperform alternatives for running ML workflows in distributed clusters.

PVLDB Reference Format:
Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois,and Chris Jermaine. Tensor Relational Algebra for Distributed Machine Learning System Design. PVLDB, 14(8): 1338 - 1350, 2021.
doi:10.14778/3457390.3457399

1 INTRODUCTION

In widely-used machine learning (ML) systems such as TensorFlow [4] and PyTorch [3], computations are usually specified operationally: a sequence of mathematical operations such as matrix multiplications, activation functions (ReLU, tanh), convolutions, etc., are applied to a set of input tensors to define a computation. HPC systems such as ScaLAPACK [17] and distributed analytic packages such as Dask [1] offer a similar, operational interface. Operations are "black boxes" in the sense that the internals of the operation are mostly opaque to the system. An operation such as a matrix multiply is not a logical operation that the ML system figures out how to best optimize at runtime. Instead, it is a physical operation that has to run somewhere, on some hardware, via the invocation of a computational kernel.

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457399

This operational approach has certain drawbacks, namely that the system has limited latitude for automatic optimization. The programmer is responsible for making sure that the operations can run successfully using available resources (such as the amount of RAM on each GPU), and if the operations cannot run successfully, the programmer must figure out how to break the operations up into smaller pieces that can run. Tasks such as getting a computation to run on multiple GPUs or on multiple machines in a distributed cluster so as to minimize communication are left to the programmer.

Toward declarative tensor programming. There has been work on programming models that are more declarative. PyTorch and TensorFlow now both support variants of Einstein notation—a classical notation for specifying operations over tensors, and work on optimizing such computations has made its way into commercial systems [8]. Researchers have proposed variants of the Ricci calculus as a specification language for ML [32]. There have been other proposals for declarative tensor programming languages that allow for automatic generation of compute kernels that can be optimized to handle the data at hand such as Tensor Comprehensions [43].

Nevertheless, while there has been attention paid at the question of how to specify ML computations declaratively, there has been little attention paid to the question of what the correct implementation abstraction for ML system design should be. That is, what target should a ML system *back-end* present to the *front-end*?[1] There are several requirements for such a back-end interface. It should be able to express most/all important ML or numerical computations. It should be hardware agnostic, but computations specified using the interface should be easily scaled to multiple ASICs and multiple machines. It should allow for computations over very large input data. It should facilitate easy, automatic optimization. And it should provide for execution times that are as fast as a carefully-designed, "hand-built" computation on top of the very best tools such as ScaLAPACK, Dask, TensorFlow, or PyTorch.

The tensor relational algebra. In this paper, we argue that the implementation abstraction that should be offered by a ML system back-end is the *tensor relational algebra*, or TRA for short. The TRA is a simple variant of the classical relational algebra (RA), which serves as the standard implementation abstraction for modern relational database system design. The key difference is that in the TRA, the relations over which computations are performed are always binary relations between k-dimensional vectors (keys) and r-dimensional tensors.

[1]Throughout the paper, we use the term *front-end* to refer to the programmer-facing API and compiler; in PyTorch, for example, this would be the part of the system that accepts Einstein notation and transforms it into a set of executable operations. We use the term *back-end* to refer to the sub-system that actually executes the computation.

Of course, it is widely understood that a k-dimensional tensor can be stored as a binary relation between k-dimensional keys and real number values, e.g. [25]. Thus, why not use classical RA as the implementation abstraction? There is good evidence that a compute engine based upon such an abstraction will perform very poorly over the dense-tensor computations common in deep learning [35]: the overhead associated with storing each entry in a high-dimensional tensor as a separate tuple and moving billions of tuples through a system can result in poor performance, especially when the competition is a high-performance CPU or GPU kernel. This is why the TRA specifically allows for tensors or "chunks" of a larger tensor to be stored in each tuple—the fixed, per-tuple cost is incurred by a much smaller number of tuples.

We argue that the TRA has three key advantages as an implementation abstraction: expressivity, easy optimization, and high performance. The joins and aggregations offered by the TRA can implement the indexed operations required by the Einstein notation and related specification languages. It is easy to implement and optimize relational operations in a distributed environment, as the decades of success enjoyed by relational database systems has demonstrated. And by design, an ML system back-end implementing the TRA is able to run classical HPC algorithms which rely on decomposing matrices into chunks, and it can run them almost as well as special-purpose HPC softwares such as ScaLAPACK.

Our Contributions. We propose the TRA as well as an implementation algebra (IA) which is easy to implement in a distributed system. We consider how computations in the TRA can be transformed into computations in the IA, and propose a set of transformations or equivalences that allow re-writes of computations in the IA. Finally, we implement a prototype of the IA, and show that it can enable efficient, distributed implementations of ML computations, which can reach or even significantly outperform the existing HPC and ML systems including ScaLAPACK, Dask, TensorFlow, and PyTorch.

2 TRA OVERVIEW

2.1 Motivation

There has been significant interest in tensor manipulation languages, which allow for declarative specification of ML and numerical computations. This simplest of these is the Einstein notation, which provides a way to write a tensor computation like the form:

$$\forall i, j : \mathbf{C}_{i,j} \leftarrow \sum_{k} \mathbf{A}_{i,k} \times \mathbf{B}_{k,j}.$$

This example describes matrix multiplication. The value i-th row and j-th column of the output is the dot product of the i-th row of input matrix $\mathbf{A}$ and the j-th column of input matrix $\mathbf{B}$. Different proposals have different variations on this idea [8, 32, 43], but the common features are that (1) values from different tensors are fed into a scalar function (such as the multiplication above) by matching indices, and (2) those dimensions are summed over.

Languages such as the Einstein notation provide an excellent, declarative interface for programming an ML system—much as SQL provides the interface for relational systems. But the question remains: what is the correct implementation abstraction for ML systems? That is, what is the interface that the back-end should export, which can be targeted by an ML programming system compiler and auto-differentiation engine that make up the ML system's front-end?

2.2 TRA: The Basics

We propose the TRA as this ML implementation abstraction. The TRA operates over *tensor relations* containing pairs of the form:

```
(key, array).
```

Conceptually, these tensor relations store sets of arrays. Each key value serves, not surprisingly, as the key for the pair.

Tensors are decomposed into sets of sub-tensors to represent them as tensor relations. For example, consider the matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix},$$

we may store this as a tensor relation

$$\mathsf{R}_A = \left\{ \left(\langle 0, 0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \right), \left(\langle 0, 1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right), \left(\langle 1, 0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \right), \left(\langle 1, 1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \right) \right\}.$$

The TRA offers a similar set of operations to the RA: joins, aggregations, selections. It makes an excellent compilation target for tensor manipulation languages like the Einstein notation for several reasons. First, these languages typically match elements in tensors based on shared index values (which can be implemented as relational joins) and then sum out various dimensions (implemented as aggregations). The problem with using the RA as an implementation abstraction for tensors, where tensors are stored relationally as keys identifying a single non-zero value in a tensor, is performance. Tensors can have millions or billions of entries, and it seems impossible to build a back-end with acceptable performance if it has to process one tuple per non-zero value.

Thus, the TRA operates over sub-tensors or "chunks", and expects the ML system front-end to supply a kernel function (typically a high-performance CPU or GPU operation) to operate over the sub-tensors themselves. This does complicate the TRA compared to the RA, but as we show experimentally, this modification makes it possible for a TRA-based back-end to significantly outperform the back-ends offered by TensorFlow and PyTorch.

2.3 TRA: the Implementation Algebra

One of the key advantages of the TRA is that like the RA, there are a number of re-writes, inherited from the RA (such as push-down of selections) that can be used to optimize TRA computations.

However, when designing an ML system back-end, one of our key goals is to distribute ML computations across multiple ASICs or multiple machines. Such distributed computations have many different implementations. Consider the matrix multiply required to push a batch of feature vectors (represented as one matrix) through the links into a fully connected layer in a neural network (the weights are represented as a second matrix). This could be implemented by decomposing the feature matrix into sub-matrices at each site (each containing a subset of the feature vectors) and broadcasting the

weight matrix to all sites. This is the common "data parallel" implementation, in ML system parlance. Or, one could fully decompose both matrices and apply a complex distributed algorithm, such as the 3D algorithm [9]. This would be a combined data and "model parallel" implementation in ML system parlance. Crucially, the TRA cannot express the distinctions among such implementations.

As such, we also propose an *implementation algebra* (IA) that can express these different distributed computations, as well as a simple compilation strategy from the TRA to the IA. Further, we propose an extensive set of re-writes for the IA that allow such disparate implementations to be reached from an initial TRA computation, and a simple cost model. Given this, an ML system back-end would export a simple, TRA-based interface which says nothing about distribution to multiple machines or ASICs. A TRA computation can then be compiled into a computation in the IA, and optimized into a high-performance, distributed computation.

3 TENSOR RELATIONAL ALGEBRA

The RA operates over `(key, array)` pairs. Define an *array type* that consists of:

(1) A *rank* $r \in \mathbb{Z}^*$
(2) A *bound* $\mathbf{b} \in (\mathbb{Z}^*)^r$.

For two vectors $\mathbf{u} = \langle u_i \rangle$ and $\mathbf{v} = \langle v_i \rangle$, define $\mathbf{u} \leq \mathbf{v} \equiv \wedge_i (u_i \leq v_i)$. Define $\mathbf{u} < \mathbf{v}$ similarly. Informally, we say that an array of rank r is *bounded* by vector $\mathbf{b}$ if the array is r dimensional, and for any r-dimensional vector $\mathbf{i}$ that is less than the bound, $\texttt{array}_\mathbf{i}$ returns a real number. Formally:

(1) For any index $\mathbf{i} \in (\mathbb{Z}^*)^r, \vec{0} \leq \mathbf{i} < \mathbf{b} \implies \texttt{array}_\mathbf{i} \in \mathbb{R}$.
(2) $\neg(\vec{0} \leq \mathbf{i} < \mathbf{b}) \implies \texttt{array}_\mathbf{i} = \perp$. That is, for any index $\mathbf{i}$ outside of the bound $[\vec{0}, \mathbf{b}]$, $\texttt{array}_\mathbf{i}$ is undefined.

Subsequently, we denote the set of all arrays of rank r and bound $\mathbf{b}$ as $T^{(r,\mathbf{b})}$. Thus, $T^{(r,\mathbf{b})}$ defines an array type.

We denote the power set of $(\mathbb{Z}^*)^k \times T^{(r,\mathbf{b})}$ as $R^{(k,r,\mathbf{b})}$; this is the set of all possible tensor relations with k-dimensional keys, storing arrays of type $T^{(r,\mathbf{b})}$.

3.1 Operations in Tensor Relational Algebra

Given this, the TRA is essentially a set of higher-order functions over tensor relations. That is, each operation takes as input a kernel function defined over multi-dimensional arrays (in practice, this function is likely be an array-based MKL, CUDA, or Verilog kernel) and returns a function over tensor relations.

We begin by giving an overview of the higher-order functions taking binary functions as input: aggregation (denoted using Σ) and join (denoted using $\bowtie$).

(1) *Aggregation* is a function:

$$\Sigma : \left((\mathbb{Z}^*)^g \times \left(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b})}\right)\right) \rightarrow \left(R^{(k,r,\mathbf{b})} \rightarrow R^{(g,r,\mathbf{b})}\right)$$

$\Sigma_{(\texttt{groupByKeys, aggOp})}$ takes as input a list of key dimensions to aggregate according to `groupByKeys` as well as an array kernel operation `aggOp`, and then returns a function that takes as input a tensor relation, groups the arrays in the relation based upon the indicated key values, and applies `aggOp` to the arrays in the group.

Consider the matrix $\mathbf{A}$ from the last section. We can sum up the individual arrays vertically using

$$\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}(\mathsf{R}_A)$$

which gives:

$$\left\{\left(\langle 0 \rangle, \begin{bmatrix} 10 & 12 \\ 14 & 16 \end{bmatrix}\right), \left(\langle 1 \rangle, \begin{bmatrix} 18 & 20 \\ 22 & 24 \end{bmatrix}\right)\right\}.$$

Because of the argument $\langle 1 \rangle$, the call $\Sigma_{(\langle 1 \rangle, \texttt{matAdd})}$ constructs an aggregation function that groups all pairs having the same value for the key in position 1, and sums them. Or we could sum up the individual arrays into a single array using:

$$\Sigma_{(\langle \rangle, \texttt{matAdd})}(\mathsf{R}_A)$$

which gives:

$$\left\{\left(\langle \rangle, \begin{bmatrix} 28 & 32 \\ 36 & 40 \end{bmatrix}\right)\right\}.$$

(2) *Join* is a function:

$$\bowtie : \left((\mathbb{Z}^*)^g \times (\mathbb{Z}^*)^g \times \left(T^{(r_l,\mathbf{b}_l)} \times T^{(r_r,\mathbf{b}_r)} \rightarrow T^{(r_o,\mathbf{b}_o)}\right)\right) \rightarrow \left(R^{(k_l,r_l,\mathbf{b}_l)} \times R^{(k_r,r_r,\mathbf{b}_r)} \rightarrow R^{(k_l+k_r-g,r_o,\mathbf{b}_o)}\right)$$

$\bowtie_{(\texttt{joinKeysL, joinKeysR, projOp})}$ takes as input a set of key dimensions to join on from the left and from the right, as well as an operation to run over all `(leftArray, rightArray)` pairs that are created during the join, and returns a function that performs the join and applies `projOp` to all pairs. Similar to a natural join in classical databases systems, the output key is all of the key values from the left input, with all of the key values from the right input appended to them, subject to the constraint that no value in `joinKeysR` is repeated a second time.

With join and aggregation we may implement matrix multiply over two matrices stored as tensor relations. Imagine that we want to implement $\mathbf{A} \times \mathbf{A}$ for the matrix $\mathbf{A}$ defined previously, where $\mathbf{A}$ is stored as a tensor relation R_A. This can be written as:

$$\Sigma_{(\langle 0,2 \rangle, \texttt{matAdd})}\left(\bowtie_{(\langle 1 \rangle, \langle 0 \rangle, \texttt{matMul})}(\mathsf{R}_A, \mathsf{R}_A)\right)$$

This computes a matrix multiply of the matrix $\mathbf{A}$ because all of the pairs in R_A are first joined on key index 1 from the first instance of R_A equaling key index 0 from the second instance of R_A. Each pair of arrays are then multiplied using the kernel `matMul`. For example,

$$\left(\langle 0, 1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right) \text{ and } \left(\langle 1, 0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right)$$

are joined to produce

$$\left(\langle 0, 1, 0 \rangle, \begin{bmatrix} 111 & 122 \\ 151 & 166 \end{bmatrix}\right).$$

The index $\langle 0, 1, 0 \rangle$ in this output pair is a combination of $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$ from the two input pairs, with the redundant index entry dropped (redundant because we know that two of the entries in positions 1 and 0, respectively, are repeated due to the join). Next, the arrays are aggregated using `matAdd`, summing out index 1 (keeping indices $\langle 0, 2 \rangle$ as `groupByKeys`), to complete the matrix multiply.

In contrast to join and aggregation, rekey, filter and transform are higher-order functions taking a unary function as input.

(3) *ReKey* allows manipulation of keys:

$$\textsc{ReKey} : \left((\mathbb{Z}^*)^{k_i} \to (\mathbb{Z}^*)^{k_o}\right) \to \left(R^{(k_i,r,\mathbf{b})} \to R^{(k_o,r,\mathbf{b})}\right)$$

$\textsc{ReKey}_{(\texttt{keyFunc})}$ applies the `keyFunc` on every key in the relation and generates a new key.
(4) *Filter* is a function:

$$\sigma : \left((\mathbb{Z}^*)^{k} \to \{\texttt{true}, \texttt{false}\}\right) \to \left(R^{(k,r,\mathbf{b})} \to R^{(k,r,\mathbf{b})}\right)$$

$\sigma_{(\texttt{boolFunc})}$ returns a function that accepts a tensor relation and filters each of the tuples in the tensor relation by applying `boolFunc` to the keys in the tuples.
(5) *Transform* is a function:

$$\lambda : \left(T^{(r_i,\mathbf{b}_i)} \to T^{(r_o,\mathbf{b}_o)}\right) \to \left(R^{(k,r_i,\mathbf{b}_i)} \to R^{(k,r_o,\mathbf{b}_o)}\right)$$

$\lambda_{(\texttt{transformFunc})}$ returns a function that accepts a tensor relation and applies the kernel function `transformFunc` to the array in each tuple from the tensor relation.

For an example of the rekey, filter and transform operations, assume we have a kernel operation `diag` that diagonalizes a matrix block, a function $\texttt{isEq}(\langle \texttt{k}_0, \texttt{k}_1\rangle) \mapsto \texttt{k}_0 = \texttt{k}_1$ that accepts a key and returns true if entries in position 0 and 1 in the key are identical to one another, and a function $\texttt{getKey0}(\langle \texttt{k}_0, \texttt{k}_1\rangle) \mapsto \langle \texttt{k}_0\rangle$ that returns the first dimension of a key. We can use these functions along with filter, rekey, and transform to diagonalize a matrix $\mathbf{A}$ represented as a tensor relation $\texttt{R}_A$, by first examining the keys to remove all pairs that do not contain entries along the diagonal, and then diagonalizing the resulting arrays:

$$\lambda_{(\texttt{diag})}\left(\textsc{ReKey}_{(\texttt{getKey0})}\left(\sigma_{(\texttt{isEq})}\left(\texttt{R}_A\right)\right)\right).$$

In addition, there are a number of operations that can be used to alter the organization of arrays within a tensor relation. This allows the manipulation of how a tensor is represented as a tensor relation. For this purpose, we have tile and concat:
(6) *Tile*:

$$\textsc{Tile} : \left(\mathbb{Z}^* \times \mathbb{Z}^*\right) \to \left(R^{(k,r,\mathbf{b})} \to R^{(k+1,r,\mathbf{b}')}\right)$$

$\textsc{Tile}_{(\texttt{tileDim, tileSize})}$ returns a function that decomposes (or tiles) each array along a dimension `tileDim` to arrays of the target `tileSize` (by applying the `arrayTileOp` function on the array). As a result, a new key dimension is created, that effectively counts which tile the tuples holds along the tiling dimension.

For example, consider the matrix $\mathbf{B}$,

$$\mathbf{B} = \begin{bmatrix} 1 & 2 & 5 & 6 & 9 & 10 & 13 & 14 \\ 3 & 4 & 7 & 8 & 11 & 12 & 15 & 16 \end{bmatrix},$$

partitioned by columns and stored in tensor relation:

$$\texttt{R}_B = \left\{\left(\langle 0\rangle, \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix}\right), \left(\langle 1\rangle, \begin{bmatrix} 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix}\right)\right\}$$

If we make the call $\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)$, we will decompose each array along dimension 1, creating one new array for each two columns. In addition, a new key dimension is created, that effectively counts which tile the pair holds along the tiling dimension:

$$\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right) = \left\{\left(\langle 0,0\rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1\rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right)\right\}.$$

We may sometimes find ourselves in a situation where it is necessary to manipulate the key in each pair in a tensor relation so that the key is consistent with the desired interpretation. For example, the tensor relation $\texttt{R}_B$ defined above can represent a matrix with eight columns and two rows, so $\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)$ is inconsistent with this, logically representing a matrix having four columns and four rows. For this purpose, we can leverage the ReKey operator as we defined before.

For example, we can rekey the output of $\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)$ so that logically, it corresponds to a two-by-eight matrix:

$$\textsc{ReKey}_{(\langle \texttt{k}_0,\texttt{k}_1\rangle \to \langle 2\texttt{k}_0+\texttt{k}_1\rangle)}\left(\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)\right)$$

This will result in:

$$\left\{\left(\langle 0\rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 1\rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 2\rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 3\rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right)\right\}$$

Finally, we have the ability to undo a tiling.
(7) *Concat*:

$$\textsc{Concat} : \left(\mathbb{Z}^* \times \mathbb{Z}^*\right) \to \left(R^{(k,r,\mathbf{b})} \to R^{(k-1,r,\mathbf{b}')}\right)$$

$\textsc{Concat}_{(\texttt{keyDim, arrayDim})}$ is an inverse to tile, which first groups all pairs in the relation using all of the key dimensions *other than* `keyDim`, then concatenates all of the arrays in each group along `arrayDim`, with the concatenation ordering provided by `keyDim`.

A call to $\textsc{Concat}_{(1,1)}\left(\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)\right)$ first groups all pairs in $\textsc{Tile}_{(1,2)}\left(\texttt{R}_B\right)$ using all of the key dimensions other than key dimension 1, and then concatenates the arrays in each group along array dimension 1, with the ordering provided by key dimension 1. Hence, this computation simply results in the recovery of $\texttt{R}_B$.

3.2 Integrity Constraints and Closedness

There are two important integrity constraints that each tensor relation must follow: uniqueness of keys, and a lack of "holes" in the tensor relation. The primary reason for defining these constraints is facilitating easy, cost-based optimization. With such constraints, cardinality estimation, one of the most vexing problems in relational optimization, goes away—see Section 5.3. Further, neither is particularly burdensome when expressing computations using the TRA. In fact, if the interpretation of a tensor relation of type $R^{(k,r,\mathbf{b})}$ is that it represents a r-dimensional tensor decomposed into chunks, these constraints are quite natural:

- *Uniqueness*: every key should be unique in a tensor relation.
- *Continuity*: there are no "holes". Given a tensor relation R, of key-arity k, define the *frontier* of R as $\textsc{Front}(\texttt{R})$. $\mathbf{f} = \textsc{Front}(\texttt{R})$ is a k-dimensional vector that bounds all keys in R. That is, for each key vector $\mathbf{k}$ in R, $\mathbf{k} < \mathbf{f}$. Further, the frontier is the "smallest" vector bounding R, in that for any other vector $\mathbf{f}'$ bounding R, $\mathbf{f} \leq \mathbf{f}'$. *Continuity* requires that for any vector $\mathbf{k} < \mathbf{f}$, some tuple in R have the key $\mathbf{k}$.

It is easy to show that for the majority of TRA operations—the exceptions being the rekey and filter operations—tensor relations are closed. That is, if the input(s) are tensor relation(s) that obey uniqueness and continuity, then the output must be a tensor relation that obeys these constraints. Filtering a tensor relation or altering the keys can obviously violate the constraints, where the the former probably leads to holes in the resulting relation, and the latter can result in repeated key values. Analyzing a TRA expression to automatically detect whether it can violate these constraints is left as future work; we conjecture that if the filtering predicate (or rekeying computation) are limited to simple arithmetic expressions, it may be possible to check for closedness using an SMT solver [19].

4 IMPLEMENTATION ALGEBRA

We now describe TRA's *implementation algebra* (IA) that is suitable for execution in a parallel/distributed environment.

In IA, we extend each `(key, array)` tuple in a tensor relation with an additional `site` attribute, so that a *physical tensor relation* R will consist of triples:

`(key, array, site)`.

The `site` attribute takes a value in $\{1, 2, ..., s\}$ where s is the number of computation sites. Conceptually, the `site` value indicates the location where the tuple is stored; this could be a machine in a distributed cluster, or a compute unit like a GPU.

Each physical tensor relation can map a particular `(key, array)` pair to one or more sites. There are a few especially important mappings, recognized by the predicates $\textsc{All}()$ and $\textsc{Part}_D()$:

(1) If $\textsc{All}(R) = \text{true}$, it indicates that if we project away the `array` attribute, the resulting set will take the value:

$$\{\mathbf{k} \text{ s.t. } \mathbf{k} \leq \textsc{Front}(R) \times \{1...s\}\}$$

where $\textsc{Front}(R)$ is the frontier of R (the frontier of a physical tensor relation is defined as in a "regular" tensor relation). In other words, this means that each possible `(key, array)` tuple in R appears at all sites.

(2) If $\textsc{Part}_D(R) = \text{true}$ for some set $D \subseteq \{1...k\}$, it indicates that: (i) for a given `key` value, there is only one tuple in R and (ii) two tuples with the same key values for all dimensions in D must be found on the same site. In other words, R is partitioned according to the key dimensions in the set D.

We are ready to describe the IA. Let $\mathcal{R}^{(k,r,\mathbf{b},s)}$ specify the set of all valid physical tensor relations with key-arity of dimension k, storing arrays of type $T^{(r,\mathbf{b})}$, and partitioned across s sites.

The first two operations are concerned with manipulating the assigning of tuples in a physical relation to sites, while the later four operations operate over the `key` and `array` attributes.

(1) *Broadcast* is defined as

$$\textsc{Bcast} : \mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)}$$

Given a physical tensor relation, $\textsc{Bcast}$ simply ensures that each tuple takes each site value, so that (i) the set of `(key, array)` pairs is unchanged after $\textsc{Bcast}$, but (ii) in any physical relation R output from a broadcast, $\textsc{All}(R) = \text{true}$.

(2) *Shuffle* is defined as:

$$\textsc{Shuf} : 2^{\{1...k\}} \rightarrow \left(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)}\right)$$

$\textsc{Shuf}_{\texttt{(partDims)}}$ is a function that accepts a set of key dimensions, and returns a function that accepts physical tensor relation, and then repartitions the physical tensor relation, so that (i) the set of `(key, array)` pairs is unchanged after $\textsc{Shuf}$, but (ii) in any physical relation R output from a shuffle, $\textsc{Part}_{\texttt{partDims}}(R) = \text{true}$.

(3) *Local join* is an extension of the TRA's join operation:

$$\bowtie^L : \left((\mathbb{Z}^*)^g \times (\mathbb{Z}^*)^g \times \left(T^{(r_l,\mathbf{b}_l)} \times T^{(r_r,\mathbf{b}_r)} \rightarrow T^{(r_o,\mathbf{b}_o)}\right)\right)$$
$$\rightarrow \left(\mathcal{R}^{(k_l,r_l,\mathbf{b}_l,s)} \times \mathcal{R}^{(k_r,r_r,\mathbf{b}_r,s)} \rightarrow \mathcal{R}^{(k_l+k_r-g,r_o,\mathbf{b}_o,s)}\right)$$

Similar to TRA join ($\bowtie$), $\bowtie^L{}_{\texttt{(joinKeysL, joinKeysR, projOp)}}$ takes as input a set of key dimensions to join on from the left and from the right, as well as a kernel operation to run over all `(leftArray, rightArray)` pairs that are created during the join. The key difference that the local join combines *only* on pairs from the left and right inputs that have the **same** `site` values. If two tuples successfully join, the corresponding output tuple will have the `site` value as those input tuples.

(4) *Local aggregation* is an extension of TRA aggregation:

$$\Sigma^L : \left((\mathbb{Z}^*)^k \times \left(T^{(r,\mathbf{b})} \times T^{(r,\mathbf{b})} \rightarrow T^{(r,\mathbf{b})}\right)\right)$$
$$\rightarrow \left(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(g,r,\mathbf{b},s)}\right)$$

Like TRA aggregation (Σ), $\Sigma^L{}_{\texttt{(groupByKeys, aggOp)}}$ takes as input a list of key dimensions to aggregate over `groupByKeys` as well as a kernel function `aggOp`. However, it returns a function that takes as input a physical tensor relation, groups the arrays in the relation based upon the indicated key values *and* the site value, and applies `aggOp` to the arrays in the group. Each output tuple in the resulting, physical tensor relation will take its `site` value from the `site` value of the set of input tuples that were aggregated to produce it.

(5) IA has a filter:

$$\sigma^L : \left((\mathbb{Z}^*)^g \rightarrow \{\texttt{true}, \texttt{false}\}\right) \rightarrow \left(\mathcal{R}^{(k,r,\mathbf{b},s)} \rightarrow \mathcal{R}^{(k,r,\mathbf{b},s)}\right)$$

The only difference is that each accepted input tuple's `site` value is carried through the filter.

(6) Map provides two functionalities:

$$\lambda^L : \left(\left((\mathbb{Z}^*)^{k_i} \rightarrow \left((\mathbb{Z}^*)^{k_o}\right)^m\right) \times \left(T^{(r_i,\mathbf{b}_i)} \rightarrow \left(T^{(r_o,\mathbf{b}_o)}\right)^m\right)\right)$$
$$\rightarrow \left(\mathcal{R}^{(k_i,r_i,\mathbf{b}_i,s)} \rightarrow \mathcal{R}^{(k_o,r_o,\mathbf{b}_o,s)}\right)$$

$\lambda^L{}_{\texttt{(keyMapFunc,arrayMapFunc)}}$ is a multi-map. It returns a function that applies `keyMapFunc` to each `key` value in the input and applies `arrayMapFunc` to each `array` value in the input. Both `keyMapFunc` and `arrayMapFunc` return m output tuples per input tuple; the `site` value is simply copied from input to output. We subsequently call m the *arity* of `keyMapFunc`/`arrayMapFunc`. In most cases the arity of these functions will be one, but on some cases (such as replication-based matrix multiply, see Section 4.2.2), the arity will be greater.

5 COMPILATION AND OPTIMIZATION

To distribute tensor-based computations so that they can run efficiently requires an optimization framework. We consider three core questions related to actually distributing a computation specified in the TRA: (1) How is that TRA computation compiled into an

equivalent statement in IA? (2) What are a set of equivalence rules that allow computations in IA to be re-written, so as to produce different implementations that are known to produce the same results, but may run more efficiently? And (3) How to cost those different, equivalent implementations, so that a search strategy can be used to choose the most efficient one?

5.1 Compiling the TRA

A complete set of rules mapping from TRA operations to IA operations are listed in Table 1[2]. Note that though there can be multiple IA computations for a given TRA computation, the compiler will generate one of such IA computation as the initial plan, and an optimizer will typically be responsible for applying a search algorithm to produce the optimal physical plan represented by IA.

5.2 Equivalence Rules

Once a (possibly inefficient) computation in IA is produced, it can be optimized via the application of a set of equivalence rules. An *equivalence rule* for IA expressions holds if, for any input physical tensor relations, the two expressions produce equivalent outputs—two physical tensor relations are said to be equivalent if they contain the same set of `(key, array)` pairs after projecting away the `site` attribute.

There are two classes of equivalence rules we consider: *simple equivalence rules* which are often extensions of classic relational equivalence rules (e.g., commutative property of selections), and *domain-specific equivalence rules* that are more complex transformations that always hold, and tend to be useful for mathematical computations, such as matrix multiplications.

5.2.1 Simple Equivalence Rules. In Table 2, we give an extensive list of two types of simple equivalence rules: (i) those based on kernel function composition, and (ii) equivalence rules based on optimization of re-partitions.

Kernel function composition targets the order or location of the application of kernel functions in order to reduce the computation load and memory consumption. This is closely related to the idea of ML operator fusion, which has been explored in systems such as TVM [16] (though TVM does not consider the sort of distributed computations considered here). Re-partition rules formalize notions of distributed query optimization over tensor relations, and are primarily designed to reduce communication.

Such rules are surprisingly effective for optimizing distributed tensor manipulations. Consider the example of extracting the diagonal elements of matrix $\mathbf{X}$ plus matrix $\mathbf{Y}$: $\texttt{diag}(\mathbf{X}+\mathbf{Y})$, where matrix $\mathbf{X}$ and $\mathbf{Y}$ are stored in physical tensor relations R_X and R_Y. This computation can be represented by the following TRA expression, where $\texttt{isEq}(\langle \mathsf{k}_0, \mathsf{k}_1\rangle) \mapsto \mathsf{k}_0 = \mathsf{k}_1$, $\texttt{merge}(\langle \mathsf{k}_0, \mathsf{k}_1\rangle) \mapsto \langle \mathsf{k}_0\rangle$, `matAdd` is an element-wise sum of two arrays, and `diag` diagonalizes the array. Then $\text{diag}(\mathbf{X}+\mathbf{Y})$ can be written as:

$$\lambda_{(\texttt{diag})}\left(\textsc{ReKey}_{(\texttt{merge})}\left(\sigma_{(\texttt{isEq})}\left(\bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\mathsf{R}_X,\mathsf{R}_Y\right)\right)\right)\right).$$

This TRA expression can be translated to the IA expression:

$$\lambda^L_{(\texttt{idOp},\texttt{diag})}\left(\lambda^L_{(\texttt{merge},\texttt{idOp})}\left(\sigma^L_{(\texttt{isEq})}\left(\bowtie^L_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right),\mathsf{R}_Y\right)\right)\right)\right).$$

We can apply the following equivalence rules for the above IA expression:

$$\begin{aligned}
&\lambda^L_{(\texttt{idOp},\texttt{diag})}\left(\lambda^L_{(\texttt{merge},\texttt{idOp})}\left(\sigma^L_{(\texttt{isEq})}\left(\bowtie^L_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right),\mathsf{R}_Y\right)\right)\right)\right)\\
&\overset{R1\text{-}2}{\equiv}\ \lambda^L_{(\texttt{merge},\texttt{diag})}\left(\sigma^L_{(\texttt{isEq})}\left(\bowtie^L_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right),\mathsf{R}_Y\right)\right)\right)\\
&\overset{R1\text{-}6}{\equiv}\ \lambda^L_{(\texttt{merge},\texttt{diag})}\left(\bowtie^L_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\sigma^L_{(\texttt{isEq})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right)\right),\sigma^L_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right)\right)\\
&\overset{R2\text{-}2}{\equiv}\ \lambda^L_{(\texttt{merge},\texttt{diag})}\left(\bowtie^L_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{matAdd})}\left(\textsc{Bcast}\left(\sigma^L_{(\texttt{isEq})}\left(\mathsf{R}_X\right)\right),\sigma^L_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right)\right)\\
&\overset{R1\text{-}7}{\equiv}\ \bowtie^L_{(\texttt{merge}(\langle 0,1\rangle),\texttt{merge}(\langle 0,1\rangle),\texttt{matAdd}\circ\texttt{diag})}\left(\textsc{Bcast}\left(\sigma^L_{(\texttt{isEq})}\left(\mathsf{R}_X\right)\right),\sigma^L_{(\texttt{isEq})}\left(\mathsf{R}_Y\right)\right).
\end{aligned}$$

The transformation will significantly reduce both the communication overhead and the computation load: by applying R1-6, the `isEq` functions will be pushed down, this transformation not only reduces the input tuple pairs for the join to execute the `matAdd` function but also the enables reduction of communication overhead where the the filter operation is commuted with the broadcast operation by R2-2; lastly, R1-7 leverages the property that kernel functions `diag` and `matAdd` are distributive, as a result, addition will only be applied for the diagonal elements for the paired blocks after kernel function composition.

5.2.2 Domain-Specific Equivalence Rules. Such rules encode specific knowledge from parallel and distributed computing algorithms. Adding such rules to a system allows IA to have at its disposal common implementation strategies, that it can choose from in a cost-based manner.

We do not attempt to produce an exhaustive list of such rules, but rather we consider in detailing one example: distributed matrix multiplication over tensor relations R_X and R_Y:

$$\Sigma_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathsf{R}_X,\mathsf{R}_Y\right)\right).$$

For physical tensor relations R_X and R_Y, using the rules of Table 1, this would be compiled into:

$$\Sigma^L_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\textsc{Bcast}\left(\mathsf{R}_X\right),\mathsf{R}_Y\right)\right)\right).$$

This is a simple, broadcast-based matrix multiply. Applying simple equivalence rules brings us to cross product-based matrix multiplication, which partitions R_X on columns, and R_Y on rows. The IA program is:

$$\Sigma^L_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\textsc{Shuf}_{(\langle 0,2\rangle)}\left(\bowtie^L_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\textsc{Shuf}_{(\langle 1\rangle)}\left(\mathsf{R}_X\right),\textsc{Shuf}_{(\langle 0\rangle)}\left(\mathsf{R}_Y\right)\right)\right)\right).$$

However, more complicated schemes are possible, which are expressible in IA, but not derivable using the simple equivalence rules. For example, replication-based matrix multiplication can be viewed as a relational version of the 3D parallel matrix multiplication [9]. The algorithm first replicates matrix $\mathbf{X}$ and $\mathbf{Y}$'s blocks multiple times, viewing the result as a 3-D array, and shuffles them using the index of the corresponding voxel as a key; then each site joins the tuples with the same keys and performs local multiplications, aggregating to obtain the final results. If `xDups` is defined as $\textsc{Front}(\mathsf{R}_Y)[1]$ and `yDups` is $\textsc{Front}(\mathsf{R}_X)[0]$, the shuffle stage can be implemented in IA as:

[2] In Table 1, tensor relations R, R_l and R_r are stored as the corresponding physical tensor relations R, R_l and R_r; `idOp` represents an identity map for key or array; `arrayTileOp` is a kernel function splitting the array to chunks of the indicated size on the indicated dimension; `arrayConcatOp` reverses this. `keyTileOp` is similar to `arrayTileOp`, but operates on keys; $\langle \texttt{keyDim}\rangle^c$ represents the complement set of $\langle \texttt{keyDim}\rangle$.

Table 1: Translation from TRA to IA.

TRA expression	Corresponding IA
$\Sigma_{(\texttt{groupByKeys, aggOp})}\ (\mathsf{R})$	$\Sigma^{L}_{(\texttt{groupByKeys, aggOp})}\ \left(\textsc{Shuf}_{(\texttt{groupByKeys})}\ (\mathsf{R})\right)$
$\bowtie_{(\texttt{joinKeysL, joinKeysR, projOp})}\ (\mathsf{R}_l, \mathsf{R}_r)$	$\bowtie^{L}_{(\texttt{joinKeysL, joinKeysR, projOp})}\ (\textsc{Bcast}\ (\mathsf{R}_l), \mathsf{R}_r)$
$\textsc{ReKey}_{(\texttt{keyFunc})}\ (\mathsf{R})$	$\lambda^{L}_{(\texttt{keyFunc,idOp})}\ (\mathsf{R})$
$\sigma_{(\texttt{boolFunc})}\ (\mathsf{R})$	$\sigma^{L}_{(\texttt{boolFunc})}\ (\mathsf{R})$
$\lambda_{(\texttt{transformFunc})}\ (\mathsf{R})$	$\lambda^{L}_{(\texttt{idOp,transformFunc})}\ (\mathsf{R})$
$\textsc{Tile}_{(\texttt{tileDim, tileSize})}\ (\mathsf{R})$	$\lambda^{L}_{(\texttt{keyTileOp(tileDim, tileSize), arrayTileOp(tileDim, tileSize)})}\ (\mathsf{R})$
$\textsc{Concat}_{(\texttt{keyDim, arrayDim})}\ (\mathsf{R})$	$\Sigma^{L}_{(\langle \texttt{keyDim} \rangle^{c}, \texttt{arrayConcatOp})}\ \left(\textsc{Shuf}_{\langle \texttt{keyDim} \rangle^{c}}\ (\mathsf{R})\right)$

$$\mathsf{R}^{*}_{X} = \textsc{Shuf}_{(\langle 0,2 \rangle)}\left(\lambda^{L}_{(\texttt{insertDim(2,xDups)},\texttt{duplicate(xDups)})}\ (\mathsf{R}_X)\right)$$
$$\mathsf{R}^{*}_{Y} = \textsc{Shuf}_{(\langle 0,2 \rangle)}\left(\lambda^{L}_{(\texttt{insertDim(0,yDups)},\texttt{duplicate(yDups)})}\ (\mathsf{R}_Y)\right)$$

where kernel functions `insertDim` and `duplicate` add a new dimension, and duplicate each existing array the specified number of times. For example, applying $\lambda^{L}_{(\texttt{insertDim(2,xDups)},\texttt{duplicate(xDups)})}$ to the tensor relation

$$\left\{ \left(\langle 0,0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right) \right\}$$

will produce:

$$\left\{ \left(\langle 0,0,0 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,0,1 \rangle, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\right), \left(\langle 0,1,0 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 0,1,1 \rangle, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right), \left(\langle 1,0,0 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,0,1 \rangle, \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}\right), \left(\langle 1,1,0 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right), \left(\langle 1,1,1 \rangle, \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right) \right\}$$

Next we execute:

$$\Sigma^{L}_{(\langle 0,2 \rangle, \texttt{matAdd})}\left(\bowtie^{L}_{(\langle 0,1,2 \rangle, \langle 0,1,2 \rangle, \texttt{matMul})}\ (\mathsf{R}^{*}_{X}, \mathsf{R}^{*}_{Y})\right).$$

The equivalence of these three implementations is an example of a set of domain-specific equivalence rules.

5.3 Cost Model

One of the beneficial aspects of the TRA is that cost-based optimization is much easier than that for classical relational algebra: If the uniqueness and continuity constraints hold, tuple counts need not be estimated and can be computed exactly.

In the simple cost model presented here, we use the number of floating point values that must be transferred between sites as the cost metric. There are two reasons for this decision.

Fist, the number of floating point operations in distributed ML computation is fixed. For example, all of the classical distributed matrix multiply algorithms—2.5D [40], SUMMA [42], etc., have the same floating point cost. While this is not a hard and fast rule—it is possible to push the filtering of tuples in a tensor relation past the application of a kernel function, which would change the number of floating-point operations—in many applications, network transfer is the dominant cost, and is a reasonable cost metric.

Second, skew, which could slow down a computation with low network cost, is generally not an issue in a TRA computation, unlike in classical relational database system. The TRA continuity and uniqueness constraints imply that joins and aggregations cannot encounter skew. Consider a join. For two tuples t_1 and t_2 in tensor relation R, when that relation is joined with another relation S, the number of tuples from S that join with t_1 and t_2 must be the same. This, along with the fact that the TRA requires all arrays in a tensor relation to be of the same type, implies that skew will be very rare. In a TRA implementation, the only source of delay where the entire computation is blocked on a machine is likely to be a machine that is, simply stated, slower to perform computations than the others in the cluster. Such slowness may be due to hardware heterogeneity—a challenging issue for future work—or for unpredictable reasons, such as other workloads running on the machine, which are eventualities that cannot be planned for and must be handled by the runtime.

To compute the network transfer cost for a plan in IA, we need to be able to compute the frontier of each physical relation R: $\mathbf{f} = \textsc{Front}(\mathsf{R})$. The reason is that, assuming that uniqueness and continuity constraints hold, we can compute the number of floating point numbers in R using $\mathbf{f}$. If R is of type $\mathcal{R}^{(k,r,\mathbf{b},s)}$, and $\mathbf{f} = \textsc{Front}(\mathsf{R})$, then the number of tuples in the corresponding tensor relation is $n = \prod_i \mathbf{f}_i$, and the number of floating point numbers in the tensor relation is $n \times \prod_i \mathbf{b}_i$.

Once the frontier is known, it is used to compute the transfer cost for each Bcast and Shuf operation. The cost to broadcast a tensor relation of type $\mathcal{R}^{(k,r,\mathbf{b},s)}$ and having f floating point numbers, is simply $f \times s$. The cost to shuffle a tensor relation of f floating point numbers is simply f.

Thus, the task of costing a physical TRA plan reduces to computing the type of each intermediate relation, as well as its frontier.

Computing the type is relatively easy: we work up from the leaves to the end result(s) of a physical plan, using the type signature of each of the physical operations (Section 4) to infer the type of each intermediate physical relation.

Computing the frontier in this way, working up from leaves to outputs, is also possible, but it requires a bit more thought. We now consider how the frontier of an output is computed for each of the various operations in the physical algebra:

(1) $\bowtie^{L}_{(\texttt{joinKeysL, joinKeysR, projOp})}\ (\mathsf{R}_l, \mathsf{R}_r)$. For local join, assuming that R_l and R_r have an appropriate partitioning to sites, let $\mathbf{f}_l$ and $\mathbf{f}_r$ be the left and right input frontiers

Table 2: Simple equivalence rules for kernel function composition and re-partition enumeration.

Kernel function composition based rules:
R1-1. Filter operations can be merged. For a physical relation R: $\sigma^L{}_{(\texttt{boolFunc1})}\left(\sigma^L{}_{(\texttt{boolFunc2})}\left(\mathrm{R}\right)\right) \equiv \sigma^L{}_{(\texttt{boolFunc1}\wedge\texttt{boolFunc2})}\left(\mathrm{R}\right).$
R1-2. Map operations can be merged, if the output arity of the key and array mapping functions is one. For a physical relation R: $\lambda^L{}_{(\texttt{keyMapFunc1, arrayMapFunc1})}\left(\lambda^L{}_{(\texttt{keyMapFunc2, arrayMapFunc2})}\left(\mathrm{R}\right)\right) \equiv \lambda^L{}_{(\texttt{keyMapFunc1}\circ\texttt{keyMapFunc2,arrayMapFunc1}\circ\texttt{arrayMapFunc2})}\left(\mathrm{R}\right).$
R1-3. Map and filter are commutative if keyMapFunc is an identify function (idOp). For a physical relation $\mathrm{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, if $\forall\ \texttt{key} \in (\mathbb{Z}^*)^k$, keyMapFunc(key) = key: $\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\sigma^L{}_{(\texttt{boolFunc})}\left(\mathrm{R}\right)\right) \equiv \sigma^L{}_{(\texttt{boolFunc})}\left(\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\mathrm{R}\right)\right).$
R1-4. The arrayMapFunc in map can be composed with local aggregation if keyMapFunc is an identify function (idOp): For a physical relation $\mathrm{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, if $\forall\ \texttt{key} \in (\mathbb{Z}^*)^k$, keyMapFunc(key) = key: $\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\mathrm{R}\right)\right) \equiv \Sigma^L{}_{(\texttt{groupByKeys, arrayMapFunc}\circ\texttt{aggOp})}\left(\mathrm{R}\right)$ And if the kernel function arrayMapFunc and aggOp is distributive; that is, $\forall\ \texttt{array}_1, \texttt{array}_2 \in T^{(r,\mathbf{b})}$, $\texttt{arrayMapFunc}\left(\texttt{aggOp}\left(\texttt{array}_1, \texttt{array}_2\right)\right) = \texttt{aggOp}\left(\texttt{arrayMapFunc}\left(\texttt{array}_1\right), \texttt{arrayMapFunc}\left(\texttt{array}_2\right)\right)$: $\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\mathrm{R}\right)\right) \equiv \Sigma^L{}_{(\texttt{groupByKeys, aggOp}\circ\texttt{arrayMapFunc})}\left(\mathrm{R}\right)$
R1-5. The boolFunc in filter can be composed with local aggregation if the kernel function only depends on groupByKeys. For a physical relation $\mathrm{R} \in \mathcal{R}^{(k,r,\mathbf{b},s)}$, $\forall\ \texttt{key}_1, \texttt{key}_2 \in (\mathbb{Z}^*)^k$, if $\texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(\texttt{key}_1\right)\right) = \texttt{boolFunc}\left(\Pi_{\texttt{groupByKeys}}\left(\texttt{key}_2\right)\right) \Rightarrow \texttt{boolFunc}\left(\texttt{key}_1\right) = \texttt{boolFunc}\left(\texttt{key}_2\right)$: $\sigma^L{}_{(\texttt{boolFunc})}\left(\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\mathrm{R}\right)\right) \equiv \Sigma^L{}_{(\texttt{boolFunc(groupByKeys), aggOp})}\left(\mathrm{R}\right)$
R1-6. The kernel function in local filter can be pushed down with local join if the boolFunc only checks the joined keys . For physical relations R_l and R_r: $\sigma^L{}_{(\texttt{boolFunc})}\left(\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathrm{R}_l, \mathrm{R}_r\right)\right) \equiv \bowtie^L{}_{(\texttt{joinKeysL, joinKeys, projOp})}\left(\sigma^L{}_{(\texttt{boolFunc})}\left(\mathrm{R}_l\right), \sigma^L{}_{(\texttt{boolFunc})}\left(\mathrm{R}_r\right)\right).$
R1-7. The kernel function in local map can be composed with local join, if the output arity of the key and array mapping functions is one. For physical relations R_l and R_r: $\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathrm{R}_l, \mathrm{R}_r\right)\right) \equiv \bowtie^L{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), arrayMapFunc}\circ\texttt{projOp})}\left(\mathrm{R}_l, \mathrm{R}_r\right).$ And if the kernel function arrayMapFunc and projOp is distributive, that is, $\forall\ \texttt{array}_1, \texttt{array}_2 \in T^{(r,\mathbf{b})}$, $\texttt{arrayMapFunc} : \left(\texttt{projOp}\left(\texttt{array}_1, \texttt{array}_2\right)\right) = \texttt{projOp}\left(\texttt{arrayMapFunc}\left(\texttt{array}_1\right), \texttt{arrayMapFunc}\left(\texttt{array}_2\right)\right)$: $\lambda^L{}_{(\texttt{keyMapFunc, arrayMapFunc})}\left(\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathrm{R}_l, \mathrm{R}_r\right)\right) \equiv \bowtie^L{}_{(\texttt{keyMapFunc(joinKeysL), keyMapFunc(joinKeysR), projOp}\circ\texttt{arrayMapFunc})}\left(\mathrm{R}_l, \mathrm{R}_r\right).$
Re-partition based rules:
R2-1. Only the final broadcast/shuffle in a sequence of broadcast/shuffle operations is needed. For a physical relation R: $\textsc{Bcast}\left(\textsc{Bcast}\left(\ldots\textsc{Bcast}\left(\mathrm{R}\right)\right)\right) \equiv \textsc{Bcast}\left(\mathrm{R}\right)$ $\textsc{Shuf}_{(\texttt{partDims}_n)}\left(\ldots\textsc{Shuf}_{(\texttt{partDims}_2)}\left(\textsc{Shuf}_{(\texttt{partDims}_1)}\left(\mathrm{R}\right)\right)\right) \equiv \textsc{Shuf}_{(\texttt{partDims}_n)}\left(\mathrm{R}\right).$
R2-2. The re-partition operations are commutative with the local filter operation. For a physical relation R: $\textsc{Bcast}\left(\sigma^L{}_{(\texttt{boolFunc})}\left(\mathrm{R}\right)\right) \equiv \sigma^L{}_{(\texttt{boolFunc})}\left(\textsc{Bcast}\left(\mathrm{R}\right)\right);$ $\textsc{Shuf}_{(\texttt{partDim})}\left(\sigma^L{}_{(\texttt{boolFunc})}\left(\mathrm{R}\right)\right) \equiv \sigma^L{}_{(\texttt{boolFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathrm{R}\right)\right).$
R2-3. The re-partition operations are commutative with the local map operation. For a physical relation R: $\textsc{Bcast}\left(\lambda^L{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\mathrm{R}\right)\right) \equiv \lambda^L{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\textsc{Bcast}\left(\mathrm{R}\right)\right);$ And, if keyMapFunc is the identity function: $\textsc{Shuf}_{(\texttt{partDims})}\left(\lambda^L{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\mathrm{R}\right)\right) \equiv \lambda^L{}_{(\texttt{keyMapFunc,arrayMapFunc})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathrm{R}\right)\right).$
R2-4. A shuffle can be avoided if the physical relation is already partitioned by a local aggregation's groupByKeys. For a physical relation R, if partDims $\subseteq$ groupByKeys: $\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathrm{R}\right)\right) \equiv \Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\mathrm{R}\right)$
R2-5. An aggregation can be split to two phases, if the physical relation is only partially partitioned. For a physical relation R, if groupByKeys $\subset$ partDims: $\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\mathrm{R}\right)\right) \equiv \Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\textsc{Shuf}_{(\texttt{partDims})}\left(\Sigma^L{}_{(\texttt{groupByKeys, aggOp})}\left(\mathrm{R}\right)\right)\right).$
R2-6. A Join $\bowtie$ defined by the TRA can be implemented in the following equivalent ways. For physical relations R_l and R_r: $\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Bcast}\left(\mathrm{R}_l\right), \mathrm{R}_r\right) \equiv \bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\mathrm{R}_l, \textsc{Bcast}\left(\mathrm{R}_r\right)\right)$ $\equiv \bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathrm{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathrm{R}_r\right)\right).$
R2-7. The local join can be pushed through shuffle. For physical relations $\mathrm{R}_l \in \mathcal{R}^{(k_l,r_l,\mathbf{b}_l,s)}$ and $\mathrm{R}_r \in \mathcal{R}^{(k_r,r_r,\mathbf{b}_r,s)}$, if partDims $\subseteq$ joinKeysL: $\textsc{Shuf}_{(\texttt{partDims})}\left(\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathrm{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathrm{R}_r\right)\right)\right) \equiv$ $\bowtie^L{}_{(\texttt{joinKeysL, joinKeysR, projOp})}\left(\textsc{Shuf}_{(\texttt{joinKeysL})}\left(\mathrm{R}_l\right), \textsc{Shuf}_{(\texttt{joinKeysR})}\left(\mathrm{R}_r\right)\right)$

of dimensionality k_l and k_r, respectively. Then the output frontier $\mathbf{f}$ is computed as follows. For $k < k_l$ and k not in `joinKeysL`, $\mathbf{f}[k] = \mathbf{f}_l[k]$, as the frontier value for that dimension is inherited from the left. For $k < k_l$ and where $k = \texttt{joinKeysL}[i]$, $\mathbf{f}[k] = \min(\mathbf{f}_l[k], \mathbf{f}_r[i])$, as the frontier value for that dimension results from the join of the two relations. And finally, for all other k, $\mathbf{f}[k]$ is inherited from the corresponding dimension in the right frontier.

(2) $\Sigma^L_{(\texttt{groupByKeys, aggOp})}(\mathsf{R})$. For local aggregation, assuming an appropriate partitioning, let $\mathbf{f}_i$ denote the input frontier, and let n be the number of key dimensions in `groupByKeys`. In this case, for $k \leq n$, $\mathbf{f}[k] = \mathbf{f}_l[\texttt{groupByKeys}[k]]$.

(3) $\sigma^L_{(\texttt{boolFunc})}(\mathsf{R})$. This performs a filter in the physical tensor relation R. For an n-dimensional input frontier $\mathbf{f}_i$, for any $k \leq n$, by definition:

$$\mathbf{f}[k] = 1 + \max\{\mathbf{k}[k] \text{ s. t. } \mathbf{k} < \mathbf{f}_i \text{ and } \texttt{boolFunc}(\mathbf{k}) = \text{ true}\}.$$

That is, the k-th dimension in the frontier is inherited from the largest key value in that dimension accepted by `boolFunc`. In many cases, especially if `boolFunc` consists of simple arithmetic expressions any comparisons, symbolic methods can be used to compute this. But in practice, it may simply be easier to use a brute-force approach, where each key value is fed into `boolFunc` to compute the required maximum. Since the size of a tensor relation is typically small—tens of thousands of tuples would be very large—this is a very practical approach.

(4) $\lambda^L_{(\texttt{keyMapFunc,arrayMapFunc})}(\mathsf{R})$. Similarly, for an n-dimensional input frontier $\mathbf{f}_i$, for any $k \leq n$, by definition:

$$\mathbf{f}[k] = 1 + \max\{\texttt{keyMapFunc}(\mathbf{k})[k] \text{ s. t. } \mathbf{k} < \mathbf{f}_i\}.$$

Again, a brute force-approach is appropriate for computing the frontier in this case.

6 EVALUATION

The goal of our paper is to design a computational abstraction that could be exported by the back-end of a machine learning system. To be specific, we have detailed that: (1) such an abstraction should be expressive enough to express a variety of computations; (2) the computations expressed using the abstraction should be competitive with hand-coded or special-purpose solutions; and (3) the abstraction should be amenable to automatic optimization.

To determine whether the TRA meets these goals, we have implemented a simple Python back-end that exports the TRA/IA interface. Across three different large-scale ML computations, we experimentally evaluate this TRA-based back-end. To see whether a TRA-based back-end can provide suitable performance, we compare with a number of other options, including hand-coded MPI solutions, high-performance libraries such as ScaLAPACK, distributed data science tools such as Dask, and ML systems such as TensorFlow and PyTorch. To see whether it is amenable to automatic optimization, for each ML computation, we apply a series of transformations to obtain multiple implementations in the IA, and evaluate whether the cost model is able to predict which is preferable.

Benchmark Tasks. (i) distributed matrix multiplication, (ii) distributed nearest neighbor search in a Riemannian metric space, and (iii) distributed stochastic gradient decent (SGD) in a two-layer, feed-forward neural network (FFNN).

TRA Implementation. We implement an execution engine for the IA in Python. While it may seem surprising that Python is appropriate for implementing a relational engine, for even very large ML problems, the number of tuples in a TRA computation is small; most data are stored in the large arrays. Our Python execution engine makes heavy use of PyTorch to handle those arrays. PyTorch is used to actually execute the compute kernels on the various sites in a compute cluster, and our IA implementation uses PyTorch's optimized communication library to move the arrays stored in tensor relations between machines.

6.1 Matrix Multiplication

Multiplication of $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times J}$ can be formalized:

$$\Sigma_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathsf{R}_{\mathbf{A}}, \mathsf{R}_{\mathbf{B}}\right)\right)$$

where matrix $\mathbf{A}$ and $\mathbf{B}$ are stored in tensor relations $\mathsf{R}_{\mathbf{A}}$ and $\mathsf{R}_{\mathbf{B}}$.

To test the effectiveness of IA optimization, as others have done [22, 23], we consider three different multiplications: (i) general matrices ($I = K = J = 4 \times 10^4$), (ii) matrices with a common large dimension ($K = 6.4 \times 10^5$, $I = J = 10^4$), and (iii) matrices with two large dimensions ($I = J = 8 \times 10^4$, $K = 10^4$). Matrices are filled with random data following uniform distribution $\mathcal{U}(-1, 1)$.

As discussed, the above TRA as three equivalent IA plans: broadcast based matrix multiplication (BMM), cross-product based matrix multiplication (CMM), and replication-based matrix multiplication (RMM). We compare these three IA implementations with Intel's version of ScaLAPACK [17] which realizes the classic SUMMA [42]. We also compare with our own, hand-coded version of the classical 2.5D matrix multiply algorithm [40], implemented on top of MPI [12]; with Dask [1], a popular distributed analytic tool with a Python interface [39]; and with PETSc [2], a popular high-performance distributed computing library [10]. All methods are benchmarked over Amazon EC2 clusters with 5, 10 or 15 `r5d.2xlarge` instances (each with 8 vCPU, 64 GB RAM, and connected by up to 10 Gb/s interconnect). Note that we have made reasonable amount of effort to tune the hyper-parameters in the alternative solutions (e.g., grid size, thread number, initial layout, etc.) and report the best results. Results are in Table 3. In Table 4, we report the IA cost (as computed in Section 5.3) predicted for a 10-node cluster.

6.2 Nearest Neighbor Search

We use TRA to implement a nearest neighbor search problem in a Riemannian metric space encoded by matrix $\mathbf{A} \in \mathbb{R}^{D \times D}$, where given a query vector $\mathbf{x}_q \in \mathbb{R}^{1 \times D}$ and a candidate set $\mathbf{X} \in \mathbb{R}^{N \times D}$, the goal is to find the i-th row in the matrix that minimizes: $d_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}_q) = (\mathbf{x}_i - \mathbf{x}_q)\,\mathbf{A}\,(\mathbf{x}_i - \mathbf{x}_q)^T$. Suppose $\mathbf{x}_q$, $\mathbf{X}$, $\mathbf{A}$ are stored in tensor relation $\mathsf{R}_{\mathbf{x}_q}$, $\mathsf{R}_{\mathbf{X}}$ and $\mathsf{R}_{\mathbf{A}}$, the corresponding TRA program can be encoded as:

$$\begin{aligned}
\mathsf{R}_{\mathbf{diff}} &= \bowtie_{(\langle 1\rangle,\langle 1\rangle,\texttt{matVecSub})}\left(\mathsf{R}_{\mathbf{x}_q}, \mathsf{R}_{\mathbf{X}}\right)\\
\mathsf{R}_{\mathbf{proj}} &= \Sigma_{(\langle 0,2\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 1\rangle,\langle 0\rangle,\texttt{matMul})}\left(\mathsf{R}_{\mathbf{diff}}, \mathsf{R}_{\mathbf{A}}\right)\right)\\
\mathsf{R}_{\mathbf{dist}} &= \lambda_{(\texttt{rowSum})}\left(\Sigma_{(\langle 0\rangle,\texttt{matAdd})}\left(\bowtie_{(\langle 0,1\rangle,\langle 0,1\rangle,\texttt{elemMul})}\left(\mathsf{R}_{\mathbf{proj}}, \mathsf{R}_{\mathbf{diff}}\right)\right)\right)\\
\mathsf{R}_{\mathbf{min}} &= \Sigma_{(\langle\rangle,\texttt{minIndex})}\left(\mathsf{R}_{\mathbf{dist}}\right).
\end{aligned}$$

Table 3: Distributed matrix multiply runtimes.

	Two General Matrices			A Common Large Dim			Two Large Dims		
Cluster Size	5	10	15	5	10	15	5	10	15
BMM	61.18s	46.48s	38.54s	106.24s	104.67s	101.63s	57.23s	37.60s	31.64s
CMM	63.14s	40.08s	29.38s	51.52s	30.58s	23.09s	106.82s	82.72s	75.63s
RMM	60.71s	43.56s	44.55s	91.19s	74.40s	68.43s	59.91s	41.12s	33.26s
ScaLAPCK	66.11s	37.05s	28.30s	83.96s	58.17s	35.45s	53.06s	28.13s	22.34s
2.5D	62.93s	29.60s	23.11s	83.59s	46.43s	34.82s	61.13s	28.36s	21.21s
Dask	200.64s	161.23s	104.12s	Fail	Fail	Fail	Fail	Fail	Fail
PETSc	1034.40s	535.85s	430.80s	1071.62s	801.26s	550.74s	1051.71s	810.61s	598.24s

Table 4: Predicted costs for a 10-node cluster.

	BMM	CMM	RMM
Two General	1.6×10^{10}	1.6×10^{10}	1.6×10^{10}
A Common Large Dim	6.4×10^{10}	1.0×10^{9}	6.4×10^{10}
Two Large Dims	8.0×10^{9}	6.4×10^{10}	8.0×10^{9}

where matVecSub is matrix-vector subtraction, elemMul is element-wise matrix multiplication (Hadamard product), minIndex finds the minimal element's index. We hand-compile this into an expression in the IA, and then use the various equivalence rules to produce two different implementations: Opt4Horizontal and Opt4Vertical. Opt4Horizontal will broadcast $\mathsf{R}_{\mathbf{x}_q}$ and $\mathsf{R}_{\mathbf{A}}$ to each compute site and partition $\mathsf{R}_{\mathbf{X}}$ by dimension 0; then the computation of $\mathsf{R}_{\mathbf{diff}}$, $\mathsf{R}_{\mathbf{proj}}$, and $\mathsf{R}_{\mathbf{dist}}$ will be conducted by local operations. Opt4Vertical will first broadcast $\mathsf{R}_{\mathbf{x}_q}$ to each site and compute $\mathsf{R}_{\mathbf{diff}}$, then partition $\mathsf{R}_{\mathbf{diff}}$ by dimension 1 and partition $\mathsf{R}_{\mathbf{A}}$ by dimension 0 so that $\mathsf{R}_{\mathbf{proj}}$ is computed in a cross-product based matrix multiplication. For the the Opt4Horizontal IA implementation, $\mathsf{R}_{\mathbf{x}_q}$, $\mathsf{R}_{\mathbf{X}}$ and $\mathsf{R}_{\mathbf{A}}$ are initially partitioned by dimension 0. For the the Opt4Vertical IA implementation, $\mathsf{R}_{\mathbf{x}_q}$, and $\mathsf{R}_{\mathbf{A}}$ are initially partitioned by dimension 0, while $\mathsf{R}_{\mathbf{X}}$ is initially partitioned by 1.

We generate two data sets: (i) Large, with a large number of data points ($N = 1.5{\times}10^5, 1.5{\times}10^6$) but small feature space ($D = 6{\times}10^3$); and (ii) Wide, with a small number of data points ($N = 6{\times}10^3$), with a large feature space ($D = 3{\times}10^4, 10^5$). We execute this computation on compute clusters with 4, 8 or 12 r5d.2xlarge instances.

We also implemented the same computation using Dask[1]. And as a baseline, we compare the execution time with a PyTorch implementation that runs on a single site equipped with the same computing power as the TRA implementation: an r5d.8xlarge instance (with 32 vCPU, 256 GB RAM), an r5d.16xlarge instance (with 64 vCPU, 512 GB RAM) and an r5d.24xlarge instance (with 96 vCPU, 768 GB RAM). Since the single-site implementation has zero communication overhead, this should be something of a lower-bound on the time required to run the computation. The results and predicted costs are enumerated in Table 5 and Table 6.

6.3 Feed-Forward Neural Network

Lastly, we benchmark a training iteration of a two-layer FFNN for multiple label classification, computed over an input matrix.

Again, we compile the TRA program for FFNN learning by hand into the IA, and use the equivalence rules to produce two implementations. The first, called TRA-DP, resembles the classic data parallel implementation. The second, called TRA-MP, corresponds to an intra-operation model parallel plan.

We compare these two IA plans with the state-of-the-art data parallel implementation provided by PyTorch 1.7.1 [33] and TensorFlow 2.4.1 [5]. We also compare with the same computation written on top of Dask [1], and hand-coded using ScaLAPACK [17]. Note that these two options do not fully support GPU.

Two data sets are considered. First, the data from the Google speech recognition task [44], where a 1600 feature vector is extracted from audio wave-forms; the goal is to identify 10 keywords ($D = 1600$ and $L = 10$); for this task, we train a very wide hidden layer with large number of neurons where $H = 1 \times 10^5$, 1.5×10^5, or 2×10^5; a batch size of 10^4 ($N = 10^4$) are used for min-batch SGD. Second, we consider the AmazonCat-14K [36, 37] benchmark, which is an extreme multi-label (XML) classification dataset including a large number of features ($D = 597540$) and labels ($L = 14588$); we train a relatively narrow network with $H = 0.5 \times 10^3$, 1×10^3, 3×10^3, 5×10^3, or 7×10^3; a batch size of 10^3 ($N = 10^3$) are used for mini-batch SGD. Each is executed on CPU clusters with 2, 5 or 10 r5dn.2xlarge instances connected by up to 25 Gb/s interconnect) and GPU clusters with 2, 5 or 10 p3.2xlarge instances (each with a NVIDIA Tesla V100 GPU, and connected by 10 Gb/s interconnect). The results for Google speech are listed in Table 7; for Amazon-XML in Table 8. Predicted costs are given in Table 9.

6.4 Discussion

First, the experiments do seem to show that the TRA provides an abstraction upon which a variety of ML computations can be mapped. Further, the TRA seems to provide for good performance. On the matrix multiplication experiments, the best TRA-based implementations were at least competitive with ScaLAPACK as well as our hand-coded MPI-based implementation (we observed 29s for the TRA-based CMM vs. 23s for hand-coded MPI in the "two general marices" case, 31s for the TRA-based BMM vs. 21s for hand-coded MPI in the "two large dims case") even beating them both (23s for the TAR-based CMM vs. 35s for hand-coded MPI) in the "common large dim case". On the FFNN experiments, the best TRA-based implementation for each task was about 1.5× times as slow as the hand-constructed ScaLAPACK implementation for the Google data set, but considerably faster than ScaLAPACK for the more challenging Amazon data set. In general it is fair to say that the best TRA-based implementation for each task was at least competitive with the ScaLAPACK and MPI-based codes. The fact that there is not a significant performance hit moving from a special-purpose tool

Table 5: Nearest neighbor search runtimes.

	Wide ($N = 1.5 \times 10^5$)			Wide ($N = 1.5 \times 10^6$)			Large ($D = 3 \times 10^4$)			Large ($D = 10^5$)		
Cluster Size	4	8	12	4	8	12	4	8	12	4	8	12
Opt4Horizontal	5.64s	4.71s	3.36s	55.62s	39.24s	24.63s	13.26s	17.71s	28.25s	159.69s	229.40s	315.17s
Opt4Vertical	10.44s	9.26s	9.88s	120.09s	112.69s	108.59s	5.93s	4.52s	3.82s	57.81s	35.61s	26.43s
Single big machine	5.31s	4.65s	3.33s	51.54s	35.82s	23.01s	5.21s	4.41s	3.79s	48.22s	31.12s	24.88s
Dask	485.87s	289.50s	223.55s	Fail	Fail	Fail	437.31s	420.25s	381.10s	Fail	Fail	Fail

Table 6: Predicted nearest neighbor search costs, 8 machines.

	Opt4Horizontal	Opt4Vertical
Wide data set	2.9×10^8	8.0×10^{10}
Large data set	7.2×10^{10}	4.8×10^9

Table 7: SGD iteration time: FFNN for Google Speech.

Cluster	CPU			GPU		
Nodes	2	5	10	2	5	10
100k Neurons						
PyTorch-DP	11.16s	6.15s	4.75s	0.99s	1.19s	1.27s
TF-DP	11.93s	7.32s	5.51s	0.87s	1.13s	1.17s
ScaLAPCK	8.52s	4.97s	2.79s	NA	NA	NA
Dask	62.57s	56.57s	49.63s	NA	NA	NA
TRA-DP	11.62s	6.51s	5.20s	1.49s	1.59s	1.63s
TRA-MP	26.56s	28.71s	29.09s	7.01s	11.56s	Fail
150k Neurons						
PyTorch-DP	14.28s	9.46s	6.54s	1.18s	1.65s	1.78s
TF-DP	16.68s	10.69s	8.43s	1.16s	1.62s	1.75s
ScaLAPCK	13.45s	7.48s	3.87s	NA	NA	NA
Dask	96.56s	85.32s	77.07s	NA	NA	NA
TRA-DP	14.52s	9.68s	7.56s	2.15s	2.22s	2.23s
TRA-MP	33.20s	42.80s	43.10s	Fail	Fail	Fail
200k Neurons						
PyTorch-DP	17.25s	11.94s	9.30s	Fail	2.09s	2.42s
TF-DP	21.36s	13.21s	11.21s	1.52s	2.12s	2.46s
ScaLAPCK	17.18s	10.05s	5.06s	NA	NA	NA
Dask	136.66s	112.72s	104.01s	NA	NA	NA
TRA-DP	17.89s	12.51s	9.67s	2.94s	2.80s	2.85s
TRA-MP	37.82s	54.23s	59.84s	Fail	Fail	Fail

Table 8: SGD iteration time: FFNN for Amazon-XML.

Cluster	CPU			GPU		
Nodes	2	5	10	2	5	10
0.5k Neurons						
PyTorch-DP	3.58s	4.51s	6.41s	1.46s	2.11s	2.19s
TF-DP	5.94s	7.81s	8.96s	1.21s	1.85s	2.11s
ScaLAPCK	4.92s	2.91s	1.73s	NA	NA	NA
Dask	27.96s	27.01s	22.69s	NA	NA	NA
TRA-DP	4.77s	5.13s	7.84s	2.42s	2.48s	2.61s
TRA-MP	2.18s	1.41s	0.83s	0.15s	0.12s	0.09s
1k Neurons						
PyTorch-DP	9.74s	10.29s	10.34s	2.67s	3.76s	4.20s
TF-DP	Fail	Fail	Fail	Fail	Fail	Fail
ScaLAPCK	8.16s	6.65s	2.47s	NA	NA	NA
Dask	45.40s	42.15s	29.34s	NA	NA	NA
TRA-DP	12.50s	14.29s	15.68s	4.67s	4.69s	4.73s
TRA-MP	3.86s	2.79s	1.70s	0.40s	0.37s	0.35s
3k Neurons						
PyTorch-DP	25.46s	29.04s	30.51s	Fail	Fail	Fail
TF-DP	Fail	Fail	Fail	Fail	Fail	Fail
ScaLAPCK	17.56s	9.59s	7.91s	NA	NA	NA
Dask	103.83s	89.09s	81.56s	NA	NA	NA
TRA-DP	26.59s	38.15s	46.06s	Fail	12.74s	13.13s
TRA-MP	10.57s	6.36s	3.88s	Fail	0.54s	0.44s
5k Neurons						
PyTorch-DP	34.05s	46.53s	50.17s	Fail	Fail	Fail
TF-DP	Fail	Fail	Fail	Fail	Fail	Fail
ScaLAPCK	23.21s	11.65s	8.33s	NA	NA	NA
Dask	246.56s	143.86s	127.26s	NA	NA	NA
TRA-DP	44.12s	68.54s	75.15s	Fail	Fail	Fail
TRA-MP	18.59s	8.07s	5.75s	Fail	0.59s	0.48s
7k Neurons						
PyTorch-DP	Fail	Fail	Fail	Fail	Fail	Fail
TF-DP	Fail	Fail	Fail	Fail	Fail	Fail
ScaLAPCK	29.19s	14.04s	9.57s	NA	NA	NA
Dask	Fail	Fail	Fail	NA	NA	NA
TRA-DP	60.28s	89.36s	107.86s	Fail	Fail	Fail
TRA-MP	21.35s	12.12s	7.854s	Fail	Fail	0.73s

requiring significant programmer expertise to a general-purpose implementation abstraction seems to argue that in fact, a TRA-based back-end can provide state-of-the-art performance.

It is also instructive to compare our TRA-based implementations with the other, more user-friendly tools tested, which in practice would be more reasonable alternatives to an ML system with a TRA-based back-end. Dask was not competitive, and was often one or two orders of magnitude slower. On the FFNN experiments, PyTorch was generally better performing than TensorFlow in CPU clusters, while both systems perform almost identically in GPU clusters. For Google speech, the optimal partition schema is identical to data parallelism, where the best TRA-based option is able to closely match PyTorch's speed. Further, while PyTorch failed on the larger Google computations in a 2-GPU cluster, the TRA implementation was able to run to completion. On the even larger, extreme classification problem, the TRA-MP (model parallel) IA was much, much faster than PyTorch, (where TensorFlow fails in most cases since it does not allow a parameter matrix to exceed 2 GB), and much more scalable. PyTorch also cannot handle the huge matrices required to power this computation in some settings.

The final question we wanted to address was whether the TRA is amenable to automatic optimization. Note that in each case, there was one IA implementation that was suitable for the input data, and one that was not; the difference between the two was often significant. In a system based upon the TRA, it would be crucial to

Table 9: Predicted FFNN costs for a 5-node cluster.

	TRA-DP	TRA-MP
Google speech 100k	9.7×10^8	1.0×10^{10}
Google speech 150k	1.5×10^9	1.5×10^{10}
Google speech 200k	1.9×10^9	2.0×10^{10}
Amazon XML 1k	3.7×10^9	1.0×10^7
Amazon XML 3k	1.1×10^{10}	3.0×10^7
Amazon XML 5k	1.8×10^{10}	5.0×10^7
Amazon XML 7k	2.6×10^{10}	7.0×10^7

automatically choose the suitable implementation. We found that in each case, the simple cost model from Section 5.3 would have chosen the correct implementation. For example, consider Table 9. In each case, the cost metric correctly assigns the lower cost to the appropriate IA computation: TRA-DP for the smaller, Google problem, and TRA-MP for the larger, extreme classification problem. These results suggest that it should easily be possible to perform cost-based optimization over the IA.

7 RELATED WORK

Our focus has been on the proper implementation abstraction for ML systems. The TRA is "front-end agnostic." Still, there has been considerable interest in programming and compilation for such systems. FAQ, by Khamis et al. [8], considers how to compute Einstein-notation-like expressions over semi-rings. Effectively, FAQ re-writes such expressions so that they can easily be computed using the "OutsideIn" method for first determining the non-zero entries using a series of joins, followed by the computation of the values. Laue et al. [32] propose a variant on the Ricci calculus for computing tensor-based derivatives using the Einstein notation. Tensor Comprehensions are an Einstein-like programming language and associated compiler that is able to produce efficient CUDA kernels [43]; the tensor algebra compiler is a similar effort [31]. Our efforts are complementary. One could imagine, for example, using FAQ-like algorithms along with a compiler for high-performance kernels to generate expressions for a TRA-based back-end.

Classic data-flow systems have been modified to support distributed machine learning. Both Spark [45] and SystemML [21] provide native libraries for deep learning. A set of deep learning frameworks can run on top of Spark, such as TensorFrames [24], Deeplearning4j [41], SparkNet[38] and BigDL [18]. Conceptually, these deep learning frameworks are related to the TRA as they allow the distribution of ML computations. Consider TensorFrames. TensorFrames allows the items in a Spark DataFrame to be operated on by a TensorFlow computation. One could view those TensorFlow computations as being similar to the kernels applied by TRA, and the Spark operations used to manipulate the data as being similar to the the joins, aggregations, and so on offered by the TRA. The key difference is that while these systems are each significant engineering efforts aimed at marrying different technologies (TensorFlow and Spark in the case of TensorFrames), the TRA is designed as a generic back-end. In fact, a TensorFrames-like programming model could easily be mapped onto TRA, with `mapRows`, `aggregate`, etc., being mapped to the appropriate TRA operations, and the TensorFlow computations run as kernels.

Relational systems have long been proposed for ML. MLog [34] is a declarative relational system managing data movement, data persistency, and training batch generation. Similar ideas have been applied in [7] for feature extraction queries over multi-relation databases, and [28] for optimizing sparse tensor computations constructed from relational tables. Recently, relational systems have also been considered as runtime engine (instead of an efficient data loader) for distributed ML. DB4ML [27] proposes user-defined iterative transactions. Multi-dimensional-recursion has been built on top of SimSQL [15], a distributed analytic database system, that can support neural network training [26].

The idea of moving past relations onto arrays as a database data model, is long-standing (e.g., consider Baumann's work on Rasdaman [13]). SciDB [14] is a well-known system following this idea. LevelHeaded [6] uses a special key-value structure to support linear operations. MATLANG [11] introduces a language for matrix manipulation. TensorDB [29, 30] is a database system that can perform tensor manipulation. LARA[25] proposes an algebra with tuple-wise operators, attribute-wise operators, and tuple extensions, then defines linear and relational algebra operations using these primitives. RMA [20] attempts to bridge the gap between relations and matrices. While related, these systems attempt to implement tensor computations as algebraic expressions (e.g., a join followed by an aggregation) over relations of `(key, value)` pairs. This requires pushing a huge number of pairs through the system, which introduces significant overhead.

8 CONCLUSION

We have introduced the tensor relational algebra (TRA), and suggested this as the interface that could be exported by the back-end of a machine learning system. We have showed through extensive experimentation that a computation expressed in the TRA then transformed into the implementation algebra and optimized, is competitive with (and often faster than) other options, including HPC softwares such as ScaLAPACK, and ML softwares such as TensorFlow and PyTorch.

There are many avenues for future work. TRA is not meant to be a user-facing programming language. Thus, a key question is: can a language such as Tensor Comprehensions or Einstein notation be compiled into TRA? At a high level, this should not be too difficult, as these languages match indices in different tensors (which is easily implemented as a join) and then sum out dimensions (aggregation). But there are many details to consider. The TRA uses arrays or "chunks" for speed. How to automatically block or chunk a tensor computation? How to automatically generate the compute kernels? Sparsity is also an important issue. A compiler could also decide to store a sparse tensor using arrays that do not have zero dimensions, but where those arrays are stored sparsely, with a high-performance kernel generated to handle the specific sparsity pattern.

ACKNOWLEDGMENTS

This work was supported by an NIH CTSA, award no. UL1TR003167 and by the NSF under grant nos. 1918651, 1910803, 2008240 and 1842494. We also thank the anonymous reviewers for their insightful comments on earlier versions of the paper.

REFERENCES

[1] [n.d.]. Dask. https://dask.org/.
[2] [n.d.]. PETSc. https://www.mcs.anl.gov/petsc/.
[3] [n.d.]. PyTorch. https://pytorch.org/.
[4] [n.d.]. TensorFlow. https://www.tensorflow.org/.
[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[6] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 449–460.
[7] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 325–340.
[8] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 13–28.
[9] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.
[10] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. 2019. PETSc users manual. (2019).
[11] Pablo Barceló, Nelson Higuera, Jorge Pérez, and Bernardo Subercaseaux. 2019. Expressiveness of Matrix and Tensor Query Languages in terms of ML Operators. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*. ACM, 9.
[12] Brandon Barker. 2015. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, Vol. 262.
[13] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The multidimensional database system RasDaMan. In *SIGMOD Record*, Vol. 27. ACM, 575–577.
[14] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*. 963–968.
[15] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J Haas, and Christopher Jermaine. 2013. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 637–648.
[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.
[17] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 120–121.
[18] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. 2019. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*. 50–60.
[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[20] Oksana Dolmatova, Nikolaus Augsten, and Michael H Böhlen. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2573–2587.
[21] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*. 231–242.
[22] Rong Gu, Yun Tang, Chen Tian, Hucheng Zhou, Guanru Li, Xudong Zheng, and Yihua Huang. 2017. Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2539–2552.
[23] Donghyoung Han, Yoon-Min Nam, Jihye Lee, Kyongseok Park, Hyunwoo Kim, and Min-Soo Kim. 2019. DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs. In *Proceedings of the 2019 International Conference on Management of Data*. 759–774.
[24] T Hunter. 2016. Tensorframes on google's tensorflow and apache spark. *Bay Area Spark Meetup* (2016).
[25] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM, 2.
[26] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2019. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *Proceedings of the VLDB Endowment* 12, 7 (2019), 822–835.
[27] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML-An In-Memory Database Kernel with Machine Learning Support. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 159–173.
[28] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: in-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–10.
[29] Mijung Kim and K Selçuk Candan. 2014. Efficient static and dynamic in-database tensor decompositions on chunk-based array stores. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. 969–978.
[30] Mijung Kim and K Selçuk Candan. 2014. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 2039–2041.
[31] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
[32] Sören Laue, Matthias Mitterreiter, and Joachim Giesen. 2020. A simple and efficient tensor calculus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4527–4534.
[33] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. [n.d.]. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment* 13, 12 ([n. d.]).
[34] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.
[35] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2018), 1224–1238.
[36] Julian McAuley, Rahul Pandey, and Jure Leskovec. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 785–794.
[37] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 43–52.
[38] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. 2015. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051* (2015).
[39] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 126. Citeseer.
[40] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.
[41] D Team et al. 2016. Deeplearning4j: Open-source distributed deep learning for the jvm. *Apache Software Foundation License* 2 (2016), 2.
[42] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.
[43] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).
[44] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).
[45] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.