Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra

Shangyu Luo, Dimitrije Jankov, Binhang Yuan, Chris Jermaine
Rice University
Houston, TX, USA
{sl45,dj16,by8,cmj4}@rice.edu

ABSTRACT

Machine learning (ML) computations are often expressed using vectors, matrices, or higher-dimensional tensors. Such data structures can have many different implementations, especially in a distributed environment: a matrix could be stored as row or column vectors, tiles of different sizes, or relationally, as a set of (`rowIndex`, `colIndex`, `value`) triples. Many other storage formats are possible. The choice of format can have a profound impact on the performance of a ML computation. In this paper, we propose a framework for automatic optimization of the physical implementation of a complex ML or linear algebra (LA) computation in a distributed environment, develop algorithms for solving this problem, and show, through a prototype on top of a distributed relational database system, that our ideas can radically speed up common ML and LA computations.

CCS CONCEPTS

• **Information systems** → **Computing platforms**; *Database management system engines*; *Data analytics.*

KEYWORDS

Distributed systems and machine learning

ACM Reference Format:
Shangyu Luo, Dimitrije Jankov, Binhang Yuan, Chris Jermaine . 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457317

1 INTRODUCTION

As machine learning (ML) has become an increasingly important class of computation, a lot of attention has been focused on building high-performance computing systems targeted at running ML or linear algebra (LA) computations. TensorFlow [3], PyTorch [1], MXNet [11], Pandas [30], and scikit-learn [31] are just a few examples of such systems.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457317

Lack of abstraction in ML system design. High-performance ML systems can be surprisingly inflexible, as they are often built to make one particular style of ML computation run fast. These systems often perform poorly (or simply do not work at all) when running distributed computations or computations that require a lot of RAM [23]. We assert that the core problem is lack of abstraction: in popular ML systems such as TensorFlow, an operation such as a matrix multiply is not an abstract or logical operation that the system will figure out how to run efficiently; it is an actual physical operation that needs to be run somewhere, using an available computational kernel. There is little built-in support for figuring out how to (automatically) distribute the computation across multiple machines or even multiple GPUs on the same machine.

A database engine as an ML engine. This is where key concepts from database systems, particularly declarativity and data independence, can be very useful. This was one of the main motivations for the SystemML project [17], for example. A programmer specifying a matrix multiplication as part of an ML computation on a "database style" system needs not worry about how the matrices are physically represented, how they are stored, or what hardware is being used. The programmer simply asks for the multiply, and the system figures out how to run it automatically and efficiently.

For this reason, a database-like system—or even a traditional database system—can serve as an excellent platform for running ML computations. For example, one can augment a database system with a special `MATRIX` type, and declare two 2×10^4 by 2×10^4 matrices, stored tiled (or chunked) into relations having 400 tuples:

```
myMatrix (tileRow INTEGER, tileCol INTEGER,
                mat MATRIX[1000][1000])
anotherMat (tileRow INTEGER, tileCol INTEGER,
                mat MATRIX[1000][1000])
```

Then simple SQL code runs an efficient, distributed matrix multiply:

```
SELECT lhs.tileRow, rhs.tileCol,
       SUM (matrix_multiply (lhs.mat, rhs.mat))
FROM myMatrix AS lhs, anotherMat AS rhs
WHERE lhs.tileCol = rhs.tileRow
GROUP BY lhs.tileRow, rhs.tileCol
```

One of the key advantages of using a database-style engine in this way is automatic optimization. For two large matrices such as `myMatrix` and `anotherMat`, the database will choose to co-partition the matrices before joining. But if one of the matrices (for example, `myMatrix`) is relatively small, a high-quality database engine will decide to broadcast it to each site, while partitioning the other matrix (`anotherMat`) across sites, and then at each site joining the full `myMatrix` with a portion of `anotherMat`. In this

way, a database engine has important advantages over a special-built ML engine such as TensorFlow, because a matrix multiplication is a logical construct that the system "figures out" how to run.

While there are considerable advantages to using a database engine as a distributed ML or LA engine, there is still a gap between the simple code that an ML programmer would ideally expect to write, and the required SQL code given above. The SQL programmers have to implicitly make several difficult decisions:

- They decided to decompose (tile) each input matrix.
- They decided that there should be 400 tiles in each matrix, where each tile is sized 1000 by 1000.
- They decided on a particular SQL code to logically specify how to run the multiplication over the tiles.

The problem is that each of these decisions can have a significant impact on the performance of the multiplication, and the implications can be difficult or impossible to reason through. One could, for example, have chosen to decompose the matrices into vertical and horizontal strips, respectively:

```
myMatrix (tileRow INTEGER, mat MATRIX[50][20000])
anotherMat (tileCol INTEGER, mat MATRIX[20000][50])
```

Then, the matrix multiplication would not require aggregation:

```
SELECT lhs.tileRow, rhs.tileCol,
       matrix_multiply (lhs.mat, rhs.mat)
FROM myMatrix AS lhs, anotherMat AS rhs
```

The SQL code then results in an output matrix that consists of 160,000, 50 by 50 tiles. Would this have been a preferred implementation? It is likely impossible for a programmer to know.

The problem of choosing a physical representation for the input matrices is even more difficult in the case of a complicated ML computation, which may require hundreds of individual operations (matrix multiplications, Hadamard products, function applications, etc.) to perform the full computation. The problem is harder because the different operations *interact*: a particular implementation of a matrix multiplication may leave the matrix in a physical organization that is inappropriate for the next operation (or next *operations* if the result of an operation has multiple consumers).

Our contributions. We consider the problem of automatic, physical "database" design for ML and LA computations, such as the matrix multiply above, that are to be run on a database-style engine. The goal is to automatically choose an optimal storage for each input and intermediate matrix/tensor, as well as an appropriate implementation for each operation, to minimize the overall running time. Some specific contributions are:

(1) We propose a framework to automatically explore the physical design space for the vectors/matrices that are used to power complex ML/LA computation.
(2) We define a novel optimization problem where the goal is to compute the set of data formats and associated logical ML operator implementations so as to minimize the running time of the input ML computation. To achieve this goal, we also propose a cost model to compute the cost of a ML/LA operation in a specific format.
(3) We adapt and extend a classical dynamic programming algorithm from computational genetics (called Felsenstein's algorithm [15]) to solve this optimization problem.
(4) We implement our ideas on top of SimSQL [9], a parallel relational database system, and PlinyCompute [41], a high-performance relational algebra system. Our experiments show that the physical design selected by our framework tends to have a better performance than the formats manually picked up by an expert user.

2 PRELIMINARIES

2.1 Motivating Example

We begin with a simple example that illustrates some of the trade-offs that must be managed when optimizing the physical design of a distributed LA or ML computation. Suppose we have three matrices `matA`, `matB` and `matC`, sized 100×10^4, $10^4 \times 100$ and 100×10^6, respectively. The matrix `matA` is stored as ten "row-strips" (that is, sub-matrices where the number of columns is equal to the number of columns in the original matrix) and the matrix `matB` is stored as ten "column-strips". The bigger matrix `matC` is stored as one hundred column-strips. Relational schemas are:

```
matA (tileRow INTEGER, mat MATRIX[10][10000])
matB (tileCol INTEGER, mat MATRIX[10000][10])
matC (tileCol INTEGER, mat MATRIX[100][10000])
```

Now, a programmer wants to compute $\texttt{matA} \times \texttt{matB} \times \texttt{matC}$. At a first glance, the multiply between `matA` and `matB` is straightforward. Assuming data are stored in a relational database system, as `matA` is stored in row-strips, and `matB` is in column-strips, the multiplication between them can be written in one SQL query:

```
CREATE VIEW matAB (tileRow, tileCol, mat) AS
    SELECT x.tileRow, m.tileCol,
           matrix_multiply(x.mat, m.mat)
    FROM matA AS x, matB AS m;
```

One benefit of this implementation is that no aggregation is needed. The resulting matrix, `matAB`, will then have schema:

```
matAB (tileRow INTEGER, tileCol INTEGER,
   mat MATRIX[10][10])
```

A natural choice for the multiplication between `matAB` and `matC` is a tile-based matrix multiply, where the `matC` is chunked into 10×10 tiles before the multiply. SQL code for performing the chunking and the subsequent multiplication are:

```
CREATE VIEW matCTile(tileRow, tileCol, mat) AS
    SELECT bi.rowID, bi.colID,
           get_tile(C.mat, bi.rowID, bi.colID -
                           C.tileCol * 1000, 10, 10)
    FROM matC AS C, tileIndex AS bi
    WHERE bi.colID / 1000 = C.tileCol;

CREATE VIEW matABC(tileRow, tileCol, mat) AS
    SELECT x.tileRow, m.tileCol,
           sum(matrix_multiply(x.mat, m.mat))
    FROM matAB AS x, matCTile AS m
    WHERE x.tileCol = m.tileRow
    GROUP BY x.tileRow, m.tileCol;
```

	Implementation 1	Implementation 2
matA × matB		
Multiply	row-strip × col-strip	row-strip × col-strip
Join Type	Shuffle Join	Pipelined
Time	15 sec	16 sec
matAB × matC		
Transform	col-strip ⇒ tile	tile ⇒ single
Trans. Time	2 min 7 sec	8 sec
Multiply	tile × tile	single × col-strip
Join Type	Shuffle Join	Broadcast Join
Mult. Time	16 min 27 sec	14 sec
Total		
Total Time	19min11sec	56sec

Figure 1: Comparison of matmul implementations.

Alternative implementation. There is, however, an alternative implementation that may have some runtime advantages.

Specifically, the matrix matAB has a total size of 100 × 100. Since it is relatively small, we may store it as a single tuple (that is, without chunking); in this case, the multiplication between matAB and matC can be run as an inexpensive broadcast join.

The difficulty is that the matrix matAB is stored as a set of 100, 10 × 10 chunks after the first multiply. If a database is augmented with simple vector and matrix operations [27], transforming matAB so that it can be stored in a single attribute can be performed by executing two aggregate functions, ROWMATRIX and COLMATRIX, which aggregate the tiles along rows and columns, respectively. The SQL code for this transformation is:

```
CREATE VIEW matABStrip(tileRow, mat) AS
    SELECT x.tileRow,
           ROWMATRIX(label_matrix(x.mat, x.tileCol))
    FROM matAB AS x
    GROUP BY x.tileRow;

CREATE VIEW matABSingle(mat) AS
    SELECT COLMATRIX(label_matrix(x.mat, x.tileRow))
    FROM matABStrip AS x;
```

After we obtain the single-tuple matrix matABSingle, the last multiply can be implemented as:

```
CREATE VIEW matABC(tileCol, mat) AS
    SELECT m.tileCol, matrix_multiply(x.mat, m.mat)
    FROM matABSingle AS x, matC AS m;
```

Performance. We run these two implementations on a five-node Amazon compute cluster (see the Experimental section of the paper for details of the cluster setup), and give their runtime performance in Figure 1. For the multiplication between matAB and matC, implementation 1 is much slower than implementation 2 as the latter uses a broadcast join for the multiply, and produces fewer tiles and intermediate data. The difference in total running time is significant.

2.2 Problem Statement

As shown, different physical data design choices for the same LA/ML computation can lead to very different runtime behaviors, and choosing a performant implementation for a LA computation is not an easy task. First, it is difficult for a programmer to anticipate

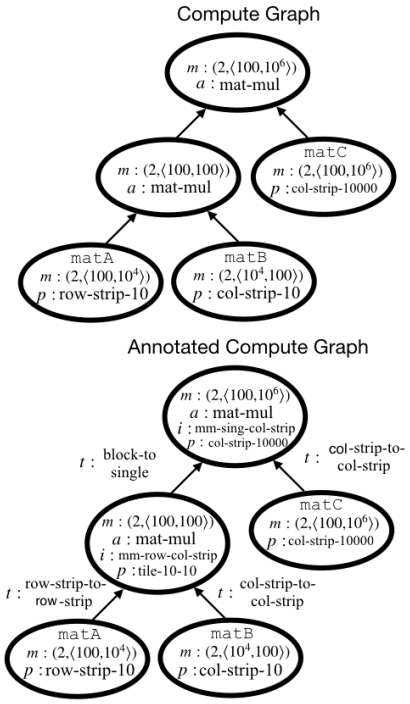

Figure 2: A compute graph and an annotated compute graph for the example of Section 2.

the effect of choosing one physical implementation for a matrix versus another. And second, even if a programmer is able to choose the best physical design for one particular operation, the design space increases in size exponentially with the number of physical designs and the complexity of the input computation.

Our goal is to allow a programmer to specify a distributed ML/LA computation at a very high level, without committing to a particular physical design. For example, in a database system, the programmer might simply give the logical specification:

```
CREATE TABLE matA (
    mat MATRIX[100][10000])
```

and then load the table with data in whatever physical format is desired. All computations are specified without reference to a physical data format. For example:

```
CREATE VIEW matAB(mat) AS
    SELECT matrix_multiply(x.mat, m.mat)
    FROM matA AS x, matB AS m;

CREATE VIEW matABC(mat) AS
    SELECT matrix_multiply(x.mat, m.mat)
    FROM matAB AS x, matC AS m;
```

The system would accept this high-level specification and explore the various physical implementations available for all of the matrices and vectors in the computation—as well as the implementations of the operations over them—with the goal of choosing the set of physical implementations that minimize the running time.

We now formalize the problem of choosing the optimal physical implementation for a distributed LA/ML computation.

3 MATRIX TYPES AND OPERATIONS

To formalize our problem definition, it is necessary to define a few key concepts: matrix types and physical matrix implementations, as well as the atomic computations and atomic computation implementations and transformations that operate over them.

Let $\mathcal{M}$ be the set of *matrix types*. A matrix type is a pair of the form $(d, \mathbf{b})$ where d is the dimensionality of the matrix. $d = 1$ for a vector, $d = 2$ for a classical matrix, and $d \geq 3$ for a higher dimensional tensor. $\mathbf{b} \in \mathbb{N}^d$ is a vector giving the number of entries of the matrix along each dimension. So, for example, the type corresponding to all two by two matrices $m_{2\times2}$ is $(2, \langle 2, 2\rangle)$.

Let $\mathcal{P}$ be the set of *physical matrix implementations*. Intuitively, a physical matrix implementation is a storage specification such as "single tuple" or "tile-based with 500 by 500 tiles" or "row strips with rows of height 50." Each physical matrix implementation has a *matrix type specification function* $f : \mathcal{M} \rightarrow \{\text{true}, \text{false}\}$. That is, given a matrix type $m \in \mathcal{M}$ and physical matrix implementation $p \in \mathcal{P}$, $p.f(m)$ evaluates to true if p can be used by the system to implement m, and false otherwise. For example, imagine that $m = (2, \langle 10^5, 10^5\rangle)$, and p corresponds to a "single tuple" implementation. We would expect that $p.f(m)$ would evaluate to false, as one could not typically store a 40GB matrix in a single tuple.

Let $\mathcal{A}$ be the set of *atomic computations* over matrices. Intuitively, an atomic computation is an operation such as a "matrix multiply" or a "3-D convolution". Each has an input arity n, and a type specification function $f : \mathcal{M}^n \rightarrow \mathcal{M} \cup \{\perp\}$. For an atomic computation a, given a set of input matrix types, $a.f$ either returns the output type of a, or else it returns $\perp$ indicating that a cannot accept the input types. For example, if a is a matrix multiply, $a.f\left(\left(2, \langle 5, 10\rangle\right), \left(2, \langle 10, 5\rangle\right)\right)$ would return $(2, \langle 5, 5\rangle)$, as multiplying a 5×10 matrix and a 10×5 matrix results in a 5×5 matrix.

Let $\mathcal{I}$ be a set of *atomic computation implementations*. Whereas an atomic computation is an abstract computation, without an implementation, each $i \in \mathcal{I}$ is an implementation for a specific computation. An atomic computation implementation has an atomic computation $a \in \mathcal{A}$ of an input arity n that it implements, as well as a type specification function $f : (\mathcal{M} \times \mathcal{P})^n \rightarrow \mathcal{P} \cup \{\perp\}$. This type specification function is analogous to the type specification function associated with an atomic computation, except that it considers *both* the matrix type *and* the associated physical matrix implementation. The function returns $\perp$ if the atomic computation implementation cannot process the input types. For example, we may have a particular matrix multiplication implementation that works when both inputs are chunked into $10^3 \times 10^3$ chunks (in which case f may specify that the implementation outputs $10^3 \times 10^3$ chunks) but does not work if one input is chunked into $10^3 \times 10^3$ chunks and the other decomposed into column-strips. Here, the associated type specification function would output $\perp$ in this case.

Finally, let $\mathcal{T}$ be the set of *physical matrix transformations*. These transformations are associated with algorithms that move from one physical matrix implementation to another (for example, an algorithm that moves from a 1000×1000 tiling for a matrix, to a row-strip implementation with a row height of 10). Such transformations allow us to chain implementations of atomic computations (such as matrix multiplies) whose output and input physical implementations do not match. Each physical matrix transformation has a *matrix type specification function* $f : \mathcal{M} \times \mathcal{P} \rightarrow \mathcal{P} \cup \{\perp\}$. This function takes two arguments: an input matrix type and an input physical implementation, and it returns an output physical implementation if the transformation is feasible. Otherwise, $\perp$.

4 FORMAL PROBLEM DEFINITION

4.1 Compute Graph

Given these preliminaries, a *compute graph* $G = (V, E)$ is defined to correspond to the computation that we want to develop an optimized physical implementation for. A compute graph is a directed acyclic graph (DAG), whose structure corresponds to the logical computation that we wish to perform, where vertices are operations and edges control the flow of data. Since not all of the atomic computations are commutative, the input edges into a vertex have an implicit ordering that corresponds to the order of arguments.

Each source vertex (that is, each vertex with no incoming edges) in V corresponds to a matrix that is input into the computation, and hence it is labeled with both a matrix type m and an associated physical matrix implementation p. Each non-source vertex is labeled with an atomic computation a. Implicitly, each non-source vertex also has a matrix type m, which can be inferred by traversing the compute graph from the source vertices. Specifically, if a vertex v has input edges (v_1, v), (v_2, v), ..., then $v.m = v.a.f(v_1.m, v_2.m, ...)$.

The compute graph corresponding to the example of the Section 2 is shown in Figure 2.

4.2 Problem: Annotating a Compute Graph

The central problem studied in this paper is the problem of *annotating a compute graph*. When annotating a compute graph G to produce an annotated graph G', we have two subtasks:

(1) Label each non-source vertex in G with an atomic computation implementation i that will actually be run.
(2) Label each edge $e = (v_1, v_2)$ in G with a physical matrix transformation t. This transformation handles the case where the physical matrix implementation output from $v_1.i$ cannot be processed by $v_2.i$ (effectively, when it does not match the type requirement of $v_2.i$). In this case, the transformation $e.t$ is used to transform the output implementation so that it can be processed.

Note that annotating a compute graph implicitly assigns a physical matrix implementation p to each vertex, where p serves to describe the implementation associated with the output of the vertex. There are two cases: if v is a source vertex, then v is explicitly given, as every input matrix has a physical implementation.

If v is not a source vertex, then assume v has input edges $e_1 = (v_1, v), e_2 = (v_2, v), ...$. Then, the physical matrix implementation associated with the vertex is determined as:

$$v.p = v.i.f(v_1.m, e_1.t.f(v_1.m, v_1.p), \\ v_2.m, e_2.t.f(v_2.m, v_2.p), ...)$$

That is, the physical implementation of a vertex v is determined by taking the physical implementations of all vertices feeding into v, pushing those implementations through the transformations associated with each input edge, and then feeding those resulting implementations to $v.i.f$. Note that the physical implementations running on CPU, or accelerators such as GPUs and FPGAs would typically be different. Also note that for an implementation i, $i.f$ typically takes into account the hardware available, so that (for example) if $v.i$ requires the use of a GPU, $v.i.f$ would return $\perp$ if there was no enough GPU RAM to perform the operation.

Finally, we note that not all annotations are acceptable. The annotated compute graph G' must be *type-correct*. That is:

- The atomic computation implementation associated with each vertex must implement the correct atomic computation. It is not acceptable to annotate a vertex that corresponds to a matrix multiply with an implementation for a 3-D convolution. Formally, it must be the case that $v.i.a = v.a$.
- The atomic computation implementation i associated with each vertex must be able to correctly process all of its input physical matrix implementations. That is, intuitively, it is incorrect to feed a tile-matrix-multiply to matrices stored as single tuples. Formally, if v has input edges $e_1 = (v_1, v), e_2 = (v_2, v), \ldots$. Then, it must be the case that

$$v.i.f(v_1.m, e_1.t.f(v_1.m, v_1.p), \\ v_2.m, e_2.t.f(v_2.m, v_2.p), \ldots), \neq \perp$$

  or, equivalently, $v.p \neq \perp$.

The annotated compute graph corresponding to Implementation 2 for the example of the Section 2 is shown in Figure 2.

4.3 Optimizing the Annotation

There are many possible type-correct annotations for a given compute graph, and not all of them make sense from a computational point-of-view. Thus, our goal is not simply to produce a type-correct annotation, but to produce the *optimal* type-correct annotation.

Optimality will be defined in terms of the cost of the annotation. Thus, we augment each item in $\mathcal{I}$ and in $\mathcal{T}$ with a *cost function* c, which returns the time (or some other, appropriate notion of cost) of the implementation or transformation.

Costing atomic computation implementations. Specifically, for each $i \in \mathcal{I}$, we define the cost function $i.c : (\mathcal{M} \times \mathcal{P})^n \rightarrow \mathbb{R}$. That is, the cost function accepts a matrix type as well as a physical matrix implementation for each input, and returns a real number indicating the cost. Then, for a vertex $v \in V$ with input edges $e_1 = (v_1, v), e_2 = (v_2, v), \ldots$, we associate a cost:

$$v.c = v.i.c(v_1.m, e_1.t.f(v_1.m, v_1.p), \\ v_2.m, e_2.t.f(v_2.m, v_2.p), \ldots).$$

Costing physical matrix transformations. For each $t \in \mathcal{T}$, we define the cost function $t.c : \mathcal{M} \times \mathcal{P} \rightarrow \mathbb{R}$. That is, the cost function accepts a matrix type as well as a physical matrix implementations, and returns the cost of performing the transformation. Then, for an edge $e = (v_1, v_2)$ from E, define the cost:

$$e.c = e.t.c(v_1.m, v_1.p).$$

Algorithm 1: GetCost (V, E, v)

```
// This computes the cost associated with vertex v
// It also associates a transformation with each edge into v
Let e_1 = (v_1, v), e_2 = (v_2, v), ... be all input edges in E into v
Set each e_j.t so that v.i.f(
  v_1.m, e_1.t.f(v_1.m, v_1.p), v_2.m, e_2.t.f(v_2.m, v_2.p), ...) ≠ ⊥
v.p ←
  v.i.f(v_1.m, e_1.t.f(v_1.m, v_1.p), v_2.m, e_2.t.f(v_2.m, v_2.p), ...)
cost ← 0
cost += v.i.c(v_1.m, e_1.t.f(v_1.m, v_1.p), v_2.m, e_2.t.f(v_2.m, v_2.p)...)
for each input edge e_j ∈ {e_1, e_2, ...} do
  | cost += e_j.t.c(v_j.m, v_j.p)
end
return cost
```

Costing an annotation. Given this, the cost for an annotated graph $G' = (V, E)$ is simply:

$$Cost(G') = \sum_{v \in V} v.c + \sum_{e \in E} e.c.$$

The problem we consider is: out of all annotated, type-correct versions of G, choose the G' that minimizes the value $Cost(G')$. We denote this optimal, annotated, type-correct version of G by G^*.

A brute-force algorithm to compute G^* is given as Algorithm 2. It is invoked with $Brute(null, \infty, V, E, Copy(V), 0)$. At each invocation, Brute chooses a vertex from unset and considers all applicable atomic computation implementations for the vertex. For each, it incrementally updates the cost obtained so far, and then recursively considers all possible atomic computation implementations for the remainder of the vertices in unset.

5 OPTIMIZING TREE-SHAPED GRAPHS

5.1 Preliminaries

Fortunately, there exists a common class of compute graphs for which it is possible to compute G^* in time that is linear to the number of vertices in G. Specifically, the optimization can be done in linear time if G is "tree shaped"—that is, each vertex in G has only one directed out-edge. The optimization algorithm we develop for this particular case is a dynamic programming algorithm inspired by Felsenstein's algorithm in computational genetics [15].

We begin with some definitions. For tree-shaped G, for vertex v, we use G_v to denote the subgraph of G that consists of all vertices from which v can be reached, and all edges between those vertices. Using the notation from the last section, G_v^* is the cost-optimal, annotated, type-correct version of G_v.

Now, define the function $\mathcal{F}$ where $\mathcal{F}(v, \rho)$ returns $Cost(G_v^*)$, *subject to the constraint that* $v.p = \rho$ (that is, subject to the constraint that the physical matrix implementation resulting from the atomic computation implementation associated with v is ρ).

5.2 Recursively Computing the Optimal Cost

Next, we make an observation. Assume we have a vertex v with n input edges $e_1 = (v_1, v), e_2 = (v_2, v), \ldots, e_n = (v_n, v)$. Then, it is

Algorithm 2: Brute (bestG, lo, V, E, unset, costSoFar)

```
// Recursive algorithm to compute the optimal compute graph
if unset = {} then
    if costSoFar < lo then
        lo ← costSoFar; bestG ← Copy(V, E)
    end
    return
end
Pick a vertex v and remove v from unset
for i ∈ 𝓘 such that i.a = v.a do
    v.i ← i; cost ← costSoFar
    if it holds for all input edges into v: e1 = (v1, v),
     e2 = (v2, v), ... that v1 ∉ unset, v2 ∉ unset, and so on
     then
        cost += GetCost (V, E, v)
    end
    for v′ where (v, v′) ∈ E and forall (v″, v′) ∈ E,
     v″ ∉ unset do
        cost += GetCost (V, E, v′)
    end
    Brute (bestG, lo, V, E, unset, cost)
end
Add v back into unset
```

possible to compute $\mathcal{F}(v, \rho)$ from each $\mathcal{F}(v_j, .)$ Specifically, we enumerate all possible combinations of the following:

(1) An atomic computation implementation for v that results in the desired physical matrix implementation ρ;
(2) The physical matrix implementations output by each v_j;
(3) A set of physical matrix implementations (denoted by $\mathbf{p}^{\text{in}}$) input into the implementation for v, from each v_j; and
(4) A set of physical matrix transformations that move between the set of physical matrix implementations (denoted by $\mathbf{p}^{\text{out}}$) output by executing each v_j and the set of physical matrix implementations input into v.

For each possible combination that is type-correct, we compute a cost associated with the combination. Then, out of all enumerated combinations of implementations and transformations that result in the physical matrix organization ρ, we choose the lowest cost combination and use its cost as the value of $\mathcal{F}(v, \rho)$.

More formally, the following recurrence can be used to compute the optimal value of $\mathcal{F}(v, \rho)$ for any value of ρ:

$$\mathcal{F}(v,\rho) = \text{argmin}\ \{i \in \mathcal{I}, \mathbf{p}^{\text{in}} \in \mathcal{P}^n, \mathbf{p}^{\text{out}} \in \mathcal{P}^n, \mathbf{t} \in \mathcal{T}^n\} \left\{ \begin{array}{l} \infty \text{ if } i.a \neq v.a \\ \infty \text{ if } i.f(v_1.m, \mathbf{p}_1^{\text{out}}, v_2.m, \mathbf{p}_2^{\text{out}}, \ldots) \neq \rho \\ i.c(v_1.m, \mathbf{p}_1^{\text{out}}, v_2.m, \mathbf{p}_2^{\text{out}}, \ldots) \text{ otherwise} \end{array} \right. +$$

$$\sum_{j \in \{1 \ldots n\}} \left\{ \begin{array}{l} \infty \text{ if } \mathbf{t}_j.f(v_j.m, \mathbf{p}_j^{\text{in}}) \neq \mathbf{p}_j^{\text{out}} \\ \mathcal{F}(v_j, \mathbf{p}_j^{\text{in}}) + \mathbf{t}_j.c(v_j.m, \mathbf{p}_j^{\text{in}}) \text{ otherwise} \end{array} \right. \quad (1)$$

Algorithm 3: DPGraphOpt (V, E)

```
Forall v ∈ V, visited_v ← false
for each v with no input edges do
    // v is input data, so v.p is known
    visited_v ← true; 𝓕(v, v.p) ← 0
    Forall ρ ≠ v.p, 𝓕(v, ρ) ← ∞
end
while exists v ∈ V s.t. visited_v = false do
    Choose v ∈ V s.t. visited_v = false and where, for each
     vertex v′ s.t. (v′, v) ∈ E, visited_v′ = true
    visited_v ← true
    for ρ ∈ 𝒫 do
        Compute and record 𝓕(v, ρ) as in Equation (1)
    end
end
```

Intuitively, to compute the lowest cost solution that ensures the result of computing vertex v is the physical implementation ρ, we need to figure out how to minimize the sum of three terms:

(1) The cost to execute some atomic computation implementation i for $v.a$; and
(2) The cost to compute the cost-optimal, annotated, type-correct version of each input into v.
(3) The cost to transform each matrix associated with $\{v_1, v_2, ..., v_n\}$ to the physical matrix implementation required by i.

Those terms are all represented in Equation (1). In addition, there are a number of constraints represented by the recurrence.

(1) The chosen atomic computation implementation i must implement the atomic computation $v.a$ associated with v; if $i.a \neq v.a$, the wrong computation type is chosen and hence we incur a cost of ∞, indicating an infeasible solution.
(2) The atomic computation implementation must result in the correct output, i.e., the physical matrix implementation ρ.
(3) When we choose an atomic computation implementation, we must also be able to choose a transformation that produces the correct physical matrix implementation for that atomic computation implementation.

This recurrence optimally computes $\mathcal{F}(v, \rho)$ from each $\mathcal{F}(v_j, .)$ because first, each $\mathcal{F}(v_j, .)$ computes the optimal cost for each G_{v_j}. Second, when choosing the optimal atomic computation implementation to ensure that $v.i = \rho$, the details of the actual computation associated with each G_{v_j} are irrelevant, other than the physical matrix implementation that is output. Thus, once we compute each $\mathcal{F}(v_j, .)$, this function "hides" the details of the sub-computation.

5.3 Dynamic Programming

Then, all of this suggests a simple, dynamic programming algorithm to compute G^* from G in the case of a tree-shaped graph. This is given as Algorithm 3, which is used to compute $\mathcal{F}$. After computing $\mathcal{F}$, traverse backward through the graph. Label each edge with the physical matrix transformation that was used to produce the optimal cost, and label each vertex with the atomic computation

Algorithm 4: FrontierGraphOpt (V, E)

Forall $v \in V$, visited$_v \leftarrow$ false; front $\leftarrow \{\}$
for *each* v *with no input edges* **do**
  // v is input data, so $v.p$ is known
  visited$_v \leftarrow$ true; $\mathcal{F}(\{v\}, v.p) \leftarrow 0$
  Forall $\rho \neq v.p$, $\mathcal{F}(\{v\}, \rho) \leftarrow \infty$
  front $\leftarrow$ front + $\{v\}$ // The frontier is a set of sets
end
while *exists* $v \in V$ *s.t.* visited$_v$ = *false* **do**
  Choose $v \in V$ s.t. visited$_v$ = false and where, for each vertex v' s.t. $(v', v) \in E$, v' is in some set in front
  Find $V_1^F, V_2^F, \ldots \in$ front where each V_j^F contains a v' s.t. $(v', v) \in E$
  visited$_v \leftarrow$ true
  $V^F \leftarrow \{V_1^F, V_2^F, \ldots\}$
  $V_v^F \leftarrow (\cup_j V_j^F) - \{v'$ if there does not exist an edge from v' to an unvisited node$\}$
  front $\leftarrow$ front $- V^F + V_v^F + \{v\}$
  for $\rho \in \mathcal{P}$ **do**
    Compute and record $\mathcal{F}(V_v^F, \rho)$ as in Equation (2)
  end
end

implementation and the resulting physical matrix implementation that produced the optimal cost.

6 OPTIMIZING GENERAL DAGS

The algorithm to compute G^* for DAGs is more involved. Consider the graph associated with the computation of the matrix $\mathbf{O}$:

$$\mathbf{T}_1 \leftarrow \mathbf{S} \times \mathbf{T};\ \mathbf{T}_2 \leftarrow \mathbf{T}_1 \times \mathbf{U}$$
$$\mathbf{O} \leftarrow ((\mathbf{R} \times \mathbf{T}_1) + \mathbf{T}_2) + (\mathbf{T}_2 \times \mathbf{V})$$

$\mathbf{T}_2$ is used in multiple places, and so we no longer have a DAG. Modern back-propagation algorithms have this structure.

When two vertices v_1 and v_2 in a graph are descended from the same vertex v, we wish to avoid optimizing and executing G_v twice. For example, both the matrices $\mathbf{T}_2$ and $\mathbf{O}$ depend on intermediate matrix $\mathbf{T}_1$. Therefore, $\mathbf{T}_1$ should be computed once and re-used.

Thus, when two vertices v_1 and v_2 share the same ancestor v, the optimal costs $\mathcal{F}(v_1, \rho_1)$ and $\mathcal{F}(v_2, \rho_2)$ cannot be computed independently, as this implicitly assumes that we *are* computing all of G_{v_1} and then all of G_{v_2} separately, and not re-using the sub-computation G_v. Thus, we need to generalize the algorithm to allow for sharing.

6.1 Optimizing Costs Along a Frontier

To extend our algorithm to general DAGs, we first define the notion of a *frontier*. A frontier is simply a cut of the input graph G, into an *optimized* portion, and an *unoptimized* portion that has not yet been processed. The set of vertices that have one or more of their out-edges cut are said to be "along the frontier".

Our algorithm will begin with an initial frontier that separates all of the source vertices from the rest of the graph—the source vertices are already "optimized", because we are given the physical matrix implementation for each source vertex as input. Then, our algorithm will iteratively move one vertex from the set of unoptimized vertices to the set of optimized vertices, and update $\mathcal{F}$ as it does so. This progression is depicted in Figure 3. Once all of the vertices have been optimized, the algorithm is done.

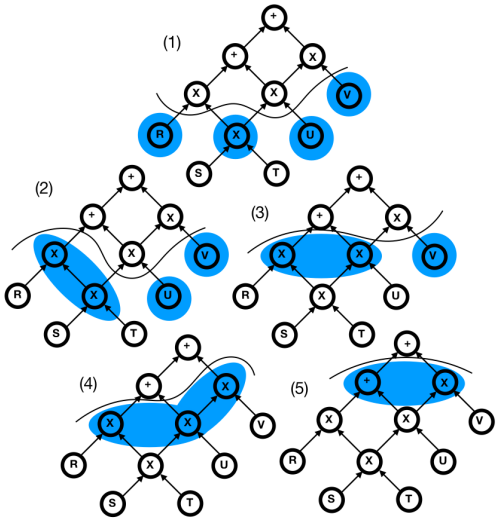

Figure 3: Moving the frontier as vertices are moved from the un-optimized set to the optimized set. The set of equivalence classes along the current frontier is shown via blue shading.

As in the algorithm for tree-like graphs, we will maintain the optimal cost as a function $\mathcal{F}$, and define a recurrence that allows us to compute $\mathcal{F}$ by looking at the optimal way to perform sub-computations. However, there is a key difference. At all times during the execution of the algorithm, as the frontier moves through the graph, *we must maintain the optimal cost not on a per-vertex basis, but jointly, for sets of vertices on the frontier that share ancestors.* Again, the intuition here is that two vertices that share the same ancestor cannot be considered independently, as we want them to share common sub-computations.

At all times during the execution of the algorithm, the set of vertices along the current frontier are partitioned into equivalence classes $V_F = \{V_1^F, V_2^F, V_3^F, \ldots\}$ so that if two vertices v_1 and $v_2 \in V_F$ have a common ancestor in G, then they must be in the same equivalence class. The partitioning of vertices along the frontier into equivalence classes is depicted in Figure 3.[1] $\mathcal{F}$ is now no longer defined over all vertices, but rather over each equivalence class that has appeared along a frontier at some time during the optimization.

For equivalence class V, let $\mathbf{p} \in \mathcal{P}^{|V|}$ be a list of physical matrix organizations, where $\mathbf{p}_v$ for $v \in V$ refers to the physical matrix organization resulting from the execution of vertex v. (Note that in the remainder of this section, we will use the convention that when we have a vector or list of candidate physical matrix implementations or matrix transformations, we subscript with the vertex to obtain the implementation/transformation associated with that

[1] To perform the partitioning, we create an undirected graph G' whose set of vertices is the set of vertices along the current frontier; two vertices v_1 and v_2 in G' are connected if they share an ancestor in G. Then, two vertices along the frontier are in the same equivalence class if and only if they are reachable from one another in G'.

vertex). Then, $\mathcal{F}(V, \mathbf{p})$ refers to the minimum cost to compute every vertex in V, subject to the constraint that the output physical matrix organizations are exactly as specified by $\mathbf{p}$.

6.2 Moving the Frontier Forward

Our generalization of the algorithm for tree-shaped graphs operates by iteratively picking a vertex v that is in the unoptimized portion of the graph, but whose input edges are *only* in the optimized portion of the graph. This vertex is then added to the the optimized portion of the graph, and $\mathcal{F}$ and the frontier are updated. As depicted in Figure 3, moving v from the unoptimized to the optimized portion changes the set of vertices along the current frontier.

Let V_v^F denote the equivalence class in the *new* version of the frontier that contains v, and let $V^F = \{V_1^F, V_2^F, V_3^F, ...\}$ denote the equivalence classes in the partitioning in the *current* version of the frontier. Further, let $V^{\text{arg}} = \{v_1^{\text{arg}}, v_2^{\text{arg}}, ...\}$ denote the set of vertices that have an out-edge to v. Consider only the equivalence classes in V^F which have a non-empty intersection with $V_v^F \cup V^{\text{arg}}$; assume that the total number of vertices in these equivalence classes is n.

The key question is: How do we update $\mathcal{F}$ in response to the update of the set of vertices along the frontier?

For the equivalence class V_F^v in the new version of the frontier that contains v, and a candidate set $\mathbf{p}$ of associated, physical matrix organizations, we have the following variant on Equation (1):

$$\mathcal{F}(V_v^F, \mathbf{p}) =$$
$$\text{argmin } \{i \in \mathcal{I}, \mathbf{p}^{\text{in}} \in \mathcal{P}^n, \mathbf{p}^{\text{out}} \in \mathcal{P}^n, \mathbf{t} \in \mathcal{T}^{|V_{\text{arg}}|}\}$$
$$\begin{cases} \infty \text{ if } i.a \neq v.a \\ \infty \text{ if } i.f(v_1^{\text{arg}}.m, \mathbf{p}^{\text{out}}_{v_1^{\text{arg}}}, v_2^{\text{arg}}.m, \mathbf{p}^{\text{out}}_{v_2^{\text{arg}}}, ...) \neq \mathbf{p}_v \\ i.c(v_1^{\text{arg}}.m, \mathbf{p}^{\text{out}}_{v_1^{\text{arg}}}, v_2^{\text{arg}}.m, \mathbf{p}^{\text{out}}_{v_2^{\text{arg}}}, ...) \text{ otherwise} \end{cases} +$$
$$\sum_{V=\{v_1,v_2,...\} \in V^F} \begin{cases} 0 \text{ if } V \cap (V^{\text{arg}} \cup V_v^F) = \varnothing \\ \infty \text{ if } \exists v \in (V - V^{\text{arg}}) \\ \qquad \text{s.t. } \mathbf{p}_v^{\text{in}} \neq \mathbf{p}_v^{\text{out}} \text{ or } \mathbf{p}_v^{\text{in}} \neq \mathbf{p}_v \\ \infty \text{ if } \exists v \in (V \cap V^{\text{arg}}) \\ \qquad \text{s.t. } \mathbf{t}_v.f(v.m, \mathbf{p}_v^{\text{in}}) \neq \mathbf{p}_v^{\text{out}} \\ \mathcal{F}(V, \langle \mathbf{p}_{v_1}^{\text{in}}, \mathbf{p}_{v_2}^{\text{in}}, ... \rangle) + \\ \qquad \sum_{v \in V \cap V^{\text{arg}}} \mathbf{t}_v.c(v.m, \mathbf{p}_v^{\text{in}}) \text{ otherwise} \end{cases} \tag{2}$$

This recurrence mirrors Equation (1). To compute $\mathcal{F}(V_v^F, \mathbf{p})$, we need to figure out how to minimize the sum of the two terms:

(1) The cost to execute some atomic computation implementation i of $v.a$ (the atomic computation associated with v) that produces the physical matrix implementation $\mathbf{p}_v$; and
(2) The cost to transform each matrix associated with a node in V^{arg} to the physical matrix implementation required by i.

Those terms are both represented in Equation (2). In addition, there are a number of constraints on any potential solution, that are also represented in Equation (2), including:

(1) When we choose an atomic computation implementation, we must also be able to choose a transformation that produces the correct physical matrix implementation for that atomic computation implementation; and
(2) For any vertex v' in V_v^F aside from v, the computation of v does not affect the physical matrix implementation for v'; thus, we must choose a subcomputation that produces exactly the physical matrix implementation $\text{p}_{v'}$.

Dynamic Programming. The dynamic programming algorithm to compute the minimum cost for general DAGs with the frontier notation is given as Algorithm 4.

6.3 Algorithm Efficiency

We now compare efficiency of the algorithms of Sections 4, 5 and 6.

Let n be the maximum number of inputs/outputs to any vertex. The brute-force algorithm (Algorithm 2) iterates through $|\mathcal{I}|^{|V|}$ different combinations of atomic computation implementations. Each of those combinations requires that we execute lines 10 to 16 of Algorithm 2. The cost of these lines can be estimated as $O(n^2)$. Thus, the total cost is $O(n^2|\mathcal{I}|^{|V|})$. This algorithm is very space efficient as it only ever stores (a) the current graph as it iterates through all possible graphs, and (b) the best graph obtained so far.

Now, consider the dynamic programming algorithm given as Algorithm 3. For a vertex v, the way we implement the loop of line 10 of Algorithm 3 is to check, for each of the atomic computation implementations available to the vertex, the best way to provide each of the required input types. If the output physical matrix implementation type is ρ, check if it gives us the best $\mathcal{F}(v, \rho)$ value; if so, update $\mathcal{F}$. In general, for atomic computation implementation, this requires that for each of the maximum of n inputs to v, and consider up to $\mathcal{P}$ different physical implementations for each. Thus, the loop of line 10 requires time $O(n|\mathcal{P}||\mathcal{I}|)$. For the entire graph, the time complexity overall is $O(n|\mathcal{P}||\mathcal{I}||V|)$. The space complexity of a straightforward implementation that never "forgets" entries in $\mathcal{F}$ that will not be used in the future is $O(|\mathcal{P}||V|)$.

Now, consider the frontier-based algorithm for general DAGs. The key difference from the original DP algorithm is that a set of nodes may be part of the same equivalence class, and the costs must be considered together. Assume that each equivalence class is bounded in size by c. Then there are at most $|\mathcal{P}|^c$ costs that must be maintained in $\mathcal{F}$ for each equivalence class. Consider line 15 in Algorithm 4. If our implementation of line 15 mirrors the implementation of line 10 in Algorithm 3, cost of the loop is $O(n|\mathcal{P}|^c|\mathcal{I}|)$. Then the overall time complexity is $O(n|\mathcal{P}|^c|\mathcal{I}||V|)$. The space complexity of a straightforward implementation is $O(|\mathcal{P}|^c|V|)$, since one equivalence class is created as each vertex is considered.

7 ESTIMATING COSTS

In the case of dense inputs, it is typically possible to develop simple, analytic formulas for a number of features describing each atomic computation implementation and physical matrix transformation. These features include: (1) the number of floating point operations required by the implementation, (2) the amount of network traffic that will be generated by the implementation in the worst case[2], (3) the bytes of intermediate data that will be pushed through the

[2]We say, "in the worst case" because, at least in a relational implementation, the system may choose from a set of different implementations for relational operations such as joins, which can affect the network traffic.

computation in the worst case, and (4) the number of tuples that will be pushed through the computation in the worst case, as each tuple tends to require a fixed overhead cost. At installation time, our implementation runs a set of benchmark computations for which it collects the running time, and then it uses the aforementioned analytically-computed features along with those running times as input into a regression that is performed for each operation. Then, the set of learned regression models will serve as the cost function.

What about sparsity? Choosing sparse tensor/matrix implementations and operations can lead to significant time savings, but costing such implementations presents a challenge. Floating point operations, bytes transferred, and intermediate data bytes are affected by sparsity. In our prototype implementation, the cost model also makes into account the level of sparsity when predicting costs (where sparsity is defined as the fraction of items in the matrix or tensor that are non-zero). Note that the sparsity for all inputs can easily be estimated as data are loaded.

This approach works for certain computations—especially machine learning computations including the feed-forward neural network tested in the experimental section—where the input is sparse, but the model is not. Operations such as matrix multiplies between sparse data matrices and dense model matrices typically result in dense matrices with no sparsity to exploit. This means that all intermediate matrices/tensors are dense.

Things are more difficult given *chains* of operations over sparse inputs, where intermediate results are *not* dense. It is not an easy task to estimate the sparsity of intermediate results for a long chain of sparse operations, which is necessary to cost the compute plans considered by our model. This is analogous to the problem of compounding statistical errors when costing relational plans [22].

One idea to handle this—left to future work—is to use our proposed optimization algorithms along with a framework such as that proposed by Sommer et al. to estimate the sparsity of all intermediate results [33] and use those estimates in the cost model. Sommer's MNC method showed remarkable accuracy, with relative errors never exceeding 1.07 for a graph of six operations over six input matrices (in Sommer's definition of relative error, 1.0 is perfect [10]). Of course, it is always possible that any sparsity estimator can fail. During execution of the plan, it is easy to compute the sparsity of each intermediate result. If the relative error in estimated sparsity exceeds some value (say, 1.2), then execution can be halted, and the remaining plan re-optimized. This is analogous to re-optimization methods used in relational databases to deal with the problem of compounding estimation errors [5, 25].

8 EXPERIMENTS

8.1 Overview

We detail several sets of experiments. The first set compares the quality of our automatically-generated plans to human-generated plans. The second compares the runtime of auto-generated plans to an equivalent computation run natively on PyTorch [1] and SystemDS [2] (formerly SystemML). Besides recording the optimization times for producing the auto-generated code for the experiments above, we added a third experiment set to specifically examine the runtime efficiency of our proposed optimization algorithms.

Input Matrix	Size Set 1	Size Set 2	Size Set 3
A	10K × 30K	50K × 1	50K × 50K
B	30K × 50K	1 × 100K	50K × 50K
C	50K × 1	100K × 30K	50K × 50K
D	1 × 50K	30K × 100K	50K × 50K
E	50K × 10K	100K × 50K	50K × 50K
F	50K × 10K	100K × 30K	50K × 50K

Figure 4: Size combinations for matrix multiplication chain.

We implement our optimization algorithm on top of SimSQL, which is a Hadoop-based parallel relational database [9], and on top of PlinyCompute [41], which is a high-performance distributed relational engine. Our implementation includes a total of 19 physical matrix implementations, 20 different physical matrix transformations, 16 different atomic computations, 38 different atomic computation implementations.

All SimSQL experiments are run on Amazon EC2 `r5d.2xlarge` machines with 8 cores, 68GB of RAM, and 300GB of NVMe SSD. All PlinyCompute, PyTorch, and SystemDS experiments are run on Amazon EC2 `r5dn.2xlarge` machines with 8 cores, 64GB of RAM, and 300GB of NVMe SSD.

8.2 Quality of Observed Plans

We first detail our experiments aimed at answering the question: *Can the optimization framework described in this paper automatically choose a set of atomic operation implementations and transformations that outperform those chosen by heuristic, or by a human programmer?* We consider four different computational tasks. All are run on our SimSQL implementation; all matrices are dense.

Feed forward neural network backprop. Consider a FFNN with three hidden layers, relu activation functions, and a softmax output later. We perform backprop using a dense input matrix having 10^4 input vectors with 6×10^4 features each. There are 17 possible possible labels. Besides the input and output layer, there are two hidden layers, whose weight matrices have size 60,000 by `layer_size`, `layer_size` by `layer_size`, respectively. FFNN input and weight matrices (and the matrices for inversion and matrix-chain mult below) are dense and generated by sampling double-precision floating point numbers from a Normal$(0, 1)$ distribution.

We initially evaluate three methods for computing the physical plan for the FFNN. The first method is to auto-generate the plan using the algorithms of this paper (note the FFNN is not a tree, so the frontier algorithm is used). For the second method we use the SimSQL FFNN code that was derived from the code used in a published paper [23]. The third method is to simply tile everything with 1K × 1K matrices. For Experiment 4, we recruit additional programmers to create plans, as described below.

There are four experiments aimed at evaluating plan quality.

Experiment 1. On ten machines, we first perform the computation required to compute the activations at the output layer during the second forward pass, which means we run one forward pass, one backpropagation, and one more forward pass, using ten machines and a hidden layer size of 80K. This results in a very large compute graph, with 57 vertices. Times are given in Figure 5.

Experiment 2. Next, we run the FFNN computation on ten machines to compute the updated weight matrix for the second hidden layer, which requires a forward pass and one backpropagation, with a variety of hidden layer sizes. We try different values for the `layer_size` from 10K, 40K, 80K to 160K. Times are given in Figure 6. "Fail" means that the system crashed, typically due to too much intermediate data.

Experiment 3. We fix the size of the hidden layer as 160K, and repeat the computation using different numbers of machines. Times are given in Figure 7. Again, "Fail" means that the system crashed.

Experiment 4. One of the key hypotheses behind the paper is that it is difficult for a human programmer to choose the "correct" matrix/tensor and operator implementations so that things run fast. To evaluate that hypothesis, we compared auto-generated code with the human-produced FFNN code from [23]. However, that is just one human-generated computation. To better evaluate that hypothesis, we reconsider the third work of Figure 6, and ask three additional ML experts to produce SimSQL implementations. The additional programmers are all PhD students working in ML, with a correspondingly-high level of ML expertise. One works in the area of high-performance distributed ML (distributed ML expertise is "high"), one works in federated learning (distributed ML experience is "medium") and one works in ML applications (distributed ML experience is "low"). Each was given a 21-page handbook describing task, which included labeling 19 different operations in the compute graph, as well as various rules and considerations to take into account while labeling the operations. The resulting labelings were translated (by us) into three different SimSQL implementations. The first attempts by the programmers with "low" and "medium" distributed ML experiences crashed, and we asked them to update the labeling accordingly. Running times are given in Figure 8.

Two-level matrix inverse. A classic implementation [18] for distributed matrix inverse is to compute $\begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{C} & \mathbf{D}\end{bmatrix}^{-1} = \begin{bmatrix}\bar{\mathbf{A}} & \bar{\mathbf{B}}\\ \bar{\mathbf{C}} & \bar{\mathbf{D}}\end{bmatrix}$ where:

$$\bar{\mathbf{A}} = \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1},$$
$$\bar{\mathbf{B}} = -\mathbf{A}^{-1}\mathbf{B}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1},$$
$$\bar{\mathbf{C}} = -(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1},$$
$$\bar{\mathbf{D}} = (\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}.$$

Since this is a *two-level* block-wise matrix inverse, the matrix inverse $\mathbf{A}^{-1}$ is computed similarly. For the experiment, we set the size of block `A`, `B`, `C`, and `D` as 10K by 10K. The input blocks of matrix `A` have size 2K by 2K, 2K by 8K, 8K by 2K, and 8K by 8K, respectively. We compare the auto-generated implementation with a hand-generated implementation (the best possible physical implementation obtained by the first author of this paper) and one simply tiling all matrices in 1K × 1K chunks. The time to run the matrix inversion is given in Figure 9, using ten machines.

Matrix Multiplication Chain. Consider six input matrices: **A**, **B**, **C**, **D**, **E**, and **F**. The specific computation is:

$$\mathbf{T}_1 \leftarrow \mathbf{A} \times \mathbf{B}; \mathbf{T}_2 \leftarrow \mathbf{C} \times \mathbf{D}$$
$$\mathbf{O} \leftarrow ((\mathbf{T}_1 \times \mathbf{E}) \times (\mathbf{T}_1 \times \mathbf{T}_2)) \times (\mathbf{T}_2 \times \mathbf{F})$$

Auto-gen	Hand-written	All-tile
0:59:02 (01:03)	1:25:34	1:54:18

Figure 5: FFNN: forward pass, backprop, and one more forward pass. Format is H:MM:SS (opt time in parens is MM:SS).

Dims	Auto-gen	Hand-written	All-tile
10K	00:06:15 (:08)	00:10:06	00:09:01
40K	00:12:18 (:11)	00:17:58	00:18:43
80K	00:23:46 (:06)	00:42:47	00:50:23
160K	00:55:16 (:04)	02:15:01	Fail

Figure 6: FFNN: forward pass, plus backpropagation to $\mathbf{W}_2$. Format is HH:MM:SS (Opt time in parens is :SS).

Num workers	Auto-gen	Hand-written	All-tile
5	01:19:32 (:04)	Fail	Fail
10	00:55:16 (:04)	02:15:01	Fail
20	00:44:19 (:04)	01:19:27	01:45:50
25	00:38:19 (:05)	01:18:59	01:31:15

Figure 7: FFNN: forward pass, plus backpropagation to $\mathbf{W}_2$. Format is HH:MM:SS (Opt time in parens is :SS).

	Auto-gen	User 1	User 2	User 3
ML expertise	NA	High	High	High
Dist. ML expertise	NA	Low	Med	High
Runtime	23:46	55:23*	36:02*	23:58

Figure 8: FFNN: forward pass, plus backprop to update $\mathbf{W}_2$. Format is MM:SS. *Computation failed, then re-designed.

We use **O** as the final output. We use three different size combinations for the input matrices. The sizes for each combination are provided in Figure 4. Again, we compare the auto-generated implementation with a hand-written implementation and one tiling all matrices in 1K × 1K chunks. The time required to run the chain of matrix multiplications is given in Figure 10.

Discussion. In every case, the auto-generated physical plans selected by the optimizer beat both the hand-developed code, and also the heuristic of simply tiling every matrix. The auto-generated code even beat the hand-written code in Figures 5, 6 an 7, where those hand-written code were derived from the one used for the FFNN experiments for a published paper [23].

Note that for experiment 4, one recruited user was able to produce an implementation that nearly matched the quality of the auto-generated implementation (Figure 8). It was likely not a coincidence that this particular user is an expert in distributed machine learning. The two recruits with less expertise produced implementations that were significantly less performant.

8.3 Comparison with Other Systems

We now detail a second set of experiments aimed at answering: *Can the algorithms of this paper auto-generate implementations that compete with implementations provided by state-of-the-art systems?*. In these experiments, we use the proposed optimization algorithm to choose the best set of formats to implement the FFNN computation on top of PlinyCompute [41], and compare with both PyTorch [1] and SystemDS [2] (formerly SystemML).

Auto-gen	Hand-written	All-tile
21:31 (:21)	28:19	34:50

Figure 9: Two-level block-wise matrix inverse. Format is MM:SS (opt time in parens is :SS).

Input size	Auto-gen	Hand-written	All-tile
Size Set 1	00:08:45 (:05)	00:20:22	00:21:38
Size Set 2	01:05:36 (:00)	02:26:32	01:56:15
Size Set 3	00:34:52 (:00)	01:46:20	02:02:54

Figure 10: Matrix multiplication chain. Format is HH:MM:SS (opt time in parens is :SS).

We perform FFNN backpropagation using the AmazonCat-14K dataset [28, 29] with 597,540 input features and 14,588 labels. We consider 1K and 10K batches. For PlinyCompute, the large input data matrix is stored as column-strips with strip width 1000 (this matrix is a $b \times 597,540$ matrix for a batch size of b). The large matrix connecting the inputs to the hidden layer is given to PlinyCompute as 1000×1000 chunks (this matrix is a $597,540 \times h$ matrix for a hidden layer size of h). All other input matrices are given to the system as whole, unchunked matrices. For PyTorch, a standard, "data parallel" implementation is used [19]; the input data matrix is sharded into column strips so each machine gets one shard. For SystemDS, the computations are implemented by using the linear algebra operations provided by the system, and is configured to use Intel MKL library [36] for BLAS [7] computations.

We run two subsets of experiments. In the first, an input batch size of 1000 is used, and PlinyCompute is constrained to use dense operations. A variety of cluster sizes and hidden layer sizes are tested. Results are given in Figure 11. In the second, we use a 10K batch size, and try three options for PlinyCompute. In the first, the input data are stored densely and PlinyCompute is constrained so that it *cannot* choose to transform the input to sparse format. In the second, the inputs are stored densely, but the constraint is removed. In the third, the inputs are stored sparsely. Results are in Figure 12.

Discussion. The optimized computations outperform PyTorch in each case. For a size 1000 batch, PlinyCompute is up to 2.5 times faster. PyTorch's data-parallel implementation broadcasts the entire model to all machines, which is problematic with such a large model—it would be better to move the data. The optimized implementation is able to avoid this. Note that PyTorch also failed in many of the cases, especially for the case of the larger batch size. This is because PyTorch is unable to multiply the matrix storing the input data with the entire matrix connecting the inputs to the first input layer without failing.

SystemDS has performance similar to PyTorch on the smaller batches. On the larger batch task, SystemDS' ability to take advantage of the very sparse one-hot-encoded input allows it to outperform PyTorch, and it is close to matching the auto-generated PlinyCompute performance, when the optimization algorithm's ability to utilize sparsity is turned off. But when the ability to choose sparse operations is turned on, the runtime drops considerably, to just 20% to 50% of the all-dense implementation.

Layer Size	PC No Sparsity	PyTorch	SystemDS
Cluster with 2 workers			
4000	0:23 (:04)	0:26	1:10
5000	0:28 (:03)	0:31	1:24
7000	0:53 (:03)	Fail	1:36
Cluster with 5 workers			
4000	0:18 (:04)	0:39	0:56
5000	0:20 (:04)	0:46	1:01
7000	0:30 (:03)	Fail	0:39
Cluster with 10 workers			
4000	0:20 (:04)	0:40	0:44
5000	0:22 (:03)	0:50	0:52
7000	0:25 (:04)	Fail	0:34

Figure 11: FFNN forward pass plus backprop, 1K batch. Format is M:SS (opt time in parens is :SS).

8.4 Optimization Runtimes

Finally, we detail a third set of experiments aimed at answering the question: *How important is the efficient dynamic programming algorithm? Is the brute-force algorithm sufficient?*. We consider three specific computations.

Computation 1: DAG2. Consider the following matrix computation:

$$\mathbf{T}_1 \leftarrow \mathbf{A} \times \mathbf{B}; \mathbf{T}_2 \leftarrow \mathbf{C} \times \mathbf{D}$$
$$\mathbf{O}_1 \leftarrow (\mathbf{T}_1 \times \mathbf{T}_2) \times \mathbf{E}; \mathbf{O}_2 \leftarrow (\mathbf{T}_1 \times \mathbf{T}_2) \times \mathbf{O}_1$$

This is a "scale 1" chain. We can construct a larger chain, called a "scale 2" chain, by repeating the above pattern and connecting to the first computation by replacing matrix $\mathbf{A}$ with $\mathbf{O}_2$, and replace matrix $\mathbf{C}$ with $\mathbf{O}_1$ (this is "DAG2" because the two scales are connected in two locations, creating a more complicated dependency). This can be repeated $n-1$ times to create a "scale n" chain. We use the lastly-produced matrix as the output matrix to trigger the computation.

Computation 2: DAG1. We can create a computation with a simpler dependency by connecting computations by replacing $\mathbf{A}$ with $\mathbf{O}_2$.

Computation 3: Tree. Lastly, consider the tree-shaped computation:

$$\mathbf{T}_1 \leftarrow \mathbf{A} \times \mathbf{B}; \mathbf{T}_2 \leftarrow \mathbf{C} \times \mathbf{D}$$
$$\mathbf{O}_1 \leftarrow (\mathbf{T}_1 \times \mathbf{T}_2) \times \mathbf{E}; \mathbf{O}_2 \leftarrow \mathbf{O}_1 \times \mathbf{F}$$

We can create a "scale n" tree computation by linking together n trees by replacing matrix $\mathbf{A}$ with $\mathbf{O}_2$.

For each of these computations, we construct tasks at a number of different scales and compare the runtime of the brute-force optimization algorithm (Algorithm 2) with either the DP algorithm (Algorithm 3) in the case of the tree computations, or the Frontier algorithm (Algorithm 4) in the case of DAG1 and DAG2. We assume all matrices are initially stored as a single tuple, with size $20,000 \times 20,000$, and we assume the computations would be run on 10 machines. Finally, we tested three cases: all formats in the SimSQL implementation are available; only single-matrix, strip and block formats are available (there are a total of 16 formats in this case), and only single-matrix and block format are available (there are a total of 10 formats in this case). Results are given in Figure 13. "Fail" means a runtime of greater than 30 minutes.

Layer Size	PC No Sparsity	PCSparse Input	PCDense Input	Py-Torch	System-DS
Cluster with 2 workers					
4000	1:34 (:05)	0:50	0:54	2:05	1:57
5000	2:47 (:05)	0:58	1:02	Fail	2:51
7000	4:24 (:05)	1:16	1:19	Fail	7:54
Cluster with 5 workers					
4000	1:15 (:06)	0:23	0:27	1:16	1:15
5000	1:20 (:05)	0:26	0:32	1:30	1:30
7000	1:55 (:05)	0:35	0:38	Fail	2:49
Cluster with 10 workers					
4000	0:53 (:06)	0:20	0:24	1:06	1:01
5000	1:02 (:05)	0:20	0:24	1:17	1:15
7000	1:16 (:05)	0:23	0:28	Fail	1:21

Figure 12: FFNN: forward pas plus back-prop, 10K batch. Format is M:SS (opt time in parens is :SS).

Scale	DP DAG2	Brute DAG2	DP DAG1	Brute DAG1	DP Tree	Brute Tree
All formats						
1	00:01	26:54	00:01	27:13	00:00	25:31
2	00:08	Fail	00:01	Fail	00:01	Fail
3	00:16	Fail	00:02	Fail	00:01	Fail
4	00:23	Fail	00:03	Fail	00:02	Fail
Single/Strip/Block formats						
1	00:00	24:04	00:00	23:57	00:00	19:14
2	00:06	Fail	00:02	Fail	00:00	Fail
3	00:11	Fail	00:02	Fail	00:01	Fail
4	00:15	Fail	00:03	Fail	00:01	Fail
Single/Block formats						
1	00:00	00:28	00:00	00:26	00:00	00:20
2	00:00	Fail	00:00	Fail	00:00	Fail
3	00:00	Fail	00:00	Fail	00:00	Fail
4	00:02	Fail	00:00	Fail	00:00	Fail

Figure 13: Opt times for matrix mult chain. Format is MM:SS.

Discussion. The exponential-complexity brute-force algorithm is a viable option only for the smallest compute graphs, with only a few matrix implementations available. On the other hand, the dynamic programming algorithms show a linear scale-up with graph size. Interestingly, there is a strong dependence on the complexity of the linkage between the subgraphs used to create the overall computation. Increasing the number of links to two increases the running time from 3 seconds to 23 seconds for the scale 4 task.

9 RELATED WORK

We have proposed an algorithm for globally optimizing algorithm choices and storage formats for large-scale, distributed matrix/tensor computations. While we have proposed an optimization algorithm and not a new system per se, several existing systems perform related optimizations. Notably, SystemDS (formerly SystemML) [13, 17] offers many of the operator implementations that were available to our SimSQL and PlinyCompute prototypes, and will automatically figure out the layouts for matrices, as well as the implementations for operations over them. Internally, SystemDS provides two layouts for dense matrices: block matrix (stored as 1000×1000 blocks), and single-tuple matrix (the matrix is stored as a single tuple). SystemDS also provides three layouts for sparse matrices: triple-values, compressed sparse row (CSR), and a modified CSR [8]. While SystemDS was one of the pioneering efforts in automated methods for choosing formats and operator implementations, its computation/format is optimized independently (or it is considered within a small group of operators fused into a compound operator), with a consideration of size, sparsity, and the processing type (local processing or distributed processing). Unlike the algorithms proposed in this paper, SystemDS does not attempt to globally optimize matrix/tensor layouts, and does not integrate the costs of transformations between the various layouts into the optimization problem, which is the key idea in this paper.

Other systems for distributed ML or linear algebra also optimize operator implementations and layouts, though no work we are aware of attempts to systematically and globally optimize operator implementations, layouts, and the transformations. MATFAST [40] optimizes the physical implementations of matrix-based programs for a distributed in-memory system. Cumulon [20] can automatically optimize R-like programs in terms of physical implementations and hardware settings. Spartan [21] is a distributed array framework that can automatically choose tile sizes for arrays to maximize the data locality. DMac [39] inspects the matrix dependencies between each operation in a matrix-based program, and utilizes those dependencies to select the optimal execution strategy for each operation. [26] proposes a hybrid representation with adaptive tiles for large sparse and dense matrices, and considers the optimization of matrix multiplication on those matrices. The deep learning community has considered automatically choosing the optimal tiling strategies for tensors to improve data locality and model parallelism. Some examples are [24], [37] and [34].

Automatically selecting the best implementation for a linear algebra computation is referred as *autotuning* in the HPC literature. Autotuning can be supported in multiple ways. Some widely-used autotuning HPC libraries are PHiPAC [6], ATLAS [38] and FFTW [16]. Better abstraction is supported by other libraries such as SPIRAL [32] and [14]. Moreover, autotuning can also be supported in an application level, where more complex LA computations can be autotuned and more algorithmic choices can be explored. Examples of such system are [4], [12] and [35].

10 CONCLUSIONS

We have proposed a framework for automatic optimization of the physical implementation of matrices and operations over them, for use during complex machine learning and linear algebra computations. We showed experimentally that the implementations chosen by our optimization framework meet or exceed those chosen by human experts, and that a system running such implementations can beat state-of-the-art systems.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their insightful comments on earlier versions of the paper. Work presented in this paper has been supported by an NIH CTSA, award no. UL1TR003167, and by the NSF under grant nos. 1918651, 1910803, and 2008240.

REFERENCES

[1] 2017. PyTorch. http://pytorch.org. Accessed Sep 1, 2018.

[2] 2021. SystemDS. https://systemds.apache.org/. Accessed Feb 1, 2021.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI 16*. USENIX Association, GA, 265–283.

[4] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. *ACM Sigplan Notices* 44, 6 (2009), 38–49.

[5] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive Re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/1066157.1066171

[6] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 253–260.

[7] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[8] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[9] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J Haas, and Christopher Jermaine. 2013. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD 2013*. ACM, 637–648.

[10] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 2000. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 268–279.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).

[12] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OÕReilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. 379Ð390.

[13] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. 2016. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.

[14] Diego Fabregat-Traver and Paolo Bientinesi. 2013. Application-tailored linear algebra algorithms: A search-based approach. *The International journal of high performance computing applications* 27, 4 (2013), 426–439.

[15] Joseph Felsenstein. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of molecular evolution* 17, 6 (1981), 368–376.

[16] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. IEEE, 1381–1384.

[17] 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.

[18] F.A. Graybill. 1983. *Matrices with applications in statistics*. Wadsworth International Group.

[19] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[20] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: optimizing statistical data analysis in the cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1–12.

[21] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. 2015. Spartan: A distributed array framework with smart tiling. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 1–15.

[22] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (Denver, Colorado, USA) *(SIGMOD '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/115790.115835

[23] 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.

[24] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 2279–2288.

[25] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. *SIGMOD Rec.* 27, 2 (June 1998), 106–117. https://doi.org/10.1145/276305.276315

[26] David Kernert, Wolfgang Lehner, and Frank Köhler. 2016. Topology-aware optimization of big sparse matrices and matrix multiplications on main-memory systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 823–834.

[27] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2017. Scalable linear algebra on a relational database system. In *ICDE 2017*. IEEE, 523–534.

[28] 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.

[29] 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.

[30] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14, 9 (2011).

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[32] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[33] Johanna Sommer, Matthias Boehm, Alexandre V Evfimievski, Berthold Reinwald, and Peter J Haas. 2019. Mnc: Structure-exploiting sparsity estimation for matrix expressions. In *Proceedings of the 2019 International Conference on Management of Data*. 1607–1623.

[34] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 342–355.

[35] Michael J Voss and Rudolf Eigemann. 2001. High-level adaptive program optimization with ADAPT. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. 93–102.

[36] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[37] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[38] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[39] Lele Yu, Yingxia Shao, and Bin Cui. 2015. Exploiting matrix dependency for efficient distributed matrix computation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 93–105.

[40] Yongyang Yu, Mingjie Tang, Walid G Aref, Qutaibah M Malluhi, Mostafa M Abbas, and Mourad Ouzzani. 2017. In-memory distributed matrix computation processing and optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1047–1058.

[41] Jia Zou, R Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. PlinyCompute: A platform for high-performance, distributed, data-intensive tool development. In *Proceedings of the 2018 International Conference on Management of Data*. 1189–1204.