EinDecomp: Decomposition of Declaratively-Specified Machine Learning and Numerical Computations for Parallel Execution

Daniel Bourgeois
Rice University
dcb10@rice.edu

Zhimin Ding
Rice University
zd21@rice.edu

Dimitrije Jankov
Rice University
dimitrijejankov@gmail.com

Jiehui Li
Rice University
jl302@rice.edu

Mahmoud Sleem
Rice University
msm15@rice.edu

Yuxin Tang
Rice University
yuxin.tang@rice.edu

Jiawen Yao
Rice University
jy75@rice.edu

Xinyu Yao
Rice University
xy38@rice.edu

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

We consider the problem of automatic parallelism in high-performance, tensor-based systems. Our focus is on *intra-operator parallelism* for inference tasks on a single GPU server or CPU cluster, where each operator is automatically broken op so that it runs on multiple devices. We assert that tensor-based systems should offer a programming abstraction based on an *extended Einstein summation notation*, which is a fully declarative, mathematical specification for tensor computations. We show that any computation specified in the Einstein summation notation can be re-written into an equivalent *tensor-relational* computation that facilitates intra-operator parallelism, and this re-write generalizes existing notations of tensor parallelism such as "data parallel" and "model parallel." We consider the algorithmic problem of optimally computing a tensor-relational decomposition of a graph of operations specified in our extended Einstein summation notation.

PVLDB Reference Format:
Daniel Bourgeois, Zhimin Ding, Dimitrije Jankov, Jiehui Li, Mahmoud Sleem, Yuxin Tang, Jiawen Yao, Xinyu Yao, and Chris Jermaine. EinDecomp. PVLDB, 18(7): 2240 - 2253, 2025.
doi:10.14778/3734839.3734858

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dcbdan/einsummable.

1 INTRODUCTION

Automatically partitioning numerical computations over matrices or multi-dimensional arrays (often referred to as *tensors*) so that they can be run in parallel on multiple computers, cores, or compute devices such as GPUs is a key problem in modern computing.

There are two types of parallelism available to tensor-based systems. In *intra-operator* parallelism [3, 31, 46], an operator (such as a matrix multiply) is decomposed so that different sites or compute devices can work on different parts of the operator at the same time. In *inter-operator* parallelism [19, 20, 33] (typically realized as *pipeline parallelism* in tensor and database systems) different operators are placed on different sites/devices and as the computation progresses, intermediate results are moved from site to site.

Both types of parallelism can increase *throughput*—the amount of data processed per time period—because they facilitate the use of additional compute resources. However, in many modern AI models, only intra-operator parallelism can reduce *latency*—the time to process a given amount of data—as parallelizing a given operator can mean that it takes less time to execute.[1] Thus, for user-facing tasks (such as AI inference, when a human is waiting for the results) or other, latency-critical tasks, intra-operator parallelism is particularly important. It is inter-operator parallelism that we are concerned with in this paper.

Effectively applying intra-operator parallelism is very challenging due to the imbalance between modern compute and network speeds. For example, consider the multiplication of two n by n matrices. The optimal 3D algorithm for parallelizing multiplication of square matrices on d devices [3] requires transferring $n^2 \times 3d^{\frac{1}{3}}$ floating point numbers.[2] Using eight H100 GPUs, the lower bound on compute time required to multiply two $10^4 \times 10^4$ matrices (half precision) using the 3D algorithm is around 0.000125 seconds. In contrast, the time required to transfer the $10^8 \times 6$ floating point numbers required to perform the computation using a super high-performance 3.2 terabit Infiniband connection is 0.003 seconds—nearly 30× the required compute time. Thus, even the fastest modern Infiniband connection between machines make it difficult to facilitate latency reduction using cross-machine parallelism in GPU clusters. GPU-to-GPU interconnects on the same server are faster[3] but point remains: modern compute devices are so fast that realizing latency reduction via inta-operator parallelism is challenging.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734858

[1] For example, in most popular transformers (such as LLaMA), the compute graph is "long and skinny" without inherent parallelism to exploit via pipelining, and intra-operator parallelism is necessary to decrease latency. Other models, such as MoEs [16], are designed with parallelism in mind and may be more amenable to pipelining.
[2] This considers the cost to shard and place the input matrices, and to aggregate intermediate results.
[3] NVLink provides for up to 7.2 terabits per second, bi-directionally, between any two GPUs on a server, for 112 terabits of connectivity on an 8-GPU server.

Automating intra-operator parallelism. Because realizing actual speedups from intra-operator parallelism is so difficult, it makes sense to develop intelligent algorithms that can automatically determine how to best decompose operators over tensors. Automation is attractive because in the general case, the design space for realizing intra-operator parallelism is huge, and made worse when the goal is not simply to decompose a single operator, but an entire, complex computation, such as inference in a modern transformer.

The classical forms of intra-operator parallelism in AI/ML are data parallelism [12, 18] and model or tensor parallelism [15, 37, 55]. In data parallelism a batch of data are sharded into sub-batches, and handled at different sites. In model parallelism, the model is partitioned and data are replicated. However, data parallelism and model parallelism are not the only options [22, 56]. The aforementioend 3D decomposition can be seen as form of joint data and model parallelism. Even the terms "data-" and "model-parallel" are problematic. Deep within an ML computation, it is unclear which dimensions are "data" dimensions and which are "model" dimensions.

Programming and intra-operator parallelism. With a few notable exceptions [8, 22, 32, 56], there has been relatively little work aimed at "hands-free" decomposition of tensor-based computations to enable fully-automated, intra-operator parallelism. The problem we consider in this paper is how to optimally and automatically decompose a computation consisting of a large number of tensor-based operators to reduce latency, via intra-operator parallelism. Because slow inter-server network connections make latency reduction via intra-operator parallelism for multi-server GPU-based computations challenging, our focus is on single-server GPU-based inference where a super-fast network (based on a technology such as NVLink) is available, or on multi-machine CPU clusters, where the slower processors make latency reduction from intra-operator parallelism easier to realize.

In this paper, we address two, closely-linked questions. First, what programming abstraction should systems for tensor-based computing offer to enable intra-operator parallelism? Second, given that abstraction, how should such systems automatically decompose an tensor-based computation? We believe that these two questions cannot be disentangled, because the selected programming abstraction needs to expose the semantics underlying the array-based computation to the system, or else the system cannot understand how to properly decompose the computation. While the programming abstraction can be hidden beneath a PyTorch-like API for programmers familiar with the current API, the current state of affairs, where operations over tensors (such as matrix multiplication) are black-box operations whose semantics are opaque to the system, cannot easily facilitate fully automated intra-operator parallelism. Systems such as PyTorch have massive APIs, and so any ad-hoc approach that does not seek to provide a unified abstraction for specifying the semantics of computations over tensors is not likely to be practical. As the developers of PyTorch state, "Writing a backend for PyTorch is challenging. PyTorch has 1200+ operators, and 2000+ if you consider various overloads for each operator [36]."

Our contributions. This paper has three main contributions.

First, we argue that tensor-based systems should offer a programming abstraction based on an *extended Einstein summation notation* [14]. Einstein summation notation is a fully declarative, mathematical specification for tensor computations, that is common in physics and is already supported in at least some fashion by both PyTorch and TensorFlow, and so it already has some buy-in. Further, it is simple and easy to understand. If one views a tensor as a relation—tensors can be seen as relations mapping keys (lists of integers that index into the tensor) to values (some sort of scalar value)—then Einstein summation notation is closely related to other relational programming languages such as SQL, as it specifies a join followed by an aggregation over the input tensors [5]. We call our expended Einstein summation notation `EinSum`.

Second, we show that any computation expressed in `EinSum` can be re-written into an equivalent *tensor-relational* computation [54]. A tensor-relational computation is a relational computation that operates not over relations mapping keys to scalars, but instead operates over relations mapping keys to tensors. This equivalence is crucial because while a classical relational system operating over scalars is never going to be competitive with PyTorch or TensorFlow, tensor-relational computations are amenable to a high-performance implementation on top of a tensor-based runtime that, in theory could be implemented on top of almost any existing system for tensor computations. A tensor-relational computation pushes tensors, not scalars, through a runtime, and operates over those tensors using high-performance *kernel functions* [9, 26, 51] that are carefully optimized to make full use of the CPU/GPU hardware.

Re-writing an operation specified in `EinSum` into an equivalent, tensor-relational computation facilitates intra-operator parallelism, as it effectively decomposes the operation into a set of kernel invocations. However, the space of possible decompositions is large, and further complicated when the input computation consists of many operations. The decompositions cannot be considered independently, as decomposing an operation to support intra-operator parallelism implicitly produces a decomposition of the output tensor that may be incompatible with the next operation.

Thus, our third contribution is to consider the problem of optimally choosing a tensor-relational decomposition for a directed, acyclic graph of `EinSum` operations. We propose an algorithm, called `EinDecomp`, that does the decomposition so as to minimize the amount of communication between kernel calls in the resulting tensor-relational computation, while ensuring that there is enough work to keep all devices (CPU cores or GPUs) busy.

An extensive set of experiments on CPUs and GPUs shows the value of the `EinDecomp` approach.

2 PAPER ROADMAP

We begin by describing the `EinSum` language through a number of examples, including how it can be used to succinctly specify multi-headed attention [52] (Section 3). We then define the notion of a *tensor relation*, which is a relation that can be used to implement a tensor as a set of keyed sub-tensors. We describe a simple relational algebra over tensor relations called the *tensor relational algebra* (TRA) that can be implemented on top of an existing runtime, like PyTorch, or as a special-purpose TRA runtime. We then describe how any `EinSum` expression can be re-written into a computation in the TRA (Section 4). The amount and exact nature of the parallelism available in the resulting TRA computation is controlled by a *partitioning vector*. Thus, given a complex computation (such

as a large language model) specified as a graph of EinSum operations, the problem of decomposing it into a TRA computation is reduced to the problem of associating a partitioning vector with every computation in the graph (Section 5). We develop a cost model for executing such a TRA computation (Section 7), as well as a dynamic programming algorithm called EinDecomp that chooses the set of partitioning vectors to minimize that cost (Section 8).

3 EINSUM BACKGROUND AND EXAMPLES

We introduce the EinSum tensor operator by generalizing from matrix multiplication. First, we define the notion of *tensor*. We use bold upper-case (for example, $\mathbf{U}$) to denote a tensor. Define the *bound* vector for $\mathbf{U}$, $\mathbf{b}_\mathbf{U}$ to be a vector of integers of length r. "r" stands for "rank;" matrices are rank-2 tensors. Next, define $\mathcal{I}(\mathbf{b}_\mathbf{U})$ to be the set $\{0...\mathbf{b}_\mathbf{U}[0]-1\}\times\{0...\mathbf{b}_\mathbf{U}[1]-1\}\times...\times\{0...\mathbf{b}_\mathbf{U}[r-1]-1\}$. This is the set of all indices or keys that obey the bound. A tensor $\mathbf{U}$ is then a function from $\mathcal{I}(\mathbf{b}_\mathbf{U})$ to the set of real numbers.

Simple EinSum examples. We start with the classic example: matrix multiplication. Let $\mathbf{X}$ and $\mathbf{Y}$ be matrices with bounds $[100, 200]$ and $[200, 50]$, respectively. Then matrix multiplication is written as: $\forall\, i, k \in \mathcal{I}([100, 50])$:

$$\mathbf{Z}_{i,k} \leftarrow \sum_{j \in \mathcal{I}([200])} \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k} \tag{1}$$

For simplicity, we may drop the subscript on the aggregation operation, as it is implied; any indices that appear in the input tensors that do not appear in the output tensor must be aggregated. We also typically drop the range specification for the output labels, as this is implied by the bound vector for the output tensor.

If instead of computing matrix multiply, we wanted to compute the squared L^2 distance between each row of $\mathbf{X}$ and each column of $\mathbf{Y}$, then we can replace the scalar multiplication $x \times y$ with $(x-y)^2$:

$$\mathbf{Z}_{i,k} \leftarrow \sum (\mathbf{X}_{i,j} - \mathbf{Y}_{j,k})^2.$$

To compute the L^∞ distance instead, (i) replace $x \times y$ with $|x-y|$ and (ii) replace the summation with maximization:

$$\mathbf{Z}_{i,k} \leftarrow \max |\mathbf{X}_{i,j} - \mathbf{Y}_{j,k}|.$$

If one views the tensors as relations mapping keys to real values, binary EinSum expressions perform a join of the two tensors to link values, followed by the application of a scalar function (multiplication, squared difference, etc.), followed by an aggregation. It is possible to have unary EinSum expressions where the join is replaced by a simple map operation that only applies a scalar function. The aggregation operator must be associative and commutative.

General form of EinSum. In full generality, EinSum applies to all rank r tensors, not just matrices. The general form involves more notation to specify indexing; the notation will be necessary to precisely describe decompositions of general EinSum expressions.

A *label* is some symbol that can be bound to a value. We use $\ell_\mathbf{U}$ to denote a list (vector) of labels used to index into tensor $\mathbf{U}$. Viewed relationally, a tensor with label vector $\ell_\mathbf{U}$ is equivalent to a database relation with schema $\mathbf{U}(\ell_\mathbf{U}[1], \ell_\mathbf{U}[2], ..., \texttt{val})$. When we "bind" $\ell_\mathbf{U}$, we are specifying specific key values we are interested in selecting for.

Often, we will need to project or permute a bound vector. Given two lists of labels ℓ_1 and ℓ_2, and a bound vector $\mathbf{b}$, define $\mathbf{b}[\ell_1; \ell_2]$ to be a vector of length $|\ell_1|$, where the ith entry is $\mathbf{b}[j]$ iff $\ell_1[i] = \ell_2[j]$. As an example, let $\mathbf{b} = [2, 3, 4]$ and let $\ell_1 = [k, i]$ and $\ell_2 = [i, j, k]$. Then $\mathbf{b}[\ell_1; \ell_2] = [4, 2]$.

Given this, binary Einsum expressions take the general form:

$$\forall\, \ell_\mathbf{Z} \in \mathcal{I}(\mathbf{b}_\mathbf{Z}) : \mathbf{Z}_{\ell_\mathbf{Z}} \leftarrow \bigoplus_{\ell_{\text{agg}} \in \mathcal{I}(\mathbf{b}_{\mathbf{XY}}[\ell_{\text{agg}}; \ell_{\mathbf{XY}}])} \bigotimes (\mathbf{X}_{\ell_\mathbf{X}}, \mathbf{Y}_{\ell_\mathbf{Y}}) \tag{2}$$

Here, $\bigoplus$ is the aggregation operator and $\bigotimes$ is the scalar function applied to joined values (EinSum is an *extended* Einstein summation notation as it allows for arbitrary $\bigoplus$ and $\bigotimes$ operations). In the above expression, to denote the concatenation of two label lists $\ell_\mathbf{X}$ and $\ell_\mathbf{Y}$, we use $\ell_{\mathbf{XY}}$. $\mathbf{b}_{\mathbf{XY}}$ similarly denotes the concatenation of two bound vectors.

Consider a more complicated EinSum expression over tensors. Assume that we have two tensors $\mathbf{X}$ and $\mathbf{Y}$ with bound vectors $\mathbf{b}_\mathbf{X} = [10, 100, 20]$ and $\mathbf{b}_\mathbf{Y} = [100, 20, 2000]$. We wish to transpose $\mathbf{X}$ to obtain a tensor with bound $[20, 10, 100]$, then transpose $\mathbf{Y}$ to obtain a new tensor with bound $[20, 100, 2000]$, and do a batch matrix multiply [2] of the two resulting tensors, and then sum out the batch dimension.

In EinSum, this is expressed in the single expression:

$$\forall i, k \in ([10, 2000]), \mathbf{Z}_{i,k} \leftarrow \sum_{b,j \in \mathcal{I}([20,100])} \mathbf{X}_{i,j,b} \times \mathbf{Y}_{j,b,k}$$

Considering the general form, we have $\ell_\mathbf{X} = [i, j, b]$, $\ell_\mathbf{Y} = [j, b, k]$, $\ell_{\text{agg}} = [b, j]$ and $\mathbf{b}_{\mathbf{XY}} = [10, 100, 20, 100, 20, 2000]$. The bound vector for the aggregation is computed as $\mathbf{b}_{\mathbf{XY}}[\ell_{\text{agg}}; \ell_{\mathbf{XY}}]$. How? ℓ_{agg} has two labels: b and j. As b occupies the third (and fifth) position in $\ell_{\mathbf{XY}}$, and j occupies the second (and fourth) position in $\ell_{\mathbf{XY}}$, we select the third (or fifth) item in $\mathbf{b}_{\mathbf{XY}}$, and the second (or fourth) item results in $\mathbf{b}_{\mathbf{XY}}$. This results in the bound vector $[20, 100]$.

When the aggregation operator is summation and the join function is multiplication, then the EinSum is often referred to as a *contraction*. Contractions include matrix multiplication and tend to be the most computationlly challenging EinSum expressions. The labels that appear in inputs but not outputs are ℓ_{agg}. If ℓ_{agg} is empty (meaning there are no indices being summed out), then the EinSum is often referred to as *element-wise* and the aggregation operator may be omitted. If $\ell_\mathbf{Z}$ contains labels not found in either $\ell_\mathbf{X}$ or $\ell_\mathbf{Y}$, then the EinSum is often referred to as a *broadcast*, as entries are being replicated across one or more dimensions. In the remainder of the paper, we ignore broadcasts (so $\mathbf{b}_\mathbf{Z} = \mathbf{b}_{\mathbf{XY}}[\ell_\mathbf{Z}; \ell_{\mathbf{XY}}]$) and focus on contractions. We assume no repeated labels in $\ell_\mathbf{X}$ or in $\ell_\mathbf{Y}$, but labels are often repeated across the two sets.

Multi-headed attention via EinSum. EinSum can be used to specify most modern ML computations. For an informatative examaple, we use EinSum expressions to specify multi-headed attention [52] used by large language models. Multi-headed attention runs the attention mechanism several times in parallel, and the attention mechanism includes a softmax term. Working backwards, we first build softmax in EinSum.

For a matrix $\mathbf{X}$, softmax can be expressed in the following EinSum:

$$\mathbf{C}_i \leftarrow \max \mathbf{X}_{i,j} \quad \mathbf{E}_{i,j} \leftarrow e^{\mathbf{X}_{i,j} - \mathbf{C}_i} \quad \mathbf{S}_i \leftarrow \sum \mathbf{E}_{i,j} \quad \mathbf{Y}_{i,j} \leftarrow \frac{\mathbf{E}_{i,j}}{\mathbf{S}_i}$$

As the extensions to $r > 2$ and $r = 1$ are straightforward, we assume there is an EinSum softmax macro that accepts any non-scalar tensor.

The attention mechanism, applied to "query," "key" and "value" matrices $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, is given by $\text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$, where d_k is the number of columns of $\mathbf{K}$. This can be expressed as the following EinSum:

$$\mathbf{T}^{(1)}_{i,k} \leftarrow \sum \mathbf{Q}_{i,j} \times \mathbf{K}_{k,j} \quad \mathbf{T}^{(2)}_{i,k} \leftarrow \frac{1}{\sqrt{d_k}}\mathbf{T}^{(1)}_{i,k}$$

$$\mathbf{T}^{(3)} \leftarrow \text{softmax}(\mathbf{T}^{(2)}) \quad \mathbf{Y}_{i,k} \leftarrow \sum \mathbf{T}^{(3)}_{i,j} \times \mathbf{V}_{j,k}$$

Multi-headed attention, applied again to query, key and value matrices, is typically presented as follows:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\mathbf{H}_1, ...\mathbf{H}_h]\mathbf{W}_O$$

$$\text{where } \mathbf{H}_i = \text{Attention}(\mathbf{Q}\mathbf{W}^Q_i, \mathbf{K}\mathbf{W}^K_i\mathbf{V}\mathbf{W}^V_i)$$

Here, the input matrices are linearly projected once for each "head" according to weight matrices. Then, the result of each attention head is concatenated together and linearly projected to the final output space.

In the following EinSum, the label "h" stands for "head", "s" for "sequence" and "a" for "attribute." First, the query, key and value matrices are linearly projected across the head dimension:

$$\mathbf{Q}^H_{s,h,d} \leftarrow \sum \mathbf{Q}_{s,a} \times \mathbf{W}^Q_{a,h,d}$$
$$\mathbf{K}^H_{s,h,d} \leftarrow \sum \mathbf{K}_{s,a} \times \mathbf{W}^K_{a,h,d}$$
$$\mathbf{V}^H_{s,h,d} \leftarrow \sum \mathbf{V}_{s,a} \times \mathbf{W}^V_{a,h,d}$$

The attention computation is "parallelized" by batching across the head dimension:

$$\mathbf{T}^{(1)}_{h,s,s'} \leftarrow \sum \mathbf{Q}^H_{s,h,d} \times \mathbf{K}^H_{s',h,d} \quad \mathbf{T}^{(2)}_{h,s,s'} \leftarrow \frac{1}{\sqrt{d_k}}\mathbf{T}^{(1)}_{h,s,s'}$$

$$\mathbf{T}^{(3)} \leftarrow \text{softmax}(\mathbf{T}^{(2)}) \quad \mathbf{O}_{s,h,d} \leftarrow \sum \mathbf{T}^{(3)}_{h,s,s'} \times \mathbf{V}^H_{s',h,d}$$

Lastly, the linear output projection is applied:

$$\mathbf{Y}_{s,a} = \sum \mathbf{O}_{s,h,d} \times \mathbf{W}^O_{a,h,d}$$

Note that in the EinSum formulation, $\mathbf{W}^O$ is not a matrix as in the standard defintion but a rank-3 tensor. This is still equivalent as the contraction producing $\mathbf{Y}$ is equivalent to first concatenating the aggregation dimensions h and d on inputs $\mathbf{O}$ and $\mathbf{W}$ and then doing matrix multiply.

4 RE-WRITING EINSUM TO TRA

In this section, we describe how any computation expressed in the EinSum can be transformed into a computation in the tensor-relational algebra (TRA) [54]. The TRA is a simple implementation abstraction that can be implemented on top of any appropriate tensor-based back-end. Like relational algebra, TRA is trivially parallelizable, and this rewrite into TRA allows for parallelization of EinSum expressions. Translating EinSum into TRA so it can be executed on a tensor backend is analogous to translating SQL into relational algebra so it can be implemented on a database backend.

4.1 Tensor Relations

The TRA operates over *tensor relations*. A tensor relation may be viewed as a set of pairs of the form

$$(\texttt{key}, \texttt{tensor})$$

Mathematically, it is a function mapping keys to tensors. Like a tensor, a tensor relation $\mathcal{R}$ has a bound vector $\mathbf{b}_\mathcal{R}$, but it also has a *partitioning vector* $\mathbf{d}_\mathcal{R}$. The tensor relation is then a function

$$\mathcal{R} : \mathcal{I}\left(\mathbf{d}_\mathcal{R}\right) \rightarrow \left(\mathcal{I}\left(\frac{\mathbf{b}_\mathcal{R}}{\mathbf{d}_\mathcal{R}}\right) \rightarrow \mathbb{R}\right)$$

Here, the division $\frac{\mathbf{b}_\mathbf{R}}{\mathbf{d}_\mathbf{R}}$ operates element-wise over the vectors. Thus, we can evaluate $\mathcal{R}$ at any value $\mathbf{i} \in \mathcal{I}\left(\mathbf{d}_\mathcal{R}\right)$ and obtain a tensor. We use $\mathcal{R}^\mathbf{i}$ to denote this evaluation. For a tensor $\mathbf{R}$, we use $\mathbf{R}_\mathbf{j}$ to denote the real number obtained when evaluating $\mathbf{R}$ at $\mathbf{j}$. When we index a tensor relation to select a particular sub-tensor, and then we index into the sub-tensor to select a scalar, we write $\mathcal{R}^\mathbf{i}_\mathbf{j}$.

We say that a tensor $\mathbf{R}$ and a tensor relation $\mathcal{R}$ having the same bound are *equivalent*, denoted $\mathbf{R} \equiv \mathcal{R}$, if, for all vectors $\mathbf{j} \in \mathcal{I}\left(\mathbf{d}_\mathbf{R}\right)$:

$$\mathbf{R}_\mathbf{j} = \mathcal{R}^{\frac{\mathbf{j}}{\mathbf{d}_\mathcal{R}}}_{\mathbf{j} \bmod \mathbf{d}_\mathcal{R}}$$

Again, in the above definition we assume the "mod" operation operates element-wise over the input vectors. $\mathbf{R} \equiv \mathcal{R}$ implies that the tensor and tensor relation are alternative implementations for the same function. Intuitively, we say that $\mathbf{R} \equiv \mathcal{R}$ when $\mathcal{R}$ stores $\mathbf{R}$, broken into a set of sub-tensors.

This may seem quite abstract; consider the matrix $\mathbf{U}$:

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

As this is a 4×4 matrix, the bound vector $\mathbf{b}_\mathbf{U} = [4, 4]$. Now, let us imagine that we instead wanted to represent $\mathbf{U}$ as a tensor relation $\mathcal{U}$ where $\mathbf{b}_\mathcal{U} = [4, 4]$ and $\mathbf{d}_\mathcal{U} = [4, 2]$. This $\mathbf{d}_\mathcal{U}$ implies that we slice the first dimension 4 ways, and the second two ways, so if $\mathbf{U} \equiv \mathcal{U}$, then

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

This can be viewed as a function from $\mathcal{I}([4, 2])$ to sub-tensors with bound vector $\frac{[4,4]}{[4,2]} = [1, 2]$.

If instead we let $\mathbf{d}_\mathcal{U} = [2, 2]$, then we slice both dimensions two ways, and so if $\mathbf{U} \equiv \mathcal{U}$, then

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

Effectively, to create a tensor relation $\mathcal{U}$ that is equivalent to $\mathbf{U}$, we simply slice up $\mathbf{U}$ according to the partitioning vector $\mathbf{d}_\mathcal{U}$, and represent $\mathbf{U}$ as a set of keyed sub-tensors.

4.2 The Tensor-Relational Algebra

The TRA is an algebra over tensor relations, that can serve as the implementation abstraction or interface that is exported by a high-performance runtime. It is closely related to the classic relational algebra. As we will show, the TRA can easily be used to implement EinSum. The three TRA operations we are concerned with are *join*, *aggregation*, and *repartition*.

Join. Given two tensor relations $\mathcal{X}$ and $\mathcal{Y}$, join applies a kernel function K to each pair of sub-tensors from the two inputs that "match". Assume that we have a tensor-valued function K accepting two sub-tensors having bounds $\frac{\mathbf{b}_\mathcal{X}}{\mathbf{d}_\mathcal{X}}$ and $\frac{\mathbf{b}_\mathcal{Y}}{\mathbf{d}_\mathcal{Y}}$. We have two label vectors $\ell_\mathcal{X}$ and $\ell_\mathcal{Y}$. Then $\bowtie_{K,\ell_\mathcal{X},\ell_\mathcal{Y}}(\mathcal{X},\mathcal{Y})$ first joins $\mathcal{X}$ and $\mathcal{Y}$, matching $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ iff $x.\texttt{key}[i] = y.\texttt{key}[i]$ whenever $\ell_\mathcal{X}[i] = \ell_\mathcal{Y}[j]$. When x and y match, a new tuple t is included in the output. $t.\texttt{key}$ is $x.\texttt{key}$ concatenated with $y.\texttt{key}$ (with redundant dimensions that are trivially equal, per the join predicate, removed, as in a natural join). $t.\texttt{tensor}$ is $K(x.\texttt{tensor}, y.\texttt{tensor})$.

Aggregation. Given a tensor relation $\mathcal{X}$, aggregation performs a grouping of the tensor relation and then applies a commutative and associative kernel function $\oplus$ to reduce the values in each group to a single value. Assume that we have a tensor-valued function $\oplus$ accepting two tensors both having bounds $\frac{b_\mathcal{X}}{d_\mathcal{X}}$. Assume we have two label vectors $\ell_\mathcal{X}$ and ℓ_{agg}. $\sum_{\oplus,\ell_\mathcal{X},\ell_{\text{agg}}}(\mathcal{X})$ first partitions $\mathcal{X}$ so two tuples x_1 and x_2 are in the same partition iff $x_1.\texttt{key}[i] = x_2.\texttt{key}[i]$ whenever $\ell_\mathcal{X}[i]$ is not in ℓ_{agg}. For each partition, produce an output tuple t by picking a tuple x from the partition, and setting $t.\texttt{key}[i] = x.\texttt{key}[j]$ whenever $\ell_{\text{agg}}[i] = \ell_\mathcal{X}[j]$. $t.\texttt{tensor}$ is produced by reducing all of the tensor values in the partition using $\oplus$. Note that if $\ell_\mathcal{X}$ and ℓ_{agg} are identical, the aggregation is the identity operation because there are no labels in $\ell_\mathcal{X}$ not in ℓ_{agg}.

Repartition. Given a tensor relation $\mathcal{X}$, let $\mathbf{X} \equiv \mathcal{X}$. Then $\Pi_\mathbf{d}(\mathcal{X})$ produces the tensor relation $\mathcal{X}'$ such that $\mathbf{d}_{\mathcal{X}'} = \mathbf{d}$ and $\mathbf{X} \equiv \mathcal{X}'$.

4.3 EinSum as a Tensor-Relational Language

We now show that any binary EinSum expression can be converted to an equivalent tensor-relational computation. The benefit is that such a rewrite allows EinSum expressions to be run in parallel, on a set of devices. This is done via the introduction of a partition vector that "explodes" the EinSum computation into a TRA expression over tensor relations, subsuming traditional notions such as "data parallel" and "model parallel."

The rewrite relies on the idea of "cloning" each list of labels in a label list such as $\ell_\mathbf{U}$ to produce a new label list $\bar{\ell}_\mathbf{U}$. The original list of labels iterates through the tuples in the tensor relation, and then the cloned list iterates through the entries in a tensor stored in the tensor relation. For example, we may take the matrix multiplication from Equation 3 and rewrite it as $\forall\, i,k \in \mathcal{I}([2,2]), \forall\, \bar{i},\bar{k} \in \mathcal{I}([50,25])$:

$$\mathbf{Z}_{\substack{i\times 50+\bar{i},\\ k\times 25+\bar{k}}} \leftarrow \sum_{\substack{j\in\mathcal{I}([2]),\\ \bar{j}\in\mathcal{I}([100])}} \mathbf{X}_{\substack{i\times 50+\bar{i},\\ j\times 100+\bar{j}}} \times \mathbf{Y}_{\substack{j\times 100+\bar{j},\\ k\times 25+\bar{k}}},$$

We can generalize this idea to any EinSum expression of the form given as Equation 3. Assume we are given a partition vector $\mathbf{d}$. We

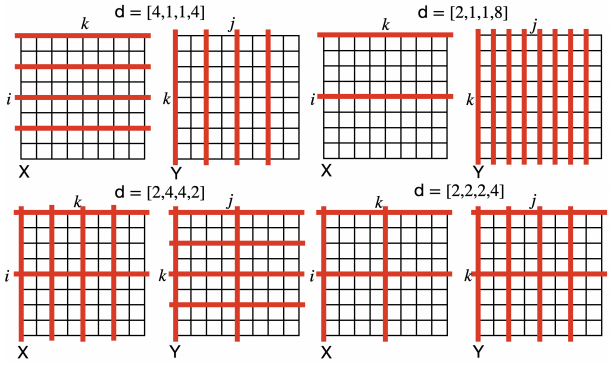

Figure 1: Four tensor-relational partitionings for $\mathbf{Z}_{i,k} \leftarrow \sum \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k}$. In each there are 16 kernel calls.

can now rewrite Equation 3 as:

$$\forall \ell_\mathbf{Z} \in \mathcal{I}\left(\mathbf{d}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right), \bar{\ell}_\mathbf{Z} \in \mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right) : \mathbf{Z}_{\ell_\mathbf{Z}\times\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]+\bar{\ell}_\mathbf{Z}}$$
$$\leftarrow \bigoplus_{\substack{\ell_{\text{agg}}\in\mathcal{I}\left(\mathbf{d}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)\\ \bar{\ell}_{\text{agg}}\in\mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)}} \bigotimes \left(\mathbf{X}_{\ell_\mathbf{X}\times\mathbf{d}\left[\ell_\mathbf{X},\ell_\mathbf{XY}\right]+\bar{\ell}_\mathbf{X}}, \mathbf{Y}_{\ell_\mathbf{Y}\times\mathbf{d}\left[\ell_\mathbf{Y};\ell_\mathbf{XY}\right]+\bar{\ell}_\mathbf{Y}}\right) \quad (3)$$

Now, assume that we have two tensor relations $\mathcal{X} \equiv \mathbf{X}$, and $\mathcal{Y} \equiv \mathbf{X}$, where $\mathbf{d}_\mathcal{X} = \mathbf{d}[\ell_\mathbf{X};\ell_\mathbf{XY}]$ and $\mathbf{d}_\mathcal{Y} = \mathbf{d}[\ell_\mathbf{Y};\ell_\mathbf{XY}]$. We can rewrite Equation 3 to compute a tensor relation $\mathcal{Z} \equiv \mathbf{Z}$ with $\mathbf{d}_\mathcal{Z} = \mathbf{d}[\ell_\mathbf{Z};\ell_\mathbf{XY}]$ as follows:

$$\forall \ell_\mathbf{Z} \in \mathcal{I}\left(\mathbf{d}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right), \bar{\ell}_\mathbf{Z} \in \mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right) : \mathcal{Z}^{\ell_\mathbf{Z}}_{\bar{\ell}_\mathbf{Z}}$$
$$\leftarrow \bigoplus_{\substack{\ell_{\text{agg}}\in\mathcal{I}\left(\mathbf{d}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)\\ \bar{\ell}_{\text{agg}}\in\mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)}} \bigotimes \left(\mathcal{X}^{\ell_\mathbf{X}}_{\bar{\ell}_\mathbf{X}}, \mathcal{Y}^{\ell_\mathbf{Y}}_{\bar{\ell}_\mathbf{Y}}\right) \quad (4)$$

Note that we effectively have a nested EinSum expression: the outer one is operating over tensor relations, and the inner one is operating over tensors in those two tensor relations.

Now, imagine that we have a *kernel function* K that accepts two sub-tensors $\mathcal{X}^{\ell_\mathbf{X}}$ and $\mathcal{Y}^{\ell_\mathbf{Y}}$ and creates a tensor $\mathbf{Z}'$ with bound vector $\frac{\mathbf{b_{XY}}}{\mathbf{d}}$ such that $\forall \bar{\ell}_\mathbf{Z} \in \mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right), \mathbf{Z}'_{\bar{\ell}_\mathbf{Z}}$ is set to $\bigoplus_{\bar{\ell}_{\text{agg}}\in\mathcal{I}\left(\frac{\mathbf{b_{XY}}}{\mathbf{d}}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)} \bigotimes \left(\mathcal{X}^{\ell_\mathbf{X}}_{\bar{\ell}_\mathbf{X}}, \mathcal{Y}^{\ell_\mathbf{Y}}_{\bar{\ell}_\mathbf{Y}}\right)$. If $\bigoplus$ over two tensors performs the $\bigoplus$ operation element-wise over entries in the tensors, then, using this kernel, Equation 4 becomes:

$$\forall \ell_\mathbf{Z} \in \mathcal{I}\left(\mathbf{d}\left[\ell_\mathbf{Z};\ell_\mathbf{XY}\right]\right) : \mathcal{Z}^{\ell_\mathbf{Z}} \leftarrow \bigoplus_{\ell_{\text{agg}}\in\mathcal{I}\left(\mathbf{d}\left[\ell_{\text{agg}};\ell_\mathbf{XY}\right]\right)} K\left(\mathcal{X}^{\ell_\mathbf{X}}, \mathcal{Y}^{\ell_\mathbf{Y}}\right) \quad (5)$$

The above equating is then implemented in the TRA by the two steps; a join to link tuples from $\mathcal{X}$ and $\mathcal{X}$ using the label lists $\ell_\mathcal{X}$ and $\ell_\mathcal{Y}$ followed by an aggregation using $\oplus$:

$$\texttt{temp} \leftarrow \bowtie_{K,\ell_\mathcal{X},\ell_\mathcal{Y}}(\mathcal{X},\mathcal{Y}) \quad \texttt{res} \leftarrow \sum_{\oplus,\ell_\mathcal{X}\odot\ell_\mathcal{Y},\ell_{\text{agg}}} (\texttt{temp})$$

$\odot$ concatenates the two label lists, removing any duplicate labels in the process (as in a classic natural join).

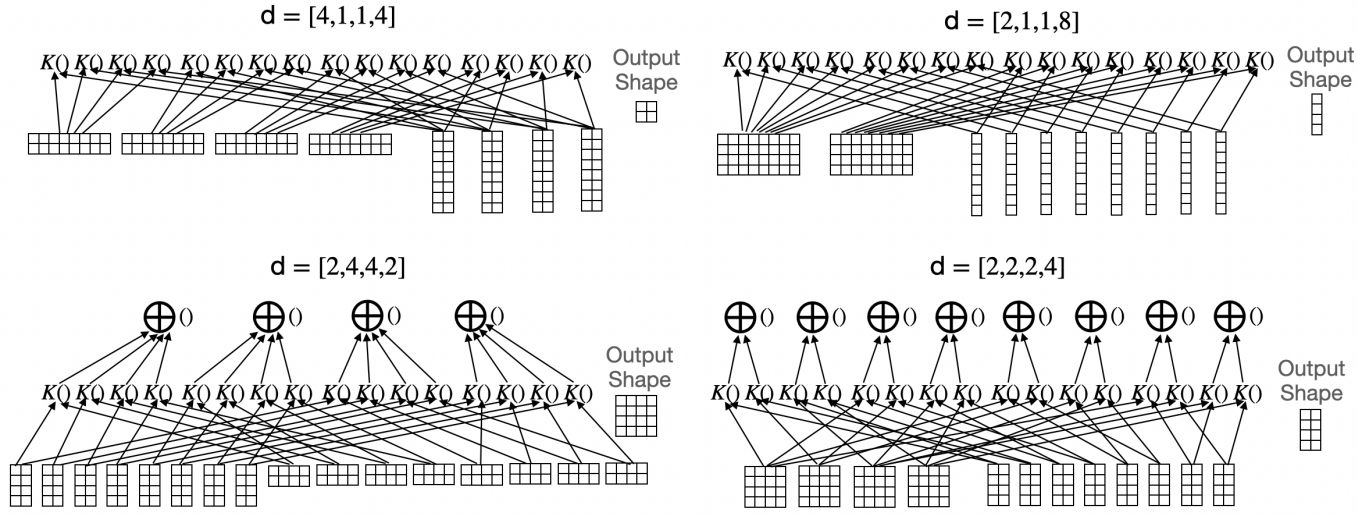

Figure 2: Dataflow graphs associated with the partitionings of Figure 1 For partitionings $\mathbf{d} = [4, 1, 1, 4]$ and $\mathbf{d} = [2, 1, 1, 8]$, there is only a join layer, as the joined dimensions are not partitioned. For $\mathbf{d} = [2, 4, 4, 2]$ and $\mathbf{d} = [2, 2, 2, 4]$ there is also an aggregation.

4.4 Parallelism via the Partitioning Vector

Thus, any binary EinSum expression can be re-written as an equivalent computation in the TRA, that can then be implemented by a TRA runtime. Crucially, the way that the EinSum expression is decomposed into tensor relations is controlled by the partitioning vector **d**; choosing a different **d** produces different partitionings of the input tensors, "exploding" the computations in different ways.

For example, the matrix multiplication of two 8×8 matrices, specified via $\mathbf{Z}_{i,k} \leftarrow \sum \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k}$. The decomposition of the input matrices, as well as the resulting TRA computation, are fully specified with a **d** vector having four entries (because both inputs are rank two). Figure 1 depicts how four different **d** vectors specify four different decompositions of the two input matrices. Figure 2 depicts the four associated dataflow (or lineage) graphs. These graphs show how the TRA computations associated with the decompositions shown in Figure 1 map tuples into kernel calls.

5 OPTIMIZING THE DECOMPOSITION

A complex computation specified in EinSum can be represented as a directed, acyclic graph called an EinGraph whose nodes are EinSum expressions and whose edges represent data flow. For each vertex, we have a triple:

$$(\texttt{bound}, \texttt{EinSum}, \texttt{inputs})$$

EinSum is the code run at the vertex, bound is the bound vector **b** for the output of the EinSum, and inputs lists vertices providing inputs. Note that inputs has an explicit ordering, as EinSum need not be commutative. inputs is empty if and only if EinSum empty; then the tensor is input to the computation. Otherwise, the bound can be deduced from the EinSum labels and the input tensor shapes.

As described in the previous section, an EinGraph can be executed by a TRA engine. Once a partition vector describing each EinSum operation is to be parallelized has been associated with each vertex in the graph, each vertex is executed as a join followed by an aggregation, with (optionally) a repartition required in between operations if the output partitioning of an operation does not match the required input partitioning of the next operation.

A key question is: How to associate a partitioning vector with each operation in the EinGraph, as this can have a radical impact on system performance? Thus, we now consider: Given an EinGraph, how do we annotate the EinGraph with partitioning vectors (see Figure 3) to describe how the inputs each vertex in the EinGraph are to be partitioned, to produce the best partitioning?

There will be two key considerations when labeling the EinGraph with partition vectors. First, we want to ensure that there is enough parallel work to do. And second, we want to ensure that the decomposition we choose is low cost. The question of how much parallel work is associated with a decomposition, and how to cost a decomposition, are considered in the next two sections.

6 ENSURING ENOUGH PARALLEL WORK

Assume the underlying system has p "processors" (in practice these may be CPU cores, GPUs, or FPGAs). We want to ensure that the tensor relational implementation of each EinSum expression is decomposed to at least p independent calls to the kernel K. At the same time, we do not want *too many* independent kernel calls, as more kernel calls tend to induce more movement across processors. Thus, we attempt to decompose each EinSum expression into exactly p kernel calls.

How to ensure this? A partitioning vector **d** controls the decomposition of an EinSum computation. **d** partitions a tensor $\mathbf{d}[i]$ ways along the ith dimension. For a binary EinSum expression over tensors $\mathbf{Z}_{\ell_Z} \leftarrow \bigoplus \bigotimes (\mathbf{X}_{\ell_X}, \mathbf{Y}_{\ell_Y})$, **d** partitions **X** according to $\mathbf{d}[\ell_\mathbf{X}; \ell_\mathbf{XY}]$ and **Y** according to $\mathbf{d}[\ell_\mathbf{Y}; \ell_\mathbf{XY}]$. Note that the elements in **d** corresponding to the same label must be the same. For example, Figure 1 shows four possible partitioning vectors (and the corresponding partitionings) for a matrix multiply $\mathbf{Z}_{i,k} \leftarrow \sum \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k}$.

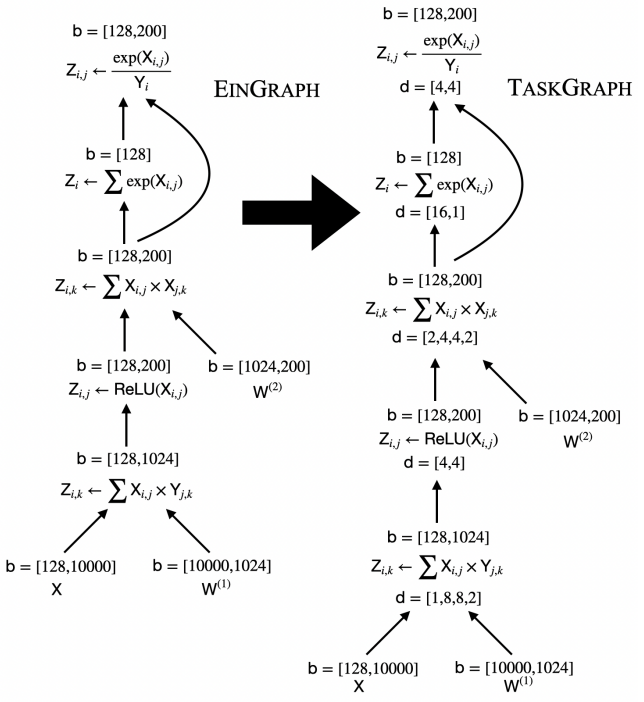

Figure 3: Modifying an EinGraph supplied by a programmer, to produce a TaskGraph, by adding bound vectors.

Here, valid partitionings will have $\mathbf{d}[1] = \mathbf{d}[2]$, corresponding to the shared j label.

When an `EinSum` expression is partitioned according to $\mathbf{d}$, the number of pairs of tuples matched during the join is $N(\ell_{\mathbf{X}}, \ell_{\mathbf{Y}}, \mathbf{d}) = \prod \mathbf{d}[\ell_{\mathbf{X}} \odot \ell_{\mathbf{Y}}; \ell_{\mathbf{XY}}]$. Recall that $\odot$ concatenates the two label lists, removing any duplicate labels in the process. What is the intuition behind this formula? Repeated labels correspond to an equality predicate associated with the join, and an equality predicate cuts down the number of tuples resulting from the join by a factor of d when the number of partitions along the corresponding dimension is d. So, in our matrix multiply example, $\mathbf{d} = [16, 2, 2, 4]$ would result in $16 \times 2 \times 4 = 128$ tuples being output from the join; as the second 2 is associated with a join predicate and does not contribute join results.

The series of dataflow graphs shown in Figure 2 depict how tuples are linked and kernel functions invoked, for four tensor-relational implementations of the matrix multiplication $\mathbf{Z}_{i,k} \leftarrow \sum \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k}$. Note that for each of these partition vectors, $N([i, j], [j, k], \mathbf{d}) = 16$.

7 COSTING A DECOMPOSITION

Our approach to costing a decomposition is to compute the number of floating point numbers that must be transferred to implement the resulting tensor relational computation, and to use that as the cost. Counting transfers makes sense as all decompositions will have the same total number of floating point operations.

Note that without knowing how all of the decomposed operations are to be placed onto compute devices, we cannot compute the exact number of numbers to be transferred (for example, if two tensors are to be added together, and they already sit on the same device, no transfer is required, but if they are on different devices, one must be transferred to the location where the other is located). However, we wish to perform this decomposition without worrying about implementation issues such as where computations are placed. To facilitate this, our tactic is to develop a cost model that assumes the worst case transfer cost for obtaining the input to each operation: the cost model assumes that every input to a node in the dataflow graph must be transferred to a different processor, where it is to be used.

By considering only the decomposition and not the placement, we can develop an algorithm that is optimal for a large class of graphs. This is similar to the way in which logical query optimization is often separated from physical query operation in database systems.

To execute a vertex in an EinGraph, there are three steps that incur data transfer, corresponding to the join, aggregation, and possible repartition of the output that are necessary to implement the vertex:

(1) Transferring sub-tensors to where two tuples are to be joined.
(2) Transferring sub-tensors resulting the join to the location where they are to be aggregated.
(3) Possibly re-partitioning the result of the aggregation to be used in subsequent EinGraph nodes.

Transferring into the join. Let $n_{\mathbf{X}}$ be the count of floating point numbers in each sub-tensor from $\mathbf{X}$. This is $\prod \frac{\mathbf{b}_{\mathbf{XY}}}{\mathbf{d}}[\ell_{\mathbf{X}}; \ell_{\mathbf{XY}}]$. Compute $n_{\mathbf{Y}}$ similarly. Then the number of floating point numbers that must be transferred into the join is $p \times (n_{\mathbf{X}} + n_{\mathbf{Y}})$, as each processor will receive one copy of a sub-tensor from the left, and from the right. Subsequently, we use $\text{cost}_{\text{join}}(\mathbf{d}, \ell_{\mathbf{X}}, \ell_{\mathbf{Y}}, \mathbf{b}_{\mathbf{XY}})$ to refer to this quantity.

For example, consider the top-left case in Figure 2, where $\ell_{\mathbf{X}} = [i, j]$, where $\ell_{\mathbf{Y}} = [j, k]$, and $\ell_{\mathbf{XY}} = [i, j, j, k]$. As $\mathbf{b}_{\mathbf{XY}} = [8, 8, 8, 8]$ and $\mathbf{d} = [4, 1, 1, 4]$, $\frac{\mathbf{b}_{\mathbf{XY}}}{\mathbf{d}} = [2, 8, 8, 2]$. Thus, $n_{\mathbf{X}} = 2 \times 8 = 16$, $n_{\mathbf{Y}} = 8 \times 2 = 16$, and thus $\text{cost}_{\text{join}}$ is $8 \times (16 + 16)$.

Transferring into the aggregation. Let $n_{\mathbf{Z}}$ be the number of floating point numbers in the sub-tensor resulting from each kernel call; this is $\prod \frac{\mathbf{b}_{\mathbf{XY}}}{\mathbf{d}}[\ell_{\mathbf{Z}}; \ell_{\mathbf{XY}}]$. Let n_{agg} be the number of sub-tensors that are aggregated down to a single sub-tensor; this is $\prod \mathbf{d}[\ell_{\text{agg}}; \ell_{\mathbf{XY}}]$. The total number of floating point numbers that must be transferred is then $\frac{p}{n_{\text{agg}}}\left(n_{\text{agg}} - 1\right) n_{\mathbf{Z}}$. There are $\frac{p}{n_{\text{agg}}}$ groups of tuples that must be aggregated. In the best case, the amount of data transferred for each group is $\left(n_{\text{agg}} - 1\right) n_{\mathbf{Z}}$, as all tuples in a group are sent to a single processor for aggregation—but if a processor that already has such a tuple is chosen as the aggregation site, no transfer happens. We use $\text{cost}_{\text{agg}}(\mathbf{d}, \ell_{\text{agg}}, \ell_{\mathbf{Z}}, \ell_{\mathbf{XY}}, \mathbf{b}_{\mathbf{XY}})$ to refer to this quantity.

For example, consider the top-left case in Figure 2. $\ell_{\mathbf{agg}} = [j]$ so $n_{\text{agg}} = 1$, so the aggregation cost is zero (there is no aggregation). Considering the bottom-right case where $\mathbf{d} = [2, 2, 2, 4]$, we have $n_{\text{agg}} = 2$. As $\ell_{\mathbf{Z}} = [i, k]$ we have $n_{\mathbf{Z}} = 2 \times 4 = 8$, and the total number of floating point numbers moved is $\frac{16}{2}(2 - 1)8 = 64$.

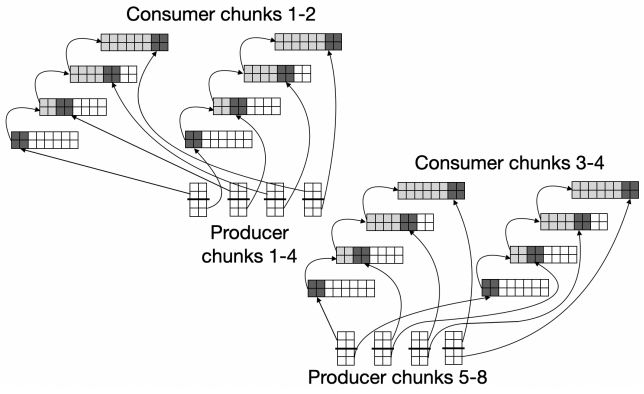

Figure 4: Modifying an EinGraph supplied by a programmer, to produce a TaskGraph, by adding bound vectors. This is done so as to minimize an upper bound on the communication required for the corresponding decomposition.

Re-partitioning across operations. If two operations are connected (one is a producer and one is a consumer) but their partitionings do not match, re-partitioning is necessary. To understand how we cost a re-partition, imagine that we have two matrix multiplies described by the `EinSum` expression $\mathbf{Z}_{i,k} \leftarrow \sum \mathbf{X}_{i,j} \times \mathbf{Y}_{j,k}$. The partitioning of the first (the "producer") is described by $\mathbf{d}^{(p)} = [2, 2, 2, 4]$ and the partitioning of the second (the "consumer") is described by $\mathbf{d}^{(c)} = [4, 1, 1, 4]$. Effectively, we are using the output of the computation in the lower left corner of Figure 2 as the left input to the computation in the upper right corner of Figure 2.

A graph showing the flow of data from the producer into the consumer is shown in Figure 4. Each producer sub-tensor must be sent to two locations—one location where its top half is used, and one location where its bottom half is used. As each producer sub-tensor has 8 floating point numbers, and there are 8 of them, the transfer cost is $2 \times 8 \times 8 = 128$. Further, as we build up the consumer sub-tensors, each consumer sub-tensor must be sent to subsequent three locations after it accepts the first producer sub-tensor: to the location where it accepts the second, then the third, and the final. Thus, the cost to move the input sub-tensors is $3 \times 16 \times 4 = 192$, for a total cost of 320.

In the general case, we have a producer producing a tensor with bound $\mathbf{b}_\mathbf{Z}$ and partitioning $\mathbf{d}_\mathbf{Z}$ and a consumer accepting a tensor with the same bound, and partitioning $\mathbf{d}_\mathbf{X}$. Let n_p be the number of floating point numbers in each producer sub-tensor, computed as $\prod \frac{\mathbf{b}_\mathbf{Z}}{\mathbf{d}_\mathbf{Z}}$. This is $4 \times 2 = 8$ in our example. Let n_c be the number of floating point numbers in each consumer sub-tensor; this is computed as $\prod \frac{\mathbf{b}_\mathbf{X}}{\mathbf{d}_\mathbf{Z}}$ ($2 \times 8 = 16$ in our example). Let n_int be the number of floating point numbers contributed by a producer sub-tensor to a consumer sub-tensor; this is $\prod \min\left(\frac{\mathbf{b}_\mathbf{Z}}{\mathbf{d}_\mathbf{Z}}, \frac{\mathbf{b}_\mathbf{X}}{\mathbf{d}_\mathbf{Z}}\right)$. Here, min() is computed element-wise. This is $2 \times 2 = 4$ in our example. And finally, let n be the number of floating point numbers in the output of the producer, or the input to the consumer (computed as $n = \prod \mathbf{d}_\mathbf{Z}$; this is $8 \times 8 = 64$ in our example). Then, the final cost is $\left(\frac{n_\text{c}}{n_\text{int}} - 1\right) \frac{n}{n_\text{c}} (n_\text{c} + n_\text{p})$, plus an additional transfer of $n_\text{p} \frac{n}{n_\text{c}}$ if $n_\text{p} \neq n_\text{int}$. This latter term corresponds to the cost to send each producer sub-tensor to the location where part of it is extracted to form the initial consumer sub-tensor. If the entire producer sub-tensor is used, then no such extraction is required. Subsequently, we use $\text{cost}_\text{repart}(\mathbf{d}_\mathbf{X}, \mathbf{d}_\mathbf{Z}, \mathbf{b}_\mathbf{Z})$ to refer to this computation.

8 THE `EINDECOMP` ALGORITHM

Given an EinGraph, we consider how to label all of the bounds to produce a TaskGraph (as in Figure 3).

8.1 Counting `EinSum` Partitionings

The core reason there exists a tractable solution is that the number of partitionings to consider for a given `EinSum` expression can be kept surprisingly small. We first start by assuming that the number of processors $p = 2^N$ for integer N and that every entry in $\mathbf{d}$ will be chosen to be a power of two. If the actual number of processors is not a power of two, p can be chosen to be larger than the number of available processors. Choosing p to be slightly larger than the number of processors physically available will increase the worst-case communication cost, but a quality assignment of operations to processors tends to alleviate this.

If D is the number of unique labels in $\ell_\mathbf{X}$ and $\ell_\mathbf{Y}$ then the number of possible values for $\mathbf{d}$ is $\frac{(N+D-1)!}{N!(D-1)!}$. Note that if two labels match across $\ell_\mathbf{X}$ and $\ell_\mathbf{Y}$ the corresponding dimensions are co-partitioned, and so they effectively count as one bucket; see Section 4. This often allows all possible partitionings for an `EinSum` expression to be enumerated, brute-force. For example, if $N = 10$ and $D = 6$, then the number of partitionings is 3003.

8.2 Dynamic Programming

In an EinGraph where there is not more than one consumer for any non-input vertex, there exists a relatively efficient dynamic programming algorithm for computing an optimal TaskGraph. This algorithm relies on a lookup table M that is a map from (vertex v, partitioning $\mathbf{d}_\mathbf{Z}$) pairs to the lowest (optimal) cost for computing the subgraph up to and including vertex v, subject to the constraint that the output partition for the vertex is $\mathbf{d}_\mathbf{Z}$.

Imagine that we have an EinGraph vertex v associated with matrix multiplication, which results in an output tensor with bound vector $[8, 8]$. If we require p = eight kernel calls in the implementation of v, the possible partitioning $\mathbf{d}$ vectors associated with implementing the `EinSum` for v are: $[2, 1, 1, 4]$; $[4, 1, 1, 2]$; $[8, 1, 1, 1]$; $[1, 1, 1, 8]$; $[2, 2, 2, 2]$; $[4, 2, 2, 1]$; $[1, 2, 2, 4]$; $[1, 8, 8, 1]$. Each of these partitioning vectors produces eight kernel calls since, after removing the repeated middle (join) index partitioning, the product over all entries is eight. Thus, the set of possible output partitionings $\mathbf{d}_\mathbf{Z}$ for v contains: $[2, 4]$; $[4, 2]$; $[8, 1]$; $[1, 8]$; $[2, 2]$; $[4, 1]$; $[1, 4]$; $[1, 1]$, as the middle dimensions are aggregated out in matrix multiplication. The lookup table M would then contain entries for $(v, [2, 4])$, $(v, [4, 2])$, $(v, [8, 1])$, and so on. $M(v, [2, 4])$, for example, would store the optimal cost for computing the EinGraph up to vertex v, subject to the constraint that the `EinSum` expression associated with vertex v produces an output partitioning of $\mathbf{d}_\mathbf{Z} = [2, 4]$.

The reason for maintaining this lookup table is that computing the lowest-cost implementation for vertex v requires having access to the lowest cost implementations of the inputs to vertex v; if we

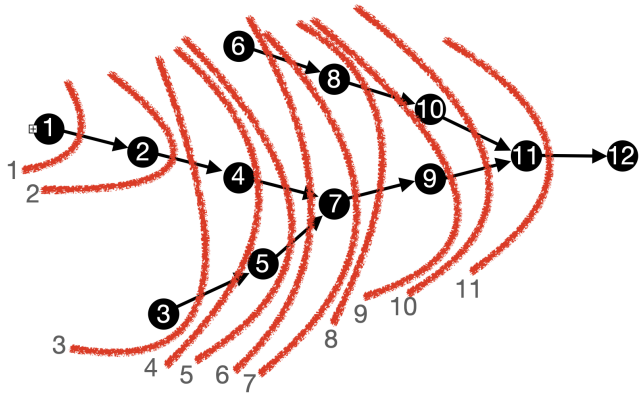

Figure 5: Progression of the `EinDecmop` dynamic programming algorithm via a topological sort. After step 1, the lookup table M holds lowest cost for producing all possible output partitionings of vertex 1. After step 2, M holds the lowest costs for both vertex 1 and 2. And in general, after step n, M holds the lowest cost for producing all possible output partitionings of vertices 1 through n.

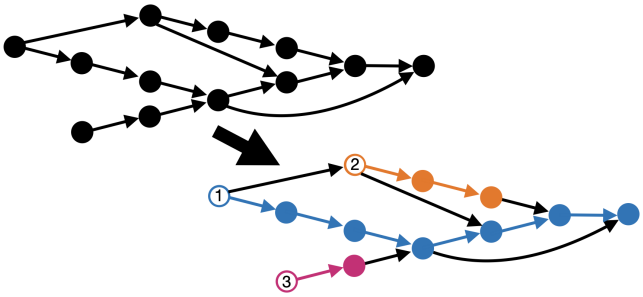

Figure 6: Linearizing an EinGraph to enable dynamic programming-based TaskGraph construction.

have access to the lowest cost for every possible input partitioning to v, we can simply enumerate all of the partition vectors for v, applying the formulas of the previous section to figure out the best way to implement vertex v. The dynamic programming algorithm proceeds according to the order provided by a topological sort of the input EinGraph, as in Figure 5. For any v with no inputs, set $M[v, \mathbf{d}]$ to zero for each $\mathbf{d}$. This reflects the fact that inputs are generally pre-computed, offline, and incur no cost.

8.3 Computing the Optimal Cost During DP

Our goal is to compute M as the dynamic programming progresses. Consider a vertex v with a binary `EinSum` expression for which we wish to compute $M[v, \mathbf{d}_\mathbf{Z}]$; let $v_\mathbf{X}$ be the first entry in v.`inputs` (so it corresponds to the $\mathbf{X}$ input to the `EinSum` expression for v) and let let $v_\mathbf{Y}$ be the second entry in v.`inputs`. Let $\ell_\mathbf{X}$, $\ell_\mathbf{Y}$, and $\ell_\mathbf{Z}$ to refer to the label vectors associated with v.`EinSum`, and $\mathbf{b}_\mathbf{XY}$ to refer to the bound vector for the `EinSum` computation. Finally, let viable(`EinSum`, p) return a list of all partitioning vectors for a tensor-relational implementation of the `EinSum` expression, subject to the constraint that the number of results of the embedded join is exactly p, ensuring p pieces of parallel work.

To compute $M[v, \mathbf{d}_\mathbf{Z}]$ we minimize:

$$\begin{aligned}&\text{over all } \mathbf{d} \in \text{viable}(v.\texttt{EinSum}, p) \text{ where } \mathbf{d}[\ell_\mathbf{Z}; \ell_\mathbf{XY}] = \mathbf{d}_\mathbf{Z};\\ &\text{over all left input partitionings } \mathbf{d}_\mathbf{X};\\ &\text{and over all right input partitionings } \mathbf{d}_\mathbf{Y};\end{aligned}$$

the following expression:

$$\begin{aligned}M[v_\mathbf{X}, \mathbf{d}_\mathbf{X}] + M[v_\mathbf{Y}, \mathbf{d}_\mathbf{Y}] + \text{cost}_\text{repart}(\mathbf{d}_\mathbf{X}, \mathbf{d}[\ell_\mathbf{Z}; \ell_\mathbf{XY}], \mathbf{b}_\mathbf{XY}[\ell_\mathbf{X}; \ell_\mathbf{XY}]) +\\ \text{cost}_\text{repart}(\mathbf{d}_\mathbf{Y}, \mathbf{d}[\ell_\mathbf{Z}; \ell_\mathbf{XY}], \mathbf{b}_\mathbf{XY}[\ell_\mathbf{Y}; \ell_\mathbf{XY}]) +\\ \text{cost}_\text{join}(\mathbf{d}, \ell_\mathbf{X}, \ell_\mathbf{Y}, \mathbf{b}_\mathbf{XY}) + \text{cost}_\text{agg}(\mathbf{d}, \ell_\text{agg}, \ell_\mathbf{Z}, \ell_\mathbf{XY}, \mathbf{b}_\mathbf{XY})\end{aligned}$$

This is: we minimize the overall cost, which is the sum of the cost for computing the graph up to and including the left and right inputs, the cost to re-partition into the current `EinSum` expression, and then the cost to perform the join and aggregation for the current `EinSum` expression. For a given output partitionings, this is minimized by considering all possible $\mathbf{d}$ vectors that will produce that output; for each, we consider all possible left input partitionings, and all possible right input partitionings.

Once all $M[.]$ entries have been computed for the input `EinGraph`, the best labeling for the graph can be found by back-tracking from the best $M[v, \mathbf{d}]$ value, where v is the output vertex for the graph. To produce the TaskGraph, at each vertex v we choose the partitioning vector $\mathbf{d}$ that produced the $M[v, \mathbf{d}_\mathbf{Z}]$ value that was used by the immediate descendant to produce its own optimal value.

8.4 Handling General DAGs

This algorithm is not applicable if there exists more than one consumer for the output of a non-input vertex, as the algorithm relies on being able to describe all possible optimal computations for a subgraph up to and including vertex v with a set of $M[v, \mathbf{d}]$ entries. If the output of v has two consumers v_1 and v_2, we would need to maintain a lookup table that stores the optimal cost for all possible combinations of output partitionings for *both* v_1 and v_2 Even if there is never more than one consumer of a non-input vertex, the algorithm may be too slow for a very large graph.

Such DAGs can be handled in approximate fashion by "linearizing" the graph. That is, we decompose the graph into a series of linear paths, and when optimizing, we consider only vertices and edges along the path. For any inputs into the path that do not come from the path, we ignore the cost to compute the associated subgraph, as well as the possible re-partition cost. So, for example, if the left ($v_\mathbf{X}$) input into a vertex v is along the path, but the right is not, to compute $M[v, \mathbf{d}_\mathbf{Z}]$ we minimize:

$$\begin{aligned}&\text{over all } \mathbf{d} \in \text{viable}(v.\texttt{EinSum}, p) \text{ where } \mathbf{d}[\ell_\mathbf{Z}; \ell_\mathbf{XY}] = \mathbf{d}_\mathbf{Z};\\ &\text{and over all left input partitionings } \mathbf{d}_\mathbf{X};\end{aligned}$$

the following expression:

$$\begin{aligned}M[v_\mathbf{X}, \mathbf{d}_\mathbf{X}] + \text{cost}_\text{repart}(\mathbf{d}_\mathbf{X}, \mathbf{d}[\ell_\mathbf{Z}; \ell_\mathbf{XY}], \mathbf{b}_\mathbf{XY}[\ell_\mathbf{X}; \ell_\mathbf{XY}]) +\\ \text{cost}_\text{join}(\mathbf{d}, \ell_\mathbf{X}, \ell_\mathbf{Y}, \mathbf{b}_\mathbf{XY}) + \text{cost}_\text{agg}(\mathbf{d}, \ell_\text{agg}, \ell_\mathbf{Z}, \ell_\mathbf{XY}, \mathbf{b}_\mathbf{XY})\end{aligned}$$

The overall algorithm is shown above in Figure 5. First, we find the longest path in the EinGraph (shown in blue), that originates from vertex 1, and optimize only along that path. To produce the **TaskGraph** partition labelings, if v is the last vertex in the path,

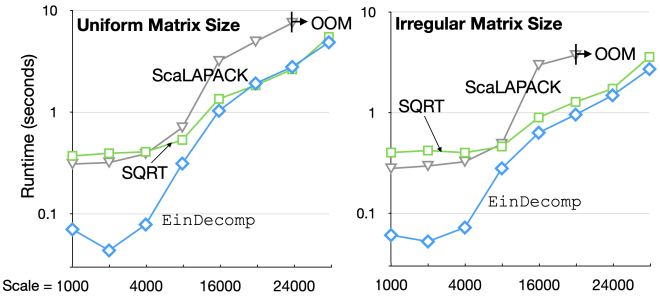

Figure 7: `EinDecomp` vs. SQRT vs. ScaLAPACK on a chain of matrix operations (16 CPU machines).

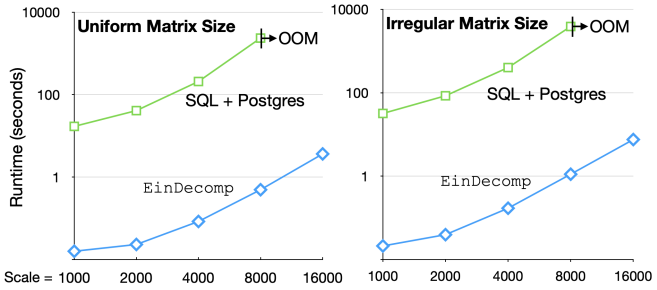

Figure 8: `EinDecomp` vs. "Portable" Einstein Notation/SQL on a chain of matrix operations (one CPU machine).

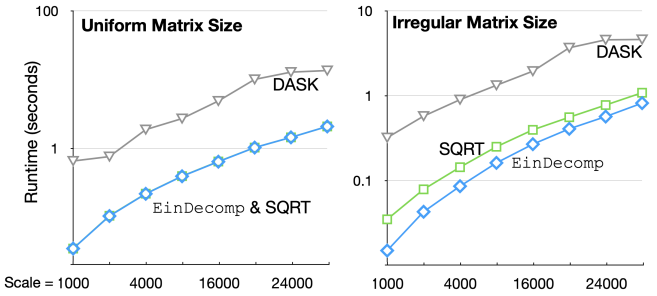

Figure 9: `EinDecomp` vs. SQRT vs. DASK, matrix chain (GPU).

we choose the smallest $M[v, \mathbf{d_Z}]$ value, and then backtrack back to vertex 1. At each vertex v, we again choose the partitioning vector $\mathbf{d}$ that produced the $M[v, \mathbf{d_Z}]$ value that was used by the immediate descendent to produce its own optimal value. Then, we find the next longest path (shown in orange, starting from vertex 2) and optimize along that path. This process is repeated for the third longest path, starting at vertex 3.

9 EXPERIMENTAL EVALUATION

We ask: how does the general `EinDecomp` approach compare to bespoke algorithms for decomposition? For example, for large-scale matrix multiplication, how does `EinDecomp` compare to the classical 3D algorithm [3]? For large-language model inference, how does `EinDecomp` compare to the tensor-parallel approach taken by Megatron [45]? A secondary question: Can an `EinDecomp`-based system be competitive with other systems?

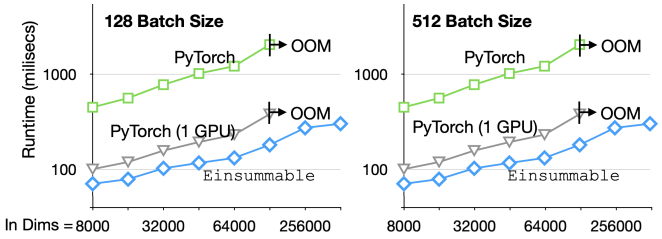

Figure 10: `EinDecomp` vs. PyTorch for training high-dimensional classifier.

9.1 Experimental Overview

We have implemented the `EinDecomp` algorithm on top of our `Einsummable` system, which is a machine learning system that utilizes an `EinSum`-based API and the `EinDecomp` decomposition algorithm to run machine learning computations on top of the Turnip execution engine [13]. `Einsummable` can run on both multi-machine CPU clusters, and on multi-GPU servers.

In the case of CPU, `EinSum` expressions are compiled by the system into kernels that (1) unpack the input tensors, (2) call Intel MKL's batch matrix multiply, and (3) re-pack the result into an output tensor. Communication is implemented using the UCX [42] library. In the case of GPU servers, `EinSum` expressions are compiled into GPU kernels using NVIDIA's CuTensor [10]. The `Einsummable` code base is currently around 27,000 lines (mostly C++).

CPU experiments are run on Amazon Web Services (AWS), using a cluster of 16 `m6in.16xlarge` machines. Each machine has 256GB of RAM and 100 Gb per second network transfer. These machines have Intel Xeon processors (Ice Lake 8375C) and 32 physical cores, each of which has two virtual cores (used by Intel's hyper-threading implementation). However, for the workloads we target (high-performance EinSum kernels), running one thread per physical core, pinned to the core, produces the best performance.

Some GPU experiments are run on AWS `P4d` instances with eight, 40 GB NVIDIA Tesla A100 GPUs. The machine has two Intel CPUs with a total of 1.1TB of RAM. Some are run on server with four, 16GB NVIDIA Tesla P100 GPUs. This machine has two Intel CPUs with a total of 1.3TB of RAM. Some experiments are run on a server with eight, 32GB NVIDIA Tesla P100 GPUs. This machine also has two Intel CPUs with a total of 1.3TB of RAM.

9.2 Experiments Run

Our evaluation of the `EinDecomp` framework considered in this paper consists of four different experiments.

Experiment 1: *Testing* `EinDecomp`*'s ability to parallelize large-scale matrix-chain arithmetic.* Parallelizing chains of matrix operations is a classical problem. We consider two chains of the form $(\mathbf{A} \times \mathbf{B}) + (\mathbf{C} \times (\mathbf{D} \times \mathbf{E}))$. In the first, all matrices are square, so for a scale of s, all matrices are sized $s \times s$. In the second, the matrices are sized as: $\mathbf{A}$: $s \times .1s$, $\mathbf{B}$: $.1s \times s$, $\mathbf{C}$: $s \times .1s$, $\mathbf{D}$: $.1s \times 10s$, $\mathbf{E}$: $10s \times s$.

Our experiments test a wide variety of s values. We run this on the CPU cluster and on the P100 GPU server. On both, we compare `Einsummable` + `EinDecomp` with `Einsummable` + "SQRT," where, to decompose a matrix into n parts, we simply slice the matrix $\sqrt{n}$ ways

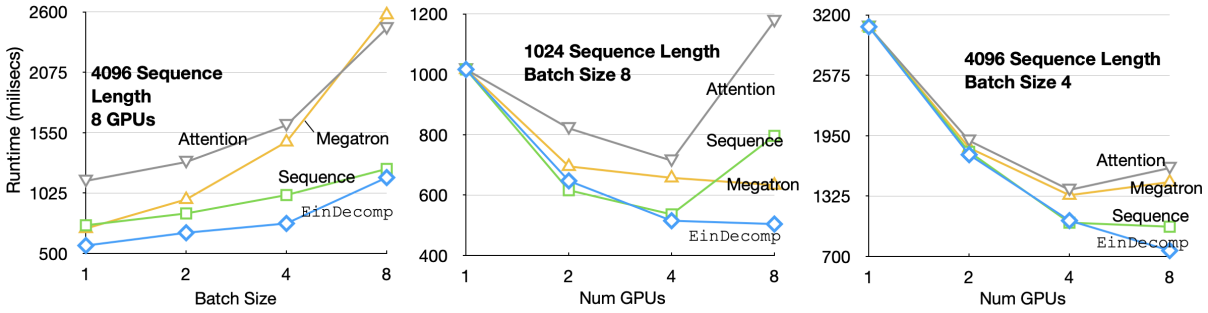

Figure 11: LLaMA large language model, run in `Einsummable` using different decomposition algorithms.

vertically and $\sqrt{n}$ ways horizontally. If the matrices are square, this gives rise to the well-known 3D matrix multiplication algorithm, which is communication-optimal for square matrices. On the CPU cluster we compare the two `Einsummable`-based algorithms with ScaLAPACK, which is the classical, high-performance, distributed matrix software. On the GPU server we compare with DASK, which is a Python library for parallel computing. We also compare with an `EinSum`-to-SQL compiler [5] on one CPU machine; the SQL is executed using Postgres. CPU results are in Figure 7. GPU results are in Figures 8 and 9. "OOM" means that the method (ScaLAPACK in this case) failed due to out-of-memory errors.

Experiment 2: *Testing* `EinDecomp`*'s ability to decompose a large feed-forward neural network classifier for training.* Here we consider training a large, feed-forward neural network (FFNN), using Pytorch (vanilla data parallel) and using `Einsummable` + `EinDecomp`. We use the AmazonCat-14K data set, which has 14,588 labels and 597,540 input features to train a FFNN having 8192 hidden neurons, using gradient descent. We start with 8192 input features, and gradually increase the number of input features until all are used. For PyTorch, we also test an option that uses only a single GPU, as opposed to all four P100 GPUs. Results are given in Figure 10, with batch sizes of 128 and 512 data points.

Experiment 3: *Comparing* `EinDecomp` *with standard algorithms for decomposing a large language models.* Our experiments target "first token" inference ("FTinf") using LLaMA large-language model [49] (or "LLM"; FTinf is also known as "prefill"): How long does it take to produce the first output token, given an input prompt? These experiments are run on the V100 server with eight GPUs.

There are a number of methods to parallelize LLM inference. One alternative is the now-classic "Megatron" parallelization scheme [45], which is a tensor-parallel or model-parallel scheme. Another is "sequence", which splits the input sequence up n ways, for inference on n GPUs. A final alternative is what we call "attention", where all attention heads are split into groups. To ensure an apples-to-apples comparison, all three of these methods were implemented on top of `Einsummable`, and compared with `Einsummable` + `EinDecomp` to compare the automatic decomposition with these other options.

We run three different FTinf experiments using the 7 billion parameter LLaMA model. In the first, we use all eight GPUs, and perform FTinf on sequences of 4096 tokens, varying the batch size. In the second, we use sequences of 1024 tokens, batch size eight,

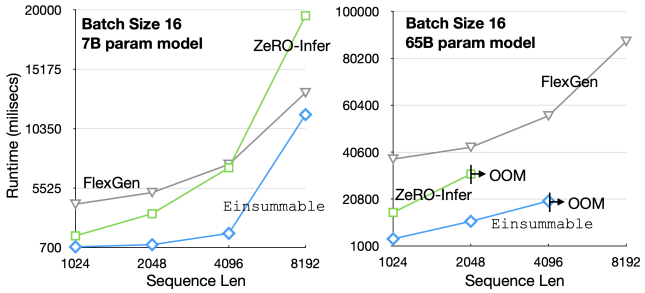

Figure 12: `Einsummable` vs ZeRO and FlexGen, LLaMA inference.

and vary the number of GPUs. In the third, we use sequences of 4096 tokens, batch size four, and vary the number of GPUs. Results are given in Figure 11.

Experiment 4: *Comparing how an* `EinDecomp`*-powered system compares with other systems for large-scale LLM inference.* Our last experiment is tasked with asking the question: can an `EinDecomp`-based system compete with a standard, hand-coded system for LLM inference? One unique aspect of `Einsummable` is that, due to the TURNIP engine, it is able to page data in GPU RAM out to CPU RAM, and avoid the ubiquitous OOM errors that plague GPU computing. Two other, well-known, PyTorch-based systems that can also do this are ZeRO Inference [4] and FlexGen [44]. We run two FTinf experiments on the AWS A100 server, both with a batch size of 16. The first uses the 7 billion parameter LLaMA model, the second the 65 billion parameter model. In each, we vary the sequence length.

9.3 Discussion

With just a few exceptions, `Einsummable` + `EinDecomp` was the best option. Crucially, the automated parallelism enabled by `EinDecomp` almost always met or outperformed the performance of bespoke decomposition/parallelization strategies.

In **Experiment 1**, as expected, SQL is much slower than `Einsummable` + `EinDecomp`, as pushing huge numbers of tuples through a database is much slower than executing a TRA-based computation using high-performance kernels. Also, it was quite surprising how poorly both ScaLAPACK and DASK performed. In the GPU experiments, we observe exactly what as expected: `Einsummable`

+ EinDecomp and Einsummable + SQRT perform the same for uniform sizes, whereas there is a consistent 2× gap for non-uniform sizes, as the simple decomposition provided by SQRT does not adapt to the skewed matrix sizes.

In **Experiment 2**, EinDecomp far outperformed data parallel PyTorch. This is perhaps the worse case for a data parallel approach: a massive model which must be broadcast across GPUs, and a comparatively small input batch. This is why PyTorch does so poorly. In fact, PyTorch on one GPU did far better than PyTorch on four GPUs, because it removes the need to broadcast the model.

In **Experiment 3**, we find that in the case of powering a LLM, EinDecomp is able to consistently do as good as, or better than, all of the obvious alternatives. One surprising finding is how well decomposing along the sequence dimension works. It is outperformed by EinDecomp, but consistently outperforms Megatron.

Finally, in **Experiment 4**, we find that Einsummable + EinDecomp is able to far outperform other PyTorch-based systems that enable inference in a memory-constrained environment. This does not directly evaluate EinDecomp, as the engine underlying Einsummable differs from the runtime for these systems. FlexGen is particularly slow at smaller sequence lengths because it utilizes pipeline parallelism to reduce memory usage—pipeline parallelism on a model such as LLaMA cannot reduce latency, but in this case it actually increases latency significantly due to pipeline startup and shutdown times (such delays would be smaller, as a fraction of total time, for larger batch sizes). ZeRO loads weights layer by layer from CPU RAM, incurring significant costs. Loading 65B weights over a PCIe bus will take four or more seconds, depending up on the setup. Further, different systems will implement expensive operations (such as attention) differently. While all of these issues undoubtedly contribute to performance differences, the results do show that the automated decompositions enabled by EinDecomp can power a high-performance machine learning system.

10 RELATED WORK

Variants of Einstein summation notation have been used for a long time (not surprisingly, Einstein used such notation [14]), and it is supported by both TensorFlow and PyTorch. Recent work has explored its use for ML [28] and its translation to SQL [5, 48], though this sort of "pure" relational implementation cannot compete with a tensor-relational implementation in terms of performance. There is a high computational overhead to push each tuple through a relational system [30, 34, 47]. The carefully-designed memory access patterns in an array-based kernel means that a CPU or GPU operates at close to its optimal floating-point operation per second rate, with little overhead.

Parallelizing ML computations has generated a lot of recent interest. Only a few recent works such as Alpa [56], FlexFlow [22] and follow-ups such as Unity [50], and Galvatron [32] have explored automatic parallelism. Unlike EinDecomp, none of these papers considered the question of how to declaratively specify the computation, and how to leverage that declarative specification to facilitate auto-parallelism. Unity, for example takes an entirely algebraic approach, directly manipulating "programs" written in their implementation abstraction (called *PCG*). Unity (and the same group's earlier TASO paper [21]) generate algebraic transformations by relying on logical rules describing valid transformations on operations, such as batch matrix multiply. EinDecomp, in contrast, starts with a general-purpose tensor calculus (EinSum) with known semantics, rather than relying on rules that specify the semantics of each operator (as in Unity) or a manual enumeration of all possible parallelization strategies for supported operators (as in Alpa), possible parallelization strategies are derived directly from the EinSum specification. Unlike these other efforts, EinDecomp works for any computation that can be specified using SimSum, with no operator-specific code required.

Other efforts at supporting parallelism offer less automation, or are more focused. One of the most influential works in this domain is Megatron-LM [45], but it proposes a specific parallelization scheme for transformers. Amazon SageMaker Model Parallelism [23] is a PyTorch-based library that is designed to make training of large and complex models easier as is Microsoft's ZeRO-Infinity [38]. GSPMD [53] is a Google-built tool for distributed/parallel machine learning, built on top of Google's XLA compiler [1]. GSPMD can be looked at as a Python API and engine for distributed tensor computations (the majority the GSPMD paper is concerned with describing the API). In that sense, the GSPMD effort sits below our work—it is suggesting an alternative implementation abstraction to the TRA that relies on ideas like distributed/sharded tensors. One could, in fact, map to the TRA onto GSPMD, rather than implementing a TRA runtime. Another effort is AutoMap [39] from DeepMind. AutoMap implements an operator-based abstraction, but programmers are asked to implement operators in a language (MLIR/PartIR [7, 27]) that forces users to expose parallelism. Mesh TensorFlow [43] allows programmers to partition tensors across a compute mesh, but partitioning decisions are left to the programmer. Other recent efforts (such as PyTorch Distributed [29]) stick to the classical data parallel paradigm.

This paper leverages the close connection between tensors and classical relations. Others have noticed this connection before. The Tensor Relational Algebra, which we use in this paper, was proposed previously [54]. Many other systems leverage this synergy such as SystemDS [6], DAPHNE [11], the Tensor Data Platform [17], TileDB [35], ArrayQL [41], STOREL [40], and TensorDB [24, 25].

11 CONCLUSIONS

We have described how to compile a computation consisting of a graph of operations expressed in the Einstein summation notation (EinSum language) and automatically decompose the EinSum operations to execute in a distributed CPU cluster, or on a GPU server. We showed, through an extensive set of experiments, that the resulting EinDecomp algorithm is, when implemented within the Einsummable system, able to perform as well as, or better than, many other standard parallelization options.

ACKNOWLEDGMENTS

This research was supported by the NSF under grant numbers 2212557, 2131294, 2008240, 1918651, NIH CTSA award No. UL1TR003167, and the US DOT Tier-1 UTC CYBER-CARE grant.

REFERENCES

[1] 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.

[2] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2020. Matrix multiplication on batches of small matrices in half and half-complex precisions. *J. Parallel and Distrib. Comput.* 145 (2020), 188–201.

[3] 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.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[5] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and Portable Einstein Summation in SQL. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–19.

[6] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*.

[7] Uday Bondhugula. 2020. High performance code generation in MLIR: An early case study with GEMM. *arXiv preprint arXiv:2003.00532* (2020).

[8] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2021. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1967–1981.

[9] 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.

[10] Nvidia Corporation. 2023. cuTENSOR: A CUDA Library for Tensor Algebra. https://docs.nvidia.com/cuda/cutensor/latest/index.html Accessed: 2024-10-01.

[11] Patrick Damme et al. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR*. https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[13] Zhimin Ding, Jiawen Yao, Brianna Barrow, Tania Lorido Botran, Christopher Jermaine, Yuxin Tang, Jiehui Li, Xinyu Yao, Sleem Mahmoud Abdelghafar, and Daniel Bourgeois. 2024. TURNIP: A" Nondeterministic" GPU Runtime with CPU RAM Offload. *arXiv preprint arXiv:2405.16283* (2024).

[14] Albert Einstein, Leopold Infeld, and Banesh Hoffmann. 1938. The gravitational equations and the problem of motion. *Annals of mathematics* (1938), 65–100.

[15] Philipp Farber and Krste Asanovic. 1997. Parallel neural network training on multi-spert. In *Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on*. IEEE, 659–666.

[16] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[17] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2022. The Tensor Data Platform: Towards an AI-centric Database System. *CoRR* abs/2211.02753 (2022). https://doi.org/10.48550/arXiv.2211.02753

[18] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. 2016. Omnivore: An optimizer for multi-device deep learning on CPUs and GPUs. *arXiv preprint arXiv:1606.04487* (2016).

[19] Waqar Hasan and Rajeev Motwani. 1994. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *VLDB*, Vol. 94. Citeseer, 12–15.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*. 103–112.

[21] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[22] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[23] Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. 2021. Amazon SageMaker Model Parallelism: A General and Flexible Framework for Large Model Training. *arXiv preprint arXiv:2111.05972* (2021).

[24] 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.

[25] 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.

[26] 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.

[27] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[28] 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.

[29] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv preprint arXiv:2006.15704* (2020).

[30] 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.

[31] Manish Mehta and David J DeWitt. 1995. Managing intra-operator parallelism in parallel database systems. In *VLDB*, Vol. 95. 382–394.

[32] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. 2022. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878* (2022).

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[34] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[35] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy G. Mattson. 2016. The TileDB Array Data Storage Manager. *Proc. VLDB Endow.* 10, 4 (2016), 349–360. https://doi.org/10.14778/3025111.3025117

[36] PyTorch. 2023. PyTocrch 2.0. https://pytorch.org/get-started/pytorch-2.0

[37] Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 873–880.

[38] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[39] Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, Georg Stefan Schmid, Tamara Norman, James Molloy, Jonathan Godwin, Norman Alexander Rink, and Vinod Nair. 2021. Automap: Towards Ergonomic Automated Parallelism for ML Models. *arXiv preprint arXiv:2112.02958* (2021).

[40] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[41] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. 2022. ArrayQL Integration into Code-Generating Database Systems. In *EDBT*. https://doi.org/10.5441/002/edbt.2022.04

[42] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, and Cliff Young. 2018. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084* (2018).

[44] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[46] 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.

[47] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[48] Yuxin Tang, Zhimin Ding, Dimitrije Jankov, Binhang Yuan, Daniel Bourgeois, and Chris Jermaine. 2023. Auto-Differentiation of Relational Computations for Very Large Scale Machine Learning. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 33581–33598. https://proceedings.mlr.press/v202/tang23a.html

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[50] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[51] 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).

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

[53] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, and Marcello Maggioni. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv preprint arXiv:2105.04663* (2021).

[54] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. (2021).

[55] Xiru Zhang, Michael Mckenna, Jill P Mesirov, and David L Waltz. 1990. An efficient implementation of the back-propagation algorithm on the connection machine CM-2. In *Advances in neural information processing systems*. 801–809.

[56] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.