Deinsum: Practically I/O Optimal Multilinear Algebra

1st Alexandros Nikolaos Ziogas
Department of Computer Science
ETH Zurich
Zurich, Switzerland
alexandros.ziogas@inf.ethz.ch

2nd Grzegorz Kwasniewski*
NextSilicon
Tel Aviv, Israel
grzegorz.kwasniewski@nextsilicon.com

3rd Tal Ben-Nun
Department of Computer Science
ETH Zurich
Zurich, Switzerland
tal.bennun@inf.ethz.ch

4th Timo Schneider
Department of Computer Science
ETH Zurich
Zurich, Switzerland
timo.schneider@inf.ethz.ch

5th Torsten Hoefler
Department of Computer Science
ETH Zurich
Zurich, Switzerland
torsten.hoefler@inf.ethz.ch

***Abstract*—Multilinear algebra kernel performance on modern massively-parallel systems is determined mainly by data movement. However, deriving data movement-optimal distributed schedules for programs with many high-dimensional inputs is a notoriously hard problem. State-of-the-art libraries rely on heuristics and often fall back to suboptimal tensor folding and BLAS calls. We present Deinsum, an automated framework for distributed multilinear algebra computations expressed in Einstein notation, based on rigorous mathematical tools to address this problem. Our framework automatically derives data movement-optimal tiling and generates corresponding distributed schedules, further optimizing the performance of local computations by increasing their arithmetic intensity. To show the benefits of our approach, we test it on two important tensor kernel classes: Matricized Tensor Times Khatri-Rao Products and Tensor Times Matrix chains. We show performance results and scaling on the Piz Daint supercomputer, with up to 19x speedup over state-of-the-art solutions on 512 nodes.**

I. Introduction

Linear algebra kernels are the fundamental building blocks of virtually all scientific applications; from physics [1], computational chemistry [2], and medicine [3]; to material science [4], machine learning [5], [6], and climate modeling [7]–[9]. It is nigh impossible for any survey of the relevant scientific codes to not stumble at every step across multitudes of vector operations, matrix products, and decompositions from the arsenal of the ubiquitous BLAS [10] and LAPACK [11] libraries. Furthermore, the execution of these kernels often dominates the overall runtime of entire applications; and with current hardware trends, their performance is frequently limited by the data movement [12] rather than FLOPs. Therefore, the design of communication-efficient parallel algorithms for (multi)linear algebra is indispensable in efficiently executing the scientific applications at scale.

Linear algebra kernels, which operate on vectors and matrices, have been studied extensively. There are a plethora of

*The author's affiliation at the time of submission was NextSilicon. However, a significant part of the research was done while he was affiliated with ETH Zurich.

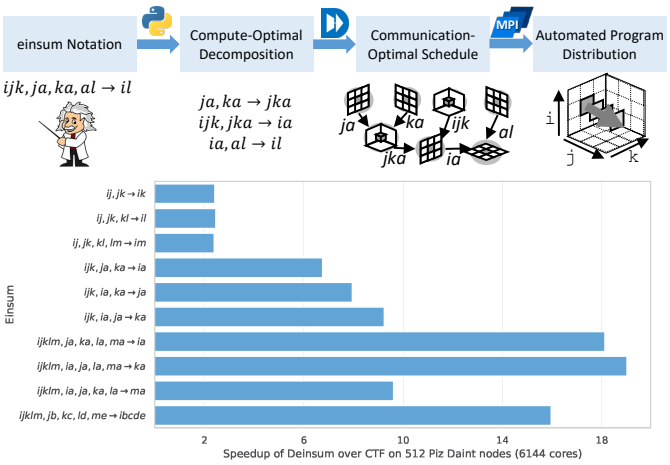

Fig. 1: Overview of Deinsum

works on lower bounds and communication-avoiding schedules, e.g., for matrix multiplication [13]–[15], and matrix factorizations such as LU and Cholesky [16], [17]. However, multilinear algebra, the extension of these methods on multi-dimensional arrays (higher-order tensors), is far less studied, especially in communication optimality. The performance of critical computational kernels in data analysis, such as the CANDECOMP/PARAFAC (CP) [18] and Tucker decompositions [19], is largely untapped due to the complexity imposed by the high dimensionality of the iteration space. Although some works study the theoretical communication complexity of chosen multilinear kernels [20], practical implementations tend to focus only on the shared-memory parallelization [21], [22] due to the intrinsic complexity of efficient communication patterns. To the best of our knowledge, the only broadly used library for distributed general tensor algebra expressible in the Einstein summation (einsum) notation is the Cyclops Tensor Framework [23].

To close the gap between the well-studied and optimized BLAS- and LAPACK-like kernels and the mostly uncharted data movement modeling in the multilinear territory, we intro-

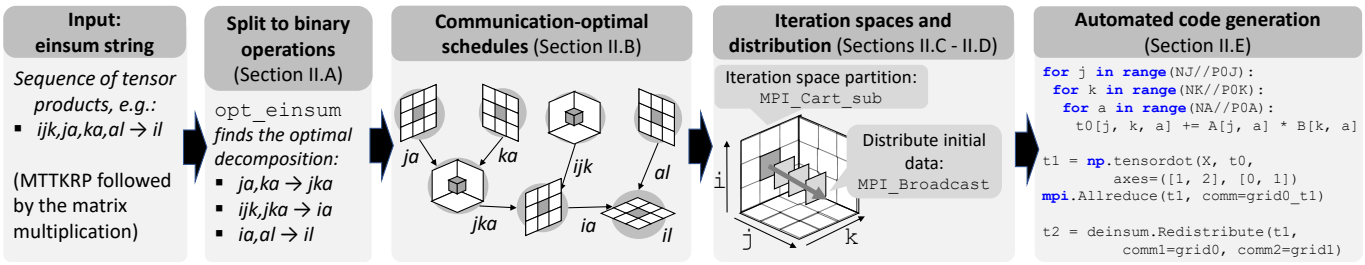

Fig. 2: Deinsum accepts arbiratry einsum strings. The single n-ary operation is decomposed into a sequence of binary operations that minimize the arithmetic complexity. Then, the framework creates the data movement model and automatically derives the tight I/O lower bound together with the corresponding parallel schedule. Next, it creates required iteration spaces, communicators, and data distribution routines. Finally, the entire schedule is automatically translated to a high-performance distributed code.

duce Deinsum, a framework for the automated derivation of I/O optimal parallel schedules of arbitrary multilinear algebra kernels described in einsum notation and operating on dense data. To the best of our knowledge, this is the first work that incorporates an analytical model of data reuse and communication minimization across multiple statements of larger kernels with fully automatic cross-platform code generation, data distribution, and high-performance computation. The presented pipeline not only provides tight I/O lower bounds for input programs, but also outputs provably communication-optimal distributed schedules. In summary, we make the following contributions:

- Code-generation framework written in Python, fully-automating data distribution and computation at scale.
- Tight I/O lower bounds for MTTKRP, the main computational kernel of the CP decomposition, improved by more than $6\times$ over the previously best-known result [20].
- Up to $19\times$ performance improvement over the current state-of-the-art, Cyclops Tensor Framework (CTF) [23].

The rest of the paper is organized as follows. First, we provide in Sec. II a top-down example that introduces Deinsum's workflow together with a high-level description of all the theoretical concepts. We proceed with a rigorous mathematical formulation of Deinsum, starting with the basic tensor algebra (Sec. III-A), continuing with our framework's underlying I/O lower bound theory (Sec. IV), and finishing with the distribution of multilinear algebra kernels (Sec. V). Then, we introduce a set of benchmarks that we subsequently use to exhibit Deinsum's superiority against the current state-of-the-art (Sec. VI). We close the paper with a related work section.

II. Workflow

To describe our framework's workflow, we use as an example the multilinear algebra kernel described by $ijk, ja, ka, al \rightarrow il$ in *Einstein index notation*. In practical terms, it describes a program with five nested loops, one for each index (i, j, k, a, l) that appears in the formula. Each loop iterates over an integer interval, for example, $i \in 0..N_i - 1$, $j \in 0..N_j - 1$, and so on, generating a 5-dimensional *iteration space* equal to the Cartesian product of the five intervals. We use the notation $\boldsymbol{I} \equiv \bigtimes_{idx \in (i,j,k,l,a)} \{0..N_{idx} - 1\}$ to describe this space. The program has four input tensors, one for each of the index strings ijk, ja, ka, and al that appear before the arrow in the formula; an order-3 tensor $\boldsymbol{\mathcal{X}}$ with size $N_i N_j N_k$, and three order-2 tensors (matrices) $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$, with sizes $N_j N_a$, $N_k N_a$, and $N_a N_l$ respectively. The program's output is represented by the index string il that appears after the arrow in the formula and corresponds to a matrix of size $N_i N_l$. A naive implementation in Python follows:

Listing 1: Naive implementation of $ijk, ja, ka, al \rightarrow il$.

```
for i in range(NI):
 for j in range(NJ):
  for k in range(NK):
   for l in range(NL):
    for a in range(NA):
     out[i,l]+=X[i,j,k]*A[j,a]*B[k,a]*C[a,l]
```

The rest of this section provides a high-level overview of Deinsum's inner workings and the workflow's steps, shown in Fig. 2, using the above program as an example. Deinsum decomposes the given einsum string into associative binary operations to expose FLOP-reduction opportunities (Sec. II-A). It then lowers the program to a data-centric intermediate representation (Sec. II-B), facilitating the extraction of the iteration spaces and the I/O lower bound analysis (Sec. II-C). It then block-distributes the program's data and computation using a near I/O optimal parallel schedule (Sec. II-D). The final step is the automated code generation and execution on distributed memory machines (Sec. II-E).

A. Decomposition of Associative Operations

The above implementation is not compute-efficient: there is a significant amount of repeated arithmetic operations since, for every operand, only a subset of iteration variables is used. For example, the same multiplication `B[k,a] * C[a,l]` is performed for each different value of `i` and `j`. Exploiting the associativity of multiplication, we can break down the above 4-ary operation operation to a series of binary operations, effectively reducing the overall arithmetic complexity from $4N_iN_jN_kN_lN_a$ to just $2N_iN_a(N_k(1+N_j)+N_l)$ FLOPs:

- $ja, ka \rightarrow jka$ (`NJ * NA * NK` iterations)
- $ijk, jka \rightarrow ia$ (`NI * NJ * NK * NA` iterations)
- $ia, al \rightarrow il$ (`NI * NA * NL` iterations)

The first binary operation is a Khatri-Rao Product (KRP), the second operation is a Tensor Dot Product (TDOT), and the last is matrix multiplication. All operations that may appear in tensor programs are formally defined in Sec. III-B. This sequence of operations roughly corresponds to the following Python program that utilizes the NumPy (`numpy` or `np`) [24] Python module for basic numerical kernels:

```
# ja,ka->jka
t0 = np.zeros((NJ, NK, NA), dtype=X.dtype)
for j in range(NJ):
  for k in range(NK):
    for a in range(NA):
      t0[j, k, a] += A[j, a] * B[k, a]
# ijk,jka->ia
t1 = np.tensordot(X, t0, axes=([1, 2], [0, 1])
# ia,al->il
out = t1 @ C
```

Our framework uses `opt_einsum` [25] – the Optimized Einsum Python module, which accepts as input arbitrary multilinear algebra kernel descriptions in Einstein index notation and breaks them down to sequences of binary tensor operations that minimize the overall FLOP count.

B. Communication-Optimal Parallel Schedules

We lower the sequence of binary tensor operations to a data flow-based intermediate representation, utilizing the Data-Centric Parallel Programming (`dace`) [26] Python framework. We extract the program's iteration space and its fine-grained parametric data access patterns from this representation. We then create a fully-symbolic data movement model, capturing data reuse both inside each binary tensor operation (e.g., via tiling) and across different kernels, potentially reusing intermediate results over the entire chain of operations (e.g., via kernel fusion). To derive tight I/O lower bounds and corresponding schedules, we implement the combinatorial data access model presented by Kwasniewski et al. [27]. The outline of the model is presented in Sec. IV. In the above example, our framework outputs the data-movement optimal schedule that fuses the first two binary operations, KRP and TDOT, forming the Matricized Times Tensor Khatri-Rao Product (MTTKRP, defined in Sec. III-B) and then multiplies it with matrix `C` using a GEMM call with a provided I/O optimal tile size. We refer to the above two groupings of the program's binary operations as the MTTKRP and MM terms.

Our framework automatically generates one of the theoretical contributions of this work: a tight parallel I/O lower bound for MTTKRP is described in Sec. IV-E, which provides more than $6\times$ improvement over the previously best-known lower bound [20]. Interestingly, a two-step MTTKRP (KRP + GEMM), which is commonly used in tensor libraries [28], [29], is not communication-optimal (Sec. IV-E).

C. Iteration Spaces

For each of the MTTKRP and MM terms we derive their corresponding iteration spaces. The first term exists in the 4-dimensional space $\bigtimes_{idx\in(i,j,k,a)}\{0..N_{idx}-1\}$, while the second one is in the 3-dimensional space $\bigtimes_{idx\in(i,l,a)}\{0..N_{idx}-1\}$. The basic idea is to distribute these iteration spaces to the available P processes. For practicality, we consider distributed programs utilizing MPI communication and, therefore, processes can be assumed to correspond to MPI ranks. For each iteration space, we arrange the P processes to a Cartesian process grid with the same dimensionality. The first space is mapped to a grid with dimensions $(P_i^{(0)}, P_j^{(0)}, P_k^{(0)}, P_a^{(1)})$, while a $(P_i^{(1)}, P_a^{(1)}, P_l^{(1)})$-sized Cartesian grid is generated for the second sub-space. The superscript of the grid dimensions identifies the term, while the subscript is the dimension index. The mapping from iteration spaces to Cartesian grids follows the block distribution.

We note that our framework is parametric in the sizes of the tensors, the optimal tile sizes, and the lengths of the Cartesian process grid dimensions (the number of grid dimensions depends on the dimensionality of the program's iteration space and is constant). The exact dimensions of the process grids depend on the available number of processes, which can be given at runtime. To provide a better intuition for the iteration space distributions, we consider for the rest of this section that there are 8 processes and that $N_{idx} = 10$, for $idx \in (i, j, k, l, a)$. However, a rigorous mathematical model can be found in Sec. V-B. According to the tile sizes generated in the previous step, the first grid has dimensions (equivalently, number of tiles) $(2, 2, 2, 1)$. This decomposition of the MTTKRP term iteration space to the 8 MPI ranks is shown in Tab. I.

TABLE I: Block distribution of the example program's MTTKRP term iteration space to $P = 8$ MPI processes.

Rank	Dimension Slices			
	i	j	k	a
0	$0..(N_i/2)-1$	$0..(N_j/2)-1$	$0..(N_k/2)-1$	$0..N_a-1$
1			$N_k/2..N_k-1$	
2		$N_j/2..N_j-1$	$0..(N_k/2)-1$	
3			$N_k/2..N_k-1$	
4	$N_i/2..N_i-1$	$0..(N_j/2)-1$	$0..(N_k/2)-1$	
5			$N_k/2..N_k-1$	
6		$N_j/2..N_j-1$	$0..(N_k/2)-1$	
7			$N_k/2..N_k-1$	

D. Data and Computation Distribution

Subsequently, our framework block-distributes the program's data and computation to the Cartesian process grids straightforwardly; each process is assigned the blocks of data and computation corresponding to its assigned blocks of iteration sub-spaces. Thus, the input order-3 tensor $\boldsymbol{\mathcal{X}}$ is tiled in half in each of its modes, resulting in 8 three-dimensional blocks. Each process is assigned one of those tiles. On the other hand, the input matrix $\boldsymbol{A}$ is partitioned to only two blocks since the dimension corresponding to the a index is not tiled. However, if we look again at Tab. I, we can see that each of these blocks is needed by multiple processes. For example, both the iteration space blocks assigned to ranks 0,

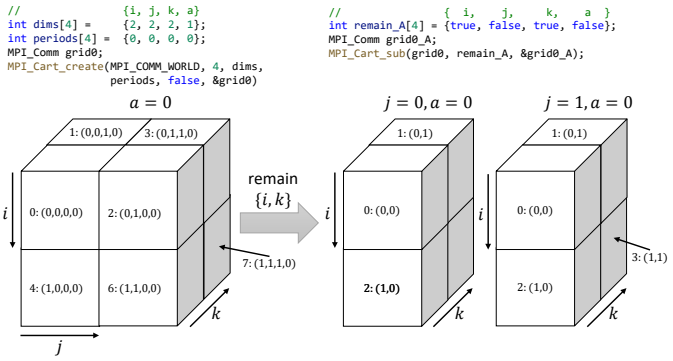

Fig. 3: The MPI process grids, ranks, and coordinates produced by Listing 2.

and 1 include the sub-block $\{0..(N_j/2)-1\}\times\{0..N_a-1\}$. Our framework handles these cases by replicating such data blocks over the necessary processes. The processes that replicate such data are defined by the Cartesian sub-grids produced by dropping the dimensions that are not relevant to the data. Readers familiar with MPI Cartesian grids may find it intuitive to consider the `MPI_Cart_create` [30], and `MPI_Cart_sub` [31] methods. For example, the Cartesian grid of the first binary operation group and sub-grid for matrix $\boldsymbol{A}$ are described by the MPI calls in Listing 2. Furthermore, the result of those calls is visualized in Fig. 3.

Listing 2: MPI calls generating the sub-grid for matrix $\boldsymbol{A}$.

```
//                {i, j, k, a}
int dims[4] =     {2, 2, 2, 1};
int periods[4] =  {0, 0, 0, 0};
MPI_Comm grid0;
MPI_Cart_create(MPI_COMM_WORLD, 4, dims,
                periods, false, &grid0);
//                { i,    j,     k,    a  }
int remain_A[4] = {true, false, true, false};
MPI_Comm grid0_A;
MPI_Cart_sub(grid0, remain_A, &grid0_A);
```

The `MPI_Cart_sub` call will produce in total $P_j^{(0)}\cdot P_a^{(0)}=2$ sub-grids, one for each of the $\boldsymbol{A}$-blocks. Each sub-grid includes $P_i^{(0)}\cdot P_k^{(0)}=4$ of the total $P=8$ processes, which replicate one of the $\boldsymbol{A}$-blocks. The assignment of the $\boldsymbol{\mathcal{X}}$- and $\boldsymbol{A}$-blocks is presented in Tab. II.

Block-distributing the data with replication ensures that each process can perform its assigned computation for a specific group of binary operations and iteration sub-space without further inter-process communication, as long as there are no dependencies on intermediate results. There are two issues to address here. First, if the output data of the group of operations does not span the whole iteration space, then, in an analogous manner to input replication, each output block is split into partial results. For example, the output `t1` of the first group of operations has size N_iN_a. Therefore, there are $P_i^{(0)}\cdot P_a^{(0)}=2$ `t1`-blocks, each one assigned to $P_j^{(0)}\cdot P_k^{(0)}=4$ MPI ranks. After completing its assigned computation, each process holds a partial result of its assigned `t1`-block. By reducing these partial results over each sub-grid with a collective operation (`MPI_Allreduce`), we achieve block distribution with replication for the output so that it can be used in next steps. The second issue relates to intermediate data that are the output of one group of operations and the input to another. In general, the distributions of the groups differ, and the data must be redistributed. In our 8 process example, the second group of operations has an iteration sub-space $\bigtimes_{idx\in(i,l,a)}\{0..N_{idx}-1\}$, assigned to a process grid with sizes $(2,2,2)$. The intermediate tensor `t1` must be redistributed from a block distribution over $P_i^{(0)}\cdot P_a^{(0)}=2$ processes to another block distribution over $P_i^{(1)}\cdot P_a^{(1)}=4$ processes. Our framework automatically infers the communication needed to redistribute tensors across different block distributions and Cartesian process grids. The redistribution's theoretical background is presented in Sec. V-C.

E. Automated Code Generation

The last step involves putting all the above analyses together and generating code that executes multilinear algebra kernels in distributed machines using MPI. To that end, we employ again `dace`, which includes basic MPI support [32]. We extend this functionality to support MPI Cartesian grids, and we use the available API to implement our redistribution scheme as a library call. We create the distributed program by adding the necessary MPI communication calls to the intermediate representation. The generated distributed code (for a specific einsum) is then compiled to a shared library that can be called by any application. Furthermore, Deinsum outputs an intermediate Python program that is functionally equivalent to the generated code. For the example presented in this section, the Python code is the following:

```
grid0 = mpi.Cart_create(dims=[P0I,P0J,P0K,P0A])
grid0_t1 = mpi.Cart_sub(
  comm=grid0, remain=[False,True,True,False])
grid1 = mpi.Cart_create(dims=[P1I, P1L, P1A])
grid1_out = mpi.Cart_sub(
  comm=grid1, remain=[False,False,True])
# ja,ka->jka
t0 = np.zeros((NJ//P0J, NK//P0K, NA//P0A),
              dtype=X.dtype)
for j in range(NJ//P0J):
  for k in range(NK//P0K):
    for a in range(NA//P0A):
      t0[j, k, a] += A[j, a] * B[k, a]
# ijk,jka->ia
t1 = np.tensordot(X, t0, axes=([1, 2], [0, 1])
mpi.Allreduce(t1, comm=grid0_t1)
# ia,al->il
t2 = deinsum.Redistribute(t1, comm1=grid0,
                                comm2=grid1)
out = t2 @ C
mpi.Allreduce(out, comm=grid1_out)
```

III. Tensor Algebra

This section describes the mathematical notation and the fundamental concepts behind data movement analysis in multilinear algebra. We use 0-based indexing for consistency.

TABLE II: Block-distribution with replication of the example program's tensors $\boldsymbol{\mathcal{X}}$, and $\boldsymbol{A}$ to $P = 8$ MPI processes, with $N_{idx} = 10$.

Rank	Coords	$\boldsymbol{\mathcal{X}}$-Block	$\boldsymbol{A}$-Block
0	$(0,0,0,0)$	`X[:5, :5, :5]`	`A[:5, :]`
1	$(0,0,1,0)$	`X[:5, :5, 5:]`	`A[:5, :]`
2	$(0,1,0,0)$	`X[:5, 5:, :5]`	`A[5:, :]`
3	$(0,1,1,0)$	`X[:5, 5:, 5:]`	`A[5:, :]`
4	$(1,0,0,0)$	`X[5:, :5, :5]`	`A[:5, :]`
5	$(1,0,1,0)$	`X[5:, :5, 5:]`	`A[:5, :]`
6	$(1,1,0,0)$	`X[5:, 5:, :5]`	`A[5:, :]`
7	$(1,1,1,0)$	`X[5:, 5:, 5:]`	`A[5:, :]`

A. Tensor Definitions and Einstein Summation Notation

Multilinear algebra programs operate on tensors, frequently represented by multidimensional arrays. The formal definition of tensors, tensor spaces, and their mathematical significance as basis-independent transformations is beyond the scope of this paper — rigorous definitions can be found in dedicated literature [**?**]. In this work, we focus on tensors from the computational and data movement perspectives, thus we refer to an order N tensor $\boldsymbol{\mathcal{X}}$ simply as an element of an N dimensional vector space $\boldsymbol{\mathcal{X}} \in \mathcal{F}^{I_0 \times \ldots I_{N-1}}$ over field $\mathcal{F}$, where $\mathcal{F}$ is typically the field of real $\mathbb{R}$ or complex numbers $\mathbb{C}$. Analogously, we refer to vectors as order 1 tensors and to matrices as order 2 tensors. We define $\boldsymbol{I} = \bigtimes_{j \in 0..N-1} I_j$ as the tensor's *iteration space*. The set of indices $(i_0, i_1, \ldots, i_{N-1})$ that iterate over $\boldsymbol{I}$ is used to access tensor elements. Given a multilinear map f

$$f : V_0 \times \cdots \times V_{N-1} \to W$$

where $V_0, \ldots, V_{N-1}$, and W are vector spaces, while f is a linear function w.r.t. each of its N arguments, this map has the associated tensor product:

$$\boldsymbol{\mathcal{W}} = \boldsymbol{\mathcal{V}}^0 \otimes \cdots \otimes \boldsymbol{\mathcal{V}}^{N-1}$$

where $\boldsymbol{\mathcal{W}} \in W$, and $\boldsymbol{\mathcal{V}}^j \in V_j$. $\boldsymbol{\mathcal{V}}^j$ are tensor *modes*. Assuming that the tensors have iteration spaces $\boldsymbol{I}^w$, and $\boldsymbol{I}^j$ respectively, the above expression can be simplified using the Einstein summation notation:

$$\mathcal{W} = \mathcal{V}^0_{I_0} \mathcal{V}^1_{I_1} \ldots \mathcal{V}^{N-1}_{I_{N-1}}$$

Repeated indices in the iteration spaces of the $\boldsymbol{\mathcal{V}}^j$ tensors are implicitly summed over, while non-repeated indices correspond to dimensions of $\boldsymbol{\mathcal{W}}$.

To provide an intuitive example, $y = A_{ji} A_{ik} x_k$ represents the equation $\boldsymbol{y} = \boldsymbol{A}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x}$, where the repeated indices i and k represent reduction over corresponding dimensions and the final result is a one-dimensional vector with index j. Analogously, $C = A_{ik} B_{kj}$ is the matrix-matrix product, and $A = u_i v_j$ is the outer product of vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. We note that it is common, especially in programming libraries that implement the Einstein notation, to drop the tensor names and keep only the indices. For example, the $y = A_{ji} A_{ik} x_k$ expression is simplified to `ji,ik,k->j`. The three index string before the right arrow are the *access indices* of the three input tensor, while `j` is the access index of the output $\boldsymbol{y}$.

B. Tensor operations

Having defined tensors, we proceed with describing basic tensor operations that frequently appear in multilinear algebra kernels. For the rest of this section we use the tensor $\boldsymbol{\mathcal{X}} \in \mathcal{F}^{I_0 \times \ldots I_{N-1}} \equiv X$. We start with a unary tensor operation, the *mode-n matricization* $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}}_{(n)}$:

$$f : \; X \to \mathcal{F}^{(I_0 \ldots I_{n-1} I_{n+1} \ldots I_{N-1}) \times I_n}, \quad \boldsymbol{\mathcal{X}} \mapsto \boldsymbol{\mathcal{X}}_{(n)}$$

In simple terms, this operation transposes a tensor by permuting the order of its modes so that the n-th mode comes last (or first, depending on convention). Subsequently, it *flattens* the first (or last) $N-1$ modes, effectively transforming the tensor to a matrix. The flattening of the modes cannot be expressed as an einsum, however, the transposition can be written as $i_0 \ldots i_{N-1} \to i_0 \ldots i_{n-1} i_{n+1} \ldots i_{N-1} i_n$. Next is the *mode-$n$ tensor product* or Tensor Times Matrix (TTM), denoted by $\times_n$. It is an operation in mode-n between a tensor $\boldsymbol{\mathcal{X}}$ and a matrix $\boldsymbol{U} \in \mathcal{F}^{I_n \times R} \equiv U$:

$$f : \; X \times U \to \mathcal{F}^{I_0 \times \ldots I_{n-1} \times R \times I_{n+1} \times \ldots I_{N-1}}, \quad (\boldsymbol{\mathcal{X}}, \boldsymbol{U}) \mapsto \boldsymbol{\mathcal{X}} \times_n \boldsymbol{U}$$

This product is computed by multiplying each of the tensor's mode-n vectors (*fibers*) by the $\boldsymbol{U}$ matrix. Another way to compute TTM is to produce the mode-n matricization of the $\boldsymbol{\mathcal{X}}$ tensor, multiply by the matrix $\boldsymbol{U}$ and *fold* the output matrix back to an order-N tensor so that the fibers corresponding to $\boldsymbol{U}$'s columns are placed in the n-th mode. This operation's einsum is $i_0 \ldots i_{N-1}, i_n r \to i_0 \ldots i_{n-1} r i_{n+1} \ldots i_{N-1}$. The Khatri-Rao Product (KRP) is defined as the column-wise Kronecker product of two matrices:

$$f : \; \mathcal{F}^{I_0 \times R} \times \mathcal{F}^{I_1 \times R} \to \mathcal{F}^{I_0 \times I_1}, \quad \left(\boldsymbol{U}^0, \boldsymbol{U}^1\right) \mapsto \boldsymbol{U}^0 \odot \boldsymbol{U}^1$$

Its einsum representation is $i_0 r, i_1 r \to i_0 i_1$. We note that both TTM and KRP can operate on tensors that are matricized appropriately, allowing TTM to generalize to the Tensor Dot Product (TDOT). Multiple TTM operations can be chained together to form a mode-n Tensor Times Matrix chain (TTMc). This is an $(N-1)$-ary operation on an order-N tensor and $N-1$ matrices $\boldsymbol{U}^j \in \mathcal{F}^{I_j \times R_j} \equiv U^j, j \in \{0..N-1\} \setminus n$:

$$\begin{aligned} f : \; & X \times U^0 \times \ldots U^{n-1} \times U^{n+1} \times \ldots U^{N-1} \\ & \to \mathcal{F}^{R_0 \times \ldots R_{n-1} \times I_n \times R_{n+1} \times \ldots R_{N-1}} \\ & \left(\boldsymbol{\mathcal{X}}, \boldsymbol{U}^0, \ldots, \boldsymbol{U}^{n-1}, \boldsymbol{U}^{n+1}, \ldots, \boldsymbol{U}^{N-1}\right) \\ & \mapsto \boldsymbol{\mathcal{X}} \times_0 \boldsymbol{U}^0 \cdots \times_{n-1} \boldsymbol{U}^{n-1} \times_{n+1} \boldsymbol{U}^{n+1} \cdots \times_{N-1} \boldsymbol{U}^{N-1} \end{aligned}$$

TTMc is written as the einsum:

$$\begin{aligned} & i_0 \ldots i_{n-1} i_{n+1} \ldots i_{N-1}, i_0 r_0, \ldots, i_{n-1} r_{n-1}, \\ & i_{n+1} r_{n+1}, \ldots, i_{N-1} r_{N-1} \to r_0 \ldots r_{n-1} i_n r_{n+1} \ldots r_{N-1} \end{aligned}$$

The mode-n Matricized Tensor Times Khatri-Rao Product (MTTKRP) is defined in a similar manner:

$$f: \ X \times U^0 \times \dots U^{n-1} \times U^{n+1} \times \dots U^{N-1} \to \mathcal{F}^{I_n \times R}$$
$$\left(\boldsymbol{\mathcal{X}}, \boldsymbol{U}^0, \dots, \boldsymbol{U}^{n-1}, \boldsymbol{U}^{n+1}, \dots, \boldsymbol{U}^{N-1}\right)$$
$$\mapsto \boldsymbol{\mathcal{X}} \odot \boldsymbol{U}^0 \odot \boldsymbol{U}^{n-1} \odot \boldsymbol{U}^{n+1} \cdots \odot \boldsymbol{U}^{N-1}$$

It is described in Einstein notation by

$$i_0 \dots i_{n-1} i_{n+1} \dots i_{N-1}, i_0 r, \dots, i_{n-1} r, i_{n+1} r, \dots, i_{N-1} r \to i_n r$$

IV. Tight data movement lower bounds for multilinear algebra kernels

We now introduce our data movement model of multilinear algebra kernels. As discussed in Section II, the evaluation of such programs may be encoded as an scalar addition-multiplication in an n-deep loop nest (Listing 1). Each execution of this operation has an associated *iteration vector* $\boldsymbol{\psi} =$ `[i, j, k, l, a]` of iteration variables' values. The central idea behind finding the data movement lower bounds is to bound the minimum number of tensor elements that needs to be loaded/communicated to perform a given number of elementary operations.

A. Data reuse and computational intensity

Consider an arbitrary sequence of X elementary operations and their associated iteration vectors $\boldsymbol{\Psi} = \{\boldsymbol{\psi}_{t_0}, \dots, \boldsymbol{\psi}_{t_1}\}$, with $|\boldsymbol{\Psi}| = t_1 - t_0 = X$. Equivalently, $\boldsymbol{\Psi}$ is a set of X new computed values. However, the evaluation of $\boldsymbol{\Psi}$ may require $Q(\boldsymbol{\Psi}) < |\boldsymbol{\Psi}|$ I/O operations from main memory, since some elements may be reused while residing in fast memory. For example, in classical matrix multiplication `C[i,j]+=A[i,k]*B[k,j]`, for each different value of the iteration variable `j`, the previously loaded element `A[i,k]` is *reused*. It has been proven [13] that to perform any execution set $\boldsymbol{\Psi}$ of this kernel, with $|\boldsymbol{\Psi}| = X$ on a machine with fast memory of size S, at least $Q(\boldsymbol{\Psi}) \geq \frac{2X}{\sqrt{S}}$ elements have to be loaded to the fast memory. Equivalently, $\rho = \frac{\sqrt{S}}{2}$ is the *computational intensity* of this kernel. Intuitively, for each loaded element, no more than ρ new elements can be computed.

B. Automated derivation of data movement lower bounds

Kwasniewski et al. [27] defined a class of programs called SOAP - Simple Overlap Access Programs. We refer readers to the original paper for the full formal definition, but for our purposes, it suffices to observe that all multilinear algebra kernels considered in this paper, such as tensor contractions and decompositions, belong to the SOAP class. The authors further derived a proof of data movement lower bounds for programs that belong to this class. Below we present a summary of the four main lemmas.

Lemma 1 (Intuition behind Lemmas 1-4 [27]): Total data movement volume from the main memory to the fast memory of size S of a program that computes array A_0 as a function of input arrays $A_1, \dots, A_n$ inside a nested loop is bounded by

$$Q \geq \frac{|V|}{\rho},$$

where $|V|$ is the nested loop's iteration space size and ρ is the computational intensity. ρ can be further bounded by

$$\rho \leq \max_{\boldsymbol{\Psi}} \frac{(\sum_{i=1}^{n} |\mathcal{A}_i(\boldsymbol{\Psi})|) - S}{|\boldsymbol{\Psi}|},$$

where, for any given set of executions $\boldsymbol{\Psi}$, $\mathcal{A}_i$ is a set of elements of input array A_i accessed during $\boldsymbol{\Psi}$.

C. Programs containing multiple statements

In multilinear algebra, analyzed problems often require contracting or decomposing multiple tensors with many intermediate values. Due to the associativity of the addition and multiplication operations, a program expressed in the Einstein notation as $w = v^1_{I_1} v^2_{I_2} \dots v^n_{I_n}$ may be written as a sequence of $n-1$ binary operations $w^0_{W_0} = v^1_{I_1} v^2_{I_2}$, $w^1_{W_1} = w^0_{W_0} v^2_{I_2}$, $\dots$, $w^n_{W_n} = w^{n-1}_{W_{n-1}} v^n_{I_n}$. Observe that this can asymptotically reduce the iteration space (see Listing in Sec. II-A). Finding the order of contraction that minimizes the total number of arithmetic operations is NP-hard in a general case [33]. However, it is possible to exhaustively enumerate all combinations for a small enough number of tensors and select the optimal one.

While the arithmetic complexity of programs containing multiple statements is easy to analyze — the arithmetic complexity of a program is the sum of complexities of each constituent statement– this is not the case for the I/O complexity. Data reuse between multiple statements (e.g., caching intermediate results or fusing computations that share the same inputs) can asymptotically reduce the overall I/O cost. Loop fusion is one of the examples of this problem and is proven to be NP-hard [34]. However, analogous to the optimal contraction permutation problem, for small enough problems, Kwasniewski et al. [27] designed an abstraction that can precisely model data reuse between statements, proving the I/O lower bound for programs containing multiple statements. The key component of the method is the Symbolic Directed Graph (SDG) abstraction, in which every vertex is a tensor (input or intermediate), and edges represent data dependencies. Then, each subgraph of non-input SDG vertices represents one of the possible kernel fusions - vertices in the subgraph correspond to the fused kernels. Each subgraph (and its corresponding fused kernel) can be expressed as a SOAP statement, and its I/O lower bound is evaluated. By enumerating all possible SDG partitions, the one that minimizes the total I/O cost is chosen and represents the I/O lower bound of the entire program.

D. Sparse data structures

The data movement model assumes that all data structures are dense — each element of input arrays $A_1, \dots, A_n$ is non-trivial and has to be loaded to the fast memory at least once. This assumption is necessary to associate a set of computations $\boldsymbol{\Phi} = \{\boldsymbol{\phi_{t_0}}, \dots, \boldsymbol{\phi_{t_1}}\}$ with well-defined sets of required input elements $\mathcal{A}_1(\boldsymbol{\Phi}), \dots, \mathcal{A}_n(\boldsymbol{\Phi})$. However, the model can be extended to sparse data structures using probabilistic methods. Given a probability distribution of non-zero elements in the input tensors $P(A[\boldsymbol{\phi}] \neq 0)$, one can derive the expected

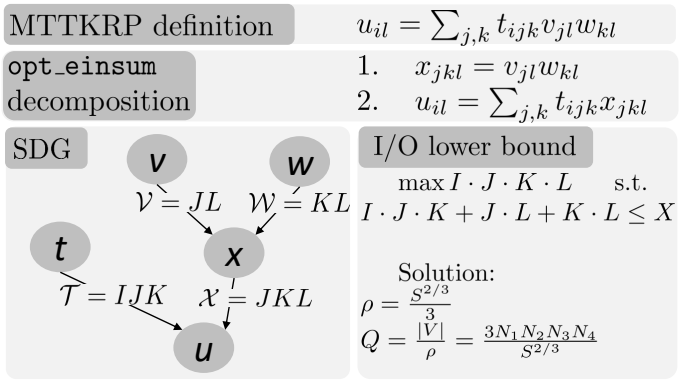

Fig. 4: MTTKRP formulation, its breakdown to two operations by the opt_einsum library, the SDG, and its I/O lower bound. $\mathcal{V}$, $\mathcal{W}$, $\mathcal{T}$, $\mathcal{X}$ are the minimum number of elements accessed from tensors v, w, T, and X during a computation that computes $IJKL$ partial products of the output tensor u.

number of non-zero elements in the access sets $E[|\mathcal{A}_j(\mathbf{\Phi})|]$. Then, the achieved lower bounds will also be probabilistic and can still be similarly used to obtain data movement-minimizing tilings and data distributions. However, the formal derivation of this extension is beyond the scope of this paper.

E. Tight MTTKRP parallel I/O lower bound

We now proceed to one of our main theoretical contributions: the MTTKRP I/O lower bound. Contrary to the state-of-the-art approaches, we show that GEMM-like parallel decomposition is communication-suboptimal. Instead, our new tiling scheme asymptotically reduces the communication by the factor of $S^{1/6}$, where S is the size of fast local memory. It also improves the previously best-known lower bound by a factor of $3^{5/3} \approx 6.24$ times [20].

The MTTKRP SDG is shown in Fig. 4. Observe that there are two possible partitions: $P_1 = \{\{x\}, \{u\}\}$ and $P_2 = \{\{x, u\}\}$. P_1 corresponds to a schedule in which the entire intermediate tensor x is computed first, and the output tensor u is evaluated next. P_2 corresponds to a schedule when these kernels are fused together and every partial result of x is immediately reused to update u. Consider an arbitrary set of computations $\mathbf{\Psi}, |\mathbf{\Psi}| = X$. Denote I the number of different values iteration variable i takes during $\mathbf{\Psi}$. Analogously, denote J, K, L the number of different values of iteration variables j, k, l. We need to express I, J, K, L as functions of the computation size X. They represent optimal tile sizes in each of the dimensions that maximize the data reuse. We now formulate the SOAP optimization problem for P_2 [27]:

$$\max I \cdot J \cdot K \cdot L \qquad \text{s.t}$$
$$I \cdot J \cdot K + J \cdot L + K \cdot L \leq X$$

which yields $I(X) = J(X) = K(X) = \sqrt[3]{\frac{2}{5}X}, \qquad L(X) = \frac{X^{2/3}}{\sqrt[3]{2}5^{2/3}}$. The interpretation is the following: for any computation $\mathbf{\Psi}, |\mathbf{\Psi}| = X$, these are the tile sizes that minimize the number of I/O operations. But since this is true for *any* X, and we want to find a tight I/O lower bound, we find the X_0 that *maximizes* the I/O cost:

$$X_0 = \text{argmin}_X \frac{I(X) \cdot J(X) \cdot K(X) \cdot L(X)}{X - S}$$

which yields $X_0 = 5S/2$. Now, substituting X_0 to the tile sizes $I(X), J(X), K(X), L(X)$ we obtain the final I/O lower bound, and the corresponding optimal tiling:

$$\rho = \frac{S^{2/3}}{3}, \qquad Q_{MTTKRP} \geq \frac{|V|}{\rho} = \frac{3N_1N_2N_3N_4}{S^{2/3}}$$
$$I = J = K = S^{1/3}, \qquad L = S^{2/3}/2$$

This result not only constitutes a tight I/O lower bound but also provides the corresponding tiling scheme and communication-optimal parallel decomposition for any size of the local memory S.

V. Distribution of Multilinear Algebra Kernels

This section defines a mathematical framework for describing iteration space distributions. We proceed by defining the block distribution of a multilinear algebra kernel's iteration space on a Cartesian process grid (notation summarized in Tab. III). We then describe the redistribution of data among different block distributions.

A. Iteration Space Distribution

Let $\boldsymbol{I} = \bigtimes_{j=0}^{N-1} \{0...I_j - 1\}$ be the N-dimensional iteration space of a mulitlinear algebra program, where I_j is the size of the j-th dimension. To *distribute* I, we first partition it to any number of disjoint subsets, which consist of consecutive elements in any dimension. This means that we can uniquely identify each subset by selecting the element with space coordinates $\boldsymbol{b} = (b_0, b_1, ..., b_{D-1})^T$, which has the shortest Euclidean distance from the origin. Any other element in the subset can be defined with respect to $\boldsymbol{b}$. The size of these subsets is defined by a vector $\boldsymbol{B} = (B_0, B_1, ..., B_{D-1})^T$. The subsets may have different sizes, so each vector component B_j may be either a constant or vary depending on $\boldsymbol{b}$ or other parameters. The coordinates $\boldsymbol{i}$ of each element in the space can be rewritten as:

$$\boldsymbol{i} = \boldsymbol{b} + \boldsymbol{o}, \tag{1}$$

where $\boldsymbol{o} = (o_0, o_1, ..., o_{D-1})^T$, with $o_j \in \{0...B_j - 1\}$, are the offset coordinates of the element relative to $\boldsymbol{b}$. We define the one-to-one mappings from the coordinates $\boldsymbol{i}$ of an element to the subset it belongs to and to its offset:

$$\boldsymbol{b} = u(\boldsymbol{i}) \tag{2}$$
$$\boldsymbol{o} = v(\boldsymbol{i}) \tag{3}$$

After partitioning the space, we assign the subsets to P processes, with each process having a unique identifier $\boldsymbol{p}$. Since each process may be assigned multiple subsets, we define a second unique subset identifier $\boldsymbol{l}$, which is process-local. We do not set any requirements in the form of $\boldsymbol{p}$ and $\boldsymbol{l}$, which may be, e.g., scalars or vectors, depending on the distribution.

TABLE III: Symbols and notations used in this section.

Name	Description
$\boldsymbol{I}$	N-dimensional iteration space. Each dimension $j \in 0...N_j - 1$ has size I_j
$\boldsymbol{i}$	Vector of size N representing the coordinates of an element of the iteration or data space
$\boldsymbol{B}$	Vector of size N representing the size of a partition of $\boldsymbol{I}$
$\boldsymbol{b}$	Given a partition of $\boldsymbol{I}$, it is a vector of size N representing the coordinates of the element that has the shortest Euclidean distance from the origin
$\boldsymbol{o}$	Given an element of the space $\boldsymbol{i}$, it is a vector of size N representing the offset coordinates of the element relative to $\boldsymbol{b}$
$\boldsymbol{p}$	Unique process identifier
$\boldsymbol{l}$	Process local partition identifier
	Mappings
$u(\boldsymbol{i})$	Returns the base element $\boldsymbol{b}$ of the partition $\boldsymbol{i}$ belongs to
$v(\boldsymbol{i})$	Returns the offset $\boldsymbol{o}$ of the partition $\boldsymbol{i}$ belongs to
$w_b(\boldsymbol{p}, \boldsymbol{l})$	Returns the subset $\boldsymbol{b}$ that belongs to $\boldsymbol{p}$ and has the unique process-local identifier $\boldsymbol{l}$
$w_p(\boldsymbol{b})$	Returns the process identifier to which $\boldsymbol{b}$ is assigned
$w_l(\boldsymbol{b})$	Returns the local subset identifier of $\boldsymbol{b}$

We define mappings among the global subset identifier $\boldsymbol{b}$, the process identifier, and the local subset identifier:

$$\boldsymbol{b} = w_b(\boldsymbol{p}, \boldsymbol{l}) \tag{4}$$

$$\boldsymbol{p} = w_p(\boldsymbol{b}) \tag{5}$$

$$\boldsymbol{l} = w_l(\boldsymbol{b}) \tag{6}$$

B. Block Distribution

We block-distribute the above space $\boldsymbol{I}$ in the following manner. First, we select a constant block size, described by the vector $\boldsymbol{B} = (B_0, B_1, ..., B_{N-1})^T$, and we tile the space to $\Pi_{j=0}^{N-1} \lceil I_j / B_j \rceil$ orthogonal blocks. This results in a regular grid of size $\lceil I_0/B_0 \rceil \times ... \times \lceil I_{N-1}/B_{N-1} \rceil$, where each block has coordinates $\boldsymbol{bi} = (bi_0, bi_1, ..., bi_{N-1})^T$, with $bi_j \in 0...\lceil I_j/B_j \rceil - 1$. Therefore, each block has the following unique identifier:

$$\boldsymbol{b} = diag(\boldsymbol{B}) \cdot \boldsymbol{bi} \tag{7}$$

$diag(\boldsymbol{B})$ is the diagonal matrix, such that $diag(\boldsymbol{B})_{jj} = B_j$. We then select the number of (MPI) processes P and arrange them in a N-dimensional Cartesian grid, the size of which is described by the vector $\boldsymbol{P} = (P_0, P_1, ..., P_{N-1})^T$, with $P = \Pi_{j=0}^{N-1} P_j$. Each process has grid coordinates $\boldsymbol{p} = (p_0, p_1, ..., p_{N-1})^T$, with $p_j \in 0...P_j - 1$. We assign a single block to each process, so that each block with coordinates $(bi_0, bi_1, ..., bi_{N-1})^T$ is assigned to process $(p_0, p_1, ..., p_{N-1})^T$. In other words, $\boldsymbol{bi} = \boldsymbol{p}$ and we rewrite the block identifier as:

$$\boldsymbol{b} = diag(\boldsymbol{B}) \cdot \boldsymbol{p} \tag{8}$$

Substituting Eq. (8) on (1), we rewrite the N-dimensional index vector $\boldsymbol{i}$ as the affine expression:

$$\boldsymbol{i} = diag(\boldsymbol{B}) \cdot \boldsymbol{p} + \boldsymbol{o} \tag{9}$$

The index $\boldsymbol{i}$ is decomposed to (a) the grid-coordinates vector $\boldsymbol{p}$ of the process to which it is assigned, and (b) the offset vector $\boldsymbol{o} = (o_0, o_1, ..., o_{N-1})^T$, with $o_j \in 0...B_j - 1$, that describes its coordinates relative to the beginning of the block to which it belongs. Eq. 9 can be decomposed to N independent affine expressions, one for each dimension:

$$i_j = p_j B_j + o_j \tag{10}$$

The mappings of Eqs. (2), (3), (5) for the block distribution are given per dimension as follows:

$$b_j = u(i_j) = B_j \left\lfloor \frac{i_j}{B_j} \right\rfloor \tag{11}$$

$$o_j = v(i_j) = i_j \ mod \ B_j \tag{12}$$

$$p_j = w_p(b) = \frac{b_j}{B_j} = \left\lfloor \frac{i_j}{B_j} \right\rfloor \tag{13}$$

We note that, in the block distribution, the mappings related to the local subsets are irrelevant, since each process is assigned a single block and the local subset identifier is the zero vector.

C. Redistributing Data

Since a multilinear algebra kernel may be decomposed into groups of statements as described in Sec. IV-C and these statements may be distributed with different block sizes, redistribution of data may be needed. Let there be two groups of statements with iteration (sub-)spaces $\boldsymbol{I}^{(x)}$ and $\boldsymbol{I}^{(y)}$ and an N-mode tensor $\boldsymbol{\mathcal{X}}$ that resides on both of them. These spaces have dimensionality $N^{(x)}, N^{(y)} \geq N$ and their intersection is a superset of the exact vector space of $\boldsymbol{\mathcal{X}}$. Without loss of generality, for the purposes of the following data movement analysis, we consider only the subsets of those spaces that coincide with the vector space of $\boldsymbol{\mathcal{X}}$. If the spaces have identical distributions, i.e., they are characterized by the same Cartesian process grids and block sizes, then no redistribution is needed. However, if the distributions are not the same, then $\boldsymbol{\mathcal{X}}$ needs to be redistributed.

Copying the data from one distribution to the other is straightforward in a *per-element* manner. Using Eq. (9), we decompose the index coordinate of each tensor element to the block sizes, process identifier, and offset coordinates that correspond to each distribution:

$$\begin{aligned} \boldsymbol{i} &= diag(\boldsymbol{B}^{(x)}) \cdot \boldsymbol{p}^{(x)} + \boldsymbol{o}^{(x)} \\ &= diag(\boldsymbol{B}^{(y)}) \cdot \boldsymbol{p}^{(y)} + \boldsymbol{o}^{(y)} \end{aligned} \tag{14}$$

The per-dimension process and offset coordinates are computed using Eqs. (12), (13). This information makes it possible to establish one-side communication and copy the data from one distribution to the other, one element at a time.

Naturally, message aggregation is a vital optimization step to reduce communication overheads by coalescing individual communication requests in fewer but larger messages. We analyze the data movement needed for redistributing a single subset of data $\boldsymbol{b}^{(x)}$ to the y-distribution. We partition the block to k (disjoint) partitions so that for each partition, communication is needed only with some (other) partition of

TABLE IV: List of benchmarks executed, together with their algebraic, and Einstein summation notations.

Name	Algebraic Notation	Definitions	Einstein Summation
Matrix-Matrix products			
1MM	$\boldsymbol{A} \cdot \boldsymbol{B}$	$\boldsymbol{A} \in \mathbb{C}^{I_0 \times I_1}, \boldsymbol{B} \in \mathbb{C}^{I_1 \times I_2}$	ij,jk->ik
2MM	$\boldsymbol{A} \cdot \boldsymbol{B} \cdot \boldsymbol{C}$	$\boldsymbol{C} \in \mathbb{C}^{I_2 \times I_3}$	ij,jk,kl->il
3MM	$\boldsymbol{A} \cdot \boldsymbol{B} \cdot \boldsymbol{C} \cdot \boldsymbol{D}$	$\boldsymbol{D} \in \mathbb{C}^{I_3 \times I_4}$	ij,jk,kl,lm->im
Matricized Tensor times Khatri-Rao products			
MTTKRP-O3-M0	$\boldsymbol{\mathcal{X}} \times_0 \left(\boldsymbol{U}^1 \odot \boldsymbol{U}^2\right)$	$\boldsymbol{\mathcal{X}} \in \mathbb{C}^{I_0 \times I_1 \times I_2}, \boldsymbol{U}^n \in \mathbb{C}^{I_n \times R}$	ijk,ja,ka->ia
MTTKRP-O3-M1	$\boldsymbol{\mathcal{X}} \times_1 \left(\boldsymbol{U}^0 \odot \boldsymbol{U}^2\right)$		ijk,ia,ka->ja
MTTKRP-O3-M2	$\boldsymbol{\mathcal{X}} \times_2 \left(\boldsymbol{U}^0 \odot \boldsymbol{U}^1\right)$		ijk,ia,ja->ka
MTTKRP-O5-M0	$\boldsymbol{\mathcal{X}} \times_0 \left(\boldsymbol{U}^1 \odot \boldsymbol{U}^2 \odot \boldsymbol{U}^3 \odot \boldsymbol{U}^4\right)$	$\boldsymbol{\mathcal{X}} \in \mathbb{C}^{I_0 \times I_1 \times I_2 \times I_3 \times I_4}, \boldsymbol{U}^n \in \mathbb{C}^{I_n \times R}$	ijklm,ja,ka,la,ma->ia
MTTKRP-O5-M2	$\boldsymbol{\mathcal{X}} \times_2 \left(\boldsymbol{U}^0 \odot \boldsymbol{U}^1 \odot \boldsymbol{U}^3 \odot \boldsymbol{U}^4\right)$		ijklm,ia,ja,la,ma->ka
MTTKRP-O5-M4	$\boldsymbol{\mathcal{X}} \times_4 \left(\boldsymbol{U}^0 \odot \boldsymbol{U}^1 \odot \boldsymbol{U}^2 \odot \boldsymbol{U}^3\right)$		ijklm,ia,ja,ka,la->ma
Tensor times Matrix Chain			
TTMc-O5-M0	$\boldsymbol{\mathcal{X}} \times_1 \boldsymbol{U}^1 \times_2 \boldsymbol{U}^2 \times_3 \boldsymbol{U}^3 \times_4 \boldsymbol{U}^4$	$\boldsymbol{\mathcal{X}} \in \mathbb{C}^{I_0 \times I_1 \times I_2 \times I_3 \times I_4}, \boldsymbol{U}^n \in \mathbb{C}^{I_n \times R_n}$	ijklm,jb,kc,ld,me->ibcde

a single subset $\boldsymbol{b}^{(y)}$ from the second distribution. To find those partitions, we rewrite the index coordinates using Eq. (1):

$$\boldsymbol{i}^{(x)} = \boldsymbol{i}^{(y)} = \boldsymbol{b}^{(y)} + \boldsymbol{o}^{(y)} \tag{15}$$

We can consider $\boldsymbol{b}^{(x)}$ to be a step function of $\boldsymbol{o}^{(y)}$. Therefore, the solution has the form:

$$\boldsymbol{b}^{(x)} = \begin{cases} \boldsymbol{b}_0, & o^{(y)} \in \mathit{partition}_0^{(y)} \\ \boldsymbol{b}_1, & o^{(y)} \in \mathit{partition}_1^{(y)} \\ ... & \\ \boldsymbol{b}_{k-1}, & o^{(y)} \in \mathit{partition}_{k-1}^{(y)} \end{cases} \tag{16}$$

Similarly, combining Eq. (3) with Eq. (2), we construct a mapping between the y-distribution offset coordinates $\boldsymbol{o}^{(y)}$ and the x-distribution offset coordinates $\boldsymbol{o}^{(x)}$:

$$\boldsymbol{o}^{(x)} = \left(v^{(x)} \circ f\right)\left(\boldsymbol{b}^{(y)} + \boldsymbol{o}^{(y)}\right) \tag{17}$$

Using the partitions of $\boldsymbol{o}^{(y)}$ found in Eq. (16), we find the corresponding partitions of $\boldsymbol{o}^{(x)}$:

$$\boldsymbol{o}^{(x)} \in \begin{cases} \mathit{partition}_0^{(x)}, & o^{(y)} \in \mathit{partition}_0^{(y)} \\ \mathit{partition}_1^{(x)}, & o^{(y)} \in \mathit{partition}_1^{(y)} \\ ... & \\ \mathit{partition}_{k-1}^{(x)}, & o^{(y)} \in \mathit{partition}_{k-1}^{(y)} \end{cases} \tag{18}$$

In general, we expect the number of partitions k to be a function of the subset sizes $\boldsymbol{B}^y$, $\boldsymbol{B}^x$.

We construct Eqs. (15) and (17) for the block distribution per dimension (the dimension subscript j is omitted for brevity):

$$p^{(x)} = \left\lfloor \frac{p^{(y)}B^{(y)} + o^{(y)}}{B^{(x)}} \right\rfloor \tag{19}$$

$$o^{(x)} = p^{(y)}B^{(y)} + o^{(y)} \ \mathit{mod}\ B^{(x)} \tag{20}$$

Using the $x \ \mathit{mod}\ y = x - y\lfloor x/y \rfloor$ property of the modulo operation, for x integer and y positive integer, we rewrite Eq. (20):

$$o^{(x)} = p^{(y)}B^{(y)} + o^{(y)} - B^{(x)} \left\lfloor \frac{p^{(y)}B^{(y)} + o^{(y)}}{B^{(x)}} \right\rfloor \tag{21}$$

The floor division $\left\lfloor \left(p^{(y)}B^{(y)} + o^{(y)}\right)/B^{(x)} \right\rfloor$ appears on both Eqs. (19), (21). To facilitate the study of the values that this expression takes, we rewrite the block identifier $p^{(y)}B^{(y)}$ in terms of the denominator, introducing auxiliary non-negative integer variables ξ and λ. ξ is the quotient of the division between $p^{(y)}B^{(y)}$ and $B^{(x)}$, while λ is the remainder:

$$p^{(y)}B^{(y)} = \xi B^{(x)} + \lambda \tag{22}$$

$$\xi = \left\lfloor \frac{p^{(y)}B^{(y)}}{B^{(x)}} \right\rfloor \in \mathbb{N} \tag{23}$$

$$\lambda = p^{(y)}B^{(y)} \ \mathit{mod}\ B^{(x)} \in 0...B^{(x)} - 1 \tag{24}$$

Using Eq. (22), we rewrite Eq. (19) as:

$$\begin{aligned} p^{(x)} &= \left\lfloor \frac{\xi B^{(x)} + \lambda + o^{(y)}}{B^{(x)}} \right\rfloor = \xi + \left\lfloor \frac{\lambda + o^{(y)}}{B^{(x)}} \right\rfloor \\ &= \xi + \begin{cases} 0, & 0 \le o^{(y)} < B^{(x)} - \lambda \\ 1, & B^{(x)} - \lambda \le o^{(y)} < 2B^{(x)} - \lambda \\ ... & \\ k-1, & (k-1)B^{(x)} - \lambda \le o^{(y)} < B^{(y)} \end{cases} \end{aligned} \tag{25}$$

where:

$$\begin{aligned} (k-1)B^{(x)} - \lambda < B^{(y)} &\Leftrightarrow k - 1 < \left\lceil \frac{B^{(y)} + \lambda}{B^{(x)}} \right\rceil \\ \Rightarrow k \equiv \left\lceil \frac{B^{(y)} + \lambda}{B^{(x)}} \right\rceil &\le \left\lceil \frac{B^{(y)} + B^{(x)} - 1}{B^{(x)}} \right\rceil \\ \Rightarrow k &\le \left\lceil \frac{B^{(y)} - 1}{B^{(x)}} \right\rceil + 1 \end{aligned} \tag{26}$$

Substituting Eqs. (22) and (25) in Eq. (21), we find the

corresponding ranges for $o^{(x)}$:

$$
\begin{aligned}
o^{(x)} &= \xi B^{(x)} + \lambda + o^{(y)} - B^{(x)} \left(\xi + \left\lfloor \frac{\lambda + o^{(y)}}{B^{(x)}} \right\rfloor \right) \\
&= \lambda + o^{(y)} - B^{(x)} \left\lfloor \frac{\lambda + o^{(y)}}{B^{(x)}} \right\rfloor \\
&\in \begin{cases}
\left[\lambda, B^{(x)}\right), & 0 \le o^{(y)} < B^{(x)} - \lambda \\
\left[0, B^{(x)}\right), & B^{(x)} - \lambda \le o^{(y)} < 2B^{(x)} - \lambda \\
\ldots & \\
\left[0, \lambda + B^{(y)} - (k-1)B^{(x)}\right), & \\
\quad (k-1)B^{(x)} - \lambda \le o^{(y)} < B^{(y)} &
\end{cases}
\end{aligned} \tag{27}
$$

We note that Eq. 19 can be used to solve message matching by substituting $o^{(y)}$ with its minimum and maximum values:

$$
\left\lceil \frac{p^{(x)} B^{(x)} + 1}{B^{(y)}} \right\rceil - 1 \le p^{(y)} < \left\lceil \frac{(p^{(x)} + 1) B^{(x)}}{B^{(y)}} \right\rceil \tag{28}
$$

Using this formula, each x-distribution process performs a loop over candidate y-distribution processes to which it may need to send data, allowing the implementation of redistribution with two-sided communication.

VI. Evaluation

We evaluate the performance of the codes generated by our framework using the benchmarks described in Tab. IV. We start with matrix multiplications; a single product (`1MM`), a chain of two (`2MM`), and three products (`3MM`). We proceed with higher-order tensor operations, specifically MTTKRP with order-3 and -5 tensors; and order-5 TTMc. We perform weak scaling experiments using the initial problem sizes (for single-node execution) and scaling factors presented in Tab. V.

TABLE V: List of benchmarks, initial problem sizes, and scaling factors as a function of the number of processes P.

Benchmark	Initial Problem Size	Scaling
`1MM`	$I^n = 4096, n \in 0..2$	$\sqrt[3]{P}$
`2MM`	$I^n = 4096, n \in 0..3$	$\sqrt[3]{P}$
`3MM`	$I^n = 4096, n \in 0..4$	$\sqrt[3]{P}$
`MTTKRP-03-M{0,1,2}`	$I^n = 1024, n \in 0..2$ $R = 24$	$\sqrt[4]{P}$ $\sqrt[4]{P}$
`MTTKRP-05-M{0,2,4}`	$I^n = 1024, n \in 0..4$ $R = 24$	$\sqrt[6]{P}$ $\sqrt[6]{P}$
`TTMc-05-M0`	$I^n = 60, n \in 0..4$ $R^n = 24, n \in 0..4$	$\sqrt[6]{P}$ $\sqrt[6]{P}$

A. Experimental Setup

We run the benchmarks on the Piz Daint supercomputer, up to 512 nodes. Each Cray XC50 compute node has a 12-core Intel E5-2690 v3 CPU @ 2.6Ghz, an Nvidia P100 GPU with 16GB of memory, and 64GB of main memory. The nodes are connected through a Cray Aries network using a Dragonfly topology. For CPU execution, we test the latest verified version of CTF (commit ID c4f89dc [35]) from its GitHub repository. The CTF C++ codes and those auto-generated by Deinsum are compiled with GCC version 9.3.0 and linked against the same libraries; Cray MPICH CUDA-aware 7.7.18 for MPI communication, and Intel oneAPI MKL 2021.3.0 for BLAS support. Furthermore, both CTF and Deinsum utilize the High-Performance Tensor Transpose library (HPTT) [36] for out-of-place tensor transpositions. Deinsum codes are linked against the latest version of HPTT (commit ID 9425386 [37]) from its GitHub repository. CTF automatically downloads and compiles a forked version of HPTT (commit ID 3c77169 [38]). For GPU execution, we test CTF's `gpu_devel_v2` branch (commit ID 0c41739b). Deinsum utilizes cuTENSOR [39] for single-GPU binary tensor operations. All GPU programs are compiled using NVCC and CUDA 11.0.

B. CPU Results

We compare Deinsum's performance with CTF's on CPU. For each benchmark and framework, we measure the runtime of at least ten executions, and we plot the median and the 95% confidence interval using bootstrapping [40]. The results are shown in Fig 5. The blue and pink bars together indicate Deinsum's runtime. The blue bar corresponds to the compute runtime, including any necessary intra-node tensor transpositions, which we measure by running a version of the code stripped of any inter-node communication for each benchmark. The pink bar represents the communication overhead, which we estimate by subtracting the compute runtime from the total execution. The green bar shows CTF's execution.

All three matrix-matrix products exhibit similar scaling behavior. The compute time is flat since it depends purely on the performance of the BLAS (MKL) GEMM kernel on the machine. Deinsum's communication overhead increases in steps at 4, 32, and 256 nodes, especially in `1MM`. This results from the SOAP-generated distribution on a three-dimensional process grid (P_0, P_1, P_2). The output product is partitioned into $P_0 P_2$ blocks, with each block further split into P_1 partial sums. In all node counts where the communication overhead increases, P_1 doubles. For example, the process grid generated for 16 nodes is $(2, 2, 4)$ and for 32 nodes the size is $(2, 4, 4)$. Therefore, the number of output blocks remains the same, while the block size increases due to weak scaling, and the depth of each `MPI_Allreduce` doubles, potentially further increasing the latency of the operation. CTF also exhibits a runtime increase in steps but at different node counts, implying a different distribution scheme. On 512 nodes, Deinsum's speedups over CTF are $2.42\times$, $2.45\times$, and $2.38\times$ for each of the three (1MM,2MM,3MM) kernels. Deinsum scales exceptionally well on the MTTKRP benchmarks exhibiting low communication overhead. The speedups against CTF on 512 nodes range from 6.75 to $19.00\times$. The performance improvements on TTMc are $15.95\times$ on 512 nodes.

C. GPU Results

We compare Deinsum's performance with CTF's on GPU, using the same statistical methods as for CPU. The results are shown in Fig. 6. We make a distinction here among executions that utilize the GPU as an accelerator, i.e, the input and output data must be copied from/to the host to/from the device, and executions where the required data are already

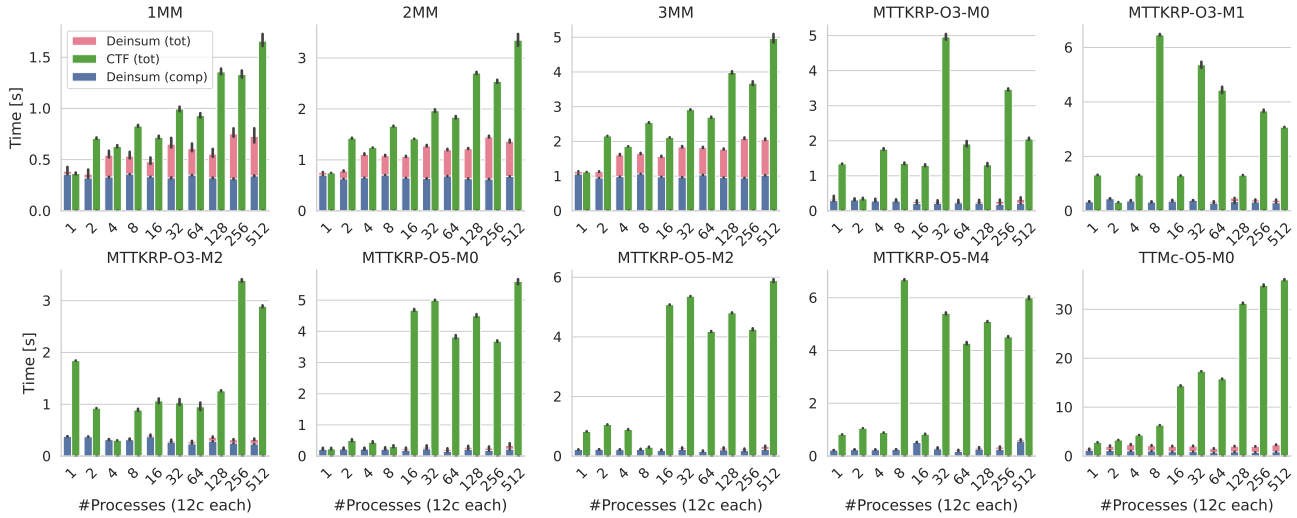

Fig. 5: Deinsum and CTF CPU runtimes on up to 512 nodes. Deinsum's computation time is also shown as part of the total runtime.

resident in global GPU memory and the output does not need to be copied back to main memory. This distinction allows us to make an apples-to-apples comparison against CTF, which supports only the first execution type, while also showcasing Deinsum's performance on the second execution type, which may be more common in large applications ran on modern GPUs with dozens of GBs of memory. The blue and pink bars together indicate Deinsum's runtime. The blue bar is Deinsum GPU-resident execution, while the pink bar is the time required to copy the input and output data between the host and the device. The green bar shows CTF's execution. Overall, we are seeing similar performance trends as in the CPU execution.

VII. Related Work

In this section, we summarize prior work related to our main contributions.

A. Multilinear Algebra Frameworks

To the best of our knowledge, the only other framework that supports automated distribution and execution of arbitrary einsums in distributed memory machines is the Cyclops Tensor Framework (CTF) [23]. TiledArray [41] is also a distributed framework that facilitates the composition of high-performance tensor arithmetic, but the user must explicitly program the data distribution into processes. There exist many frameworks that execute arbitrary einsums in shared memory: Apart from NumPy and the Optimized Einsum Python module, there are the Tensor Contraction Library (TCL) and Code Generator (TCCG) [21], and TBLIS [42] libraries that execute tensor operations and contractions on CPU. The latter led to the development of cuTENSOR [39], an Nvidia GPU-compatible library for tensor contraction, reduction, and elementwise operations. cuTENSOR also supports multi-GPU setups utilizing NVLink via the cuTENSORMg API.

B. I/O Complexity Analysis

Rigorous I/O complexity analysis dates back to the seminal work by Hong and Kung [43] who derived the first asymptotic I/O lower bound for a series of algorithms - among others, a classical matrix multiplication kernel. Their red-blue pebble game, underpinned by a two-level memory model, was extended multiple times to cover block accesses, kernel composition, and multiple memory levels [44]–[46]. The data movement model used in this paper is due to Kwasniewski et al. [27], which is also based on the red-blue pebble game. Other works that focus on the I/O complexity of linear algebra use variants of the discrete Loomis-Whitney inequality [47], [48], Holder-Brascamp-Lieb inequalities [49], or recursion-based arguments [50] to bound the I/O cost and derive communication avoiding schedules for series of linear algebra kernels. There is significantly less work on the I/O complexity of multilinear algebra kernels. Ballard et al. established a first parallel I/O lower bound for the order-n MTTKRP [20]. However, their model prohibits decomposing the kernel into a series of binary contractions.

C. Automated Data Distributions and Redistribution

Automated (re)distribution algorithms similar to the analysis presented in Sec. V are also employed by CTF. Petitet et al. [51] have presented algorithmic redistribution methods for block-cyclic distributions. Furthermore, considerable work in automating and optimizing the communication needed for multilinear algebra has been done by High Performance Fortran (HPF) compilers. We categorize it into three different approaches: (a) via linear algebraic methods to construct symbolic expressions [52], [53]; (b) using compile-time or runtime generated tables to store critical information, such as array access strides or communication mappings [54]–[58]; or (c) using the array slice expressions as index sets, such that the local and communication sets are described in terms of set operations, for example, unions and intersections [59], [60].

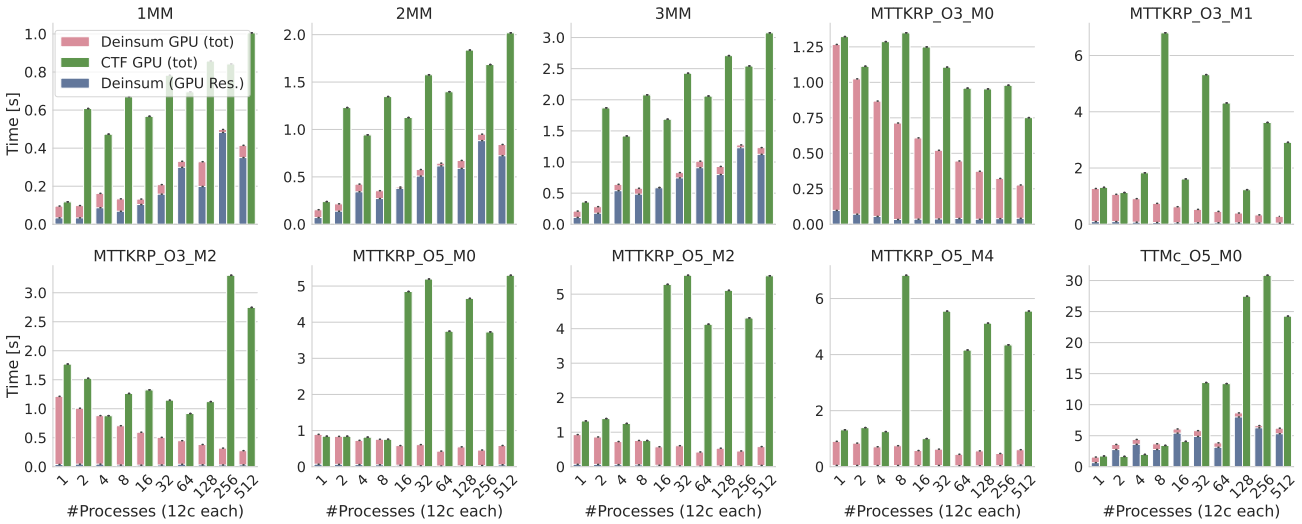

Fig. 6: Deinsum and CTF GPU runtimes on up to 512 nodes. Deinsum's runtime with input data resident in global GPU memory is also shown as part of the total runtime.

VIII. Conclusion

We present Deinsum, a framework for automatic and near I/O optimal distribution of multilinear algebra kernels expressed in Einstein notation. Deinsum leverages the strength of the SOAP theoretical framework to derive a $6\times$ improved lower bound for MTTKRP, the main computational bottleneck of the CP decomposition. Moreover, Deinsum vastly improves on CTF, the current state-of-the-art tensor computation framework, by up to $19\times$ on 512 nodes; the geometric mean of all observed speedups is $4.18\times$. These results further solidify the validity of the improved tight I/O lower bounds described in Sec. IV and confirm that the SOAP analysis provides not only theoretical but also tangible improvements to distributed computations.

IX. Acknowledgments

This work received EuroHPC-JU funding with support from the European Union's Horizon 2020 program and from the European Research Council under grant agreement PSAP, number 101002047. We also wish to acknowledge support from the DEEP-SEA project under grant agreement number 955606. The Swiss National Science Foundation supports Tal Ben-Nun (Ambizione Project No. 185778). The authors would like to thank the Swiss National Supercomputing Centre (CSCS) for access and support of the computational resources.

References

[1] W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez, and T. Williams, "Extreme scale plasma turbulence simulations on top supercomputers worldwide," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016.

[2] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, D. Golze, J. Wilhelm, S. Chulkov, M. H. Bani-Hashemian, V. Weber, U. Borštnik, M. Taillefumier, A. S. Jakobovits, A. Lazzaro, H. Pabst, T. Müller, R. Schade, M. Guidon, S. Andermatt, N. Holmberg, G. K. Schenter, A. Hehn, A. Bussy, F. Belleflamme, G. Tabacchi, A. Glöß, M. Lass, I. Bethune, C. J. Mundy, C. Plessl, M. Watkins, J. VandeVondele, M. Krack, and J. Hutter, "Cp2k: An electronic structure and molecular dynamics software package - quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, vol. 152, no. 19, p. 194103, 2020. [Online]. Available: https://doi.org/10.1063/5.0007045

[3] R. M. Hutchison, T. Womelsdorf, E. A. Allen, P. A. Bandettini, V. D. Calhoun, M. Corbetta, S. Della Penna, J. H. Duyn, G. H. Glover, J. Gonzalez-Castillo, D. A. Handwerker, S. Keilholz, V. Kiviniemi, D. A. Leopold, F. de Pasquale, O. Sporns, M. Walter, and C. Chang, "Dynamic functional connectivity: Promise, issues, and interpretations," *NeuroImage*, vol. 80, pp. 360–378, 2013. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1051116731

[4] M. Luisier, A. Schenk, W. Fichtner, and G. Klimeck, "Atomistic simulation of nanowires in the s p 3 d 5 s* tight-binding formalism: From boundary conditions to strain calculations," *Physical Review B*, vol. 74, no. 20, p. 205323, 2006.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[7] J. Dahm, E. Davis, T. Wicky, M. Cheeseman, O. Elbert, R. George, J. J. McGibbon, L. Groner, E. Paredes, and O. Fuhrer, "Gt4py: Python tool for implementing finite-difference computations for weather and climate," in *101st American Meteorological Society Annual Meeting*. AMS, 2021.

[8] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer, "Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities." *Monthly Weather Review, 139:3387–3905*, 2011.

[9] COSMO, "Consortium for small-scale modeling," oct 1998. [Online]. Available: http://www.cosmo-model.org

[10] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[11] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, Oct. 2009. [Online]. Available: https://doi.org/10.1145/1562764.1562783

[13] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, 2019.

[14] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, pp. 90–109. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23397-5_10

[15] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 193–204.

[16] G. Kwasniewski, M. Kabic, T. Ben-Nun, A. N. Ziogas, J. E. Saethre, A. Gaillard, T. Schneider, M. Besta, A. Kozhevnikov, J. VandeVondele, and T. Hoefler, "On the parallel i/o optimality of linear algebra kernels: Near-optimal matrix factorizations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. Association for Computing Machinery, 2021.

[17] E. Hutter and E. Solomonik, "Communication-avoiding Cholesky-QR2 for rectangular matrices," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 89–100.

[18] M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin, "Memory-efficient parallel tensor decompositions," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[19] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 1038–1047.

[20] G. Ballard, N. Knight, and K. Rouse, "Communication lower bounds for matricized tensor times khatri-rao product," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 557–567.

[21] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–29, 2018.

[22] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on gpus," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 85–95.

[23] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 813–824.

[24] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[25] D. G. a. Smith and J. Gray, "opt_einsum - a python package for optimizing contraction order for einsum-like expressions," *Journal of Open Source Software*, vol. 3, no. 26, p. 753, 2018. [Online]. Available: https://doi.org/10.21105/joss.00753

[26] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[27] G. Kwasniewski, T. Ben-Nun, L. Gianinazzi, A. Calotoiu, T. Schneider, A. N. Ziogas, M. Besta, and T. Hoefler, "Pebbles, graphs, and a pinch of combinatorics: Towards tight i/o lower bounds for statically analyzable programs," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 328–339.

[28] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "Hasco: Towards agile hardware and software co-design for tensor computation," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1055–1068.

[29] K. Hayashi, G. Ballard, Y. Jiang, and M. J. Tobia, "Shared-memory parallelization of mttkrp for dense tensors," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 393–394.

[30] MPICH, "Mpi_cart_create," 2022. [Online]. Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Cart_create.html

[31] ——, "Mpi_cart_sub," 2022. [Online]. Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Cart_sub.html

[32] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: Data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476176

[33] L. Chi-Chung, P. Sadayappan, and R. Wenger, "On optimizing a class of multi-dimensional loops with reduction for parallel execution," *Parallel Processing Letters*, vol. 7, no. 02, pp. 157–168, 1997.

[34] A. Darte, "On the complexity of loop fusion," in *PACT*, 1999.

[35] Cyclops Community, "Cyclops tensor framework (ctf)." [Online]. Available: https://github.com/cyclops-community/ctf

[36] P. Springer, T. Su, and P. Bientinesi, "HPTT: A High-Performance Tensor Transposition C++ Library," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2017. New York, NY, USA: ACM, 2017, pp. 56–62. [Online]. Available: http://doi.acm.org/10.1145/3091966.3091968

[37] P. Springer, "High-performance tensor transpose library." [Online]. Available: https://github.com/springer13/hptt

[38] E. Solomonik, "High-performance tensor transpose library (forked by edgar solomonik)." [Online]. Available: https://github.com/solomonik/hptt

[39] Nvidia, "cutensor," 2022. [Online]. Available: https://developer.nvidia.com/cutensor

[40] B. Efron, "The bootstrap and modern statistics," *Journal of the American Statistical Association*, vol. 95, no. 452, pp. 1293–1296, 2000.

[41] J. A. Calvin and E. F. Valeev, "Tiledarray: A general-purpose scalable block-sparse tensor framework." [Online]. Available: https://github.com/valeevgroup/tiledarray

[42] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018. [Online]. Available: https://doi.org/10.1137/16M108968X

[43] J. Hong and H. Kung, "I/O complexity: The red-blue pebble game," in *STOC*, 1981, pp. 326–333.

[44] J. S. Vitter, "External memory algorithms," in *European Symposium on Algorithms*. Springer, 1998, pp. 1–25.

[45] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "Data access complexity: The red/blue pebble game revisited," Technical Report, Tech. Rep., 2013.

[46] J. E. Savage, "Extending the hong-kung model to memory hierarchies," in *International Computing and Combinatorics Conference*. Springer, 1995, pp. 270–281.

[47] L. H. Loomis and H. Whitney, "An inequality related to the isoperimetric inequality," *Bull. Amer. Math. Soc.*, vol. 55, no. 10, pp. 961–962, 10 1949.

[48] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[49] T. M. Smith, B. Lowery, J. Langou, and R. A. van de Geijn, "A tight i/o lower bound for matrix multiplication," *arXiv preprint arXiv:1702.02017*, 2017.

[50] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.

[51] A. Petitet and J. Dongarra, "Algorithmic redistribution methods for block-cyclic decompositions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 12, pp. 1201–1216, 1999.

[52] S. P. Midkiff, "Local iteration set computation for block-cyclic distributions," in *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume II: Software*, C. D. Polychronopoulos, Ed. CRC Press, 1995, pp. 77–84.

[53] C. Ancourt, C. Fran, and I. R. Keryell, "A linear algebra framework for static hpf code distribution," *A; a*, vol. 1, no. t2, p. 1, 1993.

[54] K. Kennedy, N. Nedeljkovic, and A. Sethi, "Efficient address generation for block-cyclic distributions," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 180–184. [Online]. Available: https://doi.org/10.1145/224538.224558

[55] ——, "A linear-time algorithm for computing the memory access sequence in data-parallel programs," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 102–111. [Online]. Available: https://doi.org/10.1145/209936.209948

[56] K. "Kennedy, N. Nedeljkovic, and A. Sethi, *Communication Generation for Cyclic(K) Distributions*. Boston, MA: Springer US, 1996, pp. 185–197. [Online]. Available: https://doi.org/10.1007/978-1-4615-2315-4_14

[57] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng, "Generating local addresses and communication sets for data-parallel programs," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 149–158. [Online]. Available: https://doi.org/10.1145/155332.155348

[58] A. Thirumalai and J. Ramanujam, "Fast address sequence generation for data-parallel programs using integer lattices," in *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '95. Berlin, Heidelberg: Springer-Verlag, 1995, p. 191–208.

[59] J. M. Stichnoth, "Efficient compilation of array statements for private memory multicomputers," CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, USA, Tech. Rep., 1993.

[60] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. . Huang, and P. Sadayappan, "On compiling array expressions for efficient execution on distributed-memory machines," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 2, 1993, pp. 301–305.