Benchmarking the Linear Algebra Awareness of TensorFlow and PyTorch

Aravind Sankaran*, Navid Akbari Alashti†, and Christos Psarras‡
RWTH Aachen University, Germany
{*aravind.sankaran, †navid.alashti, ‡christos.psarras}@rwth-aachen.de

Paolo Bientinesi
Umeå Universitet, Sweden
pauldj@cs.umu.se

arXiv:2202.09888v1 [cs.MS] 20 Feb 2022

***Abstract*—Linear algebra operations, which are ubiquitous in machine learning, form major performance bottlenecks. The High-Performance Computing community invests significant effort in the development of architecture-specific optimized kernels, such as those provided by the BLAS and LAPACK libraries, to speed up linear algebra operations. However, end users are progressively less likely to go through the error prone and time-consuming process of directly using said kernels; instead, frameworks such as TensorFlow (TF) and PyTorch (PyT), which facilitate the development of machine learning applications, are becoming more and more popular. Although such frameworks link to BLAS and LAPACK, it is not clear whether or not they make use of linear algebra knowledge to speed up computations. For this reason, in this paper we develop benchmarks to investigate the linear algebra optimization capabilities of TF and PyT. Our analyses reveal that a number of linear algebra optimizations are still missing; for instance, reducing the number of scalar operations by applying the distributive law, and automatically identifying the optimal parenthesization of a matrix chain. In this work, we focus on linear algebra computations in TF and PyT; we both expose opportunities for performance enhancement to the benefit of the developers of the frameworks and provide end users with guidelines on how to achieve performance gains.**

***Index Terms*—Performance analysis, Machine Learning, Linear Algebra**

I. Introduction

The recent advances to machine learning technologies have led to the development of many latency sensitive applications, such as intelligent vehicles [1], real-time video analytics [2], and augmented reality [3], that require fast execution of costly scientific computations on heterogeneous hardware. One of the major performance bottlenecks for such scientific computations is the evaluation of linear algebra expressions, which are building blocks for countless computational problems. Libraries such as BLAS and LAPACK provide a small set of high performance kernels for some standard operations that are used to build linear algebra expressions. Implementing a linear algebra expression by directly calling said kernels is a laborious task, time consuming even for domain experts, as they should familiarize themselves with the syntax of those calls, along with its input parameters, data-types and storage formats.

By contrast, frameworks such as TensorFlow [4], PyTorch [5], Matlab [6], Julia [7], Eigen [8], Armadillo [9] (as well as many others) allow end users to input a linear algebra expression at a high level of abstraction, where the syntax closely resembles the one used on a blackboard. Internally, these tools compile the expression by automatically mapping the users' input to a sequence of function calls to basic operations that are computed with the optimized libraries. However, the mapping of a high-level expression to an optimized sequence of library calls is a task far from trivial; because, a linear algebra expression can be computed in many different ways—each corresponding to a specific sequence of library calls—which can significantly differ in terms of performance from one another [10], [11]. Unfortunately, it has been found that the mapping done by most popular high-level programming languages is still sub-optimal [12]. In this paper, we extend this investigation to popular machine learning frameworks: TensorFlow and PyTorch.

The knowledge of linear algebra can be used to derive variants for computing a given input expression; for instance, consider the following expression which appears in an image restoration application [13]:

$$\mathbf{y} := H^T\mathbf{y} + (I_n - H^T H)\mathbf{x} \tag{1}$$

where H is a square matrix in $\mathbb{R}^{n\times n}$, $\mathbf{x}$ and $\mathbf{y}$ are vectors in $\mathbb{R}^n$, and I_n is the identity matrix. This expression can be rewritten in several ways by applying the distributive and associative properties; the variants are mathematically equivalent but the sequences of instructions are different (see in Fig 1). The domain scientists may prefer to work with Variant 1 as it is more descriptive of the underlying physics. However, in terms of performance, Variant 1 would be a suboptimal choice because the product $H^T H$ is computed explicitly, performing an expensive $\mathcal{O}(n^3)$ matrix-matrix multiplication. By applying distributivity, the term $(I_n - H^T H)\mathbf{x}$ in Expression 1 is expanded, yielding:

$$\mathbf{y} := H^T\mathbf{y} + \mathbf{x} - H^T H\mathbf{x} \tag{2}$$

Due to the associativity of matrix products, the term $H^T H\mathbf{x}$ in Expression 2 can either be evaluated from "left-to-right": $(H^T H)\mathbf{x}$ or "right-to-left": $H^T(H\mathbf{x})$. The left-to-right parenthesization does not get rid of the expensive $\mathcal{O}(n^3)$ matrix multiplication. However, the right-to-left parenthesization lowers the complexity of Expression 2, which now consist of three

Financial support from the Deutsche Forschungsgemeinschaft (German Research Foundation) through grants GSC 111 and IRTG 2379 is gratefully acknowledged.

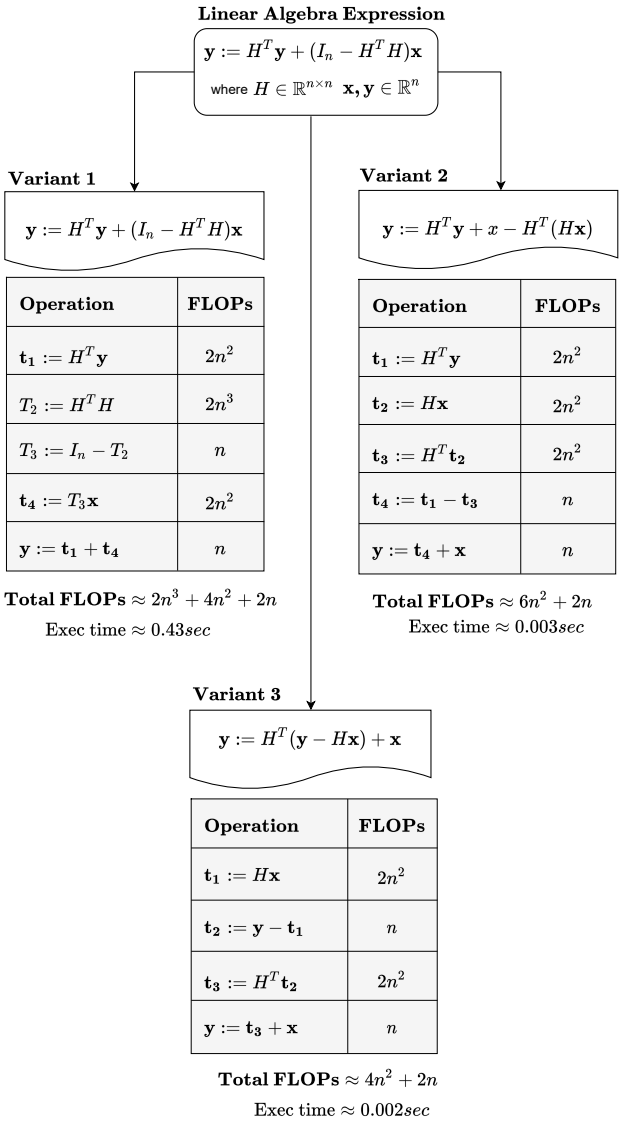

Fig. 1: Variants of Image restoration equation. The execution time is reported for $n = 3000$.

matrix-vector products, to $\mathcal{O}(n^2)$. Furthermore, applying the distributive property again, Expression 2 can be re-written as:

$$\mathbf{y} := H^T(\mathbf{y} - H\mathbf{x}) + \mathbf{x} \tag{3}$$

Now, the total number of FLOPs is further reduced, as there are only two matrix-vector products. The average execution time of the three variants implemented in TensorFlow, linked to Intel-MKL on an Intel Xeon CPU[1], is shown in Fig 1. The execution of variant 2 and 3, which does not consist of a matrix matrix product is significantly faster than variant 1.

Contributions: In this paper, we consider the two most popular machine learning frameworks TensorFlow and PyTorch, and investigate the extent to which these frameworks incorporate linear algebra knowledge to optimize mathematical expressions. To this end, we develop benchmarks consisting of simple tests that expose whether or not the frameworks carry out linear algebra optimizations such as exploiting the knowledge of matrix properties (e.g., triangular, symmetric, etc.) to reduce the number of FLOPs, optimal parenthesization of matrix chains, code motion, etc. Our benchmarks not only serve as a guide for end users to help them achieve better performance, but also expose optimization opportunities to the developers of these frameworks to help improve performance when computing linear algebra expressions.

Organization: In Sec. II, we survey the state of art. In Sec. III, we describe our experiments, evaluate the frameworks and discuss our findings. Finally, in Sec IV, we draw conclusions.

[1]for further details on the hardware used, see Sec III

II. Related Works

The difficulty in deploying and tuning the performance of scientific programs on diverse hardware and operating systems has boosted the research and development of compilers that have multiples levels of Intermediate Representations (IRs). Modern compilers such as TVM [14], Intel nGraph [15], XLA [16], Tensor Comprehension [17] and Glow [18] translate a mathematical problem through multiple levels of Intermediate Representations (IRs)—while optimising them for performance—and eventually generate machine code. Depending on the hardware supported by the compiler, machine learning frameworks such as TensorFlow [4], PyTorch [5], MXNet [19] and CNTK [20] are built with one of these compilers at their backend. Multi-level IR modularizes compiler development, so that at each level, a specific class of optimizations can be carried out [21]. At the top level, the input problem is cast to a computational graph, which captures the dependencies between data and the mathematical operations; here, algebraic optimizations such as common sub-expression elimination to remove redundant computations, algebraic simplifications, loop invariant code motion, etc., are carried out. The optimized graph is then passed on to third-part tools such as LLVM [22] for translation to lower-level IRs, where hardware-specific optimizations (such as instruction scheduling, static memory allocation, etc.) are carried out, and the code is generated for the target architecture.

The compilation of an input linear algebra expression to generate an optimized implementation can be seen as a translation to a sequence of optimized library calls. Libraries such as Intel's MKL and oneDNN [23], NVIDIA's cuBLAS and cuDNN [24], BLIS [25] etc. encapsulate operations that occur frequently in machine learning and linear algebra, and provide hardware-optimized kernels. Each expression can be computed by potentially hundreds of alternative sequences that are mathematically equivalent but significantly differ from one another in terms of performance [10], [26]. The problem of mapping a target expression into a sequence of optimized calls, while minimizing a performance-based cost function is known as the Linear Algebra Mapping Problem (LAMP); the solution to LAMP is at least NP-Complete and it has been found that

many popular frameworks for high-level languages [6]–[9] provide sub-optimal solutions [12].

LAMP can be tackled at the compilers' top level IR via graph optimization techniques. DxT [27] encodes linear algebra programs as directed acyclic graphs and uses a greedy approach to transforms them into sequence of high-performing library routines targeting distributed memory architectures. Linnea [10] is a linear algebra compiler that uses graph based approach to automatically generate variants for a given input expression and recommends an optimal variant based on the number of FLOPs performed. Grappler [28] is TensorFlows graph optimization engine that targets common machine learning operations. Since both TensorFlow and PyTorch are known to be able to perform a number of graph optimizations at their backends [21], in this paper we develop benchmarks that expose further opportunities for improving their performance when it comes to linear algebra computations.

While there are already several machine learning benchmarks such as [29]–[31], etc., which enable users to compare different frameworks and choose a more suitable one to implement their ideas with, none of them—to the best of our knowledge—evaluate the performance of linear algebra expressions within the frameworks [32]. In this paper, we follow a similar approach to [12] to investigate the performance of linear algebra expressions executed within TensorFlow and PyTorch.

III. Evaluation of the Frameworks

In our experiments, we input linear algebra test expressions in a mathematically descriptive syntax using the Python interface, and evaluate the solutions provided by TensorFlow (TF) and PyTorch (PyT) in terms of performance. The frameworks translate the python code into a language-independent intermediate representation, where the input expression is broken down into basic mathematical operations that are executed using optimized kernels. For a given test expression, we assess the quality of the solution by comparing it with several other optimized, but mathematically equivalent alternatives; the alternatives typically differ in terms of the required number of FLOPs or memory accesses. In order to compare the performance, we measure the single-threaded execution time and report the minimum over 20 repetitions; we check whether the performance differences are statistically significant (or not) using the boot-strapping approach from [11]. The experiments are run using a single core on a Linux machine with an Intel Xeon Platinum 8160 processor. The versions of the languages used are the latest stable releases as of December 2021: TensorFlow 2.7.0 and PyTorch 1.10.1. The frameworks are interfaced via Python 3.9.7 and linked to the Math Kernel Library (MKL) through the Intel OneAPI interface 2022.0.0, which provides the optimized kernels for Intel architectures. The source code for the experiments is available online[2].

The development of machine learning models typically involves two phases: Training and Deployment. The training is carried out in a research and development setting, where fast prototyping and experimentation often become more important than performance. During the deployment phase instead, the models are executed in a production environment that might demand low latency and high performance. In order to improve the productivity of developing machine learning applications and minimize code modifications between the training and deployment phases, TF and PyT support two programming modes: "Eager" and "Graph". The Eager mode is intended for research and development, while Graph mode is intended for production use cases.

Graph mode vs Eager mode

In the Graph mode, the structure of the user code is used to capture the control flow of data and operations, and to generate a computational graph, over which rigorous performance optimizations are carried out. In the Eager mode, performance optimizations are not as extensive as those in the Graph mode.

For illustration, consider the following expression which occurs in the Stochastic Newton equations [33]:

$$Y := (A^T B)^T (A^T B) \qquad A, B \in \mathbb{R}^{n \times n} \tag{4}$$

The TF code in Python that computes Expression 4 is shown in Figure 2. The operator "@" performs the matrix-matrix multiplication operation. By default, the code executes in Eager mode. In order to execute the code in Graph mode, the TF function has to be wrapped using the decorator **@tensorflow.function** (PyT functions have to be wrapped with **@torch.jit.scripts**). Unlike Eager mode, the loops inside the wrapped functions of the Graph mode cannot be programmed with the usual pythonic syntax, but have to be handled specially using framework specific constructs. This is due to the fact that, in the Graph mode, the user code is translated into a Directed-Acyclic-Graph (DAG), which does not contain loops or cycles; the nodes of the graph represent basic operations such as matrix product, transpose, etc., and the edges represent data flow.

In both Eager and Graph mode, the frameworks link to MKL, which offers highly optimized kernels for frequently encountered linear algebra operations. In order to confirm, we compare the performance of matrix matrix multiplication with a reference implementation that use the MKL kernel for matrix multiplication, GEMM, implemented in C. In the first row of Table I, we report the execution time for the matrix product sub-expression from Expression 4 for $n = 3000$ on single precision floating point data[3] in C, TF and PyT. We observe no statistically significant difference in timings (after ignoring overheads for loading the decorators[4]) between the reference implementation and the frameworks in both Graph and Eager mode; hence, we confirm that the frameworks do link to MKL.

In the Graph mode, a number of graph-optimizations are applied over the DAG; for example, duplicate nodes that

[2]https://github.com/as641651/LinearAlgebra-Awareness-Benchmark

[3]As many ML applications operate on single precision data, both TF and PyT use the single precision floating point representation by default.

[4]@tf.function = 6e-4 sec and @torch.jit,script = 2e-3 sec

$\text{ret} := (\mathbf{A^T B})^{\mathbf{T}}(\mathbf{A^T B})$ $\quad \mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$

```
import tensorflow as tf

@tf.function
def transformation(A,B){

    A^T = tf.transpose(A)
    ret =tf.transpose(A^T@B)@(A^T@B)
    return ret
}
```

Fig. 2: TensorFlow Code.

Expression	MKL-C	Eager (TF / PyT)	Graph (TF / PyT)
$A^T B$	0.39	0.40 / 0.40	0.40 / 0.40
$(A^T B)^T (A^T B)$	–	1.25 / 1.27	0.78 / 0.80

TABLE I: Execution time (in sec) for $n = 3000$

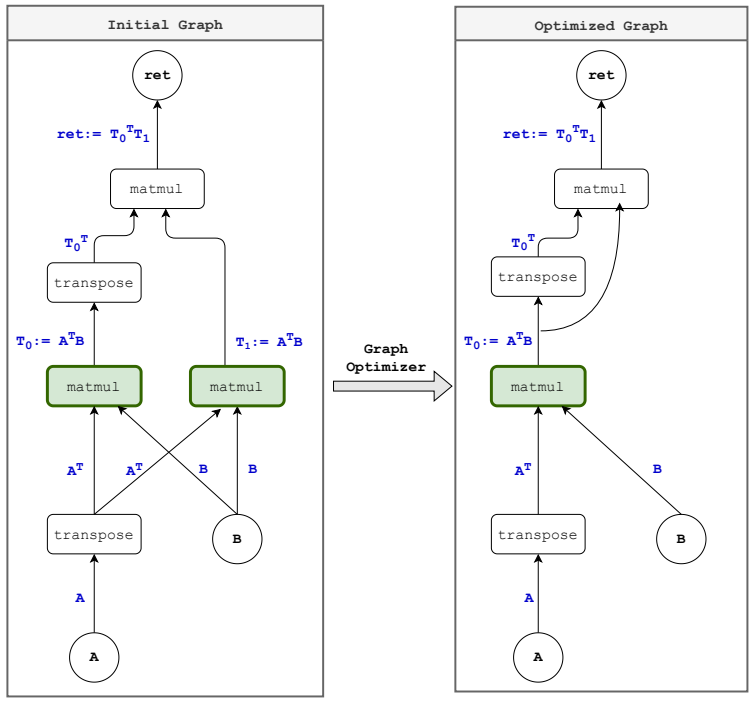

Fig. 3: The Computational Graphs for $(A^T B)^T (A^T B)$.

compute the exact same operation for the same input data can be identified and removed. The non-optimized and optimized DAG for the code-snippet in Fig. 2 is shown in Fig. 3; the circular nodes represent I/O operations, while the rounded-rectangle nodes represent mathematical operations. In the non-optimized graph, the matrix multiplication or the **matmul** operation $A^T B$ is computed twice; removing one of the nodes saves $2n^3$ FLOPs. Such optimizations are not carried out when the code is executed in Eager mode (i.e., without the decorators). In the second row of Table I, we report the execution times in the Eager and the Graph mode; the execution time in the Eager mode is approximately 1.5 times longer than in the Graph mode.

In the following, we consider only the Graph mode and develop experiments to investigate further optimization opportunities in the computation of linear algebra expressions and benchmark the performance impacts. To this end, we developed a set of 5 experiments, each one of them containing one or more test expressions, to identify whether or not the frameworks perform a particular linear algebra optimization.

A. *Experiment 1: Common Sub-expression Elimination*

Common Sub-expression Elimination (CSE) is a typical optimization incorporated in most modern compilers to reduce redundant computations. To this end, sub-expressions within an expression that evaluate to the same value are computed only once, and substituted through a temporary reference in all the subsequent instances. As illustrated earlier in this section, both TF and PyT detect common sub-expressions by searching for duplicate operations in a DAG. In this experiment, we investigate the extent to which the DAG-based approach can successfully detect common sub-expressions in some of the typical high-level input expressions.

Test expressions: We consider the following sub-expression S that occurs in the Stochastic Newton equations [33]:

$$S := A^T B \qquad A, B \in \mathbb{R}^{n \times n}$$

We develop two kinds of test expressions in which S occurs multiple times, via summation and multiplication. The test expressions and their execution times for $n = 3000$ are shown in Table II[5].

- ***Repeated in summation:*** In this expression, S occurs twice, and the occurrences are added together:

  $$E_1 := A^T B + A^T B.$$

  We compare the execution time to compute E_1 with that for S. Without CSE, the execution time for E_1 would be approximately $2\times$ higher than that for S.
  Result: In both frameworks, the execution time for E_1 is close to that for S. Therefore, we infer that the frameworks successfully identified that the S appears twice in E_1, and avoided redundant computation by instead performing $2A^T B$. The complexity of scaling S by 2 is only $\mathcal{O}(n^2)$, whose execution time is negligible in comparison to the $\mathcal{O}(n^3)$ matrix product. Moreover, in the BLAS specification, the scaling does not have to occur separately, but can be done alongside multiplication without additional overheads.

[5]In all the following experiments, the problem size is $n = 3000$, unless specified otherwise.

	Expression	TF	PyT
1	A^TB	0.40	0.40
2	$A^TB + A^TB$	0.40	0.41
3	$(A^TB)^T(A^TB)$	0.78	0.80
4	$(A^TB)^TA^TB$	1.17	1.15

TABLE II: Execution time (in sec) for $n = 3000$

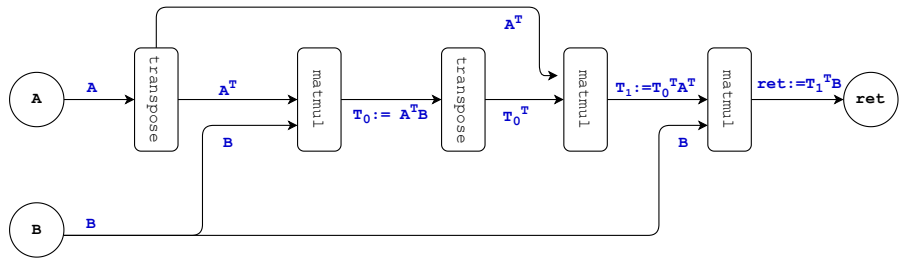

Fig. 4: The Computational Graph for $(A^TB)^TA^TB$.

- ***Repeated in multiplication:*** Let us now consider the expression B^TAA^TB, which a user can input in many different ways. Let us assume that the user explicitly rewrites the expression by distributing the transposition[6] so that S occurs twice; the input can then be either E_2 or E_3:

  $$E_2 := (A^TB)^T(A^TB) \qquad \text{(parenthesized)}$$
  $$E_3 := (A^TB)^TA^TB \qquad \text{(non-parenthesized)}$$

  With CSE, the input expressions would consist of only two matrix multiplications – one to compute S, and the other to compute S^TS – and the execution time would be approximately $2\times$ higher than that for S. Without optimization, there would be additional matrix multiplication, and the execution time would be approximately $3\times$ higher than that for S.

  Result: In both frameworks, the execution time of the parenthesized expression E_2 is $2\times$ the time for S, indicating that the common sub-expression was detected and the redundant operation was avoided. However, the execution time of the non-parenthesized expression E_3 is close to $3\times$ the time for S, which shows that if the user does not explicitly indicate the parenthesization, the frameworks fail to detect the common sub-expression.

Discussion: In-order to detect a common sub-expression, the DAG should consist of duplicate nodes with similar inputs. The graph generated by TF for the non-parenthesized matrix chain expression $(A^TB)^TA^TB$ is shown in Figure 4. The graph does not consist of any duplicate nodes. This observation shows that the frameworks do not attempt to find the optimal parenthesization for our given matrix chain input, which leads us to the next experiment, where we investigate the consequences of non-optimal parethesization of matrix chains.

B. *Experiment 2: Optimization of Matrix Chains*

Given m matrices of suitable size, the product $M := A_1A_2...A_m$ is known as a matrix chain. Because of associativity of the matrix product, matrix chains can be computed in many different ways, each identified by a specific parenthesization; for instance, $M := A_1A_2A_3$ can be computed either as $(A_1A_2)A_3$ or $A_1(A_2A_3)$. The number of different parenthesizations for a chain of length m is given by the $(m-1)^{\text{th}}$ Catalan number:

$$C_{m-1} = \frac{(2m)!}{(m+1)!m!}.$$

Although different parenthesizations evaluate to the same mathematical result (they all compute M), they require different number of FLOPs, and hence exhibit different performance. In this experiment, we input matrix chains without any parenthesization, and investigate if the frameworks automatically choose an evaluation order that requires minimum FLOPs.

Expression	TF	PyT	
	matmul	matmul	multi_dot
$H^TH\mathbf{x}$	0.40	0.41	0.006
$H^T(H\mathbf{x})$	0.006	0.004	-
$\mathbf{y}^TH^TH$	0.006	0.005	0.005
$(\mathbf{y}^TH^T)H$	0.006	0.005	-
$H^T\mathbf{y}\mathbf{x}^TH$	0.41	0.40	0.01
$(H^T\mathbf{y})(\mathbf{x}^TH)$	0.01	0.01	-

TABLE III: Execution time (in sec) of matrix chain test expression for $n = 3000$.

Test expressions: We consider three different matrix chains with sizes of the matrices chosen so that the evaluation order that minimizes the number of FLOPs is attained by evaluating the chain from *right-to-left* $A_1(A_2A_3)$, *left-to-right* $(A_1A_2)A_3$, and in *mixed* order $(A_1A_2)(A_3A_4)$:

- ***Right-to-Left:*** Recall the matrix chain sub-expression from the image restoration problem (Expression 1):

  $$\hat{\mathbf{y}}_1 := H^TH\mathbf{x} \qquad H \in \mathbb{R}^{n\times n}, \quad \mathbf{x} \in \mathbb{R}^n \tag{5}$$

  The right-most operand in the above expression is a vector; as a consequence, the evaluation from *right-to-left* $H^T(H\mathbf{x})$ avoids the expensive $\mathcal{O}(n^3)$ matrix product, and reduces the complexity to $\mathcal{O}(n^2)$, incurring a cost of $2n^2 + 2n^2$ FLOPs. On the other hand, since H is a matrix, the evaluation from left-to-right $(H^TH)\mathbf{x}$ is

[6] $U^TV^T = (VU)^T$

$\mathcal{O}(n^3)$, costing $2n^3 + 2n^2$ FLOPs. We input the expression without explicit parenthesization to check whether or not the frameworks automatically choose the *right-to-left* evaluation order.
Result: The execution times to compute the test expressions are shown in Table III. In both frameworks, when the syntax of the matrix chain is expressed using the regular matrix multiplication operator "@" (matmul), the execution time for the expression without parenthesization is significantly greater than that for the same expression evaluated from *right-to-left*, which shows that the frameworks do not automatically choose the optimum parenthesization.

- ***Left-to-Right:*** Consider the following expression:

$$\hat{\mathbf{y}}_2 := \mathbf{y}^T H^T H \qquad H \in \mathbb{R}^{n\times n}, \quad \mathbf{y} \in \mathbb{R}^{n\times 1} \tag{6}$$

Since the vector $\mathbf{y}$ is appears in the left end of the matrix chain, the evaluation from *left-to-right* $(\mathbf{y}^T H^T)H$ is $\mathcal{O}(n^2)$, whose complexity is one order of magnitude lower than the evaluation from *right-to-left* $\mathbf{y}^T(H^T H)$ that consists of an $\mathcal{O}(n^3)$ matrix product.
Result: In both frameworks, the execution time for the test expression expressed using the matmul operator without parenthesization is comparable to that of the same expression evaluated from *left-to-right*. Therefore, we infer that the default evaluation of matrix chains in both the frameworks is from *left-to-right*.

- ***Mixed:*** Consider the following expression,

$$\hat{Y} := H^T \mathbf{y}\mathbf{x}^T H \qquad H \in \mathbb{R}^{n\times n}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^{n\times 1} \tag{7}$$

where the vectors $\mathbf{x}, \mathbf{y}$ occur at the middle of a matrix chain, and neither *left-to-right* nor *right-to-left* evaluation avoids the $\mathcal{O}(n^3)$ matrix matrix product; instead, the evaluation $(H^T\mathbf{y})(\mathbf{x}^T H)$ turns out to be the optimum, having $\mathcal{O}(n^2)$ complexity.
Result: The execution time for the test expression without parenthesisation is significantly more than that for the expression evaluated in the optimized order; as usual, the frameworks do not automatically identify the optimum evaluation order when the input is expressed without explicit parenthesization using the matmul operator.

Discussion: Notice that PyTorch offers also the **torch.linalg.multi_dot** method, which solves the matrix chain problem and automatically selects the parenthesization that requires minimum FLOPs. The syntax of the input expression using the **multi_dot** function is shown in the code-snippet in Figure 5. The optimum evaluation order in terms of #FLOPS can be found using dynamic programming, where all possible parenthesizations are listed, and the variant with minimum FLOPs is chosen for execution; for a matrix chain of length 4, we show the different parenthesizations and their associated FLOPs in Figure 7. However, it should be noted that variants with comparable FLOP counts do not always exhibit comparable execution time. For instance,

$\mathbf{H^T H x}$ $\qquad \mathbf{H} \in \mathbb{R}^{n\times n} \quad \mathbf{x} \in \mathbb{R}^n$

```
@torch.jit.script
def matmul(H,x){
   H^T = torch.transpose(H)
   return H^T@H@x
}
@torch.jit.script
def multidot(H,x){
   H^T = torch.transpose(H)
   return torch.linalg.multi_dot([H^T,H,x])
}
```

Fig. 5: Variants of Matrix Chain.

the evaluation $(AB)(CD)$ can correspond to two different implementations as shown in the code-snippet in Figure 6, which differ in the order of instructions:

Y = (AB)(CD)	
Variant 1	**Variant 2**
U = A@B V = C@D Y = U@V	V = C@D U = A@B Y = U@V

Fig. 6: Variants that require same FLOPs.

Although both the variants require the same FLOP count, their execution times might differ due to the differences in memory overheads [34]. Moreover, minimizing FLOP count does not always minimize execution time, especially when the overheads due to memory references dominate; increasing parallelism tend to shift the problem to be memory-bound[7]. However, many linear algebra computations with dense matrices (having only a few non-zero elements) execute in the compute bound[8], hence FLOPs could still be a significant indicator of performance for the experiments considered in this paper [10].

The number of FLOPs required to perform a regular matrix-matrix multiplication (GEMM) is well known, hence the performance cost can be approximated using the formulae in Figure 7. However, when the matrices in multiplication have special properties such as being Symmetric, Triangular, etc, BLAS offers specialized kernels for matrix product that require fewer FLOPs than GEMM; if such specialized kernels are used, the cost function should be modified according to the FLOPs computed by that kernel. In the following experiment, we investigate if the frameworks take advantage of those specialized kernels to speed up linear algebra operations.

C. *Experiment 3: Exploiting Matrix Properties*

Matrix multiplications with matrices having certain special properties can be accelerated using specialized BLAS kernels;

[7]The performance is limited by the speed of memory access.
[8]The performance is limited by the speed of the CPU.

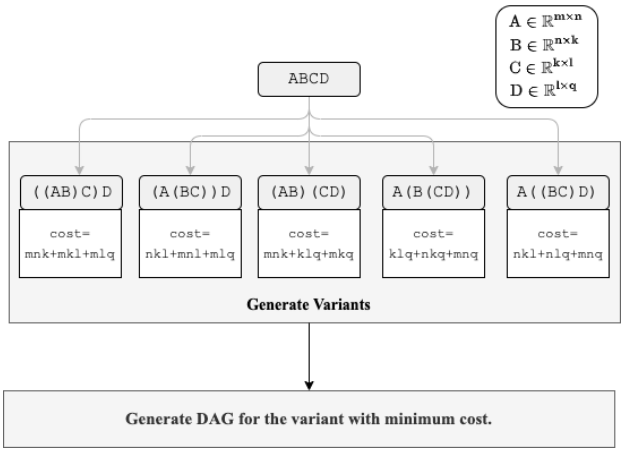

Fig. 7: Variants for Matrix Chain of length 4.

for instance, if one of the matrices in the multiplication is triangular, the product can be computed using the kernel TRMM for half the number of FLOPs than GEMM. In Python, these kernels can be accessed directly through the interface provided by the SciPy library [35], which allows the end users to code the mathematical problem by explicitly calling said kernels; however, this task is time consuming and laborious as the users should familiarize themselves with the syntax of those calls. In order to ease code development, both TF and PyT encapsulate the direct access to these kernels and offer a mathematically descriptive interface, however, it is then not clear whether or not the frameworks make use of the knowledge of matrix properties to speed up computations. Therefore, in this experiment, we compare the execution time of the hand-coded SciPy implementations – specifically written to make use of specialized kernels –, with the execution time of TF/PyT for those tests, to determine whether or not the frameworks offer similar provisions that exploit matrix properties to speed up computations.

Expr	SciPy	TF		PyT	
	BLAS	matmul	optim	matmul	optim
AB	0.40	0.40	-	0.41	-
LB	0.24	0.40	n.a	0.40	n.a
AA^T	0.24	0.41	n.a	0.39	n.a
TB	0.20	0.41	0.02	0.40	n.a
DB	0.12	0.39	0.018	0.40	n.a

TABLE IV: Exploiting Matrix Properties: Lower Triangular (L), Symmetric Output, Tridiagonal (T) and Diagonal (D). Execution time (in sec) for $n = 3000$.

Test expressions: We consider the matrix multiplication

$$Y := AB \qquad A, B \in \mathbb{R}^{n \times n}. \tag{8}$$

The computational cost of the above expression using the GEMM kernel is approximately $2n^3$ FLOPs. The required number of FLOPs can be lowered if the matrices in Expression 8 have the following special properties (we report the execution times in Table IV):

- ***Triangular:*** When all elements above the diagonal of A are zero, then the matrix is lower-triangular; for clarity, we refer to it as L. The matrix product LB can be performed using the kernel TRMM for just n^3 FLOPs, which is half the cost of GEMM.
  Result: Presently, neither frameworks offer specialized routines that a user can explicitly invoke to take advantage of the triangular matrix structure in multiplication. With the regular matrix multiplication operator, the execution time for LB is close to $1.7\times$ higher than the corresponding SciPy implementation that explicitly calls TRMM. Therefore, the frameworks do not offer provision to save the unnecessary computations.

- **Symmetric output:** In Expression 8, if B happens to be A^T, then the output of AA^T is a symmetric matrix. BLAS offers a specialized routine, SYRK ("SYmmetric Rank-K update"), which computes AA^T with only n^3 FLOPs, which is half the FLOPs of a GEMM.
  Result: The execution of AA^T using SYRK through SciPy is again close to $1.7\times$ faster than both TF and PyT; the frameworks do not take advantage of this special structure to accelerate the computation.

When the matrices in Expression 8 have the following properties, the matrix product cannot be implemented using a single specialized BLAS kernel, but can be effectively decomposed into a sequence of kernels:

- ***Tridiagonal:*** When all the elements except those in the three central diagonals of A are zero, the matrix is Tridiagonal T. The matrix product TB can be reduced from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ complexity by re-writing it as a sequence of scaling operations (using the SCAL kernel) applied to every row of B; the overall computation requires only $6n^2$ FLOPs.
  Result: The execution of TB using a sequence of SCAL via SciPy is $2\times$ faster than the regular matrix multiplication. While both the frameworks do not automatically exploit the tridiagonal structure of the matrix, TensorFlow does however offer a specialized method *linalg.tridiagonal_matmul* which the users themselves can manually invoke. The execution through this optimized call is significantly faster than even the SciPy implementation, as TensorFlow takes advantage of the fact that the scaling operations can be executed simultaneously, and hence parallelizes them.

- ***Diagonal:*** The diagonal matrix D is a special case of the tridiagonal matrix T, in which, instead of the three central diagonals, only the main diagonal is non-zero. The product DB requires only n^2 FLOPs.

Result: The execution time when using the regular matrix multiplication is $4\times$ higher than the optimized SciPy implementation. TensorFlows optimized function *linalg.tridiagonal_matmul* is $10\times$ faster than SciPy.

Discussion: If the matrix properties are known before runtime, users can explicitly annotate matrices with types that encode the properties, making it possible for the compiler to map the corresponding operations with those matrices to the specialized kernels; languages such as Julia [7] already offer this feature. The compilers in TF and PyT could also exploit the optimized kernels if matrix properties are annotated on the frameworks computational graphs. The propagation of matrix properties through the graph would also facilitate algebraic simplifications that might speed up computations; for instance, if it is known that certain matrix $Q \in \mathbb{R}^{n\times n}$ is orthogonal, then it would be straightforward to deduce that the operation Q^TQ would evaluate to an Identity matrix and avoid the explicit multiplication, hence saving $2n^3$ FLOPs.

D. *Experiment 4: Algebraic Manipulation*

Algebraic properties can be used to rewrite an expression in several alternative ways. Although these alternatives (or variants) are mathematically equivalent, the number of FLOPs required can be vastly different. In Section I, we illustrated the differences in FLOP counts as a result of rewriting Expression 1 using the distributive property. In this experiment, we develop benchmarks to expose some common algebraic manipulations that can result in significant performance improvements.

- ***Distributivity:*** In general, the application of distributive property to an expression modifies the number of multiplication operations required; for instance, consider the following test expression,

$$AB + AC = A(B + C) \qquad A, B, C \in \mathbb{R}^{n\times n} \tag{9}$$

The Left-Hand-side (LHS) of Equation 9 requires two matrix multiplication operations, while the Right-Hand-Side (RHS) requires only one, which as a result saves $2n^3$ FLOPs. However, the reduction in multiplication operations does not always reduce the required FLOPs; for instance, consider the following equation,

$$A\mathbf{x} - H^T(H\mathbf{x}) = (A - H^TH)\mathbf{x} \qquad A, H \in \mathbb{R}^{n\times n}, \mathbf{x} \in \mathbb{R}^n \tag{10}$$

The evaluation of LHS consist of three multiplication operations, i.e., three matrix-vector products of $\mathcal{O}(n^2)$ complexity, whereas the RHS, which although consist of only two multiplications, one of them is an expensive $\mathcal{O}(n^3)$ matrix-matrix multiplication; as a consequence, RHS requires more FLOPs than LHS despite the reduction in number of multiplications.

Result: The execution times for the LHS and RHS of the test equations are shown in Table V. In both frameworks, the execution of the LHS of Equation 9 is approximately $2\times$ longer than the RHS, and the RHS of Equation 10 is around $40\times$ longer than the LHS. As the execution times of LHS and RHS are not comparable, we infer that the frameworks do not rewrite expressions using distributivity to find a variant with fewer FLOPs.

- ***Blocked Matrices:*** We consider the expression where matrices $A_1, A_2 \in \mathbb{R}^{\frac{n}{2}\times\frac{n}{2}}$ and $B_1, B_2 \in \mathbb{R}^{\frac{n}{2}\times n}$ are used to form the following large blocked matrix structures:

$$A_B := \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} \in \mathbb{R}^{n\times n} \qquad B_B := \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \in \mathbb{R}^{n\times n}$$

Matrix products involving blocked matrices can be expressed as,

$$A_BB_B = \begin{bmatrix} (A_1B_1) \\ (A_2B_2) \end{bmatrix} \tag{11}$$

In many applications such as finite element methods [36], and signal processing [37], matrices exhibit such blocked structures. The matrix product A_BB_B with the large matrices (LHS of Equation 11) is more expensive than the product with individual blocks (RHS of Equation 11). In Equation 11, while the evaluation of LHS costs $2n^3$ FLOPs, the RHS that consist of matrix multiplications with the small blocks, costs only $\frac{n^3}{2} + \frac{n^3}{2} = n^3$ FLOPs.

Result: We construct the big matrix A_B by explicitly concatenating the small matrices, so that the construction is captured in the frameworks computational graph. For $n = 3000$, the execution of the LHS of Equation 11 is approximately $2\times$ longer than the RHS (see Table V). Therefore, we conclude that the frameworks do not exploit the blocked matrix structure to save computations.

Property	TF		PyT	
	LHS	RHS	LHS	RHS
Distributivity Eq [9]	0.78	040	0.81	0.41
Distributivity Eq [10]	0.01	0.42	0.01	0.41
Blocked matrices	0.40	0.20	0.40	0.20

TABLE V: Algebraic Manipulations. Execution time (in sec) for $n = 3000$.

Discussion: Derivation graphs can be used to systematically rewrite and explore the variants of an input expression. The nodes of this graph represent intermediate expressions and the edges represent the operations and its associated cost to transform the expression. Starting from the original input expression, child nodes are added as long as there exist an operation that can simplify the expression from the parent node. Upon termination, the different paths from root to leaves nodes are the alternative programs that compute the input problem and the total cost of a program can be obtained by summing the costs along the path. The program with minimum cost can be found by searching for the shortest path in the derivation graph. Linnea [10] is a linear algebra code generator that uses such derivation graph to generate variants of input expressions and find optimal programs in terms of FLOPs.

We remark that derivation graphs can serve as one of the top level intermediate representations in TF or PyT to facilitate algebraic optimizations.

E. Experiment 5: Code Motion

In this final experiment, we test if the frameworks can identify operations that can be moved around or swapped with another to improve performance.

- ***Loop-Invariant Code Motion:*** Operations that occur within a loop but yield the same result regardless of how many times the loop is executed, can be computed just once and moved outside the loop. The code snippet shown in Figure 8 updates the matrix product AB (where $A, B \in \mathbb{R}^{n \times n}$) with the outer products of the vectors $v_1, v_2, v_3 \in \mathbb{R}^n$.

Naive

```
V = [v1, v2, v3]
for i in range(3):
  Y[:,:,i] = A@B + V[i]@V[i]^T
```

Recommended

```
V = [v1, v2, v3]
tmp = A@B
for i in range(3):
  Y[:,:,i] = tmp + V[i]@V[i]^T
```

Fig. 8: Loop-Invariant code motion.

In the "naive" implementation, the expensive $\mathcal{O}(n^3)$ operation AB is recomputed in every iteration of the loop. However, the recommended implementation is to compute AB only once, outside the loop body.
Result: The execution times for the test expressions are shown in Table VI. The frameworks exhibit comparable performance for both naive and recommended implementations. Therefore, we conclude that both the frameworks perform loop-invariant code motion to move redundant operations outside the body of the loop.

- ***Partial Operand Access:*** In the code snippet shown in Figure 9, the output requires only a single element of the matrix operations: $A + B$ and AB.

Naive	**Recommended**
`#Sum` `Y = (A+B)[2,2]`	`#Sum` `Y = A[2,2]+B[2,2]`
`#Product` `Y = (A@B)[2,2]`	`#Product` `Y = dot(A[2,:],B[:,2])`

Fig. 9: Partial operand access.

The naive implementation consist of two operations: explicit matrix sum $A + B$ (or matrix product AB), followed by a slicing operation to access the required elements; however, the recommended implementation is to avoid the $\mathcal{O}(n^2)$ matrix sum (or the $\mathcal{O}(n^3)$ matrix product) by instead performing an $\mathcal{O}(1)$ element sum (or an $\mathcal{O}(n)$ dot product) of the required row and column elements (or slices) of A and B respectively.
Result: In both frameworks, the execution time for the naive implementations are significantly higher than the recommended implementation[9]. Hence, we infer that the frameworks do not swap the slicing and matrix operations to perform partial operand access.

Property	TF		PyT	
	Naive	Reco	Naive	Reco
Loop-inv code motion	0.42	0.42	0.42	0.41
Partial-op access (sum)	0.011	6e-4	0.018	2e-3
Partial-op access (product)	0.39	2e-3	0.40	3e-3

TABLE VI: Code Motion. Execution time (in sec) for $n = 3000$.

Discussion: The graph optimization systems of the frameworks are already known to perform graph re-writes that moves operations around to improve the performance for many neural network specific operations [18], [28]. Therefore, we remark that identifying code motions that speed up general linear algebra computations can further enhance the frameworks performance.

IV. Conclusion

We considered the evaluation of linear algebra expressions, which are at the heart of countless machine learning problems. We focused on two of the most popular machine learning frameworks: TensorFlow and PyTorch, and developed experiments that expose opportunities for improving the performance when computing linear algebra expressions. Specifically, we analyzed and reported the performance gains that could be obtained through optimizations such as common sub-expression elimination, optimal parenthesization of matrix chains, applying the knowledge of matrix and linear algebra properties, and code motion. For each optimization, we presented guidelines to help both the framework developers and end-users achieve better performance.

The experiments included in our benchmarks are meant to expose an initial set of optimizations that we deemed essential to attaining high performance in linear algebra computations. Natural extension to this study include further investigation on other critical aspects of linear algebra computations such as exploitation of properties in the solution of linear systems,

[9]Recall from Sec. III, the decorator overheads for TF and PyT are 6e-4 sec and 2-e3 sec respectively.

as well as the interplay of different optimizations, and taking into consideration the performance impacts due to the use of parallelism and accelerators.

References

[1] D. Grewe, M. Wagner, M. Arumaithurai, I. Psaras, and D. Kutscher, "Information-centric mobile edge computing for connected vehicle environments: Challenges and research directions," in Proceedings of the Workshop on Mobile Edge Communications, 2017, pp. 7–12.

[2] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," computer, vol. 50, no. 10, pp. 58–67, 2017.

[3] R.-S. Schmoll, S. Pandi, P. J. Braun, and F. H. Fitzek, "Demonstration of vr/ar offloading to mobile edge cloud for low latency 5g gaming application," in 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC). IEEE, 2018, pp. 1–3.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," Advances in neural information processing systems, vol. 32, pp. 8026–8037, 2019.

[6] Matlab, The MathWorks, inc. 2021. [Online]. Available: http://www.mathworks.com/.

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," SIAM review, vol. 59, no. 1, pp. 65–98, 2017.

[8] G. Guennebaud and B. Jacob, "others. eigen v3," URL http://eigen.tuxfamily. org, 2010.

[9] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," Journal of Open Source Software, vol. 1, no. 2, p. 26, 2016.

[10] H. Barthels, C. Psarras, and P. Bientinesi, "Linnea: Automatic generation of efficient linear algebra programs," ACM Transactions on Mathematical Software (TOMS), vol. 47, no. 3, pp. 1–26, 2021.

[11] A. Sankaran and P. Bientinesi, "Discriminating equivalent algorithms via relative performance," 2021.

[12] C. Psarras, H. Barthels, and P. Bientinesi, "The linear algebra mapping problem. current state of linear algebra languages and libraries," 2021.

[13] T. Tirer and R. Giryes, "Image restoration by iterative denoising and backward projections," IEEE Transactions on Image Processing, vol. 28, no. 3, pp. 1220–1234, 2018.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze et al., "{TVM}: An automated end-to-end optimizing compiler for deep learning," in 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018, pp. 578–594.

[15] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi et al., "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," arXiv preprint arXiv:1801.08058, 2018.

[16] C. Leary and T. Wang, "Xla: Tensorflow, compiled," TensorFlow Dev Summit, 2017.

[17] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," arXiv preprint arXiv:1802.04730, 2018.

[18] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein et al., "Glow: Graph lowering compiler techniques for neural networks," arXiv preprint arXiv:1805.00907, 2018.

[19] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," arXiv preprint arXiv:1512.01274, 2015.

[20] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 2135–2135.

[21] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 708–727, 2020.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004, pp. 75–86.

[23] Intel® oneAPI Deep Neural Network Library 2021, "onednn documentation." [Online]. Available: https://oneapi-src.github.io/oneDNN/

[24] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," arXiv preprint arXiv:1410.0759, 2014.

[25] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," ACM Transactions on Mathematical Software (TOMS), vol. 41, no. 3, pp. 1–33, 2015.

[26] A. Sankaran and P. Bientinesi, "Performance comparison for scientific computations on the edge via relative performance," in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021, pp. 887–895.

[27] B. Marker, D. Batory, and R. Van De Geijn, "Code generation and optimization of distributed-memory dense linear algebra kernels," Procedia Computer Science, vol. 18, pp. 1282–1291, 2013.

[28] R. M. Larsen and T. Shpeisman, "Tensorflow graph optimizations," 2019.

[29] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou et al., "Mlperf inference benchmark," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 446–459.

[30] "Baidu deepbench: Benchmarking deep learning operations on different hardware," https://github.com/baidu-research/DeepBench, 2018.

[31] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition," Training, vol. 100, no. 101, p. 102, 2017.

[32] Q. Zhang, L. Zha, J. Lin, D. Tu, M. Li, F. Liang, R. Wu, and X. Lu, "A survey on deep learning benchmarks: Do we still need new ones?" in Benchmarking, Measuring, and Optimizing, C. Zheng and J. Zhan, Eds. Cham: Springer International Publishing, 2019, pp. 36–49.

[33] J. Chung, M. Chung, J. T. Slagel, and L. Tenorio, "Stochastic newton and quasi-newton methods for large linear least-squares problems," arXiv preprint arXiv:1702.07367, 2017.

[34] E. Peise and P. Bientinesi, "A study on the influence of caching: Sequences of dense linear algebra kernels," in International Conference on High Performance Computing for Computational Science. Springer, 2014, pp. 245–258.

[35] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright et al., "Scipy 1.0: fundamental algorithms for scientific computing in python," Nature methods, vol. 17, no. 3, pp. 261–272, 2020.

[36] M. Blatt, "A parallel algebraic multigrid method for elliptic problems with highly discontinuous coefficients," Ph.D. dissertation, 2010.

[37] J. Y. Park, H. L. Yap, C. J. Rozell, and M. B. Wakin, "Concentration of measure for block diagonal matrices with applications to compressive signal processing," IEEE Transactions on Signal Processing, vol. 59, no. 12, pp. 5859–5875, 2011.