Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning and HPC Workloads

Evangelos Georganas[1]*, Dhiraj Kalamkar[1], Sasikanth Avancha[1], Menachem Adelman[1], Deepti Aggarwal[1], Cristina Anderson[1], Alexander Breuer[2], Jeremy Bruestle[1], Narendra Chaudhary[1], Abhisek Kundu[1], Denise Kutnick[1], Frank Laub[1], Vasimuddin Md[1], Sanchit Misra[1], Ramanarayan Mohanty[1], Hans Pabst[1], Brian Retford[1], Barukh Ziv[1] and Alexander Heinecke[1]

[1] Intel Corporation, Santa Clara, CA, United States, [2] Faculty of Mathematics and Computer Science, Friedrich-Schiller-Universität Jena, Jena, Germany

During the past decade, novel Deep Learning (DL) algorithms, workloads and hardware have been developed to tackle a wide range of problems. Despite the advances in workload and hardware ecosystems, the programming methodology of DL systems is stagnant. DL workloads leverage either highly-optimized, yet platform-specific and inflexible kernels from DL libraries, or in the case of novel operators, reference implementations are built *via* DL framework primitives with underwhelming performance. This work introduces the Tensor Processing Primitives (TPP), a programming abstraction striving for efficient, portable implementation of DL workloads with high-productivity. TPPs define a compact, yet versatile set of 2D-tensor operators [or a virtual Tensor Instruction Set Architecture (ISA)], which subsequently can be utilized as building-blocks to construct complex operators on high-dimensional tensors. The TPP specification is platform-agnostic, thus, code expressed *via* TPPs is portable, whereas the TPP implementation is highly-optimized and platform-specific. We demonstrate the efficacy and viability of our approach using standalone kernels and end-to-end DL & High Performance Computing (HPC) workloads expressed entirely *via* TPPs that outperform state-of-the-art implementations on multiple platforms.

Keywords: deep learning, performance portability, programming abstraction, tensor processing, high productivity, high performance computing

1. INTRODUCTION

Since the advent of Deep Learning (DL) as one of the most promising machine learning paradigms almost 10 years ago, deep neural networks have advanced the fields of computer vision, natural language processing, recommender systems, and gradually pervade an increasing number of scientific domains [1–10]. Due to the diverse nature of the problems under consideration, these DL workloads exhibit a wide range of computational characteristics and demands. Furthermore, due to the immense computational cost of such workloads, industry and academia have developed specialized hardware features on commodity processors, and even specialized accelerators in order to harness these computational needs [11].

In contrary to the fast-evolving ecosystems of DL workloads and DL-oriented hardware/accelerators, the programming paradigm of DL systems has reached a plateau [12]. More specifically, the development of novel DL workloads involves two types of components: (i) Well-established operators within DL libraries (e.g., 2D convolutions, inner-product, batch-norm layers in oneDNN [13] and cuDNN [14]), and (ii) Unprecedented, custom primitives which typically instantiate new algorithmic concepts/computational motifs. Unfortunately both of these components come with their shortcomings.

On one hand, the operators within DL libraries are heavily optimized and tuned (usually by vendors) in a platform-specific fashion, leading to monolithic, non-portable, and inflexible kernels. Additionally, such opaque and high-level operators prohibit modular design choices since the user/frameworks have to adhere to particular interfaces that may not be adapted to fit the operation under consideration. On the other hand, the custom/unprecedented primitives are typically implemented by the user *via* the available generic/reference primitives of a Machine Learning (ML) framework which are not optimized and as such yield underwhelming performance. It is up to the user to create optimized implementations for the custom primitives, leading again to code which is non-portable and potentially requires hardware expertise in order to achieve peak performance. Unfortunately, most of the times such expertise is not available to the data/ML scientist who is developing the custom DL primitive. Therefore, the deployment (or even the evaluation) of a new operator typically requires yet another stage in the development cycle where low-level optimization experts are working on the re-write/fine-tuning of the operator. Later on, in case an operator proves to be important for the community, systems researchers and vendors standardize it, and potentially create yet another monolithic kernel within a DL library for further re-use within DL frameworks. This entire development cycle potentially takes a considerable amount of time (up to years in some cases) and inadvertently impedes the efficient exploration of innovative machine learning ideas [12]. An alternative approach to optimize both types of operators is to leverage contemporary Tensor Compilers (TC) (e.g., [15–18]), however, the state-of-the-art tools are only suitable for compiling small code-blocks whereas large-scale operators require prohibitive compilation times, and often the resulting code performs far from the achievable peak [12].

We identify that the common source of the problems mentioned in the previous paragraph is the extreme levels of abstraction offered by the DL libraries and the Tensor Compilers. The DL libraries offer coarse-grain, monolithic and inflexible operators whereas the Tensor Compilers usually go to the other extreme, allowing the user to express arbitrary low-level operators without any minimal restrictions that would readily enable efficient lifting and code-generation in their back-ends (e.g., they offer no minimal/compact set of allowed operations on tensors). To exacerbate the challenge of optimal code generation, Tensor Compilers usually undertake the cumbersome tasks of efficient parallelization, loop re-ordering, automatic tiling and layout transformations, which, to date, remain unsolved in the general setup. Also, there is not a well-established

way to share state-of-the-art optimizations among the plethora of Tensor Compilers and as a result each one has its own advantages and disadvantages, which translates eventually to sub-optimal performance on real-world scenarios [19]. We note, here, the recent, promising effort of Multi-Level Intermediate Representation (MLIR) [20] toward unifying the optimization efforts in the Tensor Compiler Intermediate Representation (IR) infrastructure.

In this work, we introduce the Tensor Processing Primitives (TPP), a programming abstraction striving for efficient and portable implementation of Tensor operations, with a special focus on DL workloads. TPPs define a set of relatively low-level primitive operators on 2D Tensors, which, in turn, can be used as basic building blocks to construct more complex operators on high-dimensional tensors. TPPs comprise a minimal and compact, yet expressive set of precision-aware, 2D tensor level operators to which high-level DL operators can be reduced. TPPs's specification is agnostic to targeted platform, DL framework, and compiler back-end. As such the code which is expressed in terms of TPPs is portable. Since the level of abstraction that TPPs adopt is at the sub-tensor granularity, TPPs can be directly employed by DL workload developers within the frameworks, or could be alternatively used to back up an IR within a Tensor Compiler stack, i.e., TPPs could form the basis of an MLIR dialect.

While the TPP specification is agnostic of the targeted framework/platform/compiler stack, its implementation is platform specific, and is optimized for the target architectures. This subtle detail offers a clear separation of concerns: the user-entity of TPPs, either a developer or a compiler framework, can focus on expressing the desired algorithm and its execution schedule (e.g., parallelization, loop orders) using the TPP tensor abstraction, whereas the efficient, platform-specific code generation pertaining to the TPP operations belongs to the TPP back-end. To this extent, TPPs could be also viewed as a "virtual Tensor ISA" that abstracts the actual physical ISA of the target (e.g., SSE, AVX2, AVX512, AMX for x86, AArch64 and ARMv8 SVE, xPU).

Figure 1 shows various use-cases of TPPs within multiple software stacks. TPPs can be viewed as a layer abstraction of the actual physical target ISA, and the user-entities can rely on the TPP layer for the code generation pertaining to the tensor operations. Also, **Figure 1** illustrates the various user-entities that might leverage TPPs. First, the vendor-optimized DL libraries (e.g., oneDNN or oneDNN Graph) can use TPPs for optimized code generation in their back-end. Second, the user/developer of the DL operators can directly leverage TPPs within a DL framework extension to express the underlying tensor computations (e.g., the user may develop a framework extension for a novel DL operator by employing the TPPs as building blocks). Third, Tensor Compilers can leverage TPPs (e.g., as part of an MLIR dialect) to generate high-quality code for the corresponding tensor operators. As such, the TPP layer abstraction offers a clear separation of concerns where the Tensor Compiler may focus on higher-level optimizations (loop tiling and re-ordering, parallelization, etc.) whereas the platform-specific code generation of the tensor operations is undertaken

FIGURE 1 | Use-cases of TPPs in various software stacks.
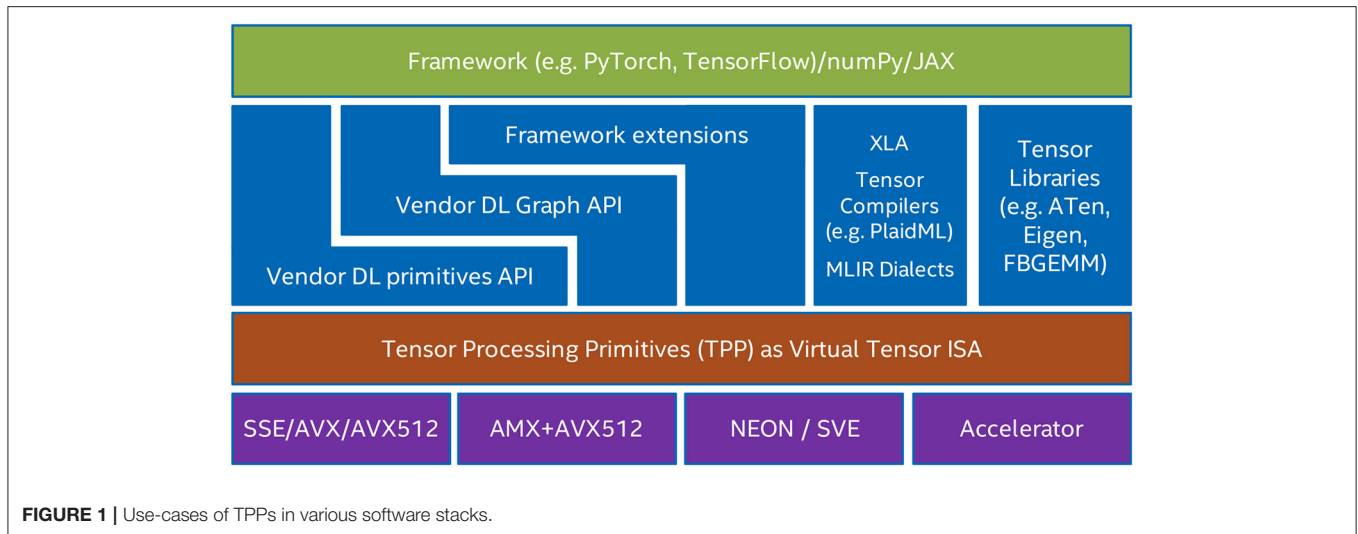
by the TPP layer. Such a synergistic Tensor Compiler - TPP paradigm is illustrated in section 7. Last but not least, TPPs could be leveraged by more general Tensor Libraries (e.g., ATen, Eigen) where tensor computations constitute the primary focus and they can be naturally mapped to TPPs.

In our Proof-Of-Concept (POC) implementation of TPPs we leverage Just-In-Time (JIT) technology to emit performant and platform-specific code during runtime. Furthermore, in our POC we define a mini embedded Domain Specific Language (mini-eDSL) where the TPPs can be combined *via* matrix equations in order to build high-level operators without sacrificing performance.

We demonstrate the efficacy of our approach on multiple platforms using standalone kernels written entirely with TPPs and compare the performance to vectorized-by-expert code and compiler generated code. Finally, we showcase the expressiveness and viability of our methodology by implementing contemporary end-to-end DL workloads using solely the TPP abstractions and show how we can outperform the state-of-the-art implementations on multiple platforms. The main contributions of this work are:

- A TPP specification/foundation for primitive tensor operations.
- A Proof-Of-Concept implementation of the TPP specification along with a mini-eDSL (called TPP Matrix Equations), enabling efficient fusion of TPPs that lead to portable, high-level tensor operations. We describe in detail various standalone TPP implementations, and also we provide a detailed analysis of our TPP Matrix Equation mini-eDSL framework.
- A demonstration of how contemporary and novel DL algorithmic motifs/workloads can be expressed in their entirety *via* TPPs.
- An experimental evaluation of the TPP-based DL workloads from all relevant fields (image processing, recommendation systems, natural language processing, graph processing, and applications in science) on multiple platforms (different

instruction set architectures (ISAs) x86_64 and aarch64, and micro-architectures for each ISA), including distributed-memory scaling. We show performance that matches/exceeds the state-of-the-art implementations, while maintaining flexibility, portability, and obviating the need for low-level platform-specific optimizations.
- We show how TPPs can be leveraged as a virtual Tensor ISA within a Tensor compiler software stack, yielding high-performance DL primitives.
- We illustrate examples of how TPPs are used outside of Deep Learning, in High Performance Computing (HPC) applications in order to accelerate tensor computations.

Section 2 details the specification of the TPPs. Then, section 3 illustrates a POC implementation of the TPP specification. Section 4 presents an infrastructure that enables efficient TPP fusion. In section 5, we exhibit how contemporary DL motifs/algorithmic paradigms can be expressed *via* TPPs. Section 6 presents an experimental evaluation of TPP-based DL kernels and workloads on multiple platforms. Section 7 outlines our POC implementation of a TPP backend within a tensor compiler (PlaidML), and also presents some results highlighting the viability of the TPP abstraction as a virtual Tensor ISA within tensor compiler stacks. Section 8 presents exemplary usage of TPPs within HPC applications in order to efficiently implement tensor computations. Sections 9 and 10 summarize the related work and conclude this article.

2. THE TPP SPECIFICATION

2.1. TPP Design Principles

The TPP specification is driven by a few design principles:

1) *Each TPP corresponds to a mathematical operator that takes a number of input(s) and produces an output*. We opt to specify TPPs that correspond to basic, well-defined mathematical tensor operations. In this way, we keep the set of TPPs *minimal* albeit *expressive*; basic TPPs can be combined to formulate more complex operators.

2) *The inputs/outputs of the TPPs are abstract 2D tensors that can be fully specified by their shape/size, leading dimensions, and precision.* Additionally, the 2D tensors hold the following complementary *runtime* information: (i) a *primary* field which corresponds to the memory address where the 2D (sub)tensor data resides, (ii) a *secondary* field holding optional data for the tensor (e.g., a mask for the tensor), and (iii) a *tertiary* field holding optional, auxiliary information of the tensor (e.g., scaling factors for a quantized tensor).

3) *TPPs are specified as "memory-to-memory" operations, or equivalently the input/output tensors are residing in memory locations specified by the user.* This design decision is critical in order to abstract the TPPs from all physical ISAs, and enables true platform-agnostic specification. For example, if the TPPs were accepting vector registers as inputs/outputs, then the number of physical registers, the vector length and dimensionality would be exposed in the Application Programming Interface (API) of TPPs, making the specification platform-specific.

4) *TPPs have declarative semantics.* As such, the TPP specification does not preclude potential parallelism [e.g., Single Instruction Multiple Data (SIMD), Single Instruction Multiple Threads (SIMT)] in the back-end implementation which is target-specific.

5) *TPPs are composable in a producer-consumer fashion.* Since the output of a TPP is a well-defined tensor O, it can be fed as input to a subsequent TPP. In such a scenario, this "intermediate" tensor O is not necessarily exposed to the user, unless the user explicitly requires it (e.g., by combining the TPPs in a manual fashion *via* an explicit temporary O buffer/tensor which lives in the user space/application). This flexibility allows the TPP implementation (which is platform-specific) to combine TPPs in the most efficient way for the target architecture (e.g., the O tensor can live at the physical register file in the composite TPP in order to avoid redundant memory movement).

6) *The TPP input/output tensors as well as the computation itself are precision aware.* This feature makes mixed precision computations (that are prominent in DL workloads) easy to express from the user point of view, and provides information to the TPP back-end that may enable efficient implementation.

2.2. TPP Arguments

As mentioned in the previous subsection, the input to TPPs are 2D tensors. Each 2D tensor can be specified by the number of rows M, columns N, its leading dimension ld and its datatype *dtype*. Additionally, during runtime each tensor gets fully characterized by specifying its location/address as *primary* info, optional companion tensor info as *secondary* (e.g., sparsity bitmask), and optionally *tertiary* info (e.g., in case the tensor shape is dynamically determined at runtime, this info may contain variables specifying M/N). Each TPP also specifies the shape/precision of the produced/output 2D tensor.

Each TPP also supports input tensors with *broadcast* semantics. More specifically, TPPs accept optional flags dictating that the input 2D tensor should be formed by broadcasting a column/row/scalar $N/M/M \times N$ times, respectively. Finally, the TPPs accept optional flags which further specify the TPP operation. For example, in case a TPP is computing a

transcendental function, the flags may be specifying various approximation algorithms used for the computation. In the next subsection, we present the TPPs in three groups: *unary*, *binary*, and *ternary* TPPs given the number of input tensors they accept.

2.3. The TPP Collection

First, we highlight the ternary *Batch-Reduce GEneral Matrix to Matrix Multiplication* (BRGEMM) TPP which is the main building block for general tensor contractions in DL kernels [21]. BRGEMM materializes the operation $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$. In essence, this kernel multiplies the specified blocks $A_i^{M \times K}$ and $B_i^{K \times N}$ and reduces the partial results to a block $C^{M \times N}$. It is noteworthy that tensors A and B can alias and also the blocks A_i and B_i can reside in any position in the input (potentially high-dimensional) tensors A and B. Previous work [21] has shown that this single building block is sufficient to express efficiently tensor contractions in the most prevalent DL computational motifs, namely: Convolution Neural Networks (CNN), Fully-Connected networks (FC), Multi-Layer Perceptrons (MLP), Recurrent Neural Networks (RNN)/Long Short-Term Memory (LSTM) Networks. In Section 5 we exhibit how BRGEMM can be further used to build efficient Attention Cells that comprise the cornerstone of modern Natural Language Processing (NLP) workloads. BRGEMM can be specialized to one of the following three variants that may enable more efficient implementations on various platforms: (i) *address-based BRGEMM*, where the addresses of the blocks A_i and B_i are explicitly provided by the user, (ii) *offset-based BRGEMM*, where the addresses of A_i and B_i can be computed as $address\_A_i = address\_A + offset_{A_i}$ and $address\_B_i = address\_B + offset_{B_i}$, and (iii) *stride-based BRGEMM*, where the addresses of A_i and B_i are: $address\_A_i = address\_A_{i-1} + stride\_A$ and $address\_B_i = address\_B_{i-1} + stride\_B$. In section 3.2, we present the implementation of the BRGEMM TPP in more depth for various ISAs and platforms.

Table 1 presents the unary TPPs that accept one 2D tensor as input. Since most of these TPPs map directly to the equivalent math function, we further elaborate only on the ones which are more complex. The *Identity* TPP essentially copies the input to the output. Since the input and output are fully specified in terms of their precision, this TPP can be also used to perform datatype conversions between tensors.

The *Quantize & Dequantize* TPPs are used to quantize/dequantize the input tensor whereas the exact algorithm employed is specified by a TPP flag.

The *Transform* TPP uses a flag to determine the exact transformation applied on the input 2D tensor. The *Transpose* transformation is the usual mathematical matrix transpose. The rest two types of transformation, namely *Vector Neural Network Instructions (VNNI) formatting*, and *VNNI to VNNI-transpose* are DL specific. More specifically, modern hardware (e.g., Intel's Cooper Lake) requires tensors to be in specific format called *VNNI* in order to employ hardware acceleration for specific operations, e.g., dot-products (see section 3.2.2 for more details). This format represents a logical 2D tensor $[D_1][D_0]$ as a 3D tensor $[D_1/\alpha][D_0][\alpha]$ where essentially the dimension D_1 is blocked in chunks of size α, which in turn are set as the inner-most tensor dimension. The *VNNI*

TABLE 1 | Unary TPPs.

Unary TPP	Description/Comments
Identity	Copies input to output. Given input/output datatype, it performs datatype conversions
Zero	Fills output with zeros
Square	Squares input and stores to output
Increment / decrement	Increments / Decrements input by 1 and stores to output
Square root	Computes the square root of input and stores to output
Reciprocal	Computes the reciprocal of input and stores to output
Rcp. Sqrt.	Computes the rcp. sqrt. of input and stores to output
Exp	Computes the exponential value of the input tensor entries and stores them to output
PRNG	Generates an output tensor with pseudo-random entries
(De)Quantize	Quantizes / Dequantizes the input
Reduce	Reduces the rows/columns of the input and stores to output. The reduction function can be SUM/MUL/MIN/MAX; (optionally) reduces the *squared* input
Transform	Transforms input and stores to output. Transformations are: Transpose, VNNI formatting, and VNNI to VNNI-transpose
Unpack	Takes each entry $x_{i,j}$ of the input tensor, splits it in two parts $x_{i,j}^{lo}$ and $x_{i,j}^{hi}$ with same bit-width, and stores them in two tensors X^{lo}, X^{hi}
Replicate columns	Takes an input column/vector, replicates it a variable number of times and forms the output
Gather / Scatter	Gathers/Scatters rows/columns from input and forms the tensor
2D Gather / 2D Scatter	Gathers/scatters elements from input using 2D offsets
2D-strided loads/stores	Loads/stores elements from/to a tensor using primary and secondary strides
Tanh &Tanh_inv	Computes the hyperbolic tangent function (or its inv used for back-propagation) on input
RELU & RELU_inv	Apply a Rectified Linear Unit function (or its inv used for back-propagation) on input
Sigmoid & Sigmoid_inv	Computes the logistic sigmoid (or its inv used for back-propagation) on input
GELU & GELU_inv	Apply a Gaussian Error Linear Unit function (or its inv used for back-propagation) on input
Dropout & Dropout_inv	Drops out values from the input tensor with probability p. For the inv/back-propagation pass, the same dropped units are zeroed out

TABLE 2 | Binary TPPs.

Binary TPP	Description/Comments
Add	Add two inputs
Sub	Subtracts two inputs
Mul	Multiples (elementwise) two inputs
Div	Divides two inputs
Max/Min	Finds element-wise max/min of two inputs
MatMul	Performs matrix multiplication of two input
Pack	Concatenates pairs of entries $x_{i,j}^{lo}$ and $x_{i,j}^{hi}$ from the inputs X^{lo}, X^{hi} into $x_{i,j}$ and stores it to the output X
Compare	Compares element-wise two inputs and stores a bitmask of the comparison

TABLE 3 | Ternary TPPs.

Ternary TPP	Description/Comments
GEMM	Performs on 2D inputs A, B, C, scalar β: $C = \beta C + A \times B$
Batch-Reduce GEMM	Performs on 2D inputs A_i, B_i (with $i = 0, 1, \ldots, n-1$), C, scalar β: $C = \beta C + \sum_{i=0}^{i=n-1} A_i \times B_i$
(N)MulAdd	Performs on 2D inputs A, B, C: $C = C + A \odot B$ (or $C = C - A \odot B$); $\odot$ denotes element-wise multiplication
Blend	Blends 2D input tensors A, B according to bitmask C

of non-linear approximations for several activation functions on various ISAs.

Tables 2, **3** present the binary/ternary TPPs that accept two/three 2D tensor as inputs, respectively.

3. TPP IMPLEMENTATION

In this section, we briefly describe our Proof-Of-Concept (POC) implementation of the TPP specification. Our implementation targets multiple CPU architectures from various vendors that support different ISAs, but could be readily extended to support even GPU ISAs. We build upon and extend the open source LIBXSMM [22] library which leverages JIT techniques. Such JIT techniques have been successfully used for optimal code generation on CPUs by taking advantage of the known (at runtime) tensor shapes/dimensions in HPC and DL applications [21–23]. Nevertheless, the TPP specification is platform-agnostic and does not preclude any TPP back-end implementation. In our POC implementation, the usage of TPPs is governed by two APIs: (i) A dispatch API with which the user can request the code generation of a specific TPP, and such a dispatch call JITs a function implementing the requested operation, (ii) an API to call the JITed TPP kernel. First, in section 3.1, we provide a generic blueprint of our TPP implementation. Then, in section 3.2, we describe in more detail the BRGEMM TPP implementation which comprises the main tensor contraction tool in the TPP abstractions. Section 3.3.1 details the implementation of the unary transform TPPs *via* shuffle networks since their efficient implementation diverts from the generic TPP blueprint. Finally, section 3.3.2

formatting TPP performs this exact transformation: $[D_1][D_0] \rightarrow [D_1/\alpha][D_0][\alpha]$ and the *VNNI to VNNI-transpose* transposes a tensor which is already laid out in VNNI format, i.e., performs $[D_1/\alpha_1][D_0][\alpha_1] \rightarrow [D_0/\alpha_0][D_1][\alpha_0]$. In section 3.3.1, we outline how the Transform TPPs are implemented *via* Shuffle Networks.

The last four entries of **Table 1** correspond to DL-specific operations. They correspond to activation functions typically encountered in DL workloads. All these activation functions have a counterpart which is required during the back-propagation pass of training DL networks. These DL specific TPPs could be built on top of other TPPs, however, since they are prevalent in DL workloads we opt to define them as self-contained TPPs for ease of usage. In section 3.3.2, we describe the TPP implementation

---

Algorithm 1 | The generic unary/binary/ternary TPP algorithm.

Inputs: $\mathbf{X}^{M \times N}$, ($\mathbf{Y}^{M \times N}$, $\mathbf{Z}^{M \times N}$ if binary/ternary)
Output: $\mathbf{O}^{M \times N}$

1: **for** $i_n = 0 \ldots N - 1$ **with step n_b do**
2:    **for** $i_m = 0 \ldots M - 1$ **with step m_b do**
3:       ▷ *Generic loads, may have broadcast/gather semantics,*
4:       ▷ *and may perform datatype conversions*
5:       $X_b \leftarrow$ load_generic $m_b \times n_b$ X-subblock$_{i_m,i_n}$
6:       **if** (unary TPP) **then**
7:          $X_b \leftarrow$ **Unary_op**(X_b)
8:       **if** (binary TPP) **then**
9:          $Y_b \leftarrow$ load_generic $m_b \times n_b$ Y-subblock$_{i_m,i_n}$
10:         $X_b \leftarrow$ **Binary_op**(X_b, Y_b)
11:       **if** (ternary TPP) **then**
12:         $Y_b \leftarrow$ load_generic $m_b \times n_b$ Y-subblock$_{i_m,i_n}$
13:         $Z_b \leftarrow$ load_generic $m_b \times n_b$ Z-subblock$_{i_m,i_n}$
14:         $X_b \leftarrow$ **Ternary_op**(X_b, Y_b, Z_b)
15:    ▷ *Generic store, may have scatter semantics, and may*
16:    ▷ *perform datatype conversion*
17:    O-subblock$_{i_m,i_n}$ $\overleftarrow{\text{store\_generic}}$ X_b

---

Algorithm 2 | The batch-reduce GEMM TPP.

Inputs: $A_i^{M \times K}$, $B_i^{K \times N}$ for $i = 0, \ldots, n\text{-}1$, $C^{M \times N}$, $\beta \in \mathbb{R}$
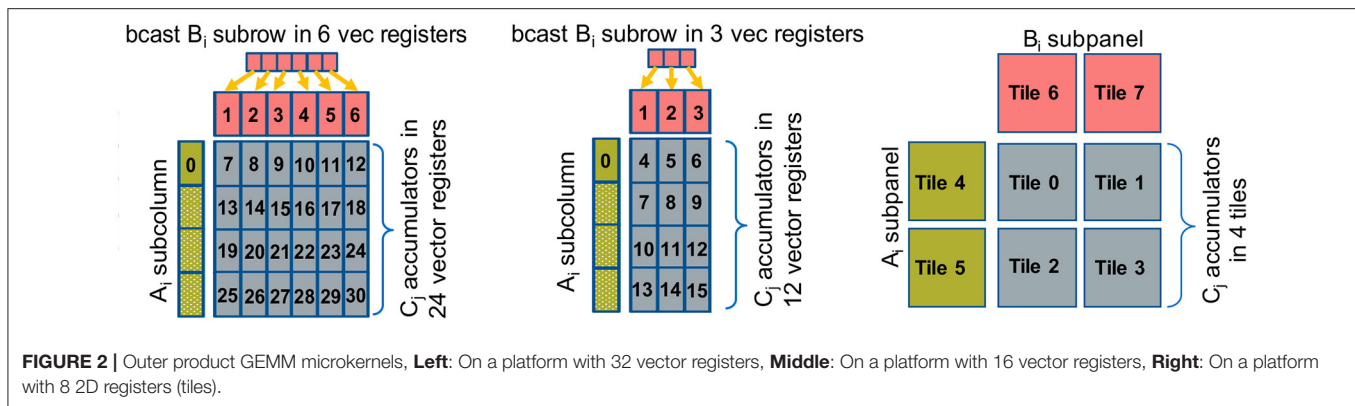**Output:** $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$

1: **for** $i_n = 0 \ldots N - 1$ **with step n_b do**
2:    **for** $i_m = 0 \ldots M - 1$ **with step m_b do**
3:       acc_regs $\leftarrow$ load_generic $m_b \times n_b$ C-subblock$_{i_m,i_n}$
4:       **for** $i = 0 \ldots n - 1$ **with step 1 do**
5:          **for** $i_k = 0 \ldots K - 1$ **with step k_b do**
6:             ▷ *Outer product GEMM microkernel*
7:             acc_regs += A_i sub-panel$_{i_m,i_k} \times B_i$ sub-panel$_{i_k,i_n}$
8:       C-subblock$_{i_m,i_n}$ $\overleftarrow{\text{store\_generic}}$ acc_regs

---

outlines the approximation techniques we leverage in our TPP implementation of non-linear activation functions; such approximations are essential in achieving high-performance, while at the same time their accuracy is sufficient for the purposes of training DL workloads.

3.1. Generic TPP Implementation Blueprint

Algorithm 1 exhibits at a high-level the pseudocode that is used to implement the Unary/Binary/Ternary TPPs in a unified fashion. The inputs of the TPPs are tensors X, Y (in case of binary/ternary TPPs) and Z (in case of ternary TPP), and an output tensor O. For the purposes of this simplified presentation we assume all tensors are of size $M \times N$, however, depending on the operation these might have different sizes. For example, if the unary OP is a reduction-by-columns and the input is $M \times N$, then the output is an $M \times 1$ vector. First, we show that the M/N loops are blocked with factors m_b/n_b such that the working sets of each microkernels fits on the available register file. The latter is architecture specific, e.g., AVX2-enabled ISAs expose 16 256-bit vector registers, AVX512-enabled ISAs expose 32 512-bit vector registers, and Aarch64 features 32 128-bit (NEON)/512-bit (SVE) vector registers. The "load_generic" function used in **Algorithm 1** denotes the loading of a sub-tensor to a register block; this load may imply row/column/scalar broadcast semantics if the user specified the TPP in that way, or even may have strided-load/gather semantics if the TPP involves a strided-load/gather operation. Also, for simplicity we do not show here the handling of "secondary" fields of the tensors that may be required (e.g., indices array for the gather operation, bitmasks arrays). Additionally, the generic load also handles datatype conversion, for instance provided the input is in bfloat16 (BF16) [24] whereas the compute is going to happen in FP32 precision. Once all

the required sub-tensors are loaded, then the corresponding Unary/Binary/Ternary operator is applied. This operator may be directly mapped to an available instruction (e.g., a vector add in case of binary addition), or to a sequence of instructions for more complicated operators (e.g., reductions, random number generation *via* xorshift algorithm [25], approximation algorithms for transcendental functions [26]). Last but not least, the optimal sequence generation depends on the available instructions and this is handled by the TPP back-end/JITer. For example, some ISAs may have masking/predicate support (e.g., AVX512 & SVE) that enable efficient handling of loop remainders, the selected unrolling degree heavily depends on the instructions in use, their latency and the number of available architectural registers. Once the result is computed, the resulting register block is stored back to the corresponding output sub-tensor position. Similarly to the generic load, the "generic" store may induce strided accesses or may be even a scatter operation. Additionally, the generic store also handles potential datatype conversions.

3.2. The BRGEMM TPP Implementation
3.2.1. The BRGEMM Kernel Structure

We present in more detail the BRGEMM TPP because it comprises the tensor contraction tool in the TPP abstraction, and is ubiquitous in the DL kernels and workloads described in section 5. **Algorithm 2** exhibits the high-level algorithm implementing: $C = \beta \cdot C + \sum_{i=0}^{n-1} A_i \times B_i$. Lines 1-2 block the computation of the result C in $m_b \times n_b$ tensor sub-blocks. Once such a subblock is loaded into the accumulation registers (line 3), we loop over all pairs A_i, B_i (line 4) and we accumulate into the loaded registers the products of the corresponding $m_b \times K$ subblocks of A_i with the relevant $K \times n_b$ subblocks of B_i (lines 5–7). In order to calculate a partial product of an $m_b \times k_b$ sub-panel of A_i with a $k_b \times n_b$ sub-panel of B_i, we follow an outer product formulation. The loading of A_i and B_i sub-panels, and the outer-product formulation is heavily dependent on the target platform. We provide BRGEMM implementations for multiple x86 ISAs: SSE, AVX, AVX2, AVX512, including the recently introduced Intel AMX (Advanced Matrix Extensions) ISA [27]. Additionally, we have implemented the BRGEMM TPP for AArch64 and ARMv8 SVE ISAs. Depending on the targeted platform, the "register" can be either a typical vector register with varying width (e.g., 128–512 bit vector length), or

FIGURE 2 | Outer product GEMM microkernels, **Left**: On a platform with 32 vector registers, **Middle**: On a platform with 16 vector registers, **Right**: On a platform with 8 2D registers (tiles).

in the case of AMX-enabled target the "register" is a 2D tile-register. Similarly, the outer-product formulation may employ the available Fused-Multiply-Add (FMA) instructions, or even 2D tile-multiplication instructions. In all these cases, the TPP implementation emits the appropriate load/store/prefetch/FMA instructions, and takes into account the available architectural registers/unrolling factors/instruction mix in order to achieve close to peak performance. Last but not least, the BRGEMM supports multiple datatypes (FP64, FP32, BF16, INT8), and whenever possible employs hardware acceleration, e.g., *via* specialized FMA instructions for INT8/BF16 datatypes. In order to highlight the differences of the outer product GEMM microkernels that are heavily dependent on the target platform, we show in **Figure 2** three different implementations.

Figure 2-Left shows an exemplary outer product microkernel on a platform with 32 available vector registers, for example an x86 with AVX512 or on ARM AArch64/SVE. In this case vector register v7-v30 constitute the accumulators, vector registers v1-v6 hold a broadcasted subrow of B, and vector register v0 is used to load a partial subcolumn of A. First, we load on v1-v6 a subrow of B *via* broadcasts, then we load on v0 the first chunk of the A subcolumn and with six fused multiply-add (FMA) instructions (v0 with v1-v6) we multiply-and-add the corresponding partial results on the accumulators v7-v12 (first logical row of accumulators). Then, we load on v0 the second chunk of the A subcolumn, and subsequently with yet another six FMA instructions (v0 with v1-v6) we multiply-and-add the computed partial results on the accumulators v13-v18 (second logical row of accumulators), etc. The registers v1-v6 are reused four times throughout the outer product computation, and v0 is reused six times for each loaded A chunk. In other words, the corresponding A subcolumn and B subrow are loaded from memory/cache into the vector registers exactly once and we get to reuse them from the register file. Also, in such a formulation, we expose 24 independent accumulation chains which is critical in order to hide the latency of the FMA instruction. Last but not least, the platform (i.e., vector register width) and the datatype of the microkernel determine the exact values of the blocking parameters m_b, n_b, and k_b. For example for single precision datatype FP32 and an x86 AVX512 platform, each vector register can hold 16 FP32 values (the vector registers are 512-bit wide).

Therefore, this microkernel operates with blocking values $m_b = 64$, $n_b = 6$, and $k_b = 1$ and it calculates a small matrix multiplication $C_{64 \times 6} += A_{64 \times 1} \times B_{1 \times 6}$.

Figure 2-Middle shows an exemplary outer product microkernel on a platform with 16 vector registers, for example an x86 with up to AVX2 ISA. The microkernel is similar with the previous case; since we have only 16 vector registers available, we dedicate 12 of those as C accumulators, 3 vector register are utilized for holding a partial B subrow, and 1 vector register is used to load a chunk of an A subcolumn. In this case 12 independent accumulation chains are also sufficient to hide the FMA latency. Analogously to the previous case, for single precision datatype FP32 and an x86 AVX2 platform, each vector register can hold now 8 FP32 values (the vector registers are now 256-bit wide). Thus, this microkernel operates with blocking values $m_b = 32$, $n_b = 3$, and $k_b = 1$ and it calculates a small matrix multiplication $C_{32 \times 3} += A_{32 \times 1} \times B_{1 \times 3}$.

Figure 2-Right shows a small GEMM microkernel on a platform with 8 2D registers (tiles), for example what is available in the recently introduced Intel AMX (Advanced Matrix Extensions) ISA. In this case each 2D tile register has size (up to) 1KB, logically holds (up to) 16 rows of a submatrix, and can be loaded with a proper tile-load instruction. In this particular example, tiles 0-3 comprise the C accumulators, tiles 4-5 are used to hold a subpanel of A and tiles 6-7 are used to hold a subpanel of B. Once we load the subpanels of A and B onto the respective tiles, we can perform 4 tile multiply-and-add instructions: *tile*0 += *tile*4 × *tile*6, *tile*1 += *tile*4 × *tile*7, *tile*2 += *tile*5 × *tile*6 and *tile*3 += *tile*5 × *tile*7, and we update the C accumulators. In such a microkernel, each A/B tile is reused 2 times. Given each tile may have size up to 1KB and may hold up to 16 rows of a submatrix, by considering BF16 datatype for A/B matrices and FP32 accumulator tiles, such a microkernel operates with blocking values $m_b = 32$, $n_b = 32$, $k_b = 32$, and can compute (up to) a small matrix multiplication $C_{32 \times 32} += A_{32 \times 32} \times B_{32 \times 32}$. Each A/B tile represents a logical 16 × 32 BF16 A/B submatrix, and each C tile represents a 16 × 16 FP32 accumulator. The AMX instructions will be available within the upcoming Intel Xeon processors code-named Sapphire Rapids, and the corresponding BF16-input/FP32-output tile multiplication instructions can deliver
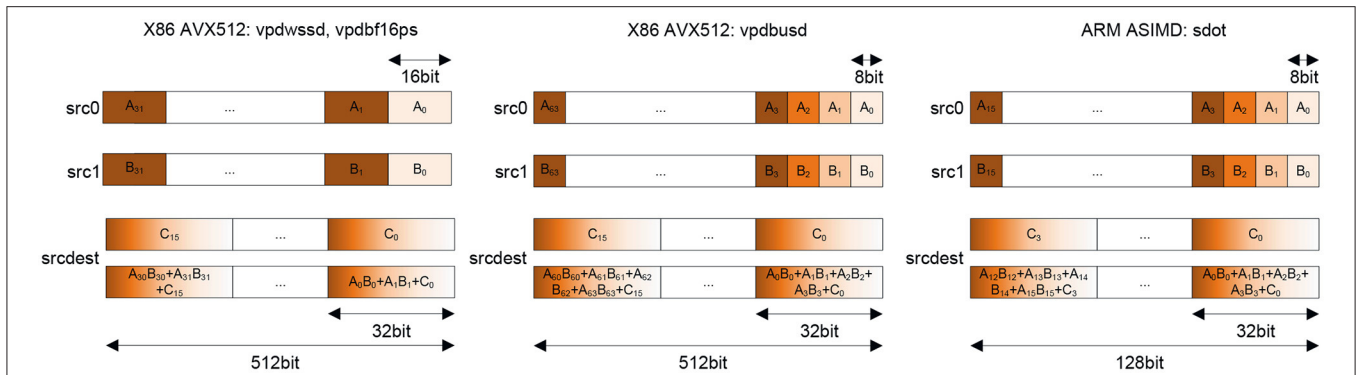
FIGURE 3 | Mixed-precision dot-product instructions, **Left**: 16 bit integer and bfloat16 on Intel AVX512, **Middle**: 8bit integer using Intel AVX512, **Right**: 8 bit integer using ARM ASIMD.

up to $16\times$ more FLOPs/cycle compared to FP32 AVX512 FMA instructions on current Xeon platforms.

These considerably different GEMM microkernel variants highlight yet another aspect of the TPPs: The TPPs specify *what* needs to be done rather than how it is done/implemented. In this case, the user may just specify/employ a BRGEMM TPP in order to perform a tensor contraction, whereas the TPP backend/implementation is responsible for generating the optimal code for each platform at hand. In this methodology, all the architectural nuances are hidden completely by the user, and the same exact user code written in terms of TPPs may be reused across platforms with different characteristic/ISAs without sacrificing performance or portability.

3.2.2. Mixed Precision BRGEMM and Its Emulation

While the previous section presents the general structure of mapping matrix multiplication to various physical ISAs, this paragraph is used to demonstrate how the idea of a virtual ISA allows to implement operations efficiently which are not natively supported by a specific physical ISA. The example we are choosing here is our GEMM kernel and its support for bfloat16 and int8 on architectures which do not support these novel ISA SIMD-extension.

Before going into the details of the emulation, we first need to introduce special memory layouts which are used by x86 and aarch64 mixed-precision dot-product instructions as shown in **Figure 3**. As we can see in all cases (x86/aarch64 and bf16/int8), the overall concept is identical: Although doing mixed-precision and mixed-datatype-length computations, these instructions are functioning from a matrix multiplication point-of-view similar to 32 bit instructions. This is achieved by having an implicit 2-wide (BF16/int16) and 4-wide (int8) dot-product of A_i and B_i values leading to a horizontal summation per single 32 bit C_i, e.g., $C_0 = A_0 \cdot B_0 + A_1 \cdot B_1 + A_2 \cdot B_2 + A_3 \cdot B_3 + C_0$ as shown for the int8 variant. If we apply blockings with these instructions as discussed in **Figure 2-Left, Middle**, then we realize that matrix B is still read *via* 32-bit broadcast (containing 2 16-bit or 4 8-bit values along the inner-product or common dimension). However, matrix A is in need of reformatting. This is due to the

fact that the GEMM kernel in **Figure 2-Left, Middle** requires full SIMD-width contiguous loads for optimal performance (which is along M and not K). Therefore, we need to reformat A into $[K^o][M][K^i]$ with $K^o \cdot K^i = K$ and $K^i = 2$ for 16-bit and $K^o = 4$ for 8-bit inputs. We refer to such a format as *VNNI-format* throughout this article. After such reformatting of A, we can perform full SIMD loads on A; combined with the 32-bit broadcast loads on B we have a 32-bit GEMM kernel which has a shorter K dimension, $2\times$ for 16-bit datatypes and $4\times$ for 8-bit datatypes.

In case these novel instructions are not available, especially for bfloat16 as this is a relatively new format, one might think, that an efficient mapping to a classic FP32 SIMD ISA is not possible. This is correct as long as the machine does not offer int16 support. However, with int16 support and SIMD masking we can implement the aforementioned non-trivial mixed-precision dot-product efficiently and even bit-accurately as shown in **Figure 4**. This is done by processing K^i in two rounds in the case of bfloat16 datatype. As shown in **Figure 4**, we first process the odd (or upper) bfloat16 number. This is done by exploiting the fact that a bfoat16 number perfectly aliases with an FP32 number in its 16 MSBs. Therefore, on AVX512 we can just execute a full SIMD load as a 16-bit-typed load with masking. As a mask we chose `0xaaaaaaaa` and as masking-mode we use zero masking. With this trick we automatically turn on-load the upper bfloat16 numbers in A into 16 valid FP32 numbers, and for B we broadcast and then perform an overriding register move. A little bit more work is needed for the lower/even bfloat16 number: In this case, we perform an unmasked load and then we use a 32-bit integer shift by 16 to create valid FP32 numbers. A simple inspection of the instruction sequence in **Figure 4** shows that we are mainly executing fused-multiply-add instructions with little overhead compared to a classic FP32 GEMM as illustrated in **Figure 2-Left, Middle**. Therefore, we can execute a bfloat16 GEMM with a reformatted matrix A with close to FP32-peak and still benefit from the smaller memory footprint (and, therefore, a small performance gain, as we will show later in section 6). Replacement sequences for int16 and int8 matrix inputs can be carried out in a similar way and their detailed discussion is skipped here.

```
__m512 c_0_0, c_0_1, c_0_2, c_0_3;
...
__m512 c_5_0, c_5_1, c_5_2, c_5_3;
__m512 a_0, a_1, a_2, a_3;
__m512 b_0, b_1, b_2, b_3, b_4, b_5;
__mmask32 m_0 = 0xaaaaaaaa;
/* load C */
...
/* load upper part of A & B and FP32 FMAs*/
a_0 = _mm512_maskz_loadu_epi16( m_0, ptrA );
a_1 = _mm512_maskz_loadu_epi16( m_0, ptrA+32 );
a_2 = _mm512_maskz_loadu_epi16( m_0, ptrA+64 );
a_3 = _mm512_maskz_loadu_epi16( m_0, ptrA+96 );
b_0 = _mm512_set1_epi32( ptrB );
b_0 = _mm512_maskz_mov_epi16( m_0, b_0 );
c_0_0 = _mm512_fmadd_ps( a_0, b_0, c_0_0 );
c_0_1 = _mm512_fmadd_ps( a_1, b_0, c_0_1 );
c_0_2 = _mm512_fmadd_ps( a_2, b_0, c_0_2 );
c_0_3 = _mm512_fmadd_ps( a_3, b_0, c_0_3 );
b_1 = _mm512_set1_epi32( ptrB+ldb );
b_1 = _mm512_maskz_mov_epi16( m_0, b_1 );
c_1_0 = _mm512_fmadd_ps( a_0, b_1, c_1_0 );
c_1_1 = _mm512_fmadd_ps( a_1, b_1, c_1_1 );
c_1_2 = _mm512_fmadd_ps( a_2, b_1, c_1_2 );
c_1_3 = _mm512_fmadd_ps( a_3, b_1, c_1_3 );
b_2 = _mm512_set1_epi32( ptrB+2*ldb );
b_2 = _mm512_maskz_mov_epi16( m_0, b_2 );
c_2_0 = _mm512_fmadd_ps( a_0, b_2, c_2_0 );
c_2_1 = _mm512_fmadd_ps( a_1, b_2, c_2_1 );
c_2_2 = _mm512_fmadd_ps( a_2, b_2, c_2_2 );
c_2_3 = _mm512_fmadd_ps( a_3, b_2, c_2_3 );
b_3 = _mm512_set1_epi32( ptrB+3*ldb );
b_3 = _mm512_maskz_mov_epi16( m_0, b_3 );
c_3_0 = _mm512_fmadd_ps( a_0, b_3, c_3_0 );
c_3_1 = _mm512_fmadd_ps( a_1, b_3, c_3_1 );
c_3_2 = _mm512_fmadd_ps( a_2, b_3, c_3_2 );
c_3_3 = _mm512_fmadd_ps( a_3, b_3, c_3_3 );
```

```
b_4 = _mm512_set1_epi32( ptrB+4*ldb );
b_4 = _mm512_maskz_mov_epi16( m_0, b_4 );
c_4_0 = _mm512_fmadd_ps( a_0, b_4, c_4_0 );
c_4_1 = _mm512_fmadd_ps( a_1, b_4, c_4_1 );
c_4_2 = _mm512_fmadd_ps( a_2, b_4, c_4_2 );
c_4_3 = _mm512_fmadd_ps( a_3, b_4, c_4_3 );
b_5 = _mm512_set1_epi32( ptrB+5*ldb );
b_5 = _mm512_maskz_mov_epi16( m_0, b_5 );
c_5_0 = _mm512_fmadd_ps( a_0, b_5, c_5_0 );
c_5_1 = _mm512_fmadd_ps( a_1, b_5, c_5_1 );
c_5_2 = _mm512_fmadd_ps( a_2, b_5, c_5_2 );
c_5_3 = _mm512_fmadd_ps( a_3, b_5, c_5_3 );
/* load lower part of A & B and FP32 FMAs*/
a_0 = _mm512_loadu_epi16( ptrA );
a_1 = _mm512_loadu_epi16( ptrA+32 );
a_2 = _mm512_loadu_epi16( ptrA+64 );
a_3 = _mm512_loadu_epi16( ptrA+96 );
a_0 = _mm512_slli_epi32( a_0, 16 );
a_1 = _mm512_slli_epi32( a_1, 16 );
a_2 = _mm512_slli_epi32( a_2, 16 );
a_3 = _mm512_slli_epi32( a_3, 16 );
b_0 = _mm512_set1_epi32( ptrB );
b_0 = _mm512_slli_epi32( b_0, 16 );
c_0_0 = _mm512_fmadd_ps( a_0, b_0, c_0_0 );
c_0_1 = _mm512_fmadd_ps( a_1, b_0, c_0_1 );
c_0_2 = _mm512_fmadd_ps( a_2, b_0, c_0_2 );
c_0_3 = _mm512_fmadd_ps( a_3, b_0, c_0_3 );
b_1 = _mm512_set1_epi32( ptrB+ldb );
b_1 = _mm512_slli_epi32( b_1, 16 );
c_1_0 = _mm512_fmadd_ps( a_0, b_1, c_1_0 );
c_1_1 = _mm512_fmadd_ps( a_1, b_1, c_1_1 );
c_1_2 = _mm512_fmadd_ps( a_2, b_1, c_1_2 );
c_1_3 = _mm512_fmadd_ps( a_3, b_1, c_1_3 );
```

```
b_2 = _mm512_set1_epi32( ptrB+2*ldb );
b_2 = _mm512_slli_epi32( b_2, 16 );
c_2_0 = _mm512_fmadd_ps( a_0, b_2, c_2_0 );
c_2_1 = _mm512_fmadd_ps( a_1, b_2, c_2_1 );
c_2_2 = _mm512_fmadd_ps( a_2, b_2, c_2_2 );
c_2_3 = _mm512_fmadd_ps( a_3, b_2, c_2_3 );
b_3 = _mm512_set1_epi32( ptrB+3*ldb );
b_3 = _mm512_slli_epi32( b_3, 16 );
c_3_0 = _mm512_fmadd_ps( a_0, b_3, c_3_0 );
c_3_1 = _mm512_fmadd_ps( a_1, b_3, c_3_1 );
c_3_2 = _mm512_fmadd_ps( a_2, b_3, c_3_2 );
c_3_3 = _mm512_fmadd_ps( a_3, b_3, c_3_3 );
b_4 = _mm512_set1_epi32( ptrB+4*ldb );
b_4 = _mm512_slli_epi32( b_4, 16 );
c_4_0 = _mm512_fmadd_ps( a_0, b_4, c_4_0 );
c_4_1 = _mm512_fmadd_ps( a_1, b_4, c_4_1 );
c_4_2 = _mm512_fmadd_ps( a_2, b_4, c_4_2 );
c_4_3 = _mm512_fmadd_ps( a_3, b_4, c_4_3 );
b_5 = _mm512_set1_epi32( ptrB+5*ldb );
b_5 = _mm512_slli_epi32( b_5, 16 );
c_5_0 = _mm512_fmadd_ps( a_0, b_5, c_5_0 );
c_5_1 = _mm512_fmadd_ps( a_1, b_5, c_5_1 );
c_5_2 = _mm512_fmadd_ps( a_2, b_5, c_5_2 );
c_5_3 = _mm512_fmadd_ps( a_3, b_5, c_5_3 );
/* store C*/
...
```

FIGURE 4 | Emulation of a bit accurate GEMM kernel using AVX512F instructions matching a GEMM kernel as depicted in **Figure 2** using vdpbf16ps AVX512 instructions. The glossary contains detailed descriptions of the used intrinsic functions.

In addition to the presented emulation of mixed-precision GEMM kernels using SIMD instructions, we have also added support for emulation of Intel AMX instructions bit-accurately on AVX512. This addition enables running numerical accuracy experiments, such as convergence studies, before the release of a chip that supports Intel AMX instructions. A similar path is possible for ARM's SME instruction set and subject to future work. These emulation capabilities further highlight the aspect of TPP as a virtual tensor ISA.

3.3. Examples of Non-trivial Non-GEMM TPPs

The previous sections covered most of the TPP implementations: straightforward element-wise unary/binary/ternary operations and various forms of mixed precision GEMMs including their emulation on older hardware. However, there are cases in which we are not operating on the data in an element-wise fashion, e.g., transpose, or the Unary_op, Binary_op, or Ternary_op is not an elementary operation. The goal of this section is to shed some light on these cases by presenting the transpose TPP in detail, and sketching fast non-linear approximations on SIMD machines that match the accuracy requirements of deep learning applications.

3.3.1. Transform-Transpose TPP *via* Shuffle Networks
When working with matrices, the transpose kernel is ubiquitous. It is needed to access the matrix's elements in various contractions along the mathematically correct dimension. However, a transpose operation is scalar at first sight. In this subsection we exhibit how transpose can be implemented using shuffle networks in a fully vectorized fashion, e.g., **Figure 5** demonstrates how a 16×16 matrix with 256 32-bit elements can be transposed in 64 cycles using AVX512 instructions.

The shuffle-network presented in **Figure 5** is a blueprint for all datatype-lengths and ISAs: in $\log_2$ *SIMD-Length* stages we can transpose a matrix held in a set of SIMD registers. In this particular example, we need $\log_2 16 = 4$ stages and in each stage we increase the shuffling/interleaving width of logical elements, and also increase the distance at which we access the 32 registers grouped into two sets of 16 registers each. More specifically, we start with registers i_0 to i_{15} and interleave elements at the same position in a pair of registers close to each other. This constructs now pairs of 32 bit values in o_0 and o_1 which are already containing the transpose's result for 2 out of 16 elements and we repeat this for all other 7 input register pairs. The analogous transformation is now repeated in the second stage with 64-bit values and accessing o_0 and o_2 as input pair pattern. This constructs a new set output registers i_0 and i_1 which are holding the transpose's result at 128-bit granularity. After that, stage 3 is shuffling at 128-bit granularity on register pairs which have a distance of "4" and creates output registers that hold 256-bit of transposed data. Finally, in stage 4, these 256-bit transposed input registers are shuffled once again creating the final set of 16 register holding the transposed 16×16 matrix. For non-square
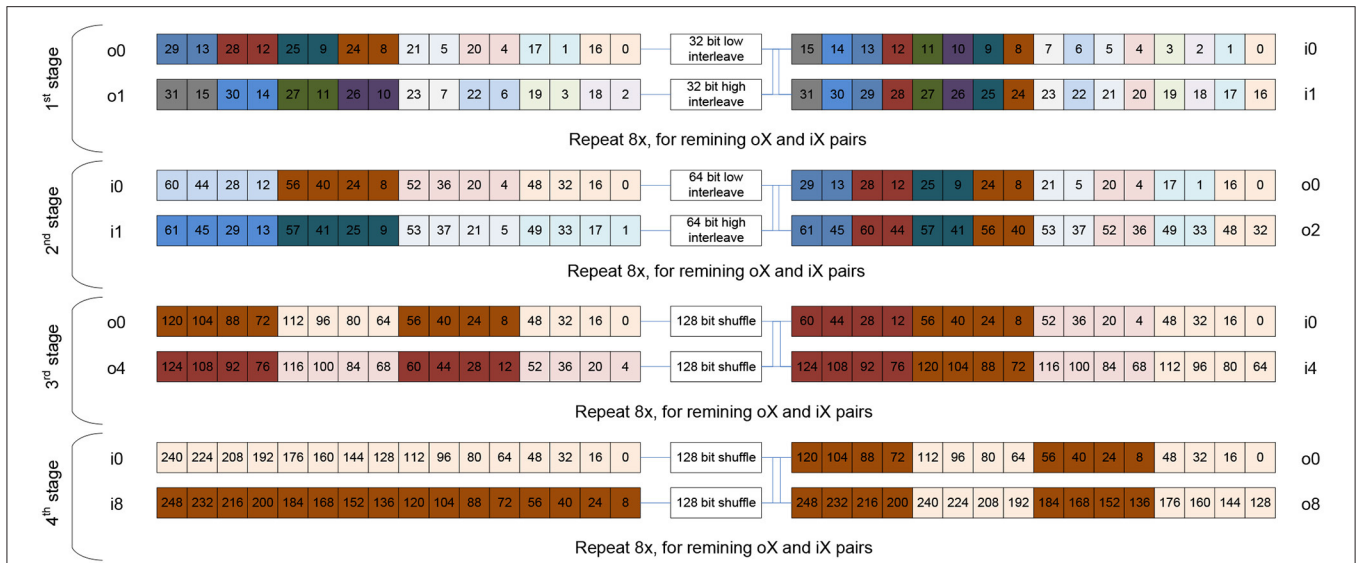
FIGURE 5 | Sketch of a shuffle network for a 32-bit transpose of a 16×16 matrix using Intel AVX512 instructions. *Via* four stages (each one having 16 independent shuffles that double in width per stage), the 16×16 matrix (256 elements) can be transposed with only 64 instructions and fully leverages the 32 architectural registers.

X86 Code

```
__m128 r0, r1, r2, r3, t0, t1, t2, t3;
r0 = _mm_loadu_ps(in + 0*ldi); // 0  1  2  3
r1 = _mm_loadu_ps(in + 1*ldi); // 4  5  6  7
r2 = _mm_loadu_ps(in + 2*ldi); // 8  9  10 11
r3 = _mm_loadu_ps(in + 3*ldi); // 12 13 14 15
t0 = _mm_unpacklo_ps(r0,r1);   // 0  4  1  5
t1 = _mm_unpackhi_ps(r0,r1);   // 2  6  3  7
t2 = _mm_unpacklo_ps(r2,r3);   // 8  12 9  13
t3 = _mm_unpackhi_ps(r2,r3);   // 10 14 11 15
r0 = _mm_unpacklo_pd(t0,t2);   // 0  4  8  12
r1 = _mm_unpackhi_pd(t0,t2);   // 1  5  9  13
r2 = _mm_unpacklo_pd(t1,t3);   // 2  6  10 14
r3 = _mm_unpackhi_pd(t1,t3);   // 3  7  11 15
_mm_storeu_ps(out + 0*ldo, r0);
_mm_storeu_ps(out + 1*ldo, r1);
_mm_storeu_ps(out + 2*ldo, r2);
_mm_storeu_ps(out + 3*ldo, r3);
```

ARM Code

```
float32x4_t r0, r1, r2, r3, t0, t1, t2, t3;
r0 = vld1q_f32(in + 0*ldi); // 0  1  2  3
r1 = vld1q_f32(in + 1*ldi); // 4  5  6  7
r2 = vld1q_f32(in + 2*ldi); // 8  9  10 11
r3 = vld1q_f32(in + 3*ldi); // 12 13 14 15
t0 = vtrn1q_f32(r0,r1);     // 0  4  1  5
t1 = vtrn2q_f32(r0,r1);     // 2  6  3  7
t2 = vtrn1q_f32(r2,r3);     // 8  12 9  13
t3 = vtrn2q_f32(r2,r3);     // 10 14 11 15
r0 = vtrn1q_f64(t0,t2);     // 0  4  8  12
r1 = vtrn2q_f64(t0,t2);     // 1  5  9  13
r2 = vtrn1q_f64(t1,t3);     // 2  6  10 14
r3 = vtrn2q_f64(t1,t3);     // 3  7  11 15
vst1q_f32(out + 0*ldo, r0);
vst1q_f32(out + 1*ldo, r1);
vst1q_f32(out + 2*ldo, r2);
vst1q_f32(out + 3*ldo, r3);
```
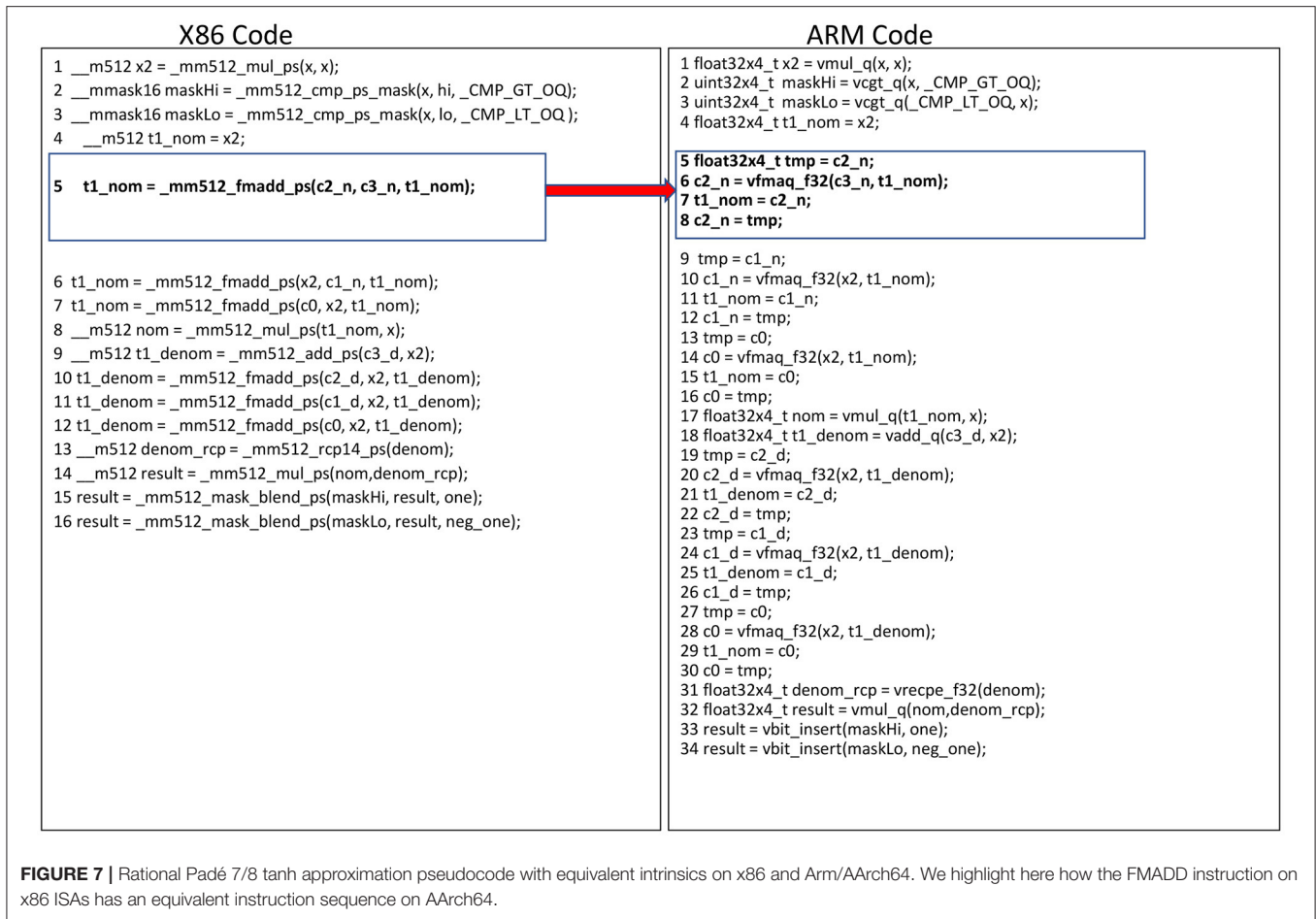
FIGURE 6 | Comparison of X86 and ARM code for a simple 4×4 single precision transpose using unpack instructions. The glossary contains detailed descriptions of the used intrinsic functions.

matrices we (a) just use masked loads or set registers to zero, (b) transpose the zeros as well, and then (c) do not store all registers or employ masked stores. This basic kernel is used as a basic building block to create large transpose operators by simply adding outer loops.

This algorithm can be implemented by any SIMD ISA which offers support for picking arbitrary values from a pair of SIMD registers to construct a result register containing values from the two sources, i.e., a general shuffler. However, "structured" shuffle instructions are adequate as shown in **Figure 6**. Both x86 and aarch64 offer instructions exactly implementing the needs for 32-bit and 64-bit interleaves as needed in the first two stages covered in the previous description. In the case of 128-bit-wide SIMD

registers this is enough to carry out the entire transpose of 4 × 4 matrices as shown in **Figure 6**.

Finally, we want to note that broadcast loads, as supported by various ISAs, can be used to implement the first stage of the shuffle network. This has the advantage that one stage of the shuffle network can be executed faster and in parallel to the shuffler. The shuffle operations needed in all of these networks are relatively expensive in hardware, therefore modern CPUs often only provide one execution unit for such operations (such "shuffle-viruses" like transposes are pretty rare in general code). However, broadcasts on the load path are cheap and can run in parallel to the shuffle unit, hence the overall performance of the transpose operation improves. This microkernel variation leads

FIGURE 7 | Rational Padé 7/8 tanh approximation pseudocode with equivalent intrinsics on x86 and Arm/AArch64. We highlight here how the FMADD instruction on x86 ISAs has an equivalent instruction sequence on AArch64.

to relatively complex code, and as such we skip its presentation. However our TPP implementation back-end employs all these microkernel variations.

3.3.2. Approximations for Non-linear TPP Activation Functions

Activation functions are used to represent non-linear behavior of neural networks. Popular known activation functions are sigmoid, tanh and Gaussian Error Linear Unit (GELU). These activation functions can be approximated to increase the efficiency of deep learning networks without effecting its non-linear characteristics. In this section, we will discuss different approximation techniques based on Padé rational polynomials, piecewise minimax polynomials and Taylor expansions, along with their TPP implementation on different ISAs. For simplicity we present the relevant algorithms in terms of x86 and arm intrinsics (see glossary for the semantics of these intrinsics), however the actual TPP implementation relies on JIT code generation.
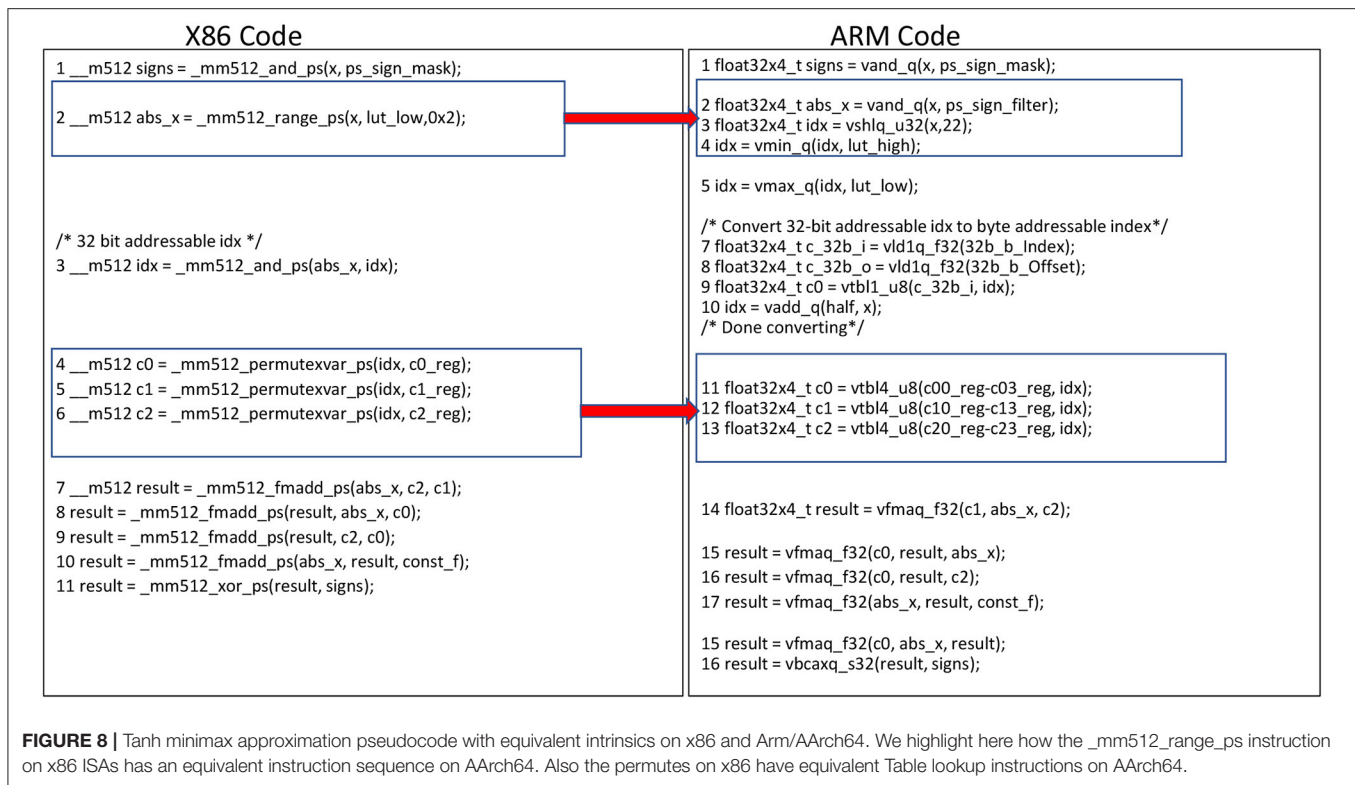
3.3.2.1. Rational Padé Polynomials

The Padé approximation of a function f is the ratio of two polynomials with degrees p and q:

$$Padé_{[p/q]f}(x) = \frac{\sum_{i=0}^{p} a_i x^i}{\sum_{i=0}^{q} b_i x^i}$$

The coefficients a_i and b_i can be calculated by considering the first $p + q$ derivatives of f at zero and solving the corresponding system of equations:

$$f(0) = Padé_{[p/q]f}(0)$$
$$f'(0) = Padé'_{[p/q]f}(0)$$
$$\vdots$$
$$f^{(p+q)}(0) = Padé^{(p+q)}_{[p/q]f}(0)$$

As an example we consider the approximation of the tanh function which has two asymptotes, hence approximating it with a Taylor expansion of lower degree polynomials may not yield good results. The implementation of the $Padé_{[7/8]}(x)$ tanh approximation is shown in **Figure 7**. FMA operations are used to compute the numerators and denominators *via* Horner's rule. The reciprocal of the denominator is multiplied by the numerator to get the final result. The accuracy of reciprocal instruction is different among different CPU's. This difference in accuracy does

X86 Code	ARM Code

```
1 __m512 signs = _mm512_and_ps(x, ps_sign_mask);

2 __m512 abs_x = _mm512_range_ps(x, lut_low,0x2);




/* 32 bit addressable idx */
3 __m512 idx = _mm512_and_ps(abs_x, idx);




4 __m512 c0 = _mm512_permutexvar_ps(idx, c0_reg);
5 __m512 c1 = _mm512_permutexvar_ps(idx, c1_reg);
6 __m512 c2 = _mm512_permutexvar_ps(idx, c2_reg);


7 __m512 result = _mm512_fmadd_ps(abs_x, c2, c1);
8 result = _mm512_fmadd_ps(result, abs_x, c0);
9 result = _mm512_fmadd_ps(result, c2, c0);
10 result = _mm512_fmadd_ps(abs_x, result, const_f);
11 result = _mm512_xor_ps(result, signs);
```

```
1 float32x4_t signs = vand_q(x, ps_sign_mask);

2 float32x4_t abs_x = vand_q(x, ps_sign_filter);
3 float32x4_t idx = vshlq_u32(x,22);
4 idx = vmin_q(idx, lut_high);

5 idx = vmax_q(idx, lut_low);

/* Convert 32-bit addressable idx to byte addressable index*/
7 float32x4_t c_32b_i = vld1q_f32(32b_b_Index);
8 float32x4_t c_32b_o = vld1q_f32(32b_b_Offset);
9 float32x4_t c0 = vtbl1_u8(c_32b_i, idx);
10 idx = vadd_q(half, x);
/* Done converting*/

11 float32x4_t c0 = vtbl4_u8(c00_reg-c03_reg, idx);
12 float32x4_t c1 = vtbl4_u8(c10_reg-c13_reg, idx);
13 float32x4_t c2 = vtbl4_u8(c20_reg-c23_reg, idx);


14 float32x4_t result = vfmaq_f32(c1, abs_x, c2);

15 result = vfmaq_f32(c0, result, abs_x);
16 result = vfmaq_f32(c0, result, c2);
17 result = vfmaq_f32(abs_x, result, const_f);

15 result = vfmaq_f32(c0, abs_x, result);
16 result = vbcaxq_s32(result, signs);
```

FIGURE 8 | Tanh minimax approximation pseudocode with equivalent intrinsics on x86 and Arm/AArch64. We highlight here how the _mm512_range_ps instruction on x86 ISAs has an equivalent instruction sequence on AArch64. Also the permutes on x86 have equivalent Table lookup instructions on AArch64.

not affect the non-linear region of the tanh function, keeping the TPP behavior same across different CPU's. The sigmoid activation function can be approximated *via* tanh by leveraging the following identity:

$$sigmoid(x) = (\tanh(x/2) + 1)/2$$

3.3.2.2. Piecewise Minimax Polynomial Approximations

In this section, we discuss the minimax polynomials approach [28] with the truncated Chebyshev series [29] for approximations of activation functions. In this approach, the input range of a function $f(x)$ is divided into intervals and for each interval $[a, b]$ we find a polynomial p of degree max n to minimize:

$$\max_{a \leq x \leq b} |f(x) - p(x)|$$

We approximate tanh and GELU activation functions using this approach in our TPP implementation. The input range is divided into 16 intervals and for each interval we investigate a polynomial p of 3rd degree (i.e., we find appropriate p's coefficients c_0, c_1, c_2 based on the minimized absolute maximum difference of f and p). **Figure 8** shows the x86 and arm implementation of evaluating such minimax polynomials. The register index (idx) is calculated using the exponent and Most Significant Bit (MSB) of the respective input values, and represents the 16 intervals where the input values are located. The range intrinsic _mm512_range_ps(A,B) is used to generate the register index (idx) on AVX512 platforms (**Figure 8-Left**, line 2). In ARM,

the range functionality is emulated with equivalent and, shlq, min and max instructions as shown in **Figure 8-Right**, lines 2–4. To evaluate the 3rd degree polynomial we need to locate 3 coefficients (c0,c1,c2) based on the values at the register index (idx), which holds 16 entries. We use 3 look up operations to find the three coefficients, each involving 16 FP32 entries. The 512-bit register length in AVX512 is sufficient to hold 16 coefficients required for each look up, resulting in using 3 registers for 3 look up operations (see **Figure 8-Left**, lines 4–6). Each ARM 128-bit wide vector register can only hold 4 FP32 entries, subsequently we are using 12 vector registers to hold the 16 entries for all 3 coefficients of the polynomial. The in-register look-up table is performed using _mm512_permutexvar_ps(A,B) instructions in x86 AVX512 as shown in **Figure 9**. In ARM we have byte addressable table look up instructions which are analogous to 32-bit addressable permutes instructions in x86. Hence, we need to convert the 32-bit addressable (0–16) register indexes to byte addressable (0-64 bytes) indexes. In order to do that, we use a constant register A with a table look up instruction to duplicate the register index (idx) to each byte in the 32-bit entry. A constant offset (0,1,2,3) is added to the duplicated byte index to get the byte addressable index for each FP32 entry in 16 FP32 entries (**Figure 8-Right**, lines 7–9). The table look up instruction in ARM provides the 64 byte look up capability, which is sufficient enough to search into 4 registers holding the 16 entries of each coefficient; we are using the generated byte indexes as shown in **Figure 10**. Finally, 4 FMA operations are used to evaluate the polynomial using Horner's rule. The FMA instruction in x86 provides the user the flexibility to decide among the sources to
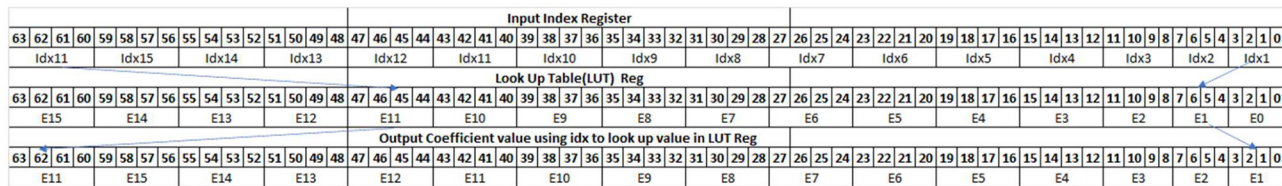
FIGURE 9 | 32Bit addressable Table look up setup on x86 AVX512 platforms.
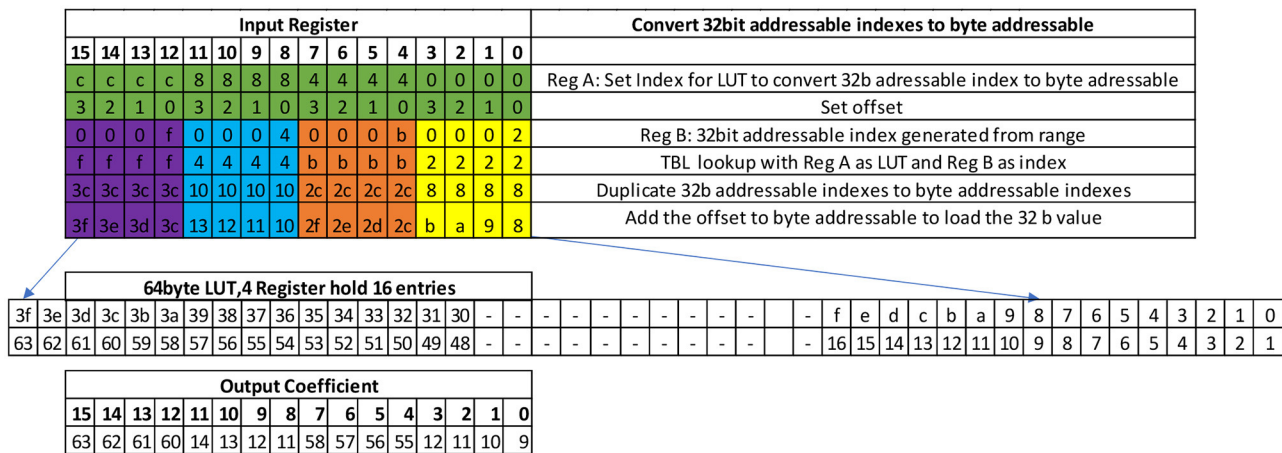

FIGURE 10 | Byte addressable table look up setup in ARM/AArch64. We highlight the conversion of 32bit indexes to byte indexes and the use of byte indexes to get the coefficients in 16 FP32 intervals.

destroy and the ones to preserve. ARM requires mov instructions to save intermediate results in order to avoid the data overwriting during FMA operations.

3.3.2.3. Approximation With Taylor Series

As an example of approximation with Taylor series we illustrate here the exp() activation function. The e^x is approximated using the identity $e^x = 2^{x\log_2 e} = 2^{n+y} = 2^n \cdot 2^y$ with $n = round(x\log_2 e)$ and $y = x\log_2 e - n$. We need to calculate 2^n with n being an integer and the term 2^y with $|y| \in [0, 1)$. A Taylor polynomial of third degree is used to calculate the term 2^y with 3 FMA instructions (see **Figure 11**-**Left**, lines 4–6). Once 2^y is calculated, we leverage the instruction `_mm512_scalef_ps(A,B)` which returns a vector register holding $a_i \cdot 2^{floor(b_i)}$ for each $a_i \in A$ and $b_i \in B$. This scale instruction concludes the exp() approximation on x86 with AVX512. On ARM we calculate 2^n and 2^y with equivalent replacement instructions as shown in **Figure 11**.

4. TPP MATRIX EQUATIONS

One of the main design principles of TPPs (as described in section 2.1) is that they can be composed in a producer-consumer fashion to form complex operations. For example consider the scenario where a user wants to implement the composite

operation $C = Tanh(A + B)$. One way to express this *via* TPPs would be to allocate an intermediate tensor *tmp* with same shape as A and B, and perform first $tmp = Add(A, B)$ *via* the binary Add TPP. Then the user can compute the final result by leveraging the Tanh Unary TPP: $C = Tanh(tmp)$. Even though this approach is functionally correct, it requires the explicit management of intermediate tensors/buffers by the user and also may result in low performance since there are redundant loads/stores to the *tmp* tensor.

In order to increase the productivity, efficiency and expressiveness pertaining to composite operators, we implemented an embedded Domain Specific Language (eDSL) in LIBXSMM [22]. Our Proof-Of-Concept implementations allows the user to express the desired composite operator as a Matrix Equation. More specifically, the user can express the composite operator as an equation tree, where the head and internal nodes are the available TPPs, whereas the leaves of the tree are the input 2D tensors of the composite operation. In the next subsections, we describe in detail the methodology we employ for JITing matrix equations of TPPs.

4.1. Definitions and Notations for TPP Matrix Equations

A TPP matrix equation is represented as a tree with unary/binary/ternary TPP operations as internal nodes and
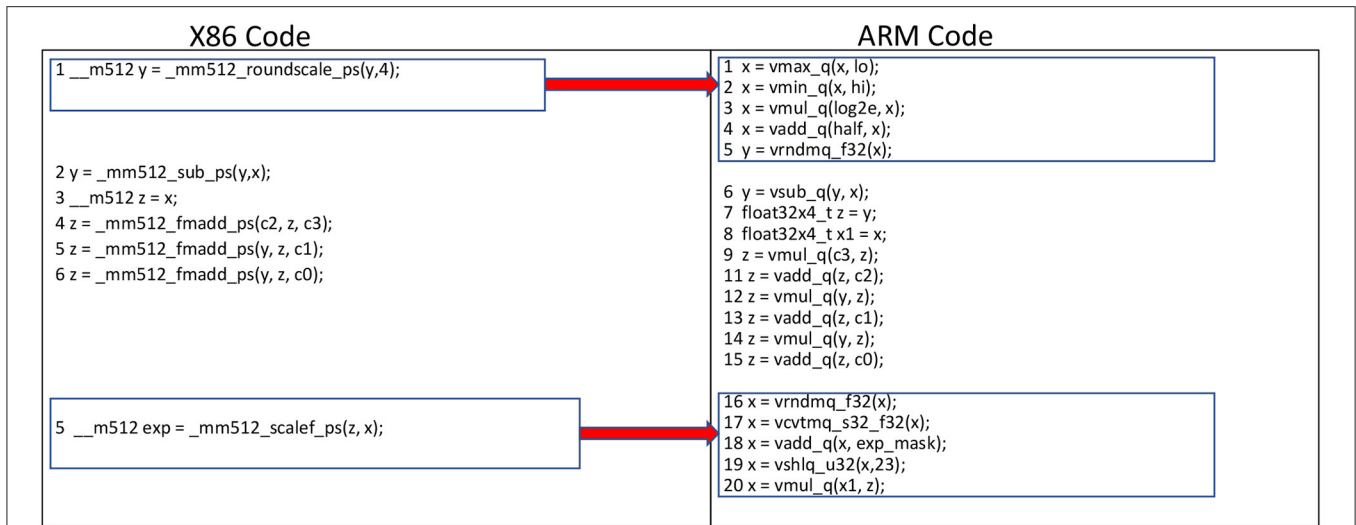
X86 Code

```
1 __m512 y = _mm512_roundscale_ps(y,4);




2 y = _mm512_sub_ps(y,x);
3 __m512 z = x;
4 z = _mm512_fmadd_ps(c2, z, c3);
5 z = _mm512_fmadd_ps(y, z, c1);
6 z = _mm512_fmadd_ps(y, z, c0);




5 __m512 exp = _mm512_scalef_ps(z, x);
```

ARM Code

```
1  x = vmax_q(x, lo);
2  x = vmin_q(x, hi);
3  x = vmul_q(log2e, x);
4  x = vadd_q(half, x);
5  y = vrndmq_f32(x);

6  y = vsub_q(y, x);
7  float32x4_t z = y;
8  float32x4_t x1 = x;
9  z = vmul_q(c3, z);
11 z = vadd_q(z, c2);
12 z = vmul_q(y, z);
13 z = vadd_q(z, c1);
14 z = vmul_q(y, z);
15 z = vadd_q(z, c0);

16 x = vrndmq_f32(x);
17 x = vcvtmq_s32_f32(x);
18 x = vadd_q(x, exp_mask);
19 x = vshlq_u32(x,23);
20 x = vmul_q(x1, z);
```

FIGURE 11 | Pseudocode for e^x approximation with Taylor series on AVX512 x86 and ARM.
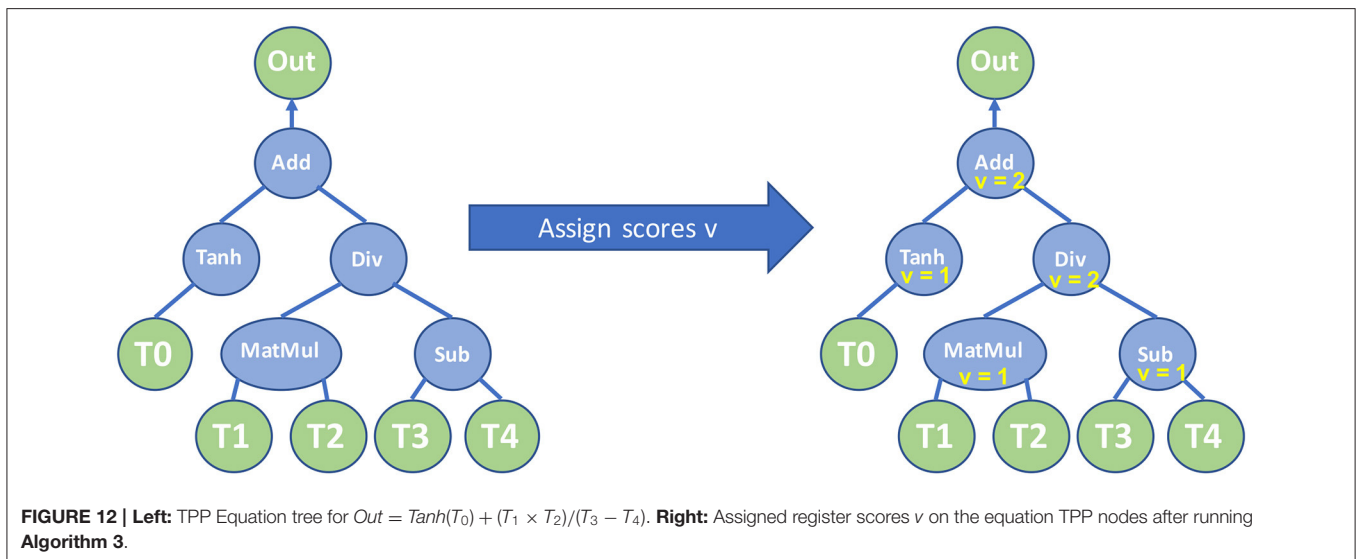
FIGURE 12 | Left: TPP Equation tree for $Out = Tanh(T_0) + (T_1 \times T_2)/(T_3 - T_4)$. **Right:** Assigned register scores v on the equation TPP nodes after running **Algorithm 3**.

the equation's input tensors are the leaves of the tree. The inputs of a TPP tree node are essentially its children in the equation tree. The output of an internal TPP node can be represented as a *temporary* intermediate tensor which in turn can be fed as input to the parent TPP node in the tree. Depending on the TPP node type (unary/binary/ternary), each internal node requires a number of inputs (one/two/three) to be computed/ready before performing the corresponding TPP operation. Let's consider for example the TPP equation tree in **Figure 12-Left** that is used to express the following operator:

$$Out = Tanh(T_0) + (T_1 \times T_2)/(T_3 - T_4) \qquad (1)$$

We will illustrate with this example how our eDSL for TPP Matrix Equations works.

4.2. Optimized Execution Plan for TPP Matrix Equations

The equation tree in **Figure 12-Left** can be naively evaluated by assigning to each intermediate node a temporary tensor to hold the corresponding TPP output, and performing, e.g., (1) the Tanh operation, (2) the Matrix Multiplication, (3) the Subtract operation, (4) the Div operation, and finally (5) the Add TPP. In such an evaluation schedule, we would need 4 intermediate tensors to hold the corresponding intermediate results. In this subsection, we illustrate how we can construct optimized execution plans for TPP Matrix Equations that minimize the number of intermediate tensors.

For each TPP node r we can assign a *register score* value v_r that essentially dictates how many temporary/intermediate tensors are required to calculate the subtree in the equation where

Algorithm 3 | Assign_Register_Score(r).

Input: TPP equation tree with root node r
Output: TPP equation tree with assigned register score values on its nodes

1: **if** is_Leaf(r) **then**
2: $\quad v_r \leftarrow 0$
3: **if** r is **unary TPP then**
4: $\quad$ Assign_Register_Score(Left_Child(r))
5: $\quad \triangleright$ *If child is leaf, then we assign current register score of 1, else we assign the child's register score*
6: $\quad$ **if** is_Leaf(Left_Child(r)) **then**
7: $\quad\quad v_r \leftarrow 1$
8: $\quad$ **else**
9: $\quad\quad v_r \leftarrow$ Register_Score(Left_Child(r))
10: **if** r is **binary TPP then**
11: $\quad$ Assign_Register_Score(Left_Child(r))
12: $\quad$ Assign_Register_Score(Right_Child(r))
13: $\quad \triangleright$ *If the register scores of children are equal, then we get the children's register score increased by one, otherwise we get the max value of the children's register score*
14: $\quad$ **if** Register_Score(Left_Child(r)) **equals** Register_Score(Right_Child(r)) **then**
15: $\quad\quad v_r \leftarrow$ Register_Score(Left_Child(r)) $+ 1$
16: $\quad$ **else**
17: $\quad\quad v_L \leftarrow$ Register_Score(Left_Child(r))
18: $\quad\quad v_R \leftarrow$ Register_Score(Right_Child(r))
19: $\quad\quad v_r \leftarrow$ MAX(v_L, v_R)
20: **if** r is **ternary TPP then**
21: $\quad$ Assign_Register_Score(Left_Child(r))
22: $\quad$ Assign_Register_Score(Middle_Child(r))
23: $\quad$ Assign_Register_Score(Right_Child(r))
24: $\quad \triangleright$ *If all children are leaves, then we assign current register score of 1. Otherwise, in a pairwise fashion we consider the register scores of the children in order of increasing value.*
25: $\quad$ **if** is_Leaf(Left_Child(r)) **AND** is_Leaf(Middle_Child(r)) **AND** is_Leaf(Right_Child(r)) **then**
26: $\quad\quad v_r \leftarrow 1$
27: $\quad$ **else**
28: $\quad\quad v_L \leftarrow$ Register_Score(Left_Child(r))
29: $\quad\quad v_M \leftarrow$ Register_Score(Middle_Child(r))
30: $\quad\quad v_R \leftarrow$ Register_Score(Right_Child(r))
31: $\quad\quad v_0, v_1, v_2 \leftarrow$ Sort_Increasing_Order(v_L, v_M, v_R)
32: $\quad\quad$ **if** v_2 **equals** v_1 **then**
33: $\quad\quad\quad v_{tmp} \leftarrow v_2 + 1$
34: $\quad\quad$ **else**
35: $\quad\quad\quad v_{tmp} \leftarrow v_2$
36: $\quad\quad$ **if** v_{tmp} **greater than** $v_0 + 1$ **then**
37: $\quad\quad\quad v_r \leftarrow v_{tmp}$
38: $\quad\quad$ **else**
39: $\quad\quad\quad v_r \leftarrow v_{tmp} + 1$

---

node r is root. We extend the methodology of Flajolet et al. [30] and we generate the register score values using the recursive **Algorithm 3**. This algorithm calculates recursively the register

Algorithm 4 | Create_Execution_Plan(r).

Input: TPP equation tree with root node r and assigned register score values on its nodes
Output: TPP equation tree with assigned traversal timestamps t and temporary tensor ids tmp

1: **if** is_Leaf(r) **then**
2: $\quad$ return
3: **if** r is **unary TPP then**
4: $\quad$ Create_Execution_Plan(Left_Child(r))
5: $\quad t_r \leftarrow$ global_timesteamp++
6: $\quad \triangleright$ *If child is leaf, reserve a new tmp, else re-use tmp from child*
7: $\quad$ **if** is_Leaf(Left_Child(r)) **then**
8: $\quad\quad tmp_r \leftarrow$ Reserve_Tmp()
9: $\quad$ **else**
10: $\quad\quad tmp_r \leftarrow$ tmp_Left_Child(r)
11: **if** r is **binary TPP then**
12: $\quad \triangleright$ *Recursively visit children in order of decreasing register score*
13: $\quad$ Create_Execution_Plan(Child_Max_Register_Score(r))
14: $\quad$ Create_Execution_Plan(Child_Min_Register_Score(r))
15: $\quad t_r \leftarrow$ global_timesteamp++
16: $\quad \triangleright$ *If all children are leaves, reserve a new tmp, else re-use the tmp from a non-leaf child and recycle the tmp of the other non-leaf child*
17: $\quad$ **if** is_Leaf(Left_Child(r) **AND** is_Leaf(Right_Child(r))) **then**
18: $\quad\quad tmp_r \leftarrow$ Reserve_Tmp()
19: $\quad$ **else**
20: $\quad\quad$ **if** not_Leaf(Left_Child(r) **then**
21: $\quad\quad\quad tmp_r \leftarrow$ tmp_Left_Child(r)
22: $\quad\quad\quad$ Recycle_Tmp(tmp_Right_Child(r))
23: $\quad\quad$ **else**
24: $\quad\quad\quad tmp_r \leftarrow$ tmp_Right_Child(r)
25: $\quad\quad\quad$ Recycle_Tmp(tmp_Left_Child(r))
26: **if** r is **ternary TPP then**
27: $\quad \triangleright$ *Recursively visit children in order of decreasing register score*
28: $\quad$ Create_Execution_Plan(Child_Max_Register_Score(r))
29: $\quad$ Create_Execution_Plan(Child_Mid_Register_Score(r))
30: $\quad$ Create_Execution_Plan(Child_Min_Register_Score(r))
31: $\quad t_r \leftarrow$ global_timesteamp++
32: $\quad \triangleright$ *If all children are leaves, reserve a new tmp, else re-use the tmp from a non-leaf child and recycle the tmps of the other non-leaf children*
33: $\quad$ **if** is_Leaf(Left_Child(r)) **AND** is_Leaf(Middle_Child(r)) **AND** is_Leaf(Right_Child(r)) **then**
34: $\quad\quad tmp_r \leftarrow$ Reserve_Tmp()
35: $\quad$ **else**
36: $\quad\quad$ **if** not_Leaf(Left_Child(r) **then**
37: $\quad\quad\quad tmp_r \leftarrow$ tmp_Left_Child(r)
38: $\quad\quad\quad$ Recycle_Tmp(tmp_Middle_Child(r)), Recycle_Tmp(tmp_Right_Child(r))
39: $\quad\quad$ **else**
40: $\quad\quad\quad$ **if** not_Leaf(Right_Child(r) **then**
41: $\quad\quad\quad\quad tmp_r \leftarrow$ tmp_Right_Child(r)

42:                  Recycle_Tmp(tmp_Middle_Child(r)),
    Recycle_Tmp(tmp_Left_Child(r))
43:          **else**
44:             $tmp_r \leftarrow$ tmp_Middle_Child(r)
45:                  Recycle_Tmp(tmp_Left_Child(r)),
    Recycle_Tmp(tmp_Right_Child(r))

scores of the children for a given node r, and in this way we know how many temporary tensors are required for the evaluation for each child. Now, if all of its children have the same register score, the node r get an increased register score value, otherwise the node gets as register score the maximum of its children's register score values. Intuitively this means that we can first evaluate a child c and its subtree with whatever intermediate tensor requirements it has, e.g., v_c temporary tensors, and eventually we need only one temporary tensor to hold c's output. We can do the same afterwards for all other siblings of c, however, we can reuse/recycle the rest $v_c - 1$ temporary tensors that were required by c since c and its subtree have been already computed.

This algorithm optimizes the number of temporary tensors/storage that are required for the equation evaluation, and it reuses the temporary storage as much as possible. For instance, for the equation in **Figure 12-Left**, after executing **Algorithm 3** on the TPP equation tree, we see that the root's register score value is 2 (see **Figure 12-Right**), meaning that only 2 intermediate tensors are required to evaluate the entire TPP tree rather than naively assigning one temporary tensor to each internal TPP node which would result in 4 intermediate tensors.

Now that we have assigned the register scores for each node we can devise an execution plan for the TPP equation tree that minimizes the number of required intermediate tensors. **Algorithm 4** recursively creates such an optimal execution plan and essentially it calculates: (1) the order/traversal timestamps t with which the TPP equation nodes have to be evaluated, and also (2) assigns to each intermediate node r a temporary tensor id tmp_r that holds the intermediate resulting tensor of that TPP node. **Figure 13-Right** shows the optimized execution plan by applying **Algorithm 4** on our example equation. This algorithm recursively visits/evaluates the children of a node r in order of decreasing register score value. This means that the child/subtree with the maximum register score value is evaluated first, one of the temporary tensors is dedicated to hold that child's intermediate output, whereas the remaining temporary tensors can be reused for the evaluation of the siblings/subtrees, which per definition/order of traversal, require less or equal number of intermediate tensors. Such a strategy guarantees that the temporary tensors are optimally reused/recycled, and as a result we can leverage the minimum required temporary tensors for the evaluation of the entire equation TPP tree. For simplicity in our description, we assumed that all intermediate temporary tensors have the same size, however, our implementation considers the actual sizes of the intermediate output tensors and takes the maximum one as representative size for all temporary tensors.

4.3. Implementation of Optimized Execution Plan for TPP Matrix Equations
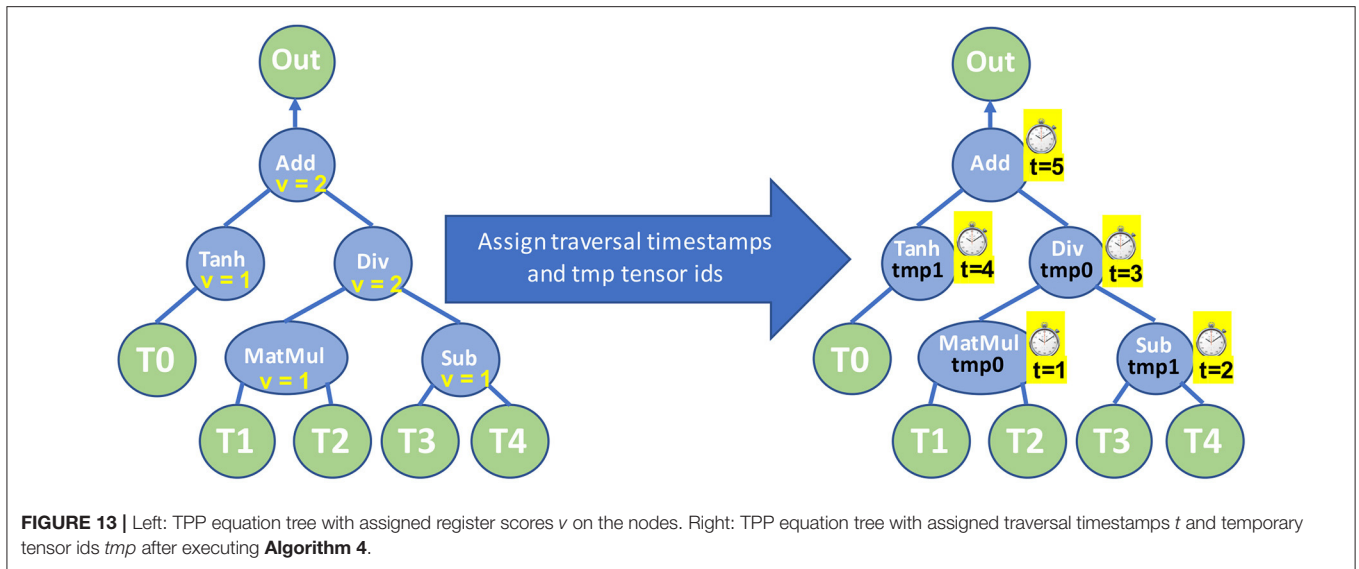
By employing **Algorithm 4**, we can devise an optimal execution plan for the TPP Matrix equation, and, here, we describe the implementation of such a plan. We consider three implementation strategies:

- *Strategy 1*: Using stack-allocated buffers as intermediate temporary tensors.
- *Strategy 2*: Using vector-register blocks as intermediate temporary tensors.
- *Strategy 3*: Hybrid implementation where some intermediate temporary tensors are stack-allocated buffers and some are vector-register blocks.

So far in our description, we have used the abstract notation "temporary tensor" without specifying how such a temporary tensor is instantiated in the implementation. The exact instantiation of a temporary/intermediate tensor is the differentiation factor among the 3 implementation strategies for the TPP matrix equations.

Strategy 1 considers each intermediate tensor as a physical buffer, and our TPP equation implementation allocates on the stack some space/buffer for each temporary tensor. Then, by following the timestamp order of the optimal execution plan (e.g., see **Figure 13-Right**), we emit/JIT the corresponding TPP code (e.g., see **Algorithms 1** and **2**) where the input tensors might be either the equation's input buffers provided by the user, or one of the stack allocated buffers representing an intermediate result. The fact that we have minimized the number of intermediate temporary buffers/tensors is critical for performance since these stack-allocated buffers may remain in some level of cache. Such a strategy is generic and can be leveraged to implement arbitrary equations. However, Strategy 1 may suffer from store-to-load forwarding inefficiencies on modern processors. Additionally, some of the intermediate tensors may spill from cache (e.g., when the intermediate outputs exceed the corresponding cache capacity) which would make the communication of temporary tensors among TPP nodes *via* loads/stores from/to stack allocated buffers quite expensive.

Strategy 2 considers each intermediate tensor as an $r_m \times r_n$ vector-register block. For example, on an AVX512 platform with 32 512-bit wide registers we have available 2 KBytes of register file that may be used for intermediate tensors. Each one of such 512-bit wide vector registers can hold 16 single-precision values and by stacking, e.g., 4 of these we can form a logical 16×4 intermediate tensor and in total we have available $32/4 = 8$ of such intermediate tensors that could be used by the equation. In Strategy 2, we block the computation of the equation's output in blocks with size $r_m \times r_n$, and we can calculate the corresponding $r_m \times r_n$ output by following the timestamp order of the optimal execution plan. We emit/JIT the corresponding TPP code for sub-tensors with size $r_m \times r_n$ where each intermediate output tensor is the assigned temporary vector-register block. Essentially this strategy performs vertical register fusion within the equation TPP nodes and incurs *no* communication *via* loads/stores from/to stack allocated buffers.

FIGURE 13 | Left: TPP equation tree with assigned register scores *v* on the nodes. Right: TPP equation tree with assigned traversal timestamps *t* and temporary tensor ids *tmp* after executing **Algorithm 4**.

However, such a methodology is limited by the number of available vector registers on each platform.

Strategy 3 combines the strengths of Strategies 1 and 2 by considering some intermediate tensors as stack-allocated buffers and some intermediate tensors as vector-register blocks. As such, in Strategy 3 the TPP operations/subtrees which exhibit *both* high register pressure and reuse (e.g., transposes, GEMM/BRGEMM, transcendental approximations), propagate the intermediate results toward the rest of the TPPs in the tree *via* stack-allocated temporal tensors. On the other hand, TPP subtrees without large register pressure are implemented using Strategy 2 that employs vertical register fusion and avoids loads/stores from/to stack-allocated buffers.

In addition to the aforementioned 3 strategies, in the TPP equation back-end we identify idioms/motifs of combined TPPs (e.g., a gather TPP followed by a reduce TPP) and we JIT an instruction sequence which is optimal for the composite access pattern. In section 5.1.5, we show an example of such a combined TPP motif that is optimized by the TPP backend.

Even though we developed a rudimentary method/POC of combining the TPPs *via* Matrix Equation Trees, we have found that it is sufficient to express all the complex operators we encountered in a wide-range of workloads discussed further in section 5. Nevertheless, we envision that when/if TPPs are widely adopted within Tensor Compiler frameworks (e.g., as an MLIR dialect) then more complicated Graphs (instead of simple trees) and more sophisticated analyses/optimization passes can be leveraged during the composition of TPPs. The key-ingredient that makes the composition of TPPs amenable to optimization opportunities is the TPP specification itself: TPPs comprise a small, well-defined compact set of tensor operators with declarative semantics as shown in section 2.

We would like also to highlight one use-case of Matrix Equations that can be beneficial for specialized DL accelerators. The BRGEMM TPP described in section 3.2 corresponds to an output-stationary flow that is suitable for CPUs and GPUs.

Given an accelerator that favors, e.g., *A*-stationary GEMM formulations, one could express the following Matrix Equation: internal nodes G_i would be GEMM ternary TPPs, for each GEMM node G_i we would have the same input leaf *A* and a varying input B_i, and the output of each node would be a result C_i. Essentially this formulation dictates an *A*-stationary flow, and the back-end could optimize accordingly for the specific accelerator.

5. TPP-BASED KERNELS AND WORKLOADS

This section covers how DL kernels and workloads (image processing, recommendation systems, natural language processing, graph processing, and applications in science) can leverage TPPs to achieve high performance. Although this article's work is targeting CPUs, we cover the entire training pipeline and not only inference. The main purpose of this is to demonstrate the versatility of TPPs which is valuable in the more complicated backward pass kernels, and to handle training's implications to the forward pass.

5.1. TPP-Based Kernels
5.1.1. Softmax Kernel
Figure 14 illustrates two Matrix Equation trees that are used to express the softmax operator [31]:

$$Y = \text{softmax}(X) \text{ with } y_{ij} = \frac{e^{\left(x_{ij} - \max_{x_{ij} \in X} x_{ij}\right)}}{\sum\limits_{x_{ij} \in X} e^{\left(x_{ij} - \max_{x_{ij} \in X} x_{ij}\right)}} \quad (2)$$

Equation 2 shows the formula for the softmax operator [31], which is often used as the last activation function of a neural network, aiming to normalize its output to a probability distribution. We can represent this operator *via* two TPP equation trees illustrated in **Figure 14**. The left tree computes the
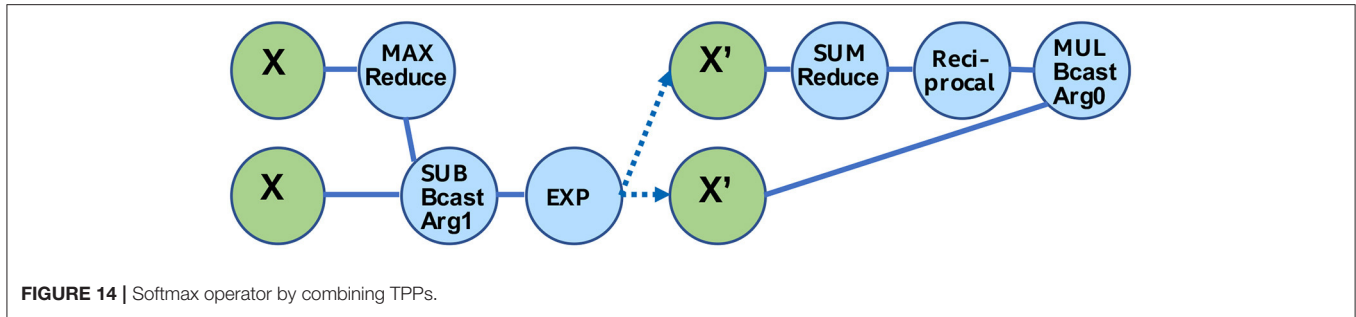
FIGURE 14 | Softmax operator by combining TPPs.

nominator of Equation 2: first the maximum value of the input tensor X is found (*via* the max-reduce TPP), then we subtract this max value from each entry of X (note the broadcast semantics in the second argument of the subtraction TPP), and a new tensor X' is computed by calculating the element-wise exponent on the earlier subtraction's outcome. Finally, in the right TPP tree each entry of the tensor X' is normalized by the sum of all values in X' to obtain the softmax output, a tensor Y. This example illustrates the expressiveness of the TPP abstractions, since the components of the mathematical formula map to TPPs in a straightforward way. At the same time, this example highlights the separation of concerns: the user does not need to worry about the efficient implementation of this equation on each different platform, since the TPP back-end is responsible for optimized code generation which is target-specific (contrary to the TPP expression itself which is platform-agnostic).

5.1.2. Normalization Kernels

Batch normalization (batchnorm) is a technique [32] that normalizes neuron layer input tensors to improve the overall training process. Batchnorm removes the need for careful parameter initialization and reduces the required training steps [32] in the neural networks. The batchnorm computations can be divided in two stages: (i) First the mean and variance of the input tensor are computed across the "batch" dimension: $\mu_j = \sum_{i=0}^{n-1} x_{ij}$, $\sigma_j^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_{ij} - \mu_i)^2$ where i is the "batch" dimension and j is the "feature" dimension, (ii) then the tensor entries x_{ij} are normalized based on μ and σ: $x'_{ij} = (x_{ij} - \mu_j)/(\sqrt{\sigma_j^2 + \epsilon})$.

Depending upon the workload, different TPPs and TPP equations can be employed to implement the batchnorm. Here, we take an example of batchnorm on a ResNet50 [33] convolution layer tensor X. The input tensor X has a four-dimensional shape of {N, C, H, W} with dimensions of batch (N), feature (C), height (H), and width (W). We first use sum-reduce TPPs on H and W dimensions to compute the sum ($m[N, C]$) and the sum of squared elements ($v[N, C]$) matrices. Subsequently, we use binary add TPPs across the batch dimension of $m[N, C]$ and $v[N, C]$ matrices for eventual computation of mean ($\mu[C]$) and variance ($\sigma^2[C]$) vectors. In the next step, we use a scaling equation to normalize each element of the input tensor. The scaling equation $Y = (m' * X + v') * G + B$ converts the input tensor X into a normalized tensor Y. Here, $G[C]$ and $B[C]$ are scaling vector inputs to batchnorm, and $m'[C]$ and
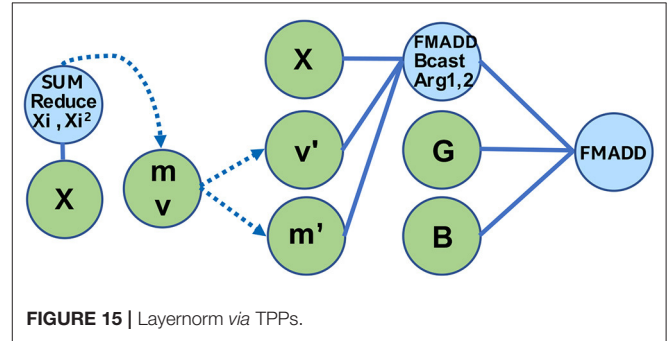

FIGURE 15 | Layernorm *via* TPPs.

$v'[C]$ are intermediate vectors that are computed from mean and variance vectors. We implement the scaling equation by a single TPP equation containing two FMADD ternary TPPs. The second equation tree of **Figure 15** shows an analogous scaling equation implementation. However, for this particular implementation, we broadcast m', v', G, B vectors into H, W, and N dimensions inside the TPP equation tree. An efficient implementation of batchnorm uses blocking on the C, H, and W dimensions along with multi-threading on the N and feature block dimension. We do not show the details of this implementation for sake of simplicity.

Layer normalization (layernorm) [34] is a technique that normalizes the neurons *within* a layer, and was motivated by the limitations of Batch Normalization [32] in Recurrent Neural Networks. The layernorm computations can be divided in two stages: (i) First the mean and variance of the input tensor are computed across the "feature" dimension: $\mu_i = \sum_{j=0}^{m-1} x_{ij}$, $\sigma_i^2 = \frac{1}{m} \sum_{j=0}^{m-1} (x_{ij} - \mu_i)^2$ where i is the batch dimension and j is the "feature" dimension, (ii) then the tensor entries x_{ij} are normalized based on μ and σ: $x'_{ij} = (x_{ij} - \mu_i)/(\sqrt{\sigma_i^2 + \epsilon})$. Depending on the workload (e.g., attention cell in BERT), the scaled tensor may be further scaled with two other tensors γ and β. **Figure 15** illustrates two TPP equation trees that implement this composite layernorm operator. The left equation is using the sum-reduce TPP to compute the sum and sum of squared elements of the input tensor, namely m and v. These two scalars are combined (not shown in the equation for simplicity), and are fed as inputs to the right TPP tree, where the FMADD ternary TPP is used to scale the input tensor X. Finally, a cascading FMADD ternary TPP computes the final result *via* the scaling tensors G and B. We illustrate this layernorm *via* means of TPPs since all DL

norming layers essentially exhibit similar computational motif, and this specific norm is used in the BERT workload described in section 5.2.3.

Group normalization (groupnorm) [35] is a technique that normalizes the neurons within a group of features. Groupnorm was proposed as an alternative to batchnorm [32] to reduce normalization error for smaller batch sizes. In groupnorm, features are divided into groups, and mean and variance are computed within each group for normalization. Groupnorm is also a generalization of the layer normalization [34] and instance normalization [36] approach. Layernorm is groupnorm with a single group, and instance norm is groupnorm with group size equal to one. Groupnorm can be implemented with the same set of TPPs and TPP equations that were used in the batchnorm kernel. We again take the example of ResNet50 [33] convolution layer tensor X and apply groupnorm on it with g number of groups. We can ignore the batch dimension (N) for this discussion as groupnorm works independently upon each batch. Therefore, the input tensor X now has a three-dimensional shape of $\{C, H, W\}$ with dimensions of feature (C), height (H), and width (W). We first use sum-reduce TPPs on H and W dimensions to compute the sum ($m[C]$) and the sum of squared elements ($v[C]$) vectors. Subsequently, we add $m[C]$ and $v[C]$ values within a feature group for eventual computation of group mean ($\mu[g]$) and group variance ($\sigma^2[g]$) vectors. Similar to batchnorm, we use a scaling equation to normalize each element of the input tensor. The scaling equation $Y = (m' * X + v') * G + B$ converts input tensor X into a normalized tensor Y. Here, $G[C]$ and $B[C]$ are scaling vector inputs to groupnorm, and $m'[C]$ and $v'[C]$ are intermediate vectors that are computed from group mean and group variance vectors. The second equation tree of **Figure 15** shows an analogous scaling equation implementation. However, for this particular implementation, we broadcast m', v', G, B vectors into H and W dimensions inside the TPP equation tree. We can also apply the same scaling equation to a single group or set of groups with few parameter changes. An efficient implementation of groupnorm uses blocking on the C, H, and W dimensions. We do not show the details of this implementation for sake of simplicity.

5.1.3. BF16 Split-Stochastic Gradient Descent Kernel

Unlike the previous kernels which are well-established in DL workloads, and as such potentially optimized in DL libraries, we present here an example of a novel operator, which per definition is not existent in DL libraries. BF16 split-SGD was recently introduced in the context of DLRM training with BF16 datatype [37]. The Split-SGD-BF16 solver aims at efficiently exploiting the aliasing of BF16 and FP32 (i.e., the 16 Most Significant Bits (MSB) on both are identical) in order to save bandwidth during the SGD-solver in training. The employed trick is that the weights are not stored as FP32 values in a single tensor. Instead, the FP32 tensors are split into their high and low 16 bit-wide parts: the 16 MSBs of the FP32 values, and the 16 LSBs of the same values are stored as two separate tensors X^{hi} and X^{lo}, respectively. The 16 MSBs represent a valid BF16 number and constitute the model/weight tensors during training. These BF16 weights are used exclusively in the forward and backward


FIGURE 16 | BF16 Split-SGD operator by combining TPPs.
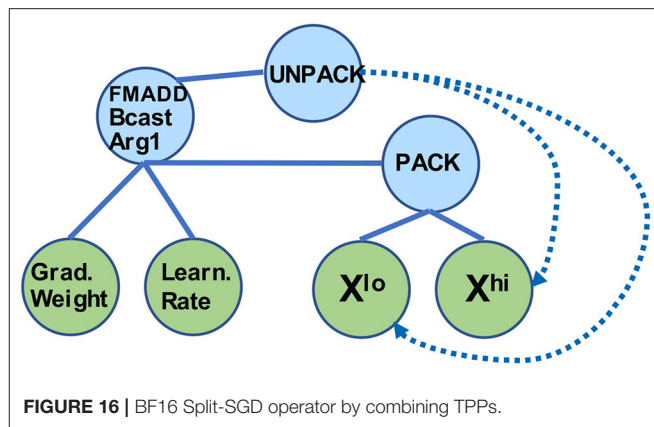
passes, whereas the lower 16 bits are only required in optimizer. More specifically, the X^{hi} and X^{lo} tensors are packed together to form an FP32 tensor, resulting in a fully FP32-accurate optimizer. **Figure 16** illustrates the BF16 Split-SGD operator written entirely *via* TPPs. First the X^{hi} and X^{lo} are packed, and the formed FP32 tensor is used in a cascading FMADD TPP that performs the SGD scaling with the corresponding Gradient Weight tensor and learning rate. Finally, the resulting FP32 tensor is unpacked to the X^{hi} and X^{lo} tensors for further use in the training process.

5.1.4. Convolutional Neural Network Kernel

Convolutional Neural Networks (CNN) consist of layers with multiple neurons connected by weights, and they have been applied with success in image recognition, semantic segmentation, autonomous driving, medical imaging and in an increasing number of scientific applications. Previous work [21, 23] has shown that CNNs, despite their seemingly complicated loop structure due to the involved high-dimensional tensors, can be mapped efficiently onto small 2D GEMMs and BRGEMMs. In this work, we adopt the same strategy to implement CNNs *via* the BRGEMM TPP. Unlike the previous work which presents only the address-based BRGEMM formulation, here, we leverage the CNN kernels with stride-based BRGEMM for 1×1 convolutions and offset-based BRGEMM for 3×3 convolutions to get even more performant implementations (see section 2.3 for a brief description of the BRGEMM variants).

5.1.5. Sparse Embedding Kernel

The sparse embedding kernel is comprised of multi-hot encoded lookups into an embedding table $W^{M\times E}$ with M being the number of rows and E the length of each row, whereas the multi-hot weight-vector is denoted as $\alpha^T = [0, \ldots, a_{p_1}, \ldots, a_{p_k}, \ldots, 0]$ with entries $a_p = 1$ for $p \in \{p_1, \ldots, p_k\}$ and 0 elsewhere (p being the index for the corresponding lookup items). Mathematically, the embedding lookup output vector o^T can be obtained *via* $o^T = a^T \times W$. This operation (assuming row-major storage for W) is equivalent to gathering the rows of W based on the non-zero indices a_p, and then adding them up to get the output vector o^T. **Figure 17** illustrates the TPP tree that is used to express the Sparse Embedding lookup kernel.

FIGURE 17 | Sparse Embedding Lookups *via* TPPs.
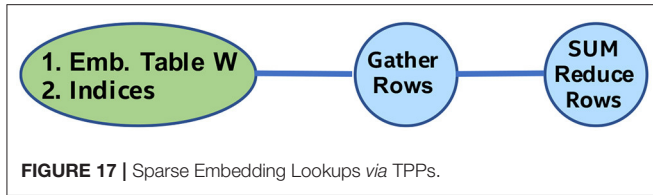
Algorithm 5 | Sparse Gather-Reduce operation.

Inputs: $\alpha^T = [0, \ldots, a_{p_1}, \ldots, a_{p_k}, \ldots, 0]$ with entries $a_p = 1$ for $p \in \{p_1, \ldots, p_k\}$ and 0 elsewhere, $W^{M \times E}$

Output: $o^T = a^T \times W$

1: **for** $j = 0 \ldots E$ **with step** $vlen \cdot U$ **do**
2:    ▷ *Initializing accumulator registers to 0*
3:    **for** $u = 0 \ldots U - 1$ **do**
4:       $vec\_out_u \leftarrow 0$
5:    ▷ *Iterating over non-zero entries/indices in α^T*
6:    **for** i **in** $1, 2, \ldots, k$ **do**
7:       $idx = p_i$
8:       $next\_idx = p_{i+1}$
9:       ▷ *Unroll innermost kernel U times: load indexed vector, prefetch next indexed vector, accumulate loaded vector to accumulator register*
10:       **for** $u = 0 \ldots U - 1$ **do**
11:          $vec\_W \leftarrow \text{load\_vector}(W[idx][j + u \cdot vlen : j + (u + 1) \cdot vlen])$
12:          $\text{prefetch}(W[next\_idx][j + u \cdot vlen : j + (u + 1) \cdot vlen])$
13:          $vec\_out_u \mathrel{+}= vec\_W$
14:    ▷ *Store accumulator registers to o^T*
15:    **for** $u = 0 \ldots U - 1$ **do**
16:       $o^T[j + u \cdot vlen : j + (u + 1) \cdot vlen] \leftarrow vec\_out_u$

We note that the TPP backend optimizes this sequence of TPPs, and performs register fusion across the gather and the reduce TPP components. More specifically, given a non-zero index a_p, the corresponding row of W is loaded in vector registers, and is added to a set of running accumulators/vector registers that hold the output o^T. **Algorithm 5** illustrates the optimized JITed implementation in our TPP backend. The E dimension is vectorized in an SIMD-fashion with vector length $vlen$. Note that in line 13 we expose multiple independent accumulation chains in order to hide the latency of the vector-add SIMD instructions. Since we JIT this sub-procedure, we know the exact value of E at runtime. As such, we can pick appropriate unrolling factor U as well as the remainder handling can be performed optimally *via* masking in case E is not perfectly divisible by the vector length $vlen$. Last but not least, the JITed aggregation procedure employs prefetching of the subsequent indexed vectors in W (line 12) in order to hide the latency of these irregular accesses.

5.1.6. Multi-Layer Perceptron Kernel

Multilayer perceptrons (MLP) form a class of feed-forward artificial neural networks. An MLP consists of (at least three) *fully connected* layers of neurons. Each neuron in the topology

---

Algorithm 6 | Fully-Connected Layer with Unary Activation Function.

Inputs: $A^{M_b \times K_b \times b_k \times b_m}$, $B^{N_b \times K_b \times b_n \times b_k}$
Output: $C^{N_b \times M_b \times b_n \times b_m}$

1: Based on *thread_id* calculate M_b_start, M_b_end, N_b_start and N_b_end to assign output work items
2: **for** $ib_n = N_b\_start \ldots N_b\_end$ **do**
3:    **for** $ib_m = M_b\_start \ldots M_b\_end$ **do**
4:       $Out = \&C[ib_n][ib_m][0][0]$
5:       ▷ *Stride-based BRGEMM, stride_A=$b_k \cdot b_m$, stride_B=$b_n \cdot b_k$*
6:       **BRGEMM**($\&A[ib_m][0][0][0], \&B[ib_n][0][0][0], Out, K_b$)
7:       $C[ib_n][ib_m][0][0] \leftarrow$ **UNARY**($C[ib_n][ib_m][0][0]$)

---

may be using a non-linear activation function. In this section, we present the implementation of the *Fully Connected* layers since they constitute the cornerstone of MLP. Even though, we illustrate the forward pass of Fully Connected layers, we also implement *via* TPPs the kernels of the back-propagation training in an analogous fashion. **Algorithm 6** shows the fully connected layer implementation which is mapped to TPPs. First we note that the input tensors are conceptually 2D matrices $A^{M \times K}$ and $B^{K \times N}$ that need to be multiplied. We follow the approach of previous work [21] and we block the dimensions M, K, and N by factors b_m, b_k, and b_n, respectively. Such a blocked layout is exposing better locality and avoids large, strided sub-tensor accesses which are known to cause Translation Lookaside Buffer (TLB) misses and cache conflict misses in case the leading dimensions are large powers of 2 [21]. We leverage the BRGEMM TPP in order to perform the tensor contraction with A and B across their dimensions K_b and b_k (which constitute the K/inner-product dimension of the original 2D matrices). We employ the stride-based BRGEMM because the sub-blocks "A_i" and "B_i" that have to be multiplied and reduced are apart by constant strides $stride\_A = b_k \cdot b_m$ and $stride\_B = b_n \cdot b_k$ respectively. Finally, we apply (optionally) a unary TPP corresponding to the requested activation function (e.g., RELU) onto the just-computed output block of C.

5.2. TPP-Based Workloads
5.2.1. 1D Dilated Convolutions and Computational Biology

In this subsection, we show the implementation of a special type of convolution *via* TPPs in their entirety, namely one-dimensional (1D) dilated convolution layer of a 1D CNN named ATACworks [38]. ATACworks is used for de-noising and peak calling from ATAC-Seq genomic sequencing data [38]. The 1D dilated convolution layer in ATACworks takes more than 90% of the training time, and it has input tensor width W, output tensor width Q, C input channels, K output channels, filter size of S, and dilation d. We employ the transpose TPPs, copy TPPs, and BRGEMM TPPs to optimize the forward pass and the backward pass of the PyTorch-based 1D convolution layer. **Algorithm 7** shows an example of the forward pass procedure with an input tensor I, a weight tensor W, and an output tensor O.

Algorithm 7 | 1D Dilated convolution forward pass using TPPs.

Inputs: $I^{C \times W}$, $W^{K \times C \times S}$, $d \in \mathbb{R}$
Output: $O^{K \times Q}$

1:   $W^T \leftarrow$ **TRANSPOSE**(W)
2:   **for** $pos = 0 \ldots Q - 1$ **with step** b_q **do**
3:     ▷ *Address-based BRGEMM, prepare arguments A_{ptrs}, B_{ptrs}*
4:     **for** $s = 0 \ldots S - 1$ **with step** 1 **do**
5:       $A_{ptrs}[s] = \&W^T[s, 0, 0]$
6:       $B_{ptrs}[s] = \&I[0, (pos + s \cdot d)]$
7:     **BRGEMM**$(A_{ptrs}, B_{ptrs}, \&O[0, pos], S)$

FIGURE 18 | Binary-Reduce aggregation kernel *via* TPPs.

5.2.2. Deep Learning Recommendation Model

Facebook recently proposed a deep learning recommendation model (DLRM) [39]. Its purpose is to assist the systematic hardware–software co-design for deep learning systems. DLRM is comprised of the following major components: (a) a sparse embedding (see section 5.1.5) involving tables (databases) of varying sizes, (b) a small dense Multi-Layer Perceptron (see section 5.1.6), and (c) a larger and deeper MLP which is fed by the interaction among (a) and (b). All three parts can be configured (number of features, mini-batch sizes, and table sizes) to stress different aspects of the system. We also note that in the case of training with BF16 datatype, we leverage the BF16 split-SGD optimizer (see section 5.1.3). For more details on the workload and CPU-oriented optimizations we refer to prior work [37].

5.2.3. Natural Language Processing - Bidirectional Encoder Representations From Transformers

The BERT model is a bidirectional transformer pre-trained *via* a combination of masked language modeling objective, and next-sentence prediction [40]. The heart of the BERT model is comprised by sequence of BERT layers which are built using smaller building blocks. For ease of use and implementation, we followed modular building blocks from Hugging Face transformers library [41] and implemented four fused layers using TPP building blocks, namely *Bert-Embeddings*, *Bert-SelfAttention*, *Bert-Output/Bert-SelfOutput*, and *Bert-Intermediate* layers.

The *SelfAttention* layer, in turn, can be formulated as a bunch of Matrix / batch Matrix-Multiplications mixed with element-wise scale, add, dropout and softmax operators. We formulate these Matrix-Multiplications as tensor contractions on blocked-tensors *via* the stride-based BRGEMM TPP (similarly to **Algorithm 6**). We opt to use blocked tensor layouts for the same reasons briefly described in section 5.1.6. Furthermore, by working on one small sub-tensor at a time we naturally follow a "dataflow" computation, which has been shown to maximize the out-of-cache-reuse of tensors among cascading operators [26, 42]. The softmax operator is also formulated entirely by TPPs as described in section 5.1.1. We note that the sequence of Matrix-Multiplications in the attention layer requires sub-tensors to be transposed (and VNNI transformed in case of BF16 implementation), and for this task we leverage the transpose/transform TPPs. *Bert-Output* and *Bert-SelfOutput*
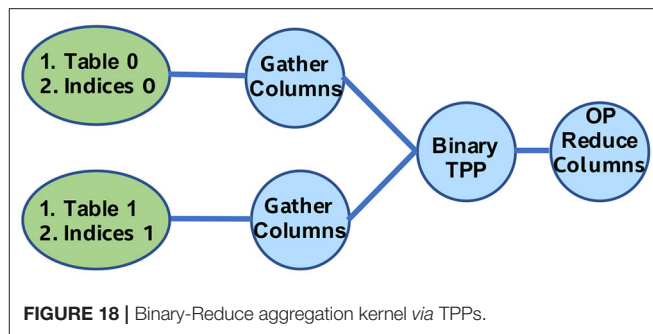
layers perform GEMM over blocked layout, and fuse bias addition, dropout, residual addition, and layernorm using TPPs. The *Bert-Embeddings* layer also performs layernorm and dropout after embedding lookups that are also implemented using TPPs. Finally, *Bert-Intermediate* layer performs blocked GEMM followed by bias addition and GELU activation function which we implement using the GELU TPP.

5.2.4. Emerging AI—Graph Neural Networks

Graph Neural Networks (GNN) [43] form an emerging class of Neural Networks for learning the structure of large, population-scale graphs. Depending on the specific algorithm and task that a GNN is designed for (e.g., node classification, link prediction), feature-vector aggregation precedes or succeeds a shallow neural network. Such a shallow neural network typically materializes one or more linear transformations, followed by a classification or regression mechanism [44], and the relevant TPP-based implementation is essentially the one we present in **Algorithm 6**.

We focus here on the TPP-based implementation of the feature-vector aggregation. This aggregation motif can be seen as a sequence of linear algebraic expressions involving node/edge features, along with the relevant operators. Prior work [44] has focused on the following two algebraic sequences: Copy-Reduce and Binary-Reduce. We elaborate here on the latter sequence Binary-Reduce (as the first is even simpler). The feature-vectors (either pertaining to vertices or edges) are represented *via* dense 2D matrices/tables. At the same time, the adjacency information in the graphs can be eventually found *via* arrays of indices. Therefore, by providing a set of indices and the appropriate Tables of feature-vectors (assuming column-major storage), one can extract selectively the desired feature-vectors *via* Gather-columns operations. Then, the extracted feature-vectors are fed into a binary operator, and the outcome of the binary operations are finally reduced (the reduce operation could be sum/max/min etc).

Figure 18 illustrates a TPP tree that is used to express the Binary-Reduce aggregation kernel. The TPP back-end optimizes this sequence of TPPs and performs horizontal register fusion across them. More precisely, two feature-vectors namely v_0 and v_1 are extracted at a time from Table 0 and Table 1 respectively by using the relevant indices arrays, and they are combined *via* the proper binary op to get an intermediate vector v_i. Subsequently, v_i is reduced with a running reduce-vector v_o
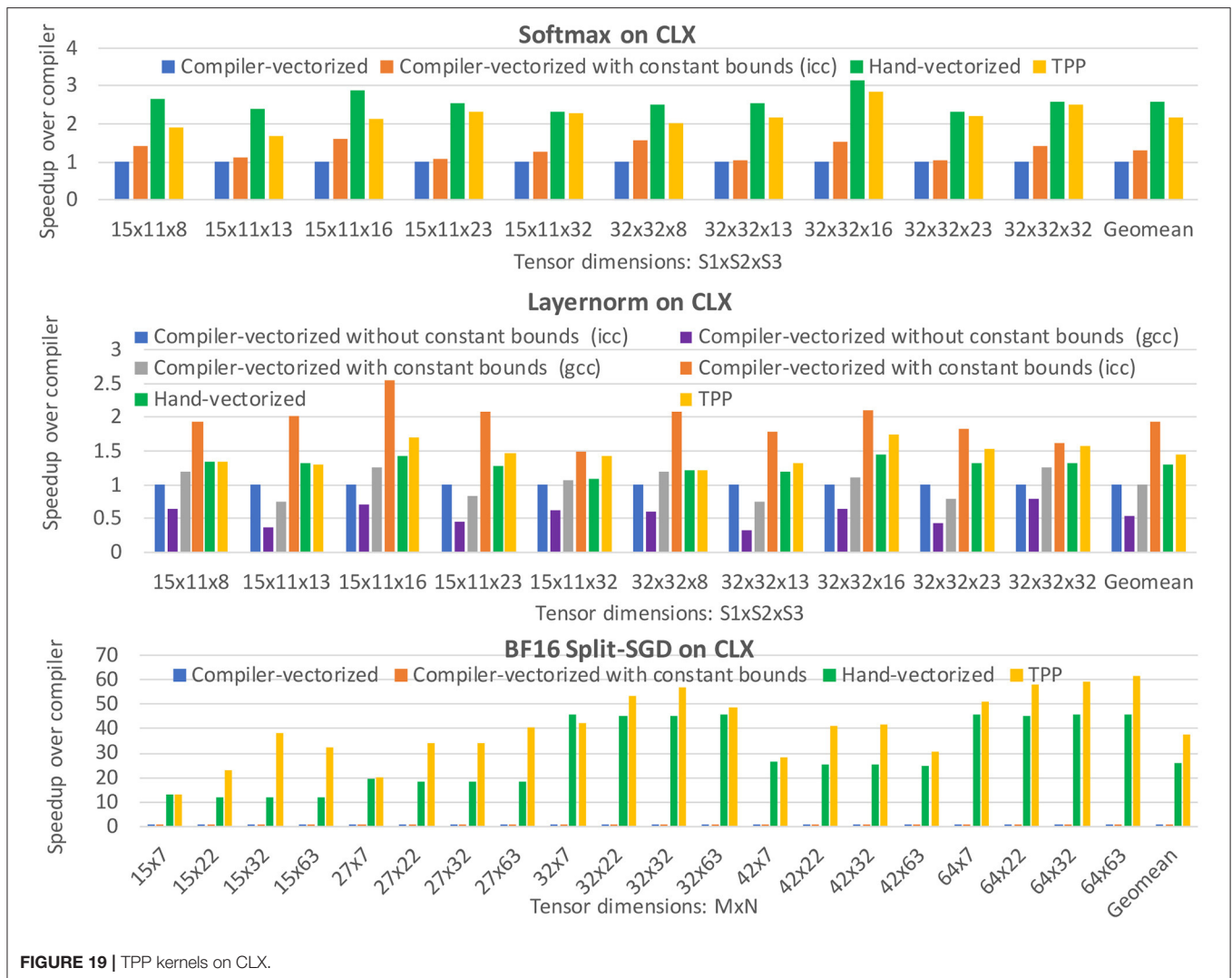
FIGURE 19 | TPP kernels on CLX.

that holds the output of this composite operator. Once the running reduction has been completed (i.e., all indexed columns from Table 0 and Table 1 have been accessed, processed and reduced), the output vector v_o is stored in the corresponding output subtensor.

## 6. EXPERIMENTAL RESULTS OF DL KERNELS AND WORKLOADS

We use a variety of platforms that span different ISAs, different vendors and micro-architectures. More specifically, our tested platforms include: (i) a 22-core Intel Xeon E5-2699 v4 (BDX) supporting up to AVX2 ISA, (ii) a 28-core Intel Xeon 8280 (CLX) supporting up to AVX512 ISA, (iii) a recently announced 40-core Intel Xeon 8380 (ICX) supporting also up to AVX512 ISA, (iv) a 28-core Intel Xeon 8380H (CPX) supporting up to AVX512 ISA, which also offers BF16 FMA acceleration, (v) a 64-core AMD EPYC 7742 (ROME) with AVX2 ISA, (vi) an AWS Graviton2 instance with 64 cores at fixed 2.5 GHz and AArch64 ISA, (vii)

a 48-core Fujitsu A64FX at fixed 1.8 GHz with ARMv8 SVE ISA, and (viii) a 4-core client Intel i7-6700 CPU (i7) supporting up to AVX2 ISA. All Intel and AMD chips are operating in Turbo mode. For the cluster experiments, we used a 32 node CLX installation with a dual-rail Intel Omnipath 100 pruned 2:1 fat-tree topology.

6.1. Performance of Standalone DL Kernels

We start the performance evaluation with standalone TPP kernels presented in section 5.1. First, we want to highlight the productivity/efficiency provided by TPPs: the high-level code expressed *via* TPPs/trees of TPPs can match or outperform code by compilers, and hand-vectorized (thus non-portable code) written by performance experts. Second, we want to show the portability aspect of TPPs, since exactly the same high-level code yields high-performance across different ISAs and micro-architectures.

Figure 19-Top shows the performance of the Softmax operator of blocked 3D tensors with size $S1 \times S2 \times S3$, on the

FIGURE 20 | TPP kernels on ROME.
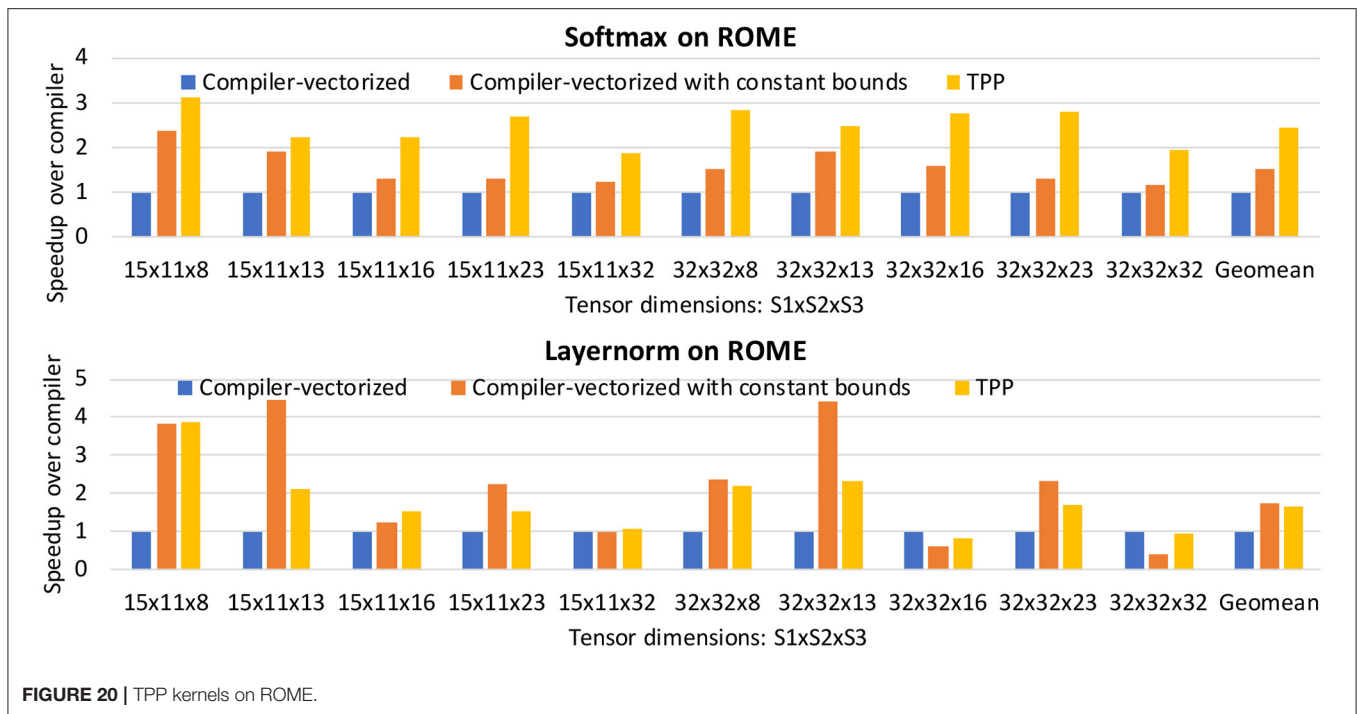
CLX platform (i.e., targeting AVX512 ISA). Here, we perform $S2$ softmax operations over blocked $S1 \times S3$ dimensions. The sizes are chosen such that some of the dimensions do not match perfectly with the vector length. The baseline is the icc generated code with -O3 optimization level and high-zmm usage flags. The second variant is also icc-generated code, but we propagate the tensor sizes/loop bounds *via* compile-time constants in order to assist the auto-vectorization/optimize remainder handling *via* masking. The third code variant is the AVX512 hand-vectorized by an expert, where the *exp* function uses fast Taylor approximation. Last, we evaluated the TPP-based softmax implementation. As we can see, by propagating the tensor sizes we achieve (geo-mean) speedup of $1.3\times$ over the baseline. The hand-vectorized code is faster by $2.6\times$ whereas the TPP-based variant shows similar speedups by being $2.2\times$ faster. The main shortcoming of the hand-vectorized code is that it is platform-dependent and as such non-portable. More specifically, we didn't have to our avail AVX2 hand-optimized code in order to experiment with it on ROME. On the contrary, **Figure 20-Top** shows the softmax performance on AVX2 enabled platform for the compiler-generated code and the TPP based code. The TPP-based softmax exhibits geo-mean speedup of $2.45\times$ over the baseline on ROME.

Figure 19-Middle shows the performance of the layernorm operator on the CLX platform. Since the layernorm code is more straightforward (i.e., no expensive *exp* function is involved), we see that icc with compile-constant bounds outperforms by $1.9\times$ the baseline. We inspected the compiler-generated code and identified that the reduction-loops were recognized and were heavily optimized with multiple accumulation chains etc. Similarly, the hand-vectorized

code and the TPP based code outperform the baseline by $1.3\times$ and $1.5\times$. We also experimented with gcc and the fast-math flag, and it just matched baseline performance. We want to emphasize that propagating the tensor sizes as compile-time constants throughout the operators is not practical for real use-cases within DL frameworks. **Figure 20-Bottom** shows similar performance speedups on ROME, where the TPP-based code is $1.6\times$ faster than the auto-vectorized baseline.

Figure 19-Bottom shows the performance of the BF16 split-SGD operator on CLX. This use-case represents a novel, mixed-precision operator where the compiler (icc with compile-time constant tensor sizes) struggles to yield good performance; the TPP-based code has geometric mean (geomean) speedup of $38\times$ over the compiler generated code.

Figure 21 illustrates the TPP-based implementation of various ResNet50 [33] Convolution layers across all available platforms. The minibatch size used on each platform equals to the number of the corresponding cores. It is noteworthy that the TPP-user code is identical for all targets, hence, truly portable; it is merely that the TPP backend optimizes the code generation (BRGEMM in this case) in a platform/ISA-aware fashion. The geomean efficiencies of these convolutions are: 69% for BDX, 72% for CLX, 72% for CPX, 77% for CPX with BF16 datatype, 70% for ICX, 78% for ROME, 81% for Graviton2 and 52% for A64FX. Previous work [21] also showed on an x86 TPP-predecessor that BRGEMM-based convolutions matched or outperformed Intel's oneDNN library [13]. Fujitsu recently contributed an A64FX back-end to oneDNN [45] and our TPP implementation outperforms this by 22% on the geomean. We observe that our TPP convolutions not only run on all of these different platforms
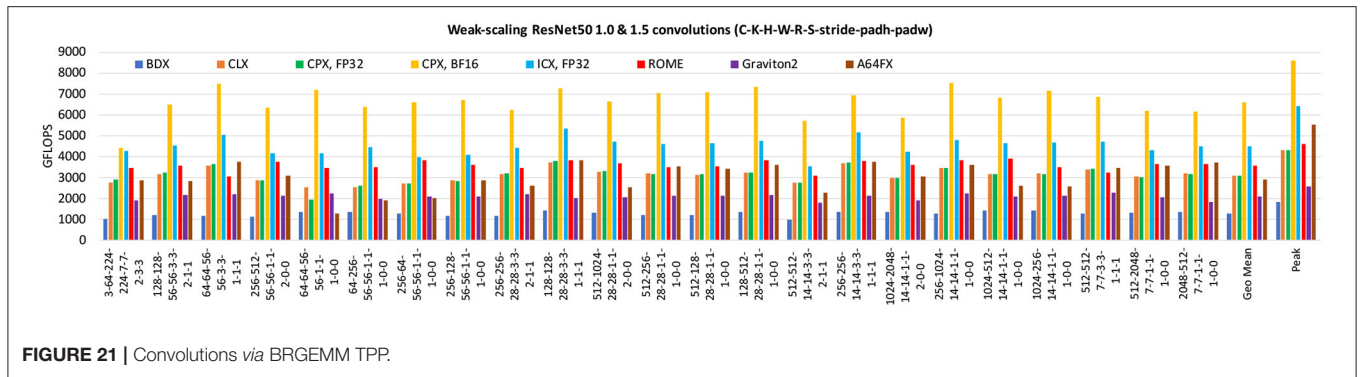
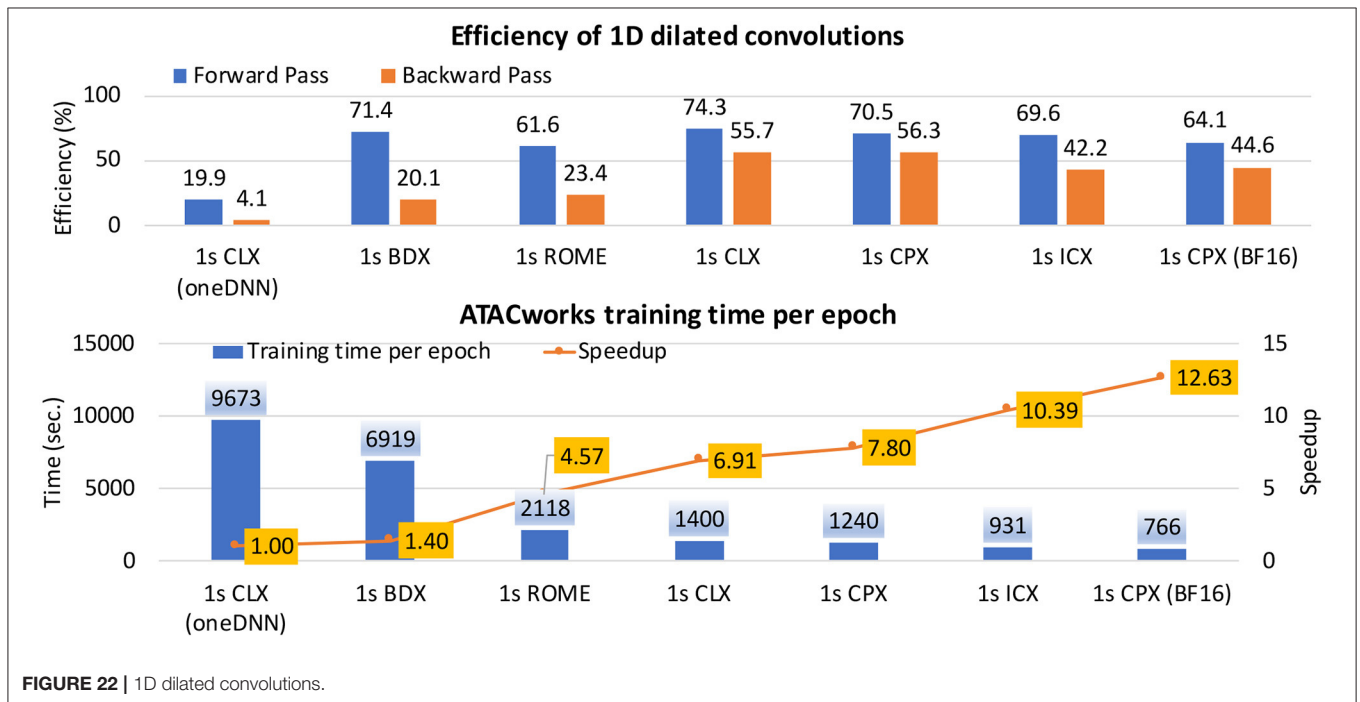**FIGURE 21** | Convolutions *via* BRGEMM TPP.

FIGURE 22 | 1D dilated convolutions.

without a single line of code change, but they run at very similar hardware utilization.

## 6.2. Performance of End-To-End DL Workloads

6.2.1. 1D Dilated Convolutions and Their Application to Computational Biology

Here, we evaluate the oneDNN [13] and TPP-based 1D dilated convolution layer of ATACworks [38] which takes more than 90% of the training time, and it has input tensor width (W) of 60,400, output tensor width (Q) of 60,000, 15 input channels (C), 15 filters (K), filter size (S) of 51, and dilation (d) of 8. **Figure 22-Top** shows the computational efficiency results of the 1D convolution layer. oneDNN is not reaching peak performance for these specialized convolutions, exhibiting 19.9% efficiency for the forward pass and only 4.1% for the backward pass on CLX. Our TPP-based implementation shows 74.3 and 55.7% efficiency for the corresponding training passes. We also highlight the

performance portability of our TPP-based approach across all tested platforms. Finally, we show training time per epoch results for ATACworks in **Figure 22-Bottom**. The TPP-based kernels provide training time speedup of 6.91× on CLX when comparing to the oneDNN based implementation. We also show that by leveraging the BF16 FMA acceleration of the CPX platform we can further obtain 1.62× speedup compared to the FP32 implementation on the same platform. In total BF16 yields 12.6× speedup over the oneDNN baseline.

6.2.2. Deep Learning Recommendation—DLRM

Figure 23-Top shows the FP32 DLRM performance on CLX using two different configurations, namely small DLRM (blue bars) and MLPerf DLRM (orange basrs). We refer to previous work for the detailed specification of these configurations [37]. We evaluated 4 different implementations of DLRM: (i) the PyTorch reference implementation, (ii) PyTorch reference + custom Embedding extension auto-vectorized by the compiler,

FIGURE 23 | DLRM performance on a small config (blue bars) and on the MLPerf config (orange bars).
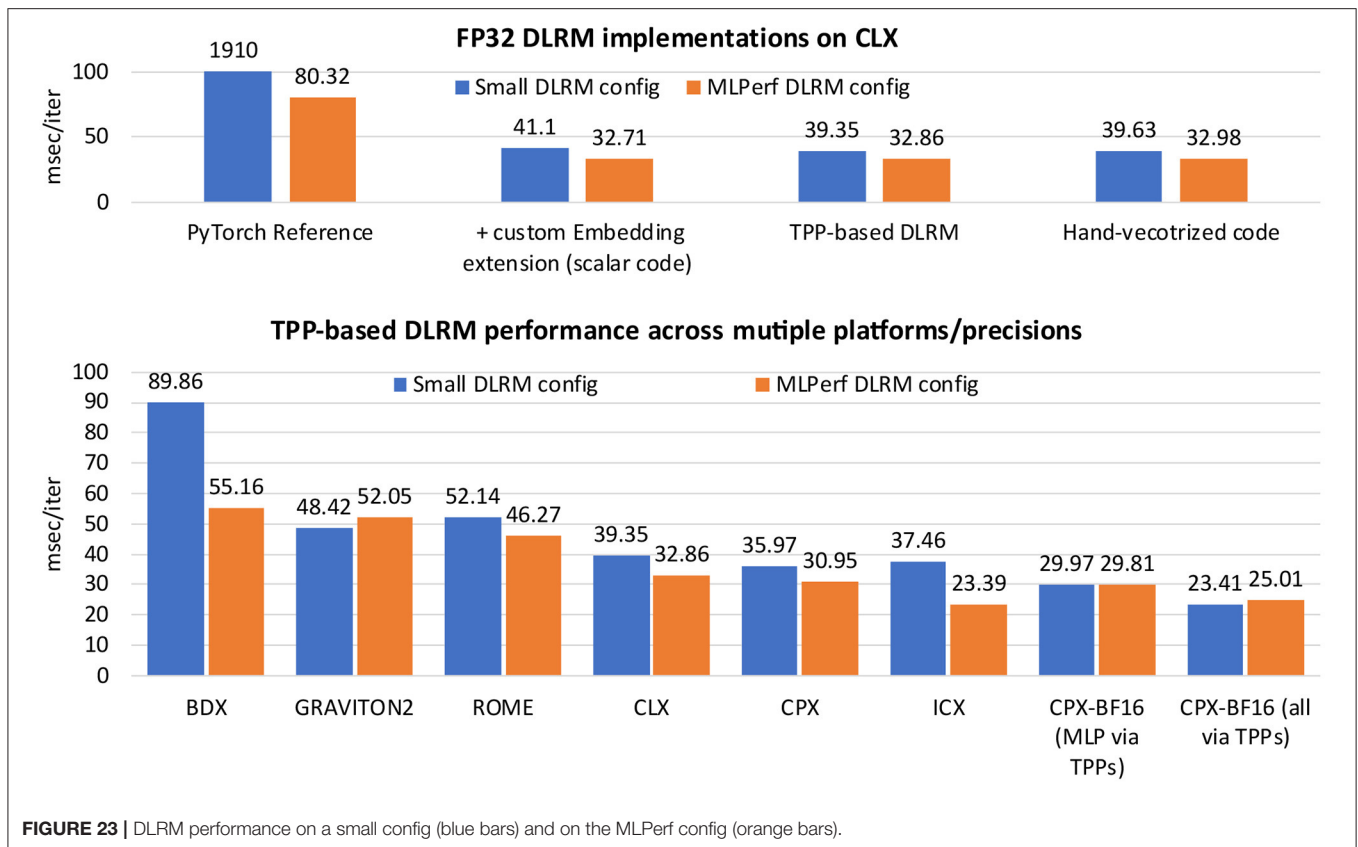
(iii) DLRM expressed entirely *via* TPPs, and (iv) hand-vectorized Embedding extension + BRGEMM-TPP based MLPs [37]. We conclude that the TPP-based implementation matches the performance of the State-Of-The-Art implementation which is hand-vectorized specifically for AVX512 targets; both of these optimized versions substantially outperform the PyTorch CPU reference implementation by up to 48×. Compared to the version with the custom, auto-vectorized variant the TPP-version is up to 4.4% faster.

Figure 23-Bottom shows the DLRM performance of our TPP-based implementation across multiple platforms and compute precisions. We want to highlight two aspects: First, we are able to run the same TPP-code without any change across all platforms, something that is not doable with the hand-vectorized SOTA variant (iv) (since it is not able to run on the AVX2-only BDX and ROME platforms, or on the Graviton2 platform with AArch64 ISA). Second, the TPP-based BF16 shows speedup up to 28% over the variant with auto-vectorized Embedding extension. The culprit here is the mixed precision operations like split-SGD where the compiler struggles to yield efficient code as shown in section 6.1.

Figure 24 illustrates the performance breakdown of the small config on multiple platforms. The blue portions of the bars correspond to the time spent on the Embedding component, the orange parts reflect the MLP portion, and finally the yellow portions correspond to the remaining components of the DLRM workload. We observe that depending on the platform, the time spent on Embedding varies from 29 to 37% of the total execution time, the time spent on MLP is in the range of 33–56% of the total time, and the rest components account for 15–23% of the time. We can also observe the correlation of the MLP performance with the compute capabilities of each platform. For example, on CPX which has native BF16 FMA support, the BF16 MLPs are sped up by ∼2× compared to the FP32 MLPs on the same platform. In regard to the time spent on the Embedding kernel which tends to be bandwidth bound, we observe correlation with the corresponding bandwidth capabilities of the machines.

6.2.3. Natural Language Processing—BERT Large
Figure 25-Top shows end-to-end performance (in examples/second) on CLX for the BERT large SQuAD fine-tuning task in FP32, using a max sequence length of 384 and minibatch of 24. We observe that the TPP-based implementation (blue bar) matches the performance of the AVX512-hand-vectorized code/orange bar. At the same time, our implementation is 1.69× faster than the Reference Hugging Faces CPU reference code [46] (green bar).

Figure 25-Bottom shows the performance of the reference Hugging Faces code (green bars) versus the TPP-based code (blue bars) across multiple platforms (x86 and AArch64/Graviton2) and compute precisions (FP32 for all platforms, and BF16 for the CPX platform). The TPP-based BERT shows speedups ranging from 1.5× to 8.5× over the Hugging Faces code. This result highlights the performance portability through the TPP

FIGURE 24 | DLRM performance breakdown of small config on multiple platforms.
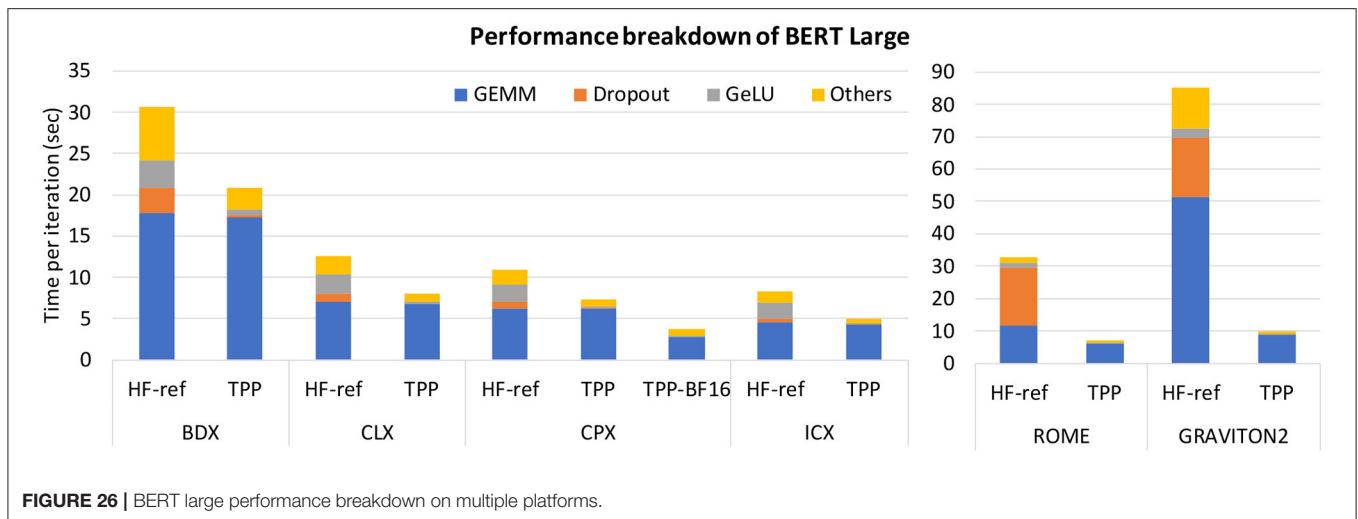

FIGURE 25 | BERT large performance.
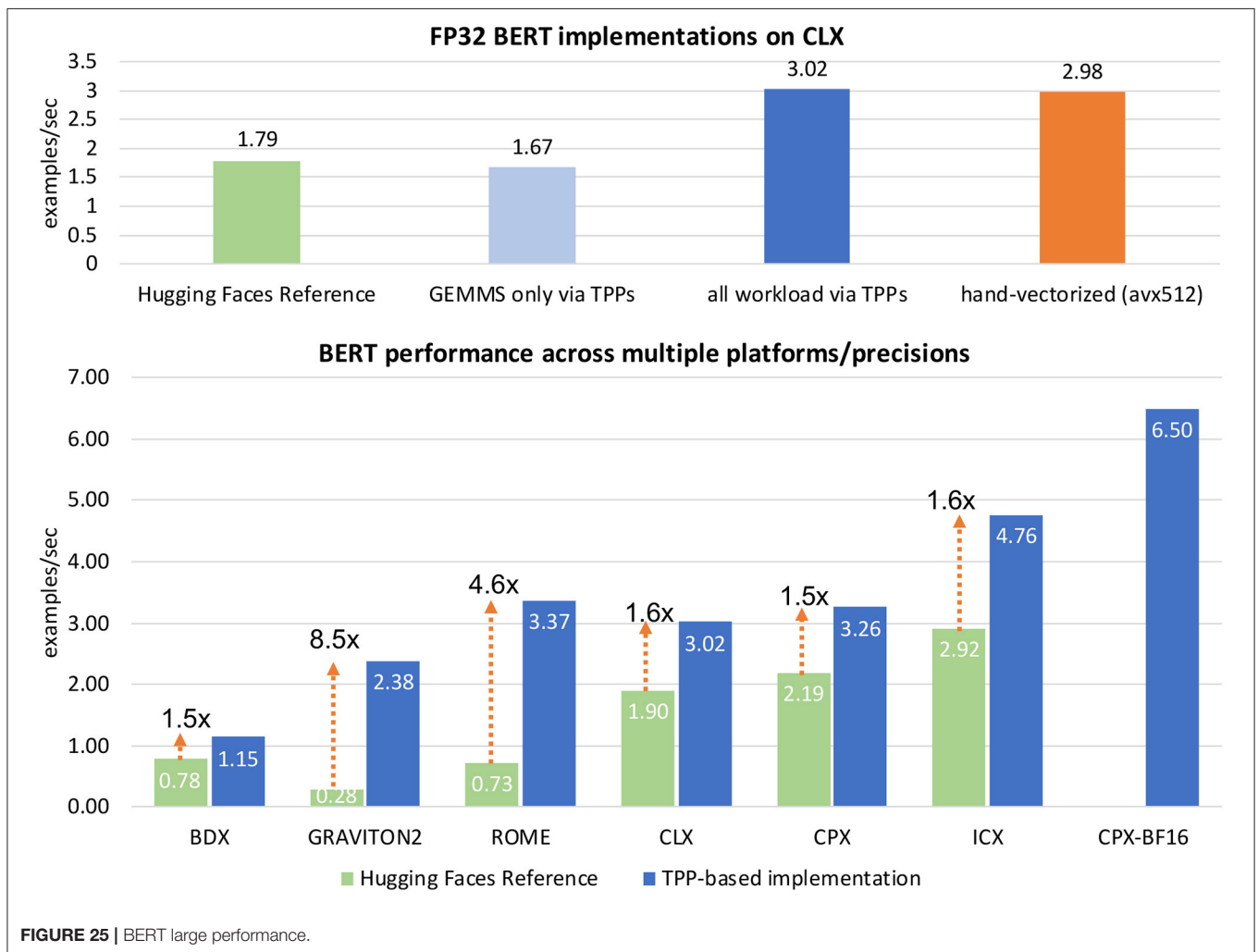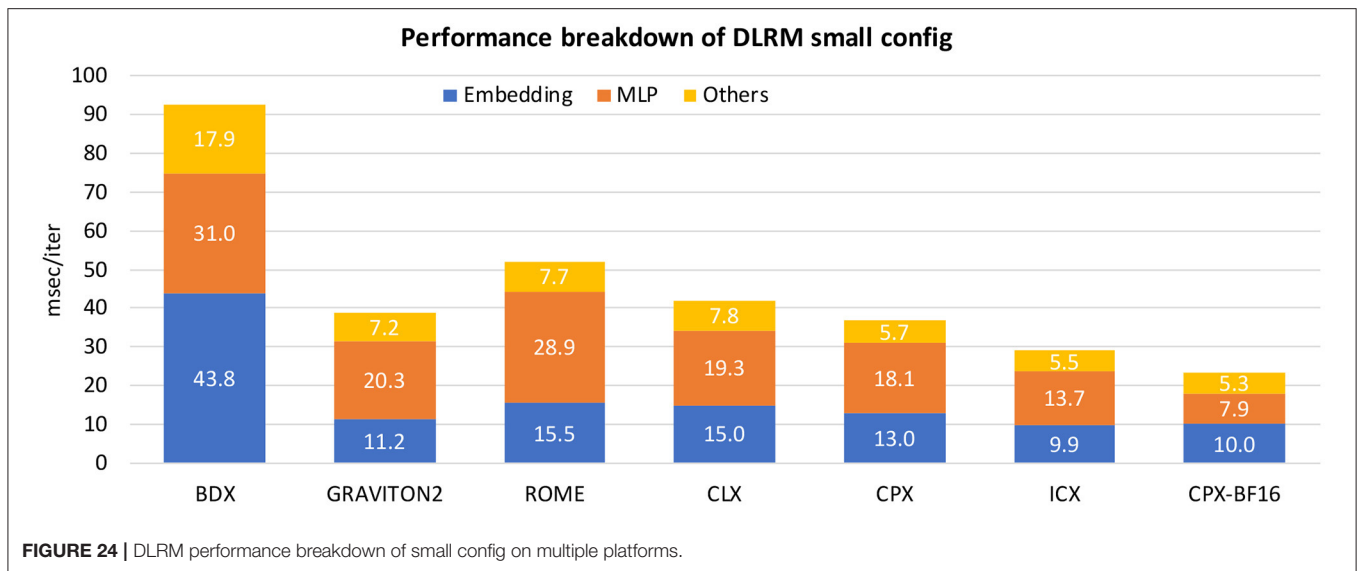
FIGURE 26 | BERT large performance breakdown on multiple platforms.

abstractions. In regard to various compute precisions, we note that with minimal changes inside the fused operators to handle the VNNI tensor layout (required for BF16 GEMM/BRGEMM), and a couple of lines changes in the application code to enable BF16 training, we were able to realize $2\times$ speed up using BF16 training on CPX (compared to FP32 training on CPX) with 28 cores, surpassing 40-core FP32-ICX performance by 37%.

In order shed light on where the benefits are coming from, we present in **Figure 26** the performance breakdown of the Hugging Faces reference code and the TPP-based implementation. In particular we focus on four components:

1. *GEMM* which corresponds to the tensor contractions implemented *via* either the BRGEMM-TPP in the TPP implementation, or it leverages optimized GEMM routines within BLAS libraries in the Hugging Faces implementation (MKL for x86 platforms and OpenBLAS for AArch64/Graviton2).
2. *Dropout* corresponding to the dropout layer in BERT, where the TPP-based implementation employs fast random number generation *via* xorshift algorithm.
3. *GeLU* corresponding to the Gaussian Error Linear Unit activation function in BERT, where the TPP-based implementation leverages fast approximations as discussed in section 3.3.2.
4. *Others* capturing the remaining operators: Transpose, Layernorm, softmax, bias addition, vnni-reformatting (in case of BF16 training), copy, add, scale, zero-kernel, reduce, optimizer. Note that all these operators map to either unary/binary/ternary TPPs (see section 2) or the can be expressed *via* Matrix Equation TPPs (see section 5).
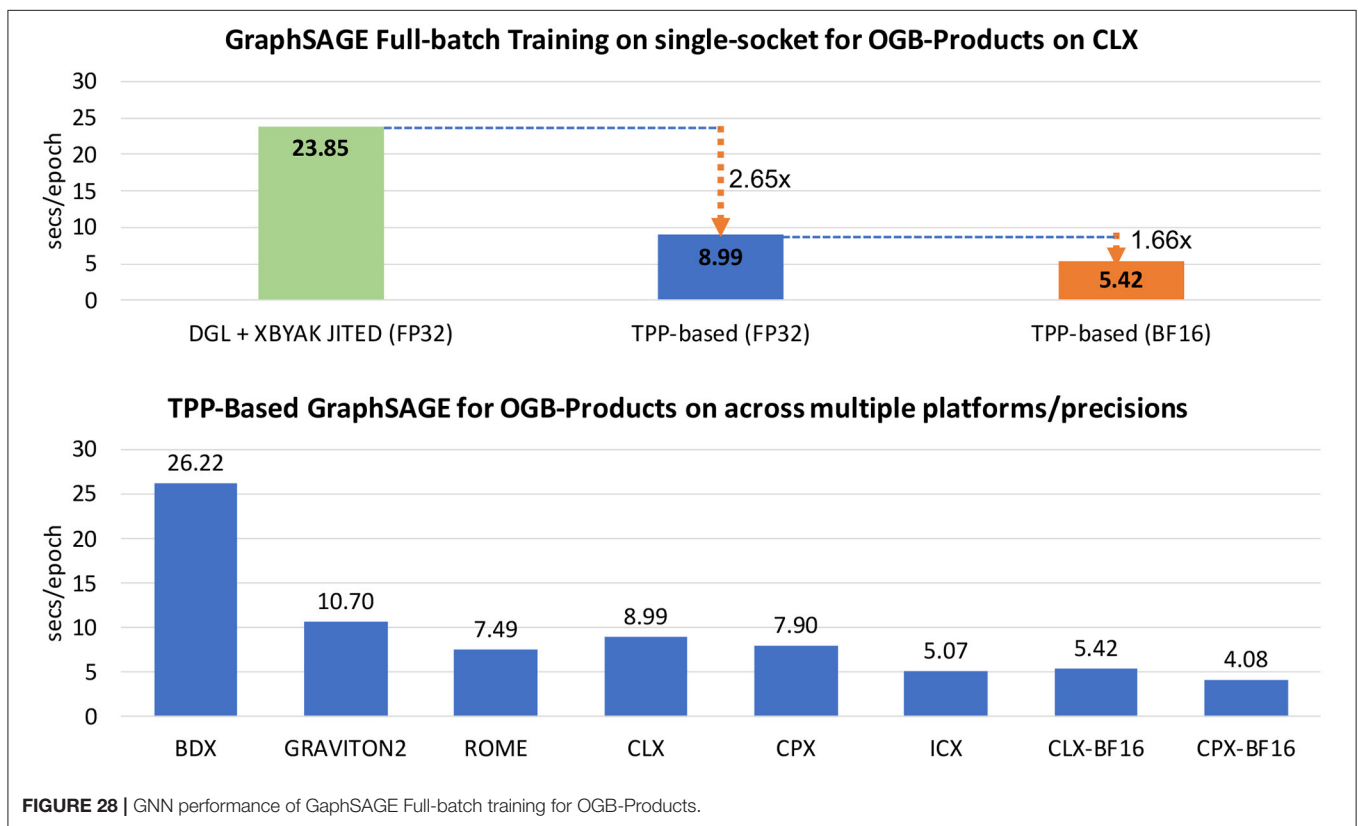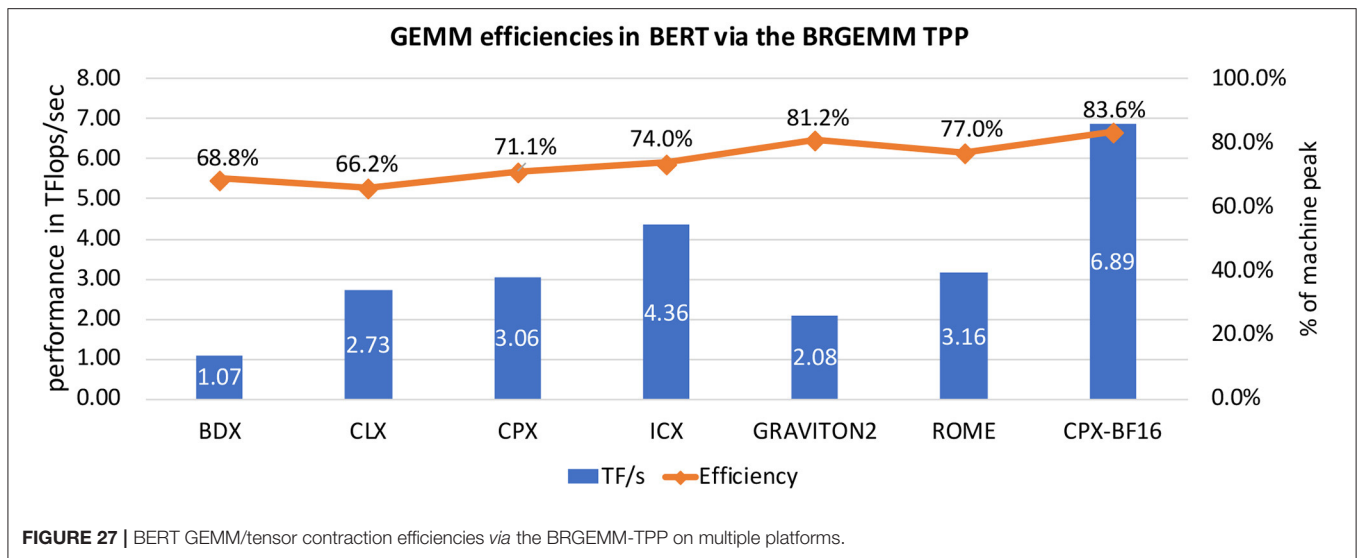
First, we note that for the Intel x86 platforms (left part of the breakdown plot) the tensor contractions show speedups over the highly-optimized MKL GEMM implementation in Hugging Faces in the range of 2–6%. On the right side of the breakdown plot we observe that the BRGEMM-TPP benefits are even larger on the non-Intel platforms. More specifically, on AMD Rome (AVX2 x86 platform) the tensor contractions are sped

up by $1.9\times$ *via* the BRGEMM-TPP, and on Graviton2 (Arm AArch64 platform) the tensors contractions are $5.7\times$ faster *via* the BRGEMM-TPP compared to the implementation relying on OpenBLAS GEMM calls. To further highlight the performance portability of the tensor contractions *via* the BRGEMM-TPP across multiple platforms and precisions, **Figure 27** shows the achieved GEMM performance (Left axis) on each platform for the entire training process (blue bars), whereas the orange line (Right axis) dictates the % of machine peak. The conclusion here is that the BRGEMM-TPP delivers high-efficiency for the corresponding tensor contractions in the range of 66–84% for all tested ISAs and micro-architectures.

The second conclusion we can draw from the performance breakdown in **Figure 26** is that our fused/dataflow TPP implementation outlined in section 5.2.3 makes the dropout and GeLU times shrink substantially, offering speedups in the range of 10–360×. The BERT implementation *via* the dropout/GeLU TPPs in tandem to the BRGEMM TPPs take advantage of temporal locality, and virtually make the corresponding times disappear from the overall execution time. Last but not least, the remaining components are sped-up in the TPP-based implementation by 2.5-14× depending on the platform. As a result of these optimizations, the TPP-based BERT implementation spends the majority of the time (75.5–88.8%) in tensor contractions which are executed at high-efficiency as **Figure 27** shows.

6.2.4. Emerging AI—Graph Neural Networks

Figure 28-**Top** shows end-to-end performance (in seconds/epoch, so lower is better) on CLX for the full-batch training of the GraphSAGE workload on OGB-Products with FP32 and BF16 precision. For the CLX BF16 experiments, since CLX doesn't have native support for BF16 FMAs, we use bit-wise accurate emulated-BF16 BRGEMM TPPs (see section 3.2.2), and we still expect savings due to the bandwidth reduction in the non-GEMM parts, e.g., graph traversal and edge/node aggregation. We observe that the TPP-based implementation outperforms the DGL with Xbyak JIT backend baseline

FIGURE 27 | BERT GEMM/tensor contraction efficiencies *via* the BRGEMM-TPP on multiple platforms.

FIGURE 28 | GNN performance of GaphSAGE Full-batch training for OGB-Products.

version by 2.65×. The TPP-BF16 version yields another 1.66× speedup over the TPP-FP32 variant mainly due to reduced bandwidth requirements.

Figure 28-Bottom shows the performance of the TPP-based code across multiple platforms (x86 and Arm AArch64) and compute precisions (FP32 and BF16). The relative differences in the performance can be justified by the different compute/bandwidth specs of the benchmarked platforms. We

highlight that with minimal changes in the MLP portion to handle VNNI layout required for BF16 BRGEMM, and a couple of lines changes in the application code to enable BF16 training, we were able to realize 1.94× speed up using BF16 training on CPX with 28 cores compared to the FP32 training on the same platform.

In order to further analyze the behavior of the various implementations on multiple platforms, we present on **Figure 29**

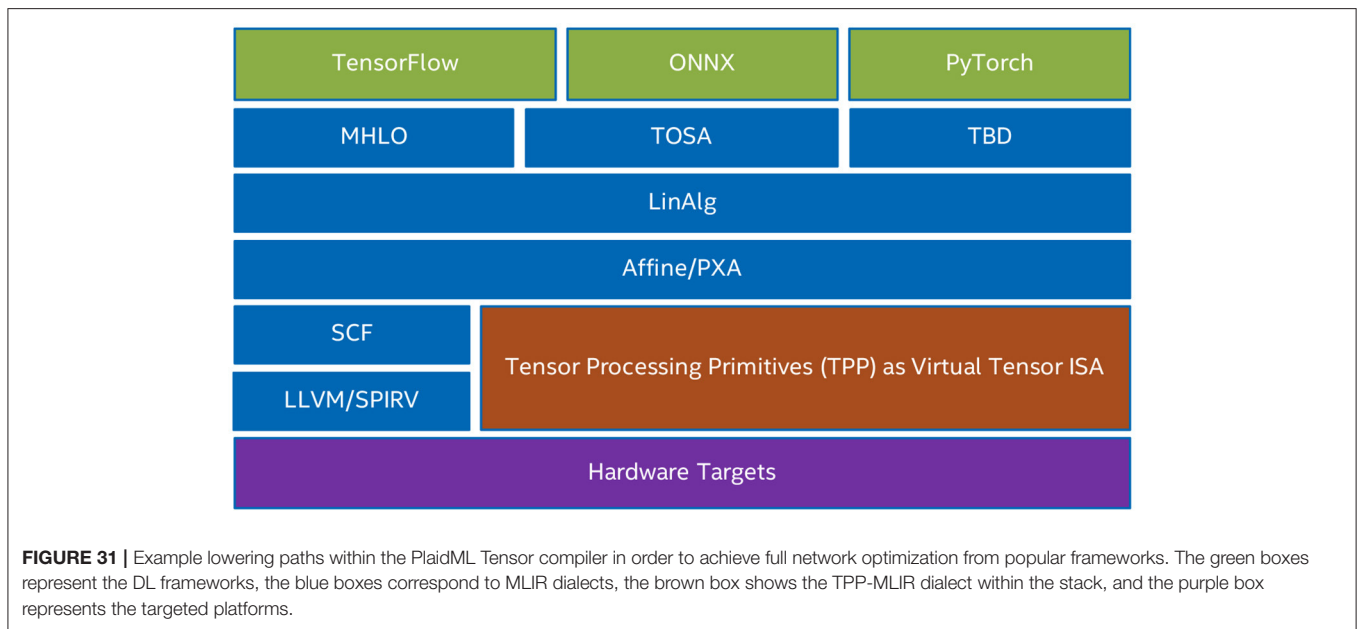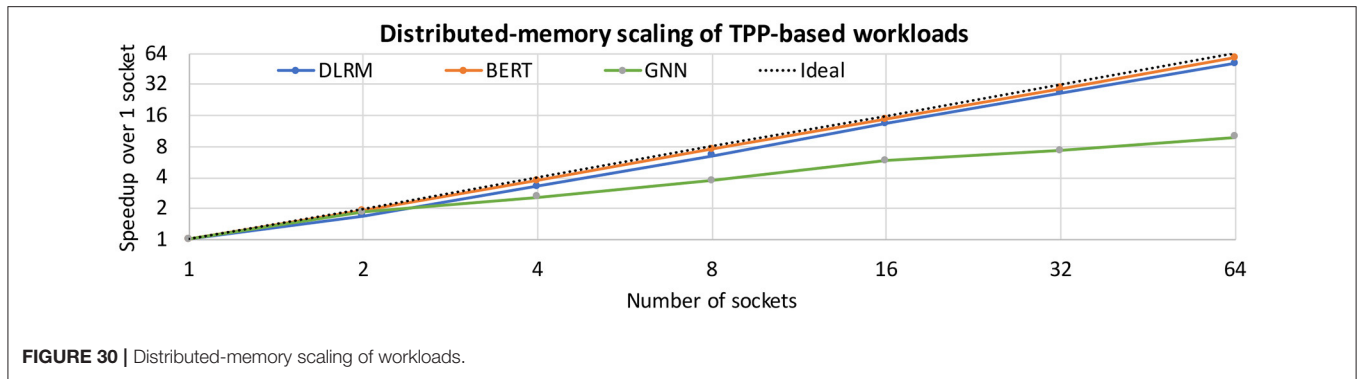**FIGURE 29 |** GNN performance breakdown of GaphSAGE Full-batch training for OGB-Products.
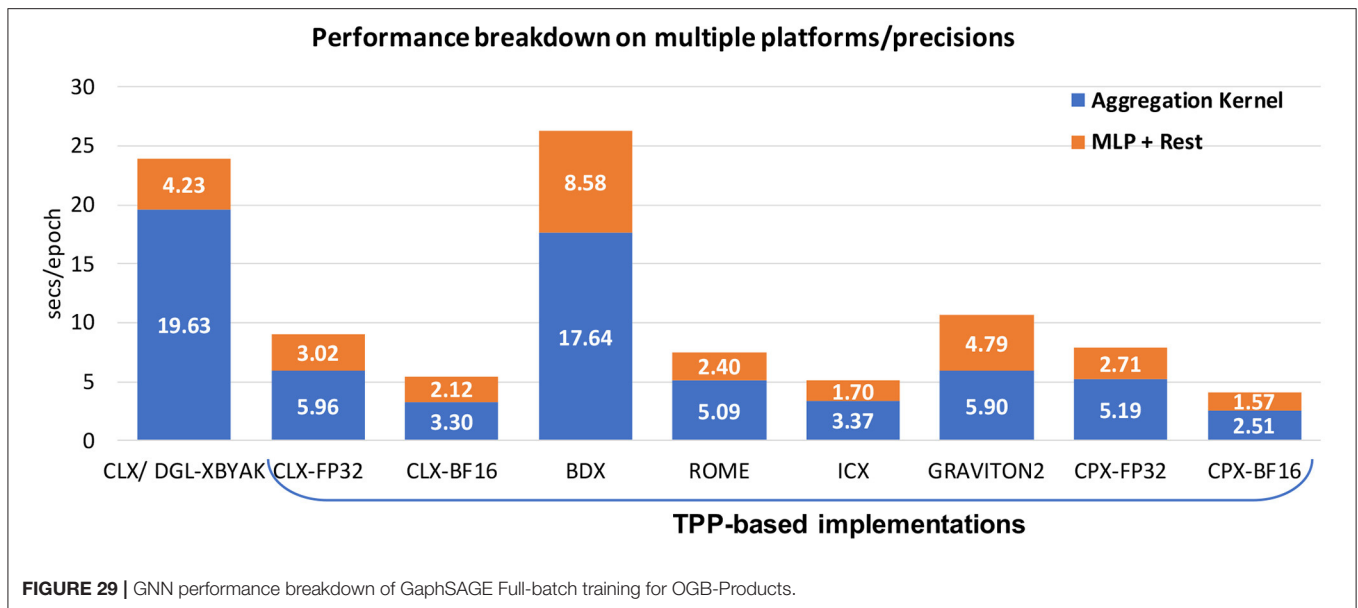
the relevant performance breakdown. The very left bar shows the performance breakdown of the FP32 optimized DGL implementation that leverages JITed kernels through Xbyak on the CLX platform. The blue part corresponds to the Aggregation kernel described in section 5.2.4 whereas the orange portion represents the time required by the remaining kernels, namely Multilayer-Peceptrons with Activation functions. In the DGL implementation the activation functions are not fused within the MLP's tensor contractions. We observe that in this optimized DGL implementation, 82.3% is spent on the Aggregation kernel and only 17.7% is spent on the MLPs. On the second from the left bar (annotated as CLX-FP32) we show the performance of the FP32 TPP-based implementation on the same CLX platform. We conclude that the TPP-based Aggregation kernel exhibits a speedup of $3.29\times$ compared to the DGL-Xbyak implementation, and the TPP-based MLP kernels (BRGEMM-TPP tensor contractions with *fused* TPP activation functions) exhibit a speedup of $1.4\times$ compared to the respective DGL-Xbyak implementation. The FP32 TPP-based implementation spends 66.4% on the aggregation kernel and 33.6% on the fused MLP kernels.

The last 8 bars on **Figure 29** illustrate the performance breakdown of the TPP-based implementation on various platforms (CLX/BDX/ROME/ICX/GRAVITON2/CPX) and various precisions (FP32 and CPX-BF16). We want to emphasize that all these performance numbers are obtained by employing the same exact TPP-based code (which is *platform-agnostic*); the only modification is pertaining to the BF16 TPP code where we changed the tensor layouts in the MLP portion in order to deal with the required VNNI format. When comparing the CPX-F32 and the CPX-BF16 performance breakdowns we observe a $2\times$ speedup on the Aggregation kernel. This kernel is typically bandwidth bound due to its irregular/indexed accesses, and the BF16 TPP code moves half of the data compared to the FP32

TPP code since all the tensors are halved in size (BF16 vs FP32 datatype). The MLP portion of the TPP-based implementation is sped up by $1.73\times$ by using the BF16 BRGEMM-TPP. The CPX platform supports the BF16 FMA instruction which has effectively $2\times$ the compute throughput compared to the FP32 FMA on the same platform. The BF16 BRGEMM-TPP internally leverages this BF16 FMA instruction within the GEMM microkernel on CPX (see section 3.2) to speed up the tensor contraction. Finally, we highlight here the speedup of the Aggregation kernel when, e.g., comparing the CPX and the ICX FP32 TPP-based performance numbers. The ICX platform has STREAM bandwidth of 175 GB/s whereas CPX has 97.7 GB/s, and this trend is reflected also in the performance of the Aggregation kernel ($1.54\times$ faster on ICX than CPX).

6.3. Distributed-Memory Scaling of DL Workloads

Even though we focused on the evaluation of the TPP-based workloads on a single node, our approach is seamlessly incorporated into the DL frameworks, hence we can scale to multiple nodes in a cluster to accelerate the training process employing the oneCCL library [47]. **Figure 30** shows the distributed-memory scaling of the TPP-based workloads. DLRM and BERT show almost perfect weak-scaling from 1 to 64 sockets of CLX (32 nodes) with speedups 51.7 and $57.9\times$, Respectively. Regarding the scaling of the GNN workload, the efficiency is directly affected by the quality of the partitions produced by the graph partitioning tools. Using 64 sockets we achieve $10\times$ speedup compared to single socket, and further scaling improvements constitute future work. We can conclude that TPPs for single node optimizations combined with small-size cluster level execution can accelerate deep learning training on CPUs by up to two orders of magnitude.
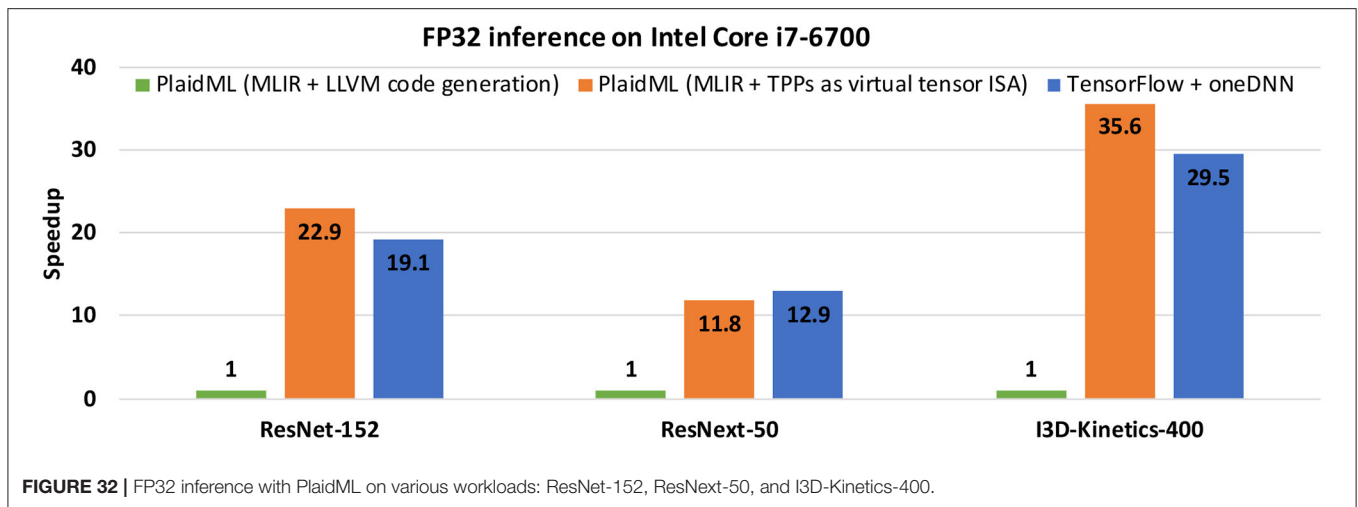
FIGURE 30 | Distributed-memory scaling of workloads.

FIGURE 31 | Example lowering paths within the PlaidML Tensor compiler in order to achieve full network optimization from popular frameworks. The green boxes represent the DL frameworks, the blue boxes correspond to MLIR dialects, the brown box shows the TPP-MLIR dialect within the stack, and the purple box represents the targeted platforms.

7. TPP WITHIN MLIR AND A TENSOR COMPILER

In order to illustrate the viability of TPPs as a virtual Tensor ISA within MLIR and Tensor Compilers, we implemented a rudimentary MLIR dialect corresponding to the TPPs. We also implemented lowering passes within the PlaidML [15] Tensor Compiler that transform intermediate MLIR representations to the TPP-MLIR dialect. The TPP-MLIR dialect is subsequently lowered to the corresponding LIBXSMM TPP calls, therefore such a flow is not relying on LLVM for the code generation of the corresponding tensor operations.

The current lowering path through MLIR supports a variety of front-end interfaces with LinAlg or Tile as the lowest level common entry points, i.e., the lowest level of abstraction that inbound programs can be specified in such that they will be subject to the full range of optimizations necessary to achieve full performance. **Figure 31** details the lowering paths currently implemented in PlaidML and where key transforms map tensor operations into the TPP dialect. The key transformation is located

in the stencil pass of the PXA dialect (Parallel eXtensions for Affine—a staging ground for PlaidML/TPP work that will be proposed upstream to the affine dialect). Operations that cannot be matched to TPP primitives are lowered through standard affine optimization pipelines.

We experimented with the use-case of FP32 inference on a client CPU (Intel i7-6700) on three different workloads: ResNet-152 [33], ResNext-50 [48], and I3D-Kinetics-400 [49]. **Figure 32** shows the results of three implementations: (i) The green bars show the performance of the code generated by PlaidML with MLIR for intermediate representations, and LLVM for the code generation, (ii) The orange bars show the performance of the code generated by PlaidML with MLIR for intermediate representations, and the TPP-MLIR dialect as virtual Tensor ISA for the code generation of the corresponding tensor contractions, and (iii) TensorFlow FP32 inference backed-up by the vendor-optimized oneDNN library. We observe that the Tensor Compiler variant which relies on the TPP-MLIR dialect for the tensor contractions outperforms the variant which relies exclusively on LLVM (for loop-tiling and vectorization) up to

FIGURE 32 | FP32 inference with PlaidML on various workloads: ResNet-152, ResNext-50, and I3D-Kinetics-400.

35.6×. At the same time, PlaidML assisted by the TPP-MLIR dialect matches/outperforms the performance of TensorFlow which uses internally oneDNN, a highly-tuned vendor library for this CPU target. These preliminary results highlight the viability of the synergistic Tensor Compiler—TPP paradigm as discussed in section 1.

8. TPP AND HPC APPLICATIONS

So far, in this article, the focus was on how the TPP abstraction can be leveraged within the Deep Learning Domain. Tensor computations are ubiquitous, and in particular they constitute the cornerstone of many HPC applications. As such, the TPP abstraction can be readily employed by HPC applications to accelerate tensor computations without sacrificing portability. In the rest of this section, we examine how TPPs are used within two HPC applications, namely CP2K and EDGE.

8.1. CP2K

The tensor based formulation originated and became common in physics, and it is well adopted in the field of engineering or applied sciences, and in electronic structure (ES) theory in particular. CP2K is an open source ES- and MD-package (molecular dynamics) for atomistic simulations of solid-state, liquid, molecular, and biological systems [50]. CP2K is striving for good performance on HPC and massively parallel systems. Even though the use of novel algorithms in CP2K is the norm for scientific reasons, implementations have not widely tapped tensors in an explicit fashion. In contrast, Machine Learning emerged with similar, yet not coherent APIs and frameworks around the notions of tensors, layers, and image processing.

While ES calculations can be formulated with tensors of ranks two to four, CP2K (and similar packages) largely remain with matrix based formulation. Various libraries for tensor contractions gained some attraction for scientific applications but the level of generality is key, e.g., as sparse representations are desired. CP2K explored an API for sparse tensor contractions

and published a proof of concept implementation built into the DBCSR library [51]. Efforts targeting accelerators in CP2K, namely GPUs, are not fully booked hence hardware specifically for Deep Learning (with focus on low and mixed precision arithmetic) is not yet a motivation of tensors as an implementation vehicle (and source of acceleration). Therefore, a collection of primitives, such as TPP is well-suited for an emerging discussion of a more general API.

CP2K 3.0 introduced LIBXSMM for Small Matrix Multiplications (SMMs). CP2K and DBCSR (previously part of CP2K's code base) since then additionally introduced element-wise operations (copy and transpose) with "elements" being small matrices based on LIBXSMM. Reformulating existing code to build on (batched) GEMM TPP and element-wise TPP operations is an established pattern for increased performance in CP2K.

To practically improve performance in CP2K one has to consider:

- Fusing kernels and increasing arithmetic intensity independent of the target being a CPU or an accelerator (performance bound by memory bandwidth).
- Specializing code at runtime based on workload/input of the application, e.g., generating code Just-In-Time (JIT) a.k.a. meta-programming.

These objectives can be delivered by TPPs as a domain-specific language (DSL), enabling the scientist to write more abstract code, e.g., by the means of meta-programming, and by relying on a specification which delivers versatile primitives deferring low-level optimizations to the TPP backend.

For CP2K's performance evaluation, we refer to BDX, CLX, ICX, and ROME as introduced earlier (section 6). To show the portability of our approach, we augmented our results by using the Oracle Cloud Infrastructure, namely the result for *Altra* processor (BM.Standard.A1.160 OCI shape). **Table 4** shows the performance benefit of LIBXSMM's GEMM-TPP in CP2K when compared to Intel's MKL GEMM routines.

TABLE 4 | CP2K performance (Cases/Day) of three workloads fitting into single systems with two processors.

System	Workload[a]	BLAS-GEMM[b]	TPP-GEMM[c]	TPP-Speedup
BDX	H2O-256	91	**101**	11%
	H2O-512	23	**27**	17%
CLX	H2O-256	154	**162**	5%
	H2O-512	39	**41**	5%
	H2O-DFT-LS4	45	**47**	4%
ICX	H2O-256	235	**249**	6%
	H2O-512	60	**65**	8%
	H2O-DFT-LS4	67	**70**	4%
ROME	H2O-256	225	**244**	8%
	H2O-512	55	**57**	4%
	H2O-DFT-LS4	**65**	65	0%
Altra	H2O-256	228	**236**	4%
	H2O-512	60	**62**	3%
	H2O-DFT-LS4	60	**66**	10%

Single-socket performance is reported here for consistency within this article. Intel MKL or OpenBLAS are always used for general BLAS operations including large GEMMs. Either (BLAS-)GEMM or TPP-GEMM was used for batched multiplication of small matrices (SMMs). Workloads utilizing CP2K's DBCSR library for distributed block-sparse matrix multiply benefit from (runtime-)specialized GEMM-TPP kernels where the set of matrix shapes is not known at compile-time of the application or depends on the workload in general.

[a] H2O-256 (CP2K bench.), H2O-512 (UEABS Case A) and H2O-DFT-LS NREP=4 (UEABS Case C) from PRACE UEABS 2.1.

[b] Intel MKL (x86-64) or OpenBLAS (otherwise).

[c] LIBXSMM.

The bold values indicate the best-performing implementation for each system and workload.

8.2. EDGE

The Extreme-Scale Discontinuous Galerkin Environment (EDGE) uses the Arbitrary high-order DERivatives (ADER) Discontinuous Galerkin (DG) finite element method to simulate seismic wave propagation [52]. The software uses unstructured tetrahedral meshes which are typically adapted to the used seismic velocity models. Additionally, modelers may introduce mountain topography. A sophisticated local time stepping scheme allows the solver to operate efficiently in very large and complex settings. The software is able to fuse multiple ensemble simulations into one execution of the software. EDGE uses an orthogonal polynomial expansion basis to discretize each of the considered variables in a tetrahedron of the mesh. In a typical setting, we use three relaxation mechanisms for the viscoelastic part, resulting in a total of 27 seismic variables. Additionally using a fifth order method gives us 35 basis functions, resulting in a total of $27 \cdot 35 = 945$ degrees of freedom per tetrahedral element. The solver advances the degrees of freedom in time by repeatedly computing a triplet of quadrature-free integrators. While the actual integrators are part of EDGE, their implementation relies heavily on TPPs. The GEMM-TPP with small and uncommon matrix sizes is the most crucial operation required by EDGE. For example, the surface integrator requires the multiplication of a 9×35 matrix with a 35×15 matrix. The solver's extension

TABLE 5 | Sustained 32-bit floating point performance on the studied systems.

System	GTS		LTS	
	Single	Fused[a]	Single	Fused[a]
Cascade lake	1.08	0.78	1.02	0.74
Ice lake	1.29	1.01	1.23	0.96
Rome	1.20	1.08	1.12	1.01
Milan	**1.39**	**1.16**	1.29	**1.07**
Altra	1.27	0.73	**1.51**	0.76

The performance is given in TFLOPS. Results are presented for Global Time Stepping (GTS) and Local Time Stepping (LTS) when using single and fused forward simulations.
[a] EDGE's fused simulations use sparse matrix kernels.
The bold values indicate the best performing system for each experimental setup.

with additional, performance-portable TPPs in all parts of the integrators is work-in-progress. Especially, EDGE's support for viscoelastic attenuation or local time stepping requires "simpler" kernels, e.g., the unary TPPs Identity and Zero, or the binary TPPs Mul, Sub and Add.

We evaluate EDGE's performance-portability through the use of TPPs by studying the performance of a full setup of the Layer Over Halfspace 3 (LOH3) benchmark with 743,066 tetrahedral elements. The same setting was also used in Breuer and Heinecke [53] to study the performance of the solver on a single processor of the Frontera supercomputer located at the Texas Advanced Computing Center (position ten in the 06/21 TOP500-list). Following this study, a sophisticated simulation of the 2014 M_w 5.1 La Habra earthquake using a mesh with 237,861,634 tetrahedral elements and EDGE's advanced features yielded a performance of 2.20 FP32-PFLOPS on 1,536 nodes.

For the EDGE application, we study the software's raw floating point performance and time-to-solution by extending our LOH3-Frontera-only study [53] with diverse processors:

- *Cascade Lake* (similar to *CLX* as introduced in section 6): 2.7 GHz 28-core Intel Xeon Platinum 8,280 processor of the Frontera system at the Texas Advanced Computing Center. We only used a single 28-core processor of Frontera's dual-socketed compute nodes in our tests.
- *Ice Lake:* 2.3 GHz 40-core Intel Xeon Platinum 8,380 processor on Intel's on-premises cluster. We only used a single 40-core processor of the dual-socket compute nodes in our tests.
- *Rome* (similar to *ROME* as introduced in section 6): 2.25 GHz AMD EPYC 7,742 (BM.Standard.E3.128 OCI shape). We only used a single 64-core processor of the bare metal instance in our tests.
- *Milan:* 2.55 GHz AMD EPYC 7J13 (BM.Standard.E4.128 OCI shape). We only used single 64-core processor of the bare metal instance in our tests.
- *Altra:* 3.0 GHz Ampere Altra Q80-30 processor (BM.Standard.A1.160 OCI shape). We only used a single 80-Armv8.2-core processor of the bare metal instance in our tests.

Table 5 shows the sustained floating point performance of the conducted runs. All numbers are given in FP32-TFLOPS. Columns two and three present the performance of Global Time

TABLE 6 | Time-to-solution speedups of the studied systems when using different configurations of the solver EDGE.

System	GTS		LTS	
	Single	Fused	Single	Fused
Cascade lake	1.00	1.80	2.50	4.52
Ice lake	1.19	2.33	3.02	5.87
Rome	1.11	2.48	2.76	6.17
Milan	**1.28**	**2.67**	3.18	**6.55**
Altra	1.18	1.69	**3.71**	4.64

*The performance of the Cascade Lake system, running EDGE with Global Time Stepping (GTS) and a single forward simulation, is used as baseline. In contrast to **Table 5**, the speedups include the higher algorithmic efficiencies of EDGE's support for Local Time Stepping (LTS) and fused forward simulations.*
The bold values indicate the best performing system for each experimental setup.

Stepping (GTS), whereas columns four and five show that of Local Time Stepping (LTS). In general, the LTS configurations have a slightly lower peak utilization when compared to their GTS counterparts. Note, however, that **Table 5** only shows raw floating point performance and does not account for time-to-solution speedups through LTS (theoretically up to $2.67\times$ in this case). The performance of GTS and LTS is further split into running a single forward simulation and fusing multiple simulations. In fused mode, the solver parallelizes over the right-hand-side by concurrently simulating seismic wave propagation for a collection of seismic sources. One of the fused mode's unique advantages is the opportunity for perfect vectorization of all small matrix multiplications, even when considering sparsity [52]. In this work, we matched the microarchitectures' SIMD-length by fusing 16 simulations on Cascade Lake and Ice Lake, eight simulations on Rome and Milan, and four simulations on Altra. Once again, note that **Table 5** does not include the respective sparsity-driven $2.49\times$ increase of the floating point operations' value when running fused simulations. Comparing the performance of the different systems, we observe very high overall performance with architectural efficiency gains originating from decreasing SIMD-lengths. This is especially noticeable when running single forward simulations. In this case, the vectorized dimension of the small dense matrix kernels coincides with the number of basis functions, i.e., $M = 35$, which is challenging when optimizing for AVX512 (Cascade Lake and Ice Lake) and AVX2 (Rome and Milan). The short 128-bit ASIMD vector instruction (Altra) reach a very high peak utilization of 33.2% for GTS and 39.2% in LTS. For the fused simulations, the differences in relative peak utilization narrow further.

Table 6 describes the obtained performance numbers in terms of time-to-solution. Here, we use the runtime of the studied LOH3 setting on Cascade Lake for GTS and a single forward simulation as baseline. All other settings are given relative to this. Further, for the fused settings, we consider the per-simulation time. We observe that EDGE's overall performance is driven by the high floating point performance through the use of TPPs and the solver's advanced algorithmic features. Here, Altra performs best for single forward simulations using LTS, accelerating the

baseline by $3.71\times$. Milan has the best time-to-solution in all other settings and is able to outperform the baseline by $6.55\times$ when using LTS and fusing simulations. This performance lead originates from Milan's high theoretical peak combined with a high peak utilization (see **Table 5**).

9. RELATED WORK

The related work in terms of the development methodology of DL workloads has been referenced in the introduction, so here we mention community efforts that share the same design philosophy with TPPs. Tensor Operator Set Architecture (TOSA) is a recent work, concurrently developed with TPPs, that provides a set of whole-tensor operations commonly employed in DL [54]. TOSA allows users to express directly operators on up to 4D/5D tensors which are not naturally mapped even on contemporary 2D systolic hardware. We believe that staying at the 2D primitive level is expressive and sufficient, as we can build higher-order ops with loops around 2D operators, e.g., see **Algorithm 6**. Despite the similarities of TPP and TOSA specifications, the TOSA back-end is reference C code and is not showcased in full DL-workloads. CUTLASS [55] and Triton [56] strive for high-performance on GPUs, while also offer flexible composition that can be easily applied to solve new problems related in DL and linear algebra, and share many design principles with TPPs.

XLA [57] is a domain-specific compiler for linear algebra and DL that targets TensorFlow models with potentially no source code changes. JAX [58] provides automatic differentiation of Python and NumPy functions, and the compilation of the desired operators happens in a user-transparent way with JIT calls, yielding optimized XLA kernels. XLA and JAX share the same philosophy with TPPs: the user is focusing on the DL kernel/workload development using high-level, platform-agnostic, declarative-style programming, whereas the tensor-aware back-end infrastructure undertakes the efficient and portable code generation. Julia [59] is a high-level, dynamic programming language, designed to give users the speed of C/C++ while remaining easy to use. Since its incarnation, Julia has evolved with a strong Deep Learning/Machine Learning ecosystem, providing optimized libraries for such workloads. We envision that TPPs and tensor compilation frameworks (like JAX and Julia) will coexist in a synergistic fashion. For example, a program written in JAX could be lowered *via* an MLIR pass to the Linalg dialect, and from there the compilation stack could follow the path illustrated in **Figure 31** (JAX→Linalg→Affine/PXA→TPP) in order to leverage TPPs for efficient code generation. To this extend, Tensor Processing Primitives serve as a virtual tensor ISA within tensor compilation frameworks rather than trying to replace them.

Tensor Compilers (TC) [15–18] attempt to optimize DL operators in a platform-agnostics way, however their applicability is restricted to relatively small code-blocks whereas full workload integration is cumbersome. Also, TC undertake the tasks of efficient parallelization, loop re-ordering, automatic tiling and

layout transformations, nevertheless the obtained performance is typically underwhelming [12]. We envision that TPPs can be used as a tool by TC in order to attain efficient platform-specific code generation, therefore, TC could focus on optimizing the higher level aspects of the tensor programs (e.g., layout transformations). Along these lines, TPPs fit in the MLIR [20] ecosystem/stack as a lowering dialect (see section 7), and in this way the TPP back-end could be leveraged by multiple TC frameworks.

Tensor computations are also ubiquitous in HPC (e.g., physics, quantum chemistry, numerical simulations) and consequently a plethora of tensor computation frameworks have emerged to facilitate the development of such applications [60–64]. Typically these frameworks are comprised of a front-end that enables the expression of the underlying tensor computations (and can be domain-specific), and a back-end that optimizes the expressed computations using both high-level and low-level techniques. Since TPPs are agnostic of the user-entity, we envision that such tensor computation frameworks can leverage TPPs as a virtual tensor ISA instead of relying on generic compilers or low-level customized generators for efficient code generation across multiple platforms.

10. CONCLUSIONS AND FUTURE WORK

In this work, we presented the Tensor Processing Primitives (TPP), a compact, yet versatile set of 2D-tensor operators, which subsequently can be utilized as building-blocks to construct efficient, portable complex DL operators on high-dimensional tensors. We also show how TPPs can be used within HPC applications in order to accelerate tensor computations. We demonstrate the efficacy of our approach using standalone kernels and end-to-end training DL-workloads (CNNs, dilated convolutions, DLRM, BERT, GNNs) expressed entirely *via* TPPs that outperform state-of-the-art implementations on multiple platforms. As future work, we plan to create a full-fledged TPP-based MLIR dialect such that Tensor Compilers can leverage the strengths of TPPs. Also, we plan to further enrich the TPP back-end implementation by supporting more ISAs, including GPUs and POWER architectures.

11. OPTIMIZATION NOTICE

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other.

AUTHOR'S NOTE

This is an extended version of a technical paper presented at SC21 [65].

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: https://github.com/hfp/libxsmm/tree/sc21.

AUTHOR CONTRIBUTIONS

EG, AH, DKa, and SA worked on the design of the TPP framework. EG, AH, DKa, SA, MA, DA, CA, AB, AK, and BZ worked on the implementation of the TPP backend. EG, DKa, SA, NC, VM, SM, RM, and AH worked on the Deep Learning applications. AB and HP worked on the HPC applications. JB, DKu, FL, and BR worked on the Tensor Compiler integration. All authors read and approved the manuscript.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fams. 2022.826269/full#supplementary-material

REFERENCES

1. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Burges L, Weinberger KQ, editors. *Advances in Neural Information Processing Systems.* Curran Associates, Inc (2012). Available online at: https://proceedings. neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

2. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* Boston, MA (2015). p. 1–9.

3. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint* arXiv:14091556. (2014).

4. Yu D, Seltzer ML, Li J, Huang JT, Seide F. Feature learning in deep neural networks-studies on speech recognition tasks. *arXiv preprint* arXiv:13013605. (2013).

5. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, et al. Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint* arXiv:160908144. (2016).

6. Cheng HT, Koc L, Harmsen J, Shaked T, Chandra T, Aradhye H, et al. Wide & deep learning for recommender systems. In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.* New York, NY: ACM (2016). p. 7–10.

7. Wolf T, Chaumond J, Debut L, Sanh V, Delangue C, Moi A, et al. Transformers: state-of-the-art natural language processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* (2020). p. 38–45.

8. Gawehn E, Hiss JA, Schneider G. Deep learning in drug discovery. *Mol. Inf.* (2016) 35:3–14. doi: 10.1002/minf.201501008

9. Goh GB, Hodas NO, Vishnu A. Deep learning for computational chemistry. *J Comput Chem.* (2017) 38:1291–307. doi: 10.1002/jcc.24764

10. Raghu M, Schmidt E. A survey of deep learning for scientific discovery. *arXiv preprint* arXiv:200311755. (2020).

11. Alom MZ, Taha TM, Yakopcic C, Westberg S, Sidike P, Nasrin MS, et al. A state-of-the-art survey on deep learning theory and architectures. *Electronics.* (2019) 8:292. doi: 10.3390/electronics8030292

12. Barham P, Isard M. Machine learning systems are stuck in a rut. In: *Proceedings of the Workshop on Hot Topics in Operating Systems.* New York, NY: (2019). p. 177–83.

13. oneDNN. *Intel oneDNN GitHub.* Available online at: https://github.com/oneapi-src/oneDNN (accessed online at March 30, 2021).

14. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, et al. cudnn: efficient primitives for deep learning. *arXiv preprint* arXiv:14100759. (2014).

15. Zerrell T, Bruestle J. Stripe: tensor compilation via the nested polyhedral model. *arXiv preprint* arXiv:190306498. (2019).

16. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, 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).* Berkeley, CA (2018). p. 578–94.

17. Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses WS, et al. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. *arXiv preprint* arXiv:180204730. (2018).

18. Zheng L, Jia C, Sun M, Wu Z, Yu CH, Haj-Ali A, et al. Ansor: generating high-performance tensor programs for deep learning. In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20).* Berkeley, CA (2020). p. 863–79.

19. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, et al. The deep learning compiler: a comprehensive survey. *IEEE Trans Parallel Distrib Syst.* (2020) 32:708–27. doi: 10.1109/TPDS.2020.3030548

20. MLIR. *Multi-Level Intermediate Representation GitHub.* Available online at: https://github.com/tensorflow/mlir (accessed online at March 30, 2021).

21. Georganas E, Banerjee K, Kalamkar D, Avancha S, Venkat A, Anderson M, et al. Harnessing deep learning via a single building block. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* New Orleans, LA: IEEE (2020). p. 222–33.

22. Heinecke A, Henry G, Hutchinson M, Pabst H. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16.* Piscataway, NJ: IEEE Press (2016). p. 84:1–84:11.

23. Georganas E, Avancha S, Banerjee K, Kalamkar D, Henry G, Pabst H, et al. Anatomy of high-performance deep learning convolutions on simd architectures. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* Dallas, TX: IEEE (2018). p. 830–41.

24. Bfloat16. *Using bfloat16 With TensorFlow Models.* Available online at: https://cloud.google.com/tpu/docs/bfloat16 (accessed online at April 3, 2019).

25. Marsaglia G, et al. Xorshift rngs. *J Stat Softw.* (2003) 8:1–6. doi: 10.18637/jss.v008.i14

26. Banerjee K, Georganas E, Kalamkar DD, Ziv B, Segal E, Anderson C, et al. Optimizing deep learning rnn topologies on intel architecture. *Supercomput Front Innov.* (2019) 6:64–85. doi: 10.14529/jsfi190304

27. Intel-ISA. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference.* Available online at: https://software.intel.com/content/dam/develop/public/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf (accessed online at March 30, 2021)

28. *city Powell MJD. *Approximation Theory and Methods.* Cambridge University Press. (1981).

29. Chebyshev-Polynomials. *Chebyshev Polynomials.* Available online at: https://en.wikipedia.org/wiki/Chebyshev_polynomials (accessed online at September 26, 2021)

30. Flajolet P, Raoult JC, Vuillemin J. The number of registers required for evaluating arithmetic expressions. *Theor Comput Sci.* (1979) 9:99–125.

31. Gibbs JW. *Elementary Principles in Statistical Mechanics: Developed with Especial Reference to the Rational Foundation of Thermodynamics.* Cambridge: Cambridge University Press (2010). doi: 10.1017/CBO9780511686948

32. Ioffe S, Szegedy C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In: *International Conference on Machine Learning.* Lille: PMLR. (2015). p. 448–56.

33. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* (2016). p. 770–78.

34. Ba JL, Kiros JR, Hinton GE. Layer normalization. *arXiv preprint* arXiv:160706450. (2016).

35. Wu Y, He K. Group normalization. In: *Proceedings of the European Conference on Computer Vision (ECCV).* Munich (2018). p. 3–19.

36. Ulyanov D, Vedaldi A, Lempitsky V. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint* arXiv:160708022. (2016).

37. Kalamkar D, Georganas E, Srinivasan S, Chen J, Shiryaev M, Heinecke A. Optimizing deep learning recommender systems training on CPU cluster architectures. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* Atlanta, GA: IEEE (2020). p. 1–15.

38. Lal A, Chiang ZD, Yakovenko N, Duarte FM, Israeli J, Buenrostro JD. AtacWorks: a deep convolutional neural network toolkit for epigenomics. *bioRxiv.* (2019).

39. Naumov M, Mudigere D, Shi HJM, Huang J, Sundaraman N, Park J, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint* arXiv:190600091. (2019).

40. Devlin J, Chang MW, Lee K, Toutanova K. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint* arXiv:181004805. (2018).

41. Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, et al. Transformers: state-of-the-art natural language processing. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Association for Computational Linguistics (2020). p. 38–45.

42. Zhang M, Rajbhandari S, Wang W, He Y. Deepcpu: serving rnn-based deep learning models 10x faster. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18).* Boston, MA (2018). p. 951–65.

43. Hamilton WL, Ying R, Leskovec J. Inductive representation learning on large graphs. *arXiv preprint* arXiv:170602216. (2017).

44. Avancha S, Md V, Misra S, Mohanty R. Deep Graph Library Optimizations for Intel (R) x86 Architecture. *arXiv preprint* arXiv:200706354. (2020).

45. oneDNN Fugaku. *A Deep Dive Into a Deep Learning Library for the A64FX Fugaku CPU - The Development Story in the Developer's Own Words Fujitsu.* Available online at: https://blog.fltech.dev/entry/2020/11/19/fugaku-onednn-deep-dive-en (accessed online at April 9, 2021).

46. Hugging-Faces. *Hugging Faces GitHub.* Available online at: https://github.com/huggingface/transformers (accessed online at April 9, 2021).

47. oneCCL. *Intel oneCCL GitHub.* Available online at: https://github.com/oneapi-src/oneCCL (accessed online at March 30, 2021).

48. Xie S, Girshick R, Dollár P, Tu Z, He K. Aggregated residual transformations for deep neural networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* Honolulu, HI (2017). p. 1492–500.

49. Carreira J, Zisserman A. Quo vadis, action recognition? a new model and the kinetics dataset. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* Honolulu, HI (2017). p. 6299–308.

50. Kühne TD, Iannuzzi M, Del Ben M, Rybkin VV, Seewald P, Stein F, et al. CP2K: an electronic structure and molecular dynamics software package - Quickstep: efficient and accurate electronic structure calculations. *J Chem Phys.* (2020) 152:194103. doi: 10.1063/5.0007045

51. Sivkov I, Seewald P, Lazzaro A, Hutter J. DBCSR: a blocked sparse tensor algebra library. *CoRR.* (2019) abs/1910.13555.

52. Breuer A, Heinecke A, Cui Y. EDGE: extreme scale fused seismic simulations with the discontinuous galerkin method. In: Kunkel JM, Yokota R, Balaji P, Keyes D, editors. *High Performance Computing.* Cham: Springer International Publishing. (2017). p. 41–60.

53. Breuer A, Heinecke A. *Next-Generation Local Time Stepping for the ADER-DG Finite Element Method. arXiv[Preprint].* (2022). arXiv: 2202.10313. Available online at: https://arxiv.org/abs/2202.10313

54. TOSA. *TOSA.* Available online at: https://developer.mlplatform.org/w/tosa/ (accessed online at March 30, 2021).

55. CUTLASS. *NVIDIA CUTLASS GitHub.* Available online at: https://github.com/NVIDIA/cutlass (accessed online at March 30, 2021).

56. Tillet P, Kung H, Cox D. Triton: an intermediate language and compiler for tiled neural network computations. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* New York, NY (2019). p. 10–9.

57. XLA. *XLA: Optimizing Compiler for Machine Learning.* Available online at: https://www.tensorflow.org/xla (accessed online at March 30, 2021).

58. JAX. *JAX: Autograd and XLA.* Available online at: https://github.com/google/jax (accessed online at March 30, 2021).

59. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: a fresh approach to numerical computing. *SIAM Rev.* (2017) 59, 65–98. doi: 10.1137/141000671

60. Solomonik E, Matthews D, Hammond JR, Stanton JF, Demmel J. A massively parallel tensor contraction framework for coupled-cluster computations. *J Parallel Distrib Comput.* (2014) 74:3176–90. doi: 10.1016/j.jpdc.2014.06.002

61. Solomonik E, Hoefler T. Sparse tensor algebra as a parallel programming model. *arXiv preprint* arXiv:151200066. (2015).

62. Springer P, Bientinesi P, Wellein G. High-performance tensor operations: tensor transpositions, spin summations, and tensor contractions. *Fachgruppe Informatik.* (2019). Available online at: http://publications.rwth-aachen.de/record/755345/files/755345.pdf

63. Hirata S. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J Phys Chem A.* (2003) 107:9887–97. doi: 10.1021/jp034596z

64. Epifanovsky E, Wormit M, Kuś T, Landau A, Zuev D, Khistyaev K, et al. *New Implementation of High-Level Correlated Methods Using a General Block Tensor Library for High-Performance Electronic Structure Calculations.* Wiley Online Library. (2013). Available online at: https://onlinelibrary.wiley.com/doi/10.1002/jcc.23377

65. Georganas E, Kalamkar D, Avancha S, Adelman M, Anderson C, Breuer A, et al. Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* New York, NY (2021). p. 1–14.

GLOSSARY

Intel Pseudo Intrinsics

1. **_mm128** Represents a vector of width 128 bits.
2. **_mm128_loadu_ps(addr)** Loads 16byte of 32 bit elements.
3. **_mm128_storeu_ps(addr)** Stores 16byte of 32 bit elements.
4. **_mm128_unpacklo_ps(A, B)** Unpacks and interleaves 32 bit elements from the low half of A and B.
5. **_mm128_unpackhi_ps(A, B)** Unpacks and interleaves 32 bit elements from the high half of A and B.
6. **_mm128_unpacklo_pd(A. B)** Unpacks and interleaves 64 bit elements from the low half of A and B.
7. **_mm128_unpackhi_pd(A, B)** Unpacks and interleaves 64 bit elements from the high half of A and B.
8. **_mm512** Represents a vector of width 512 bits.
9. **_mm512_permutexvar_ps(A,B)** Shuffle single precision floating point elements in 512 wide vector length using indexes specified in B.
10. **_mm512_roundscale_ps(A,B)** Round single precision floating point elements to the rounding mode specified by argument B.
11. **_mm512_sub_ps(A,B)** Subtract single precision floating point elements in A from B.
12. **_mm512_scalef_ps(A,B)** Scales single precision floating point elements in A using values specified in B.
13. **_mm512_range_ps(A,B, int imm8)** Calculates the min, max or absolute max for each single precision- floating point elements in A and B. Lower 2 bits of imm8[1:0] specifies the operation(min/max/absolute max) to be performed.
14. **_mm512_xor_ps(A,B)** Performs XOR operation between each single precision floating point elements in A and B vector.
15. **_mm512_and_ps(A,B)** Performs AND operation between each single precision floating point elements in A and B vector.
16. **_mm512_rcp14_ps(A,B)** Calculates approximate reciprocal of each single precision floating point element in range less then 2^{14}.
17. **_mm512_cmp_ps_mask(A,B,int C)** Compare the single precision elements in A and B specified by the comparison mode in C.
18. **_mm512_mask_blend_ps(mask A,B,C)** Copies single precision floating point element from vector A in vector C if the corresponding mask bit is set.
19. **_mm512_fmadd_ps(mask A,B,C)** Fused-Multiply-Add: Multiplies elements from vector A and B and adds them to elements of vector C.
20. **_mm512_maskz_loadu_epi16(mask, addr)** Loads 64byte of 16bit elements under zero masking from address addr.
21. **_mm512_set1_epi32(value)** sets a 32 bit value into all 16 entries of the vector, e.g. broadcast.
22. **_mm512_maskz_mov_epi16(mask, A)** Moves 16 bit-type register A under zero-masking to a different register.
23. **_mm512_slli_epi32(A, imm)** Shifts all entries in the vector registers (typed as 32 bit elements) by value imm to the left by shifting 0 in.

Arm Pseudo Intrinsics

1. **vld1q_f32(addr)** Loads 16byte of 32 bit elements.
2. **vst1q_f32(addr)** Loads 16byte of 32 bit elements.
3. **vtrn1q_f32(A, B)** Unpacks and interleaves 32 bit elements from the low half of A and B.
4. **vtrn2q_f32(A, B)** Unpacks and interleaves 32 bit elements from the high half of A and B.
5. **vtrn1q_f64(A. B)** Unpacks and interleaves 64 bit elements from the low half of A and B.
6. **vtrn2q_f64(A, B)** Unpacks and interleaves 64 bit elements from the high half of A and B.
7. **vmax_q(A,B)** Calculates the maximum between each single precision floating point elements in A and B vector.
8. **vmin_q(A,B)** Calculates the minimum between each single precision floating point elements in A and B vector.
9. **vmul_q(A,B)** Multiply single precision elements in A and B vector.
10. **vsub_q(A,B)** Subtract corresponding single precision elements in B from A.
11. **vadd_q(A,B)** Add single precision elements in B and A.
12. **vshlq_u32(A,B)** Shift left each single precision elements in A by the value specified in B.
13. **vrndmq_f32(A)** Round single precision floating point elements in A using minus infinity rounding mode.
14. **vcvtmq_s32_f32(A)** Converts single precision floating point elements in A to signed integers using minus infinity rounding mode.
15. **float32x4_t** Represents 4 single precision floating point elements in vector width of 128.
16. **vand_q(A,B)** Performs bit-wise AND operation between A and B vector.
17. **vfmaq_f32(A,B,C)** Multiply single precision elements in A and B.Add the intermediate result to C.
18. **vld1q_f32(A)** Load a single precision element from scalar to all single precision element in a vector.
19. **vtbl1_u8(A,B)** Performs a byte look up operation in vector A using byte addressable indexes specified in vector B.
20. **vtbl4_u8(A,B)** Performs a 64 byte look up operation in vector A, A+1, A+2, A+3 using byte addressable indexes specified in vector B.
21. **vbcaxq_s32(A,B)** Performs XOR operation between each single precision floating point elements in A and B vector.
22. **vcgt_q(A,B)** Compare corresponding single precision elements in A and B. If B is greater then A the corresponding bits are set in the destination vector.
23. **vrecpe_f32(A)** Calculates approximate reciprocal of each single precision floating point element in vector A.
24. **vbit_insert(A,B)** Copies single precision floating point element from vector A in destination vector if the corresponding bits are set in vector B.