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

Evangelos Georganas*, Dhiraj Kalamkar*, Sasikanth Avancha*, Menachem Adelman*, Cristina Anderson*, Alexander Breuer*, Jeremy Bruestle*, Narendra Chaudhary*, Abhisek Kundu*, Denise Kutnick*, Frank Laub*, Vasimuddin Md*, Sanchit Misra*, Ramanarayan Mohanty*, Hans Pabst*, Barukh Ziv*, Alexander Heinecke*

*Intel Corporation
#Friedrich-Schiller-Universität Jena

ABSTRACT

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/hardware ecosystems, the programming methodology of DL systems is stagnant. DL workloads leverage either highly-optimized, yet platform-specific TP implementations are highly-optimized and platform-specific. 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 MLIR [20] towards unifying the optimization efforts in the Tensor Compiler 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/compiler 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 by the TPP layer.

Such a synergistic Tensor Compiler - TPP paradigm is illustrated in Section 5.4. 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 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 efficiency 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, enabling efficient fusion of TPPs that lead to efficient, portable high-level tensor operations.
- 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.

Section 2 details the specification of the TPPs. Then, Section 3 illustrates a POC implementation of the TPP specification along with an infrastructure that enables efficient TPP fusion. In Section 4 we exhibit how contemporary DL motifs/algorithmic paradigms can be expressed via TPPs. Section 5 presents an experimental evaluation of TPP-based DL kernels and workloads on multiple platforms. Sections 6 and 7 summarize the related work and conclude this paper.

Figure 1: Use-cases of TPPs in various software stacks.
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 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. SIMD, 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 \( l d \) and its datatype \( d t y p e \). 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.

| Unary TPP | Description/Comments |
|-----------|----------------------|
| Identify  |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 reciprocal 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|
| Transform |Transforms input and stores to output. Transformations are: Transpose, VNNI, VNNI-Tanspose|
| Unpack    |Takes each entry \( x_{i,j} \) of the input tensor, splits it in two parts \( x^{hi}_{i,j} \) and \( x^{lo}_{i,j} \) with same bit-width, and stores them in two tensors \( x^{hi} \), \( x^{lo} \)|
| Replicate columns |Takes an input column/row, 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 |Applies 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 |Applies a TanhScaled 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 1: Unary TPPs

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 GEMM (BRGEMM) TPP which is the main building block for general tensor contractions in DL kernels [21]. BRGEMM materializes the operation \( C = B \cdot C + \sum_{i=0}^{n-1} A_i \times B_i \). In essence, this kernel multiplies the specified blocks \( A_{M \times K}^{i} \) and \( B_{K \times N}^{i} \) 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 4 we exhibit how BRGEMM can be 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 $address_{B_i} = address_{B_{i-1}} + stride_B$.

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 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). 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 $\alpha$, which in turn are set as the inner-most tensor dimension. The VNNI 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 $[D_1[\alpha][D_0[\alpha] \rightarrow [D_0[\alpha][D_1[\alpha]$.

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. 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. Table 2 and Table 3 present the binary/ternary TPPs that accept two/three 2D tensors 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 JITted TPP kernel.

### 3.1 Generic TPP Structure

Algorithm 1 exhibits at a high-level the pseudocode that is used to implement the Unary/Binary/Ternary TPPs in a unified fashion.
We present in more detail the BRGEMM TPP because it comprises Algorithm 2 (NEON)/512-bit (SVE) vector registers. The "load_generic" function TPPs) and $X$. The inputs of the TPPs are tensors $A$, $B$, and $C$.

### 3.2 BRGEMM TPP Structure

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.

### 3.3 Combining TPPs via 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 $tmp = \text{Add}(A, B)$ 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 effectiveness pertaining to composite operators, we implemented an embedded Domain Specific Language (eDSL) in LIBXSM [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. Our code generator parses the provided equation and emits/JITs a function implementing the desired operation. Figure 2 illustrates two Matrix Equation trees that are used to express the softmax operator [28] (see Section 4.1 for calculating):

\[
Y = \text{softmax}(X) \text{ with } y_{ij} = \frac{e^{x_{ij} - \max_{x_{ij}}(cX)}}{\sum_{x_{ij}} e^{x_{ij} - \max_{x_{ij}}(cX)}}
\]

The eDSL back-end performs multiple analyses and optimizations passes. The input equation tree is analyzed, and TPP fusion opportunities are identified. Then, a secondary analysis is performed to minimize the intermediate tensor storage requirements and to increase the register reuse when passing/communicating results across the equation’s TPPs. This analysis performs an adaptive post-order traversal of the equation tree, where the computation (if possible) is performed to smaller sub-tensors that can fit in the available register file. The reuse/recycling among register blocks in this adaptive post-order evaluation is provably optimal and necessitated to maximize the register file reuse [29].

We want to highlight two aspects of the matrix equation JIT strategy that are essential for high performance and usability. First, when combining specific operations which exhibit both high register pressure and reuse (e.g., transposes, GEMM/BRGEMM, transcendental approximations), we decide to propagate the results towards the rest of the TPPs in the tree via stack-allocated temporal tensors. This design decision is done in order to avoid excessive register spills/reloads for such operations, since fusing vertically in the tree across multiple TPPs merely increases the register pressure, and it would diminish the benefits or register reuse within these operations. This stack-allocated tensor management is done automatically by the back-end and is transparent to the user. Second, in the back-end we identify idioms/motifs of combined TPPs (e.g., a gather TPP followed by a reduce TPP) and we can JIT an instruction sequence which is optimal for the composite access pattern.

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 4. 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 \(C_j\). Essentially this formulation dictates an A-stationary flow, and the back-end could optimize accordingly for the specific accelerator.

4 TPP-BASED KERNELS & 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 paper’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.

4.1 Softmax Kernel

Equation 1 shows the formula for the softmax operator [28], 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 2. The left tree computes the nominator of Equation 1: 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).

4.2 Layernorm Kernel

Layer normalization (layernorm) [30] is a technique that normalizes the neurons within a layer, and was motivated by the limitations of Batch Normalization [31] 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: \(\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_{ij}, \sigma_i^2 = \frac{1}{m} \sum_{j=1}^{m} (x_{ij} - \mu_i)^2\) where \(i\) is the batch dimension and \(j\) is the “feature” dimension, ii) then the tensor entries \(x_{ij}\) are normalized based on \(\mu\) and \(\sigma\): \(x_{ij}' = (x_{ij} - \mu_i) / \sigma_i\). 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 paper’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.

4 TPP-BASED KERNELS & 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 paper’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.

4.1 Softmax Kernel

Equation 1 shows the formula for the softmax operator [28], 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 2. The left tree computes the nominator of Equation 1: 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).

4.2 Layernorm Kernel

Layer normalization (layernorm) [30] is a technique that normalizes the neurons within a layer, and was motivated by the limitations of Batch Normalization [31] 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: \(\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_{ij}, \sigma_i^2 = \frac{1}{m} \sum_{j=1}^{m} (x_{ij} - \mu_i)^2\) where \(i\) is the batch dimension and \(j\) is the “feature” dimension, ii) then the tensor entries \(x_{ij}\) are normalized based on \(\mu\) and \(\sigma\): \(x_{ij}' = (x_{ij} - \mu_i) / \sigma_i\).
\[
\mu_i / (\sqrt{\sigma_i^2 + \epsilon})
\]
Depending on the workload (e.g., attention cell in BERT), the scaled tensor may be further scaled with two other tensors \(G\) and \(B\). Figure 3 illustrates two TPP equation trees that implement this composite layer norm 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 layer norm via means of TTPs since all DL norming layers essentially exhibit similar computational motifs, and this specific norm is used in the BERT workload described in subsection 4.7.

4.3 BF16 Split-SGD Kernel

Unlike the two previous kernels which are well-established in DL workloads, and as such 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 [32]. 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 16bit-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 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 4 illustrates the BF16 Split-SGD operator written entirely via TTPs. 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.

4.4 Convolutional Neural Networks (CNN)

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).

4.5 1D Dilated Convolutions & Comp. Bio

In this Section, we show the implementation of a special type of convolution via TTPs in their entirety, namely one-dimensional (1D) dilated convolution layer of a 1D CNN named ATACworks [33]. ATACworks is used for de-noising and peak calling from ATAC-seq genomic sequencing data [33]. 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 \(O\), \(C\) 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 3 shows an example of the forward pass procedure with an input tensor \(I\), a weight tensor \(W\), and an output tensor \(O\).

4.6 Deep Learning Recommendation Model

Facebook recently proposed a deep learning recommendation model (DLRM) [34]. Its purpose is to assist the systematic hardware-software co-design for deep learning systems. DLRM comprises of the following major components: (a) a sparse embedding involving tables (databases) of varying sizes, (b) a small dense Multi-Layer Perceptron (MLP), 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. As such, we provide a high-level overview of how Sparse Embeddings and MLPs can be expressed via TTPs. For simplicity we focus on the forward propagation pass during training, however we have fully implemented the back-propagation in an analogous way by using solely TTPs. For more details on the workload and CPU-oriented optimizations we refer to prior work [32].
We follow the approach of previous work [21] and we block the
Algorithm 4 shows the fully connected layer implementation which
TPPs, and performs horizontal register fusion across them. More
vector is denoted as \( M \) is loaded in vector registers, and is added to a set of running
4.6.1 Sparse Embedding Kernel
The sparse embedding kernel is comprised of multi-hot encoded
lookups into an embedding table \( W \times M \times K \) with \( M \) being the number of rows and \( E \) the length of each row, whereas the multi-hot weight-vector is denoted as \( \alpha^T = [0, \ldots, a_p, \ldots, a_p, \ldots, 0] \) with entries \( a_p = 1 \) for \( p = p_1, \ldots, p_k \) and 0 elsewhere (\( p \) being the index for the corresponding lookup items). Mathematically, the embedding lookup output vector \( \alpha^T \) can be obtained via \( \alpha^T = \alpha^T \times W \). 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 \( \alpha^T \). Figure 5 illustrates the TPP tree that is used to express the Sparse Embedding lookup kernel. We note that the TPP back-end optimizes this sequence of TPPs, and performs horizontal register fusion across them. More specifically, given a non-zero index \( a_p \), the corresponding row of \( M \) is loaded in vector registers, and is added to a set of running accumulators/vector registers that hold the output \( \alpha^T \).

4.6.2 Multi-Layer Perceptron (MLP)
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 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 4 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 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 \( K \) and \( b_k \) (which constitute the \( K \)-inner-product dimension of the original 2D matrices). We employ the stride-based BRGEMM

Figure 5: Sparse Embedding Lookups via TPPs

Figure 6: Binary-Reduce aggregation kernel via TPPs

because the sub-blocks “A_i” and “B_i” that have to be multiplied and reduced are apart by constant strides \( \text{str} \_A = b_k \cdot b_m \) and \( \text{str} \_B = b_n \cdot b_k \) respectively. Finally, we apply (optionally) a unary TPP corresponding to the requested activation function (e.g. RELU) onto the just-computed output block of \( C \).

4.7 Natural Language Processing - BERT
The BERT model is a bidirectional transformer pre-trained via a combination of masked language modeling objective, and next-sentence prediction [35]. 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 [36] 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 4). We opt to use blocked tensor layouts for the same reasons briefly described in Section 4.6.2. 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, 37]. The softmax operator is also formulated entirely by TPPs as described in Section 4.1. 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.

4.8 Emerging AI - Graph Neural Networks
Graph Neural Networks (GNN) [38] form an emerging class of Neural Networks for learning the structure of large, population-scale graphs. Depending on the specific algorithm and task that
We use a variety of platforms that span different ISAs, different AVX2 ISA, vi) an AWS Graviton2 instance with 64 cores at fixed 2.5 GHz with ARMv8 SVE 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.

5 EXPERIMENTAL RESULTS
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.

5.1 Performance of standalone DL kernels
We start the performance evaluation with the standalone TPP kernels presented in Sections 4.1, 4.2, and 4.3. 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 8-Top shows the performance of the Softmax operator of blocked 3D tensors with size $S_1 \times S_2 \times S_3$, on the CLX platform (i.e. targeting AVX512 ISA). Here we perform S2 softmax operations over blocked $S_1 \times S_3$ 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× over the baseline. The hand-vectorized code is faster by 2.6× whereas the TPP-based variant shows similar speedups by being 2.2× 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 9-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× over the baseline on ROME.

Figure 8-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× 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× and 1.5×. 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 9-Top shows similar performance speedups on ROME, where the TPP-based code is 1.6× faster than the auto-vectorized baseline.

Figure 8-Bottom shows the performance of the BF16 split-SGD operator on CLX. This use-case represents a novel, mixed-precision operator where the compiler (even with compile-time constant tensor sizes) struggles to yield good performance; the TPP-based code has geo-mean speedup of 38× over the compiler generated code.

Figure 7 illustrates the TPP-based implementation of various ResNet50 [40] 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 geometric 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 [41] 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 without a single line of code change, but they run at very similar hardware utilization.

5.2 Performance of end-to-end DL workloads

5.2.1 1D Dilated Convolutions and their application to Comp. Bio

Here we evaluate the oneDNN [13] and TPP-based 1D dilated convolution layer of ATACworks [33] which takes more than 90% of the training time, and it has input tensor width (W) of 60400, output tensor width (Q) of 60000, 15 input channels (C), 15 filters (K), filter size (S) of 51, and dilation (d) of 8. Figure 10-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 10-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.

5.2.2 Deep Learning Recommendation - DLRM

Figure 11-Top shows the FP32 DLRM performance on CLX using two different configurations, namely small DLRM and MLPerf DLRM configurations. We refer to previous work for the detailed specification of these configurations [32]. We evaluated 4 different implementations of DLRM: i) the PyTorch reference implementation, ii) PyTorch reference + custom Embedding extension auto-vectorized by the compiler, iii) DLRM expressed entirely via
TPPs, and iv) hand-vectorized Embedding extension + BRGEMM based MLPs [32]. 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 49%. Compared to the version with the custom, auto-vectorized variant the TPP-version is up to 54% faster on CLX.

Figure 11-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. 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 5.1.

5.2.3 Natural Language Processing - BERT Large

Figure 12-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 matches the performance of the AVX512-hand-vectorized code (within 3%). At the same time, our implementation is 1.61x faster than the Reference HuggingFace CPU reference code [42]. Figure 12-Bottom shows the performance of the TPP-based code across multiple platforms and compute precisions. The relative differences in the performance can be justified by the different compute/bandwidth specs of the benchmarked platforms. It is noteworthy mentioning that with minimal changes inside the fused operators to handle VNNI layout required for BF16.

5.2.4 Emerging AI - Graph Neural Networks

Figure 13-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 FMA.s, we use bit-wise accurate emulated-BF16 BRGEMM TPPs, and we still expect savings due to the bandwidth reduction in the non-GEMM parts, e.g., graph traversal and edge/nodes aggregation. We observe that the TPP-based implementation outperforms the DGL [43] baseline by 3.8x and the Xbyak JIT-ed/optimized version (non-default optimization in DGL) by 2.35x. TPP-BF16 yields another 1.33x speedup over the TPP-FP32 mainly due to reduced bandwidth requirements.

Figure 13-Bottom shows the performance of the TPP-based code across multiple platforms and compute precisions. 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 1.8x speed up using BF16 training on CPX with 28 cores surpassing 40 core ICX performance by 30%.

5.3 Distributed-memory scaling of 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 [44]. Figure 14 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 51.7x and 57.9x respectively. Regarding the scaling of the GNN workload, the efficiency is directly affected by the quality of the
5.4 TPPs 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. We experimented with the use-case of FP32 inference on a client CPU (Intel i7-6700) on three different workloads: ResNet-152 [40], ResNext-50 [45], and I3D-Kinetics-400 [46].

Figure 15 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 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.

6 HPC IMPLICATIONS & RELATED WORK

While the presentation in this work focused on how TPPs can be used to express DL computations in a performance-portable way across various architectures and precisions, TPPs can be readily applied in HPC as well. Especially quantum chemistry (CP2K [47]) and higher finite element solvers (pyFR [48], EDGE [49], NeK [50], SeisSol [51], Spectre [52] and libCEED [53]) make already heavy use of the versatile (BR)GEMM TPP. An extension to the element-wise TPPs and hence even increasing performance portability in this expanded application field is straightforward.

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 4. 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.

Tensor Compilers (TC) [15–18] attempt to optimize DL operators in a platform-agnostic 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 5.4), and in this way the TPP back-end could be leveraged by multiple TC frameworks.

7 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 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.
REFERENCES

[1] Alex Krizhevsky, I. Sutskever, and G.E. Hinton. Image classification with deep convolutional neural networks. *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Christian Servex, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *In Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[4] Dong Yu, Michael L Seltzer, Jinyu Li, Jui-Ting Huang, and Frank Seide. Feature learning in deep neural networks: studies on speech recognition tasks. *arXiv preprint arXiv:1303.3665*, 2013.

[5] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Suleyman, et al. Wide & deep learning for recommender systems. *In Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10. ACM, 2016.

[7] Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, 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*, pages 38–45, 2020.

[8] Erik Gawehn, Jan A Has, and Guibert Schneider. Deep learning in drug discovery. *Molecular informatics*, 35(3):1–14, 2016.

[9] Garrett B Goh, Nathan O Hodas, and Abhinav Vishnu. Deep learning for computational chemistry. *Journal of computational chemistry*, 38(16):1291–1307, 2017.

[10] Matthijs Raghu and Eric Schmidt. A survey of deep learning for scientific discovery. *arXiv preprint arXiv:2003.11755*, 2020.

[11] Md Zahangir Alom, Tarok M Taha, Chris Yakopcic, Stefan Westberg, Paheding Garrett B Goh, Nathan O Hodas, and Abhinav Vishnu. Deep learning for computing. *arXiv preprint arXiv:1903.06498*, 2019.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning libraries. *arXiv preprint arXiv:1504.00353*, 2015.

[13] Tim Zerrell and Jeremy Breville. Tensor: Deep learning with custom node types. *In Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[14] Intel oneDNN. https://github.com/oneapi-src/onednn, Accessed on 3/30/2021.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1404.0054*, 2014.

[16] Tim Zerrell and Jeremy Breville. Stripe: Deep learning with custom node types. *In Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[17] Intel oneDNN. https://github.com/oneapi-src/onednn, Accessed on 3/30/2021.

[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1404.0054*, 2014.

[19] Tim Zerrell and Jeremy Breville. Strippe: Deep learning with custom node types. *In Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[20] Intel oneDNN. https://github.com/oneapi-src/onednn, Accessed on 3/30/2021.

[21] Thomas Wolf, Lysandre Debut, Victor Sanh, Clemens Dyer, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, 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*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[22] Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. Optimizing deep learning on large gpus. *In Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2018.

[23] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[24] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Technical Report, arXiv preprint arXiv:1901.00596*, 2019.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Aäron Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. Generative pre-training of a large language model. *In Proceedings of the 13th Workshop on Text Representation Learning (WERC 2020)*, pages 38–45, 2020.

[27] Stefan K{"{u}}hl and Marcella Iannuzzi. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[28] J Willard Gibbs. *Elementary principles in statistical mechanics*. Courier Corporation, 2014.

[29] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[30] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[31] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. *In Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[32] Intel oneDNN. https://github.com/oneapi-src/onednn, Accessed on 3/30/2021.

[33] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1404.0054*, 2014.

[34] Tim Zerrell and Jeremy Breville. Strippe: Deep learning with custom node types. *In Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[35] Intel oneDNN. https://github.com/oneapi-src/onednn, Accessed on 3/30/2021.

[36] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.
