ROLLER: Fast and Efficient Tensor Compilation for Deep Learning

Hongyu Zhu, *University of Toronto and Microsoft Research;* Ruofan Wu, *Renmin University of China and Microsoft Research;* Yijia Diao, *Shanghai Jiao Tong University and Microsoft Research;* Shanbin Ke, *UCSD and Microsoft Research;* Haoyu Li, *Columbia University and Microsoft Research;* Chen Zhang, *Tsinghua University and Microsoft Research;* Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, and Lidong Zhou, *Microsoft Research;* Asaf Cidon, *Columbia University;* Gennady Pekhimenko, *University of Toronto*

https://www.usenix.org/conference/osdi22/presentation/zhu

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ROLLER: Fast and Efficient Tensor Compilation for Deep Learning

Hongyu Zhu[†◇*] Ruofan Wu[‡◇*] Yijia Diao[$◇*] Shanbin Ke[¶◇*] Haoyu Li[§◇*] Chen Zhang[£◇*]
Jilong Xue[◇] Lingxiao Ma[◇] Yuqing Xia[◇] Wei Cui[◇] Fan Yang[◇] Mao Yang[◇]
Lidong Zhou[◇] Asaf Cidon [§] Gennady Pekhimenko [†]
[†]*University of Toronto* [‡]*Renmin University of China* [$]*Shanghai Jiao Tong University*
[¶]*UCSD* [§]*Columbia University* [£]*Tsinghua University* [◇]*Microsoft Research*

Abstract

Despite recent advances in tensor compilers, it often takes hours to generate an efficient kernel for an operator, a compute-intensive sub-task in a deep neural network (DNN), on various accelerators (e.g., GPUs). This significantly slows down DNN development cycles and incurs heavy burdens on the development of general kernel libraries and custom kernels, especially for new hardware vendors. The slow compilation process is due to the large search space formulated by existing DNN compilers, which have to use machine learning algorithms to find good solutions.

In this paper, we present ROLLER, which takes a different construction-based approach to generate kernels. At the core of ROLLER is *r*Tile, a new tile abstraction that encapsulates tensor shapes that *align* with the key features of the underlying accelerator, thus achieving efficient execution by limiting the shape choices. ROLLER then adopts a recursive *r*Tile-based construction algorithm to generate *r*Tile-based programs (*r*Program), whose performance can be evaluated efficiently with a micro-performance model without being evaluated in a real device. As a result, ROLLER can generate efficient kernels *in seconds*, with comparable performance to the state-of-the-art solutions on popular accelerators like GPUs, while offering better kernels on newer accelerators like IPUs.

1 Introduction

Deep neural networks (DNN) have been used extensively in intelligent tasks like computer vision and natural language understanding. As DNN computation is known for its complexity, the compute intensive sub-tasks (e.g., matrix multiplication) in a DNN model are abstracted as operators and implemented as kernels, executed on modern accelerators (e.g., GPUs, TPUs) to speed up the computation. DNN compilers play an important role in producing high-performance kernels for the development of DNN models. It reduces the burden of (often hand-crafted) library-based kernel development (e.g., cuDNN [6] and cuBLAS [2]) and provides a flexible way to cover the fast-growing number of custom operators, which libraries struggle to catch up with and optimize, a growing pain especially for new hardware vendors.

DNN compilers treat a DNN operator as tensor computation, which is then translated into nested multi-level loops iterated over the computation on each tensor element along different axes (dimensions). Compiler optimization techniques like loop partitioning/fusion/reordering are applied to nested loops. Due to the inherent complexity of loop rearrangement, it is a combinatorial optimization problem to find a good solution among a large search space, often with millions of choices. Therefore, advanced compilers [15, 33, 35] propose to adopt machine learning algorithms to search for a good solution. This usually takes thousands of search steps, each evaluated in a real accelerator, to find a reasonable solution. Our own experience shows that tuning an end-to-end DNN model using state-of-the-art compilers [15, 33] often requires days, if not weeks. The tuning time may be even longer if the DNN model runs on less mature accelerators (e.g., AMD GPU or Graphcore IPU [4]) (§2). To make the matter worse, a DNN model need to re-compile whenever its structure, operator types, tensor shapes and configurations are changed. This is often required when trying different configurations in model training or inference. Given that an operator could have arbitrary input shapes and configurations, such compilation could significantly slow down the overall DNN model development cycle.

In this paper, we propose ROLLER, a deep learning tensor compiler that addresses the problem in a radically different way. ROLLER is built on the following insights. First, instead of multi-level nested loops, ROLLER treats the computation in a DNN operator as a *data processing pipeline*, where data tiles (a fraction of a tensor) are moved and processed in an abstracted hardware with parallel execution units and multi-layer memory hierarchy. The goal of generating efficient kernel programs then becomes that of improving the throughput of the pipeline.

*Work is done during the internship at Microsoft Research.

Second, for an accelerator to execute efficiently, the shape of a data tile should *align* with the hardware characteristics, including memory bank, memory transaction length, and minimum schedulable unit (e.g., warp size in GPUs). To achieve the full alignment across multiple hardware features, the available tile shapes are limited. More importantly, with alignment as a constraint, to maximize the throughput of a pipeline, one only needs to *construct* an aligned tile shape that saturates the execution unit of the accelerator. This construction process is significantly more efficient than solving the original unconstrained combinatorial optimization problem.

Third, the performance of an aligned pipeline is highly predictable. Key performance metrics under the aligned pipeline (e.g., memory throughput) can be derived from the hardware specification (or through micro-benchmarking). This greatly simplifies the performance evaluation under various aligned configurations, eliminating the need of a complex cost model and/or expensive hardware-based evaluation on each aligned configuration.

With these insights, ROLLER proposes *r*Tile, a new abstraction that encapsulates data tile shapes that *align* with the key features of the hardware accelerator and the input tensor shapes (§3.1). A data processing pipeline can then be described as an *r*Tile-based program (a.k.a. *r*Program) composed by three interfaces: `Load`, `Store`, and `Compute`, acted against *r*Tile. To construct an efficient *r*Program, ROLLER follows a *scale-up-then-scale-out* approach. It first performs the scale-up process, which adopts a recursive *r*Tile-based construction algorithm (Figure 8) to gradually increase the size of the *r*Tile shape to construct an *r*Program that saturates a single execution unit of the accelerator (e.g., an SM, a streaming multi-processor in a NVIDIA GPU). It then performs the scale-out process, which simply replicates the resulting *r*Program to other parallel execution units, thanks to the homogeneity of both the computation pattern of deep learning and the parallel execution units in an accelerator.

ROLLER can evaluate the performance of different *r*Tiles without significant overheads. The peak (saturate) compute throughput can simply be measured *once per operator type*. And due to the alignment, other key performance factors like memory pressure of an *r*Tile can be derived analytically from hardware specifications. This leads to an efficient micro-performance model, avoiding the expensive online profiling on each configuration required by existing DNN compilers, thereby significantly speeding up the compilation process. In addition, due to the strict alignment requirements, the recursive construction process can produce a few desired *r*Tiles (and *r*Program) quickly. Combined, ROLLER can generate efficient kernels *in seconds*.

We have implemented ROLLER on top of TVM [15] and Rammer [26], and open-sourced the code[1]. Our evaluation on 6 types and 119 popular DNN operators from several mainstream DNN models shows that ROLLER can generate highly-optimized kernels in *seconds*, especially for large expensive custom operators. This achieves *three orders of magnitude improvement* on compilation time. The performance of ROLLER-generated kernels is comparable to and often better than the state-of-the-art tensor compilers and even vendor-provided DNN libraries. With the three *r*Tile-based interfaces (`Load`, `Compute`, `Store`) describing an *r*Program, ROLLER can easily adapt to different accelerators like AMD GPU and Graphcore IPU. ROLLER has been used to develop custom DNN kernels internally and shown to significantly speed up our development cycle. It offers potentially disruptive opportunities to new players in the compute accelerator market, who previously have to spend significant engineering efforts on efficient kernels.

[1] https://github.com/microsoft/nnfusion/tree/osdi22_artifact/artifacts

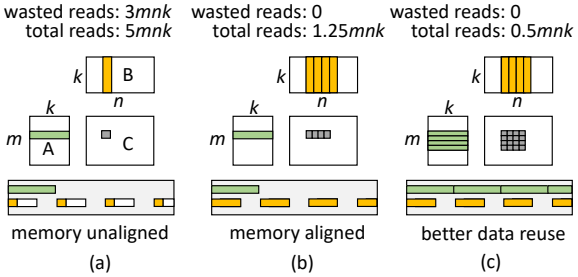

Figure 1: Access pattern of different tile shape. Matrix multiplication, $C_{m,n} = A_{m,k} \times B_{k,n}$.

2 Motivation and Key Observations

Excessive compilation time. Our own experience in a set of DNN operators (detailed setting in §5) shows that the average compile time for a single operator using Ansor [33], a state-of-the-art tensor compiler, is 0.65 hours. Among them, one convolution operator in ResNet model takes 2.17 hours. A DNN model may contain hundreds of operators, thus it easily takes days to compile the model. For example, to compile a NASNet model (§5), we reach only 32% of the overall searching progress after tuning for 41.8 hours. Our experience also shows the compilation speed is even worse on less mature devices, the compiler takes much longer time for a kernel.

Observation and insights. We observe that there exists a different view to the computation of an DNN operator. Taking matrix multiplication (MatMul), $C_{m,n} = A_{m,k} \times B_{k,n}$, as an example to illustrate our observation. Unlike existing compilers that treat MatMul as a 3-level loop iterated over each axis m,k,n, the computation process is also a data processing pipeline. One can `Load` each sub-matrix (i.e., a tile) from A and B, `Compute` the two tiles, and `Store` the resulting tile of C to memory. Thus, the performance of the computation depends on how fast one can move the data tiles in the `Load-Compute-Store` pipeline.

The key factor affecting the performance in all steps in the pipeline is the *shape* of tiles and the corresponding layout

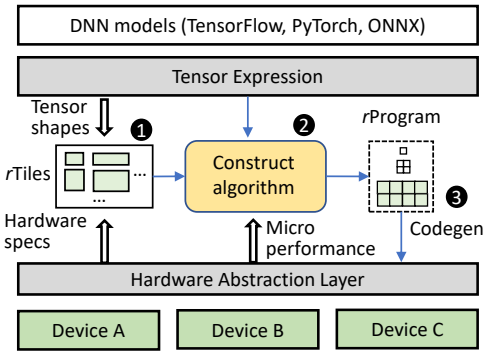

Figure 2: System overview of ROLLER.

in the one-dimension memory space. Figure 1(a) illustrates the computation of one element in C (in the top part) and the memory accessing pattern (in the bottom part). Assuming all matrices stored in a row-major layout, loading a column from B causes strided accesses in length of 1. Suppose the memory transaction length is 4, there will be $3/4$ of total redundant data reads. Thus, the data tile shape should align with the memory transaction length for efficient memory access. In Figure 1(b), when computing B in the granularity of 1×4 tile, there will be no memory bandwidth waste. Besides memory alignment, the tile shape should also align with the hardware execution unit, e.g., the parallel threads number, to avoid waste in computing cycles. Moreover, the tile shape also affects data reuse opportunities due to caching, a common feature in modern accelerators. For example, Figure 1(a) needs $2mnk$ data reads when computing a 1×1 tile each time. However, in Figure 1(b), only $1.25mnk$ reads are required, as one read from A can be reused 4 times. If setting the tile size along M dimension to 4×4, as shown in Figure 1(c), the total reads can be reduced to $0.5mnk$. A $10\times$ improvement over Figure 1(a).

These observations motivate ROLLER, a system that identifies the aligned tile shapes and constructs an efficient tile processing pipeline to improve the end-to-end throughput.

3 System Design

Figure 2 shows the system overview. ROLLER takes an operator described as a tensor expression (§3.1). The expression is generated by users or from a graph-level DNN compiler [15, 26, 33], which might further fuse multiple operators into a single expression. ROLLER extracts the tensor shapes from the tensor expression and leverage hardware specifications to construct *r*Tiles, i.e., a hardware-aligned building block (§3.1). Based on *r*Tiles, ROLLER proposes a *scale-up-then-scale-out* recursive construction algorithm to generate efficient tensor programs (named *r*Program) that describes the data processing pipeline (§3.2). When generating *r*Program, the construction algorithm identifies good *r*Tile configurations by evaluating the performance of a constructed *r*Program through a micro-performance model. It is built on top a device described through a hardware abstraction layer exposing only *r*Tile-related interfaces: `Load`, `Compute`, and `Store` (§3.3). The constructed *r*Program is finally realized through a code generator to emit the final kernel code corresponding to the specific device.

```
class rTile {
  TensorExpr expr;
  TileShape shape;
  TileShape storage_padding;
  vector<TileShape> GetInputDataTiles();
  vector<TileShape> GetOutputDataTiles();
};
```

Figure 3: The data structure of *r*Tile.

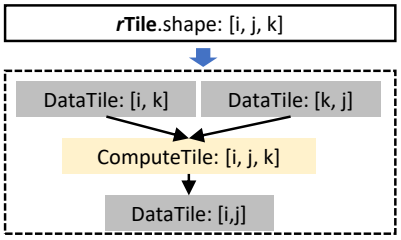

Figure 4: The data tiles and computing tile inferred by an *r*Tile for MatMul expression.

3.1 Tensor Expression and *r*Tile

ROLLER takes input of a tensor computation as an index-based lambda expression, i.e., tensor expression [15, 27]. It describes how each element in the output tensor is computed based on the corresponding elements in the input tensors. For example, a MatMul operator with output tensor C of the shape $M \times N$ can be expressed as,

```
C = compute((M,N), lambda i,j:sum(A[i,k]*B[k,j])),
```

where the element indexed by (i, j) in C is computed by a sum reduction over the elements in row i of A and column j of B, and k is the reduction axis. Such an expression can cover the majority of operators in DNN models and is widely used in existing DNN compilers including TVM [15], Ansor [33], and FlexTensor [35].

ROLLER introduces *RollingTile* (*r*Tile for short) as the basic computing unit to compose a tensor computation. As shown in Figure 3, an *r*Tile encapsulates a multi-dimensional tile `shape` defined along each loop axis of a given tensor expression `expr`. Given `shape` and `expr`, an *r*Tile can statically infer the involved input and output data tiles. For example, a tile shape $[4, 4, 2]$ along axes i, j, k denotes an *r*Tile for the above MatMul expression, where each *r*Tile loads a 4×2 data tile from A and a 2×4 tile from B, conducts total $4 \times 4 \times 2$ multiply-add computations, and stores a 4×4 data tile to C, as illustrated in Figure 4.

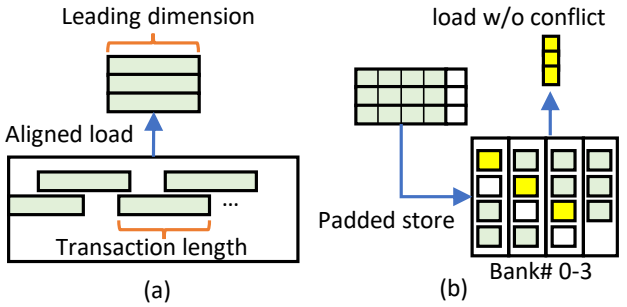

Figure 5: Illustration of (a) transaction aligned memory load and (b) bank conflict-free padding.

A unique property of an *r*Tile is that it must align with both the underlying hardware features and the tensor shapes in a given tensor expression. All these alignments are controlled by the *r*Tile *shape* and the *storage_padding* fields in Figure 3, which represent the logical form and the physical layout of an *r*Tile, respectively. We elaborate the detailed requirements of alignment next.

Alignment with the hardware execution unit. First, the shape of an *r*Tile must align with the parallelism of the execution unit it runs on. For example, if running on a warp of threads in a GPU, the size of `shape` in the *r*Tile should be a multiple of the warp size, e.g., 32, for maximal computing efficiency. When using TensorCore in NVIDIA GPUs, the *r*Tile shape should be a multiple of $16 \times 16 \times 16$. Similarly, an *r*Tile executed on a streaming multi-processor (SM) should align its size as a factor of execution unit number on the SM.

Alignment with memory transaction. Second, a data tile's shape should align with the length of memory transaction for optimal memory access. Specifically, for each data tile of an *r*Tile, we should guarantee that its leading dimension (e.g., the inner-most dimension in a row-major tensor) is a multiple of the memory transaction length, as illustrated in Figure 5(a). In ROLLER, tensors are allocated in a cache-aligned way. Thus, an *r*Tile can avoid any wasted transaction read, as its shape is aligned with the memory transaction.

Alignment with memory bank. Third, the memory layout of a data tile should align its stride with the memory bank to avoid read conflicts. For example, a $[3,4]$ data tile is kept in the memory across 4 banks and is read by an upper-memory-layer tile with a shape of $[3,1]$, as shown in Figure 5(b). A naive approach that stores all the $[3,1]$ values in the same bank will result in conflicted loading. *r*Tile avoids such inefficiency by padding a data tile. Given a data tile with a leading dimension of size N, which is read by another tile with a leading dimension of size n, we add a padding size of $(BL - N\%(BL) + L\lceil n/L \rceil)\%(BL)$ along N when storing this tile, where B and L are the bank number and the bank width, respectively. The padding sizes along each axis are calculated and stored in the `storage_padding` field in Figure 3. For the case in Figure 5(b), by a padding size of 1, all the $[3,1]$ values are distributed in different banks and can be read efficiently.

Alignment with tensor shape. Finally, an *r*Tile's shape should align with the tensor shape of an input tensor expression. Otherwise, the computation cannot be evenly partitioned by the *r*Tile, wasting compute resources or incurring heavy boundary checking overheads. A simple solution is to add a padding size P_i along a tensor dimension i with size of N_i, which makes $N_i + P_i$ a multiple of the *r*Tile shape's dimension size at axis i. However, a large padding might waste computation. ROLLER therefore restricts tensor padding under a range ε, where an *r*Tile's shape dimension size S_i has to satisfy that $\frac{S_i - N_i \% S_i}{N_i} \leq \varepsilon$, where N_i is the tensor size at dimension i. This ensures the wasted percentage of computation is bounded by ε. With this restriction, we can enumerate all the valid *r*Tile shapes that satisfy this condition.

Deriving all *r*Tiles. Given the above alignment requirements, for a specific tensor expression and hardware device, ROLLER incrementally derives all the conforming *r*Tiles using the following interface:

```
vector<int> GetNextAlignedAxisSize(rTile T, Dev d),
```

which returns the next aligned size for each axis in the `shape` of *r*Tile T given the specific device specification d. This is calculated by gradually increasing the dimension size along each axis until it satisfies all the alignment requirements. The *r*Tile abstraction allows ROLLER to be extended to support new alignment requirements (e.g., new hardware features). This is achieved by adding new alignment rules to the `GetNextAlignedAxisSize` interface.

Calculating data reuse score. An interesting property of *r*Tile is that we can implicitly control the memory traffic by adjusting its `shape`. Increasing the *r*Tile size usually brings more *data reuse* opportunities at the cost of occupying more memory space. Given an *r*Tile T and its next aligned size in each axis, we can calculate the data reuse score S_i for axis i by $S_i = \frac{Q(T) - Q(T_i')}{F(T_i') - F(T)}$, where T_i' is a newly enlarged *r*Tile by replacing the dimension size at axis i with the next aligned size from `GetNextAlignedAxisSize`. Functions $Q(T)$ and $F(T)$ calculate the memory traffic and memory footprint when the computation is executed in the granularity of T, which can be directly inferred based on the given tensor expression and hardware memory specification (§3.3). A larger S_i means better cost-efficiency, i.e., more memory traffic can be saved with the same memory usage. The memory reuse score plays a critical role in constructing an efficient *r*Program (using *r*Tiles), as shown in the next subsection.

3.2 Tensor Program Construction

***r*Tile program.** Given *r*Tile and the hierarchical memory structure of modern accelerators, a tensor computation can be naturally treated as a streaming data processing pipeline. The computation loads data tiles (specified in *r*Tile) from the lowest memory layer through the memory hierarchy to the highest layer, performs *r*Tile computation on the execution

```
for L1_iter in L2_rtile.split(L1_rtile):
  L1_input_tiles = Load(L1_iter); //L2 to L1
  for L0_iter in L1_rtile.split(L0_rtile):
    L0_input_tiles = Load(L0_iter) //L1 to L0
    L0_out_tile = Compute(L0_input_tiles);
    Store(L0_out_tile, L2_out_tile);//L0 to L2
```

Figure 6: The pseudo code of an *r*Program on a device with a 3-layer memory hierarchy (Bottom-up: layer `L2` to layer `L0`).

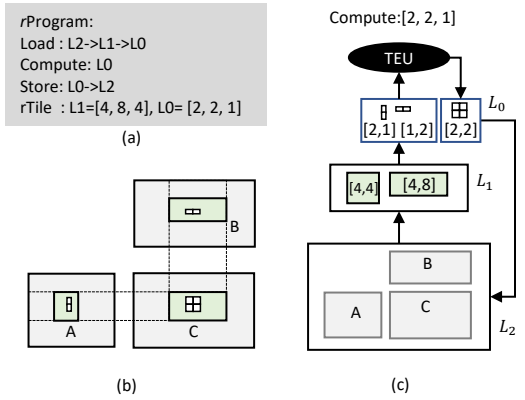

Figure 7: ROLLER computation model. (a) An *r*Tile program; (b) *r*Tiles on matrix multiplication; (c) Execution of the *r*Tile program on a hardware memory hierarchy.

units of the accelerator, and stores the resulting data tiles back to the lowest memory. For each memory layer, a specific *r*Tile is defined to align with the characteristics of this memory layer. Thus, ROLLER describes tensor computation as a data processing pipeline with a hierarchical *r*Tile configuration, which is called an *r*Tile program (i.e., *r*Program).

Figure 6 shows an *r*Program on a device with three memory layers (L0, L1 and L2). The *r*Program is described by the *r*Tile at each layer and the *r*Tile instructions (i.e., `Load`, `Store`, and `Compute`) at each memory layer. Figure 7(a) shows a MatMul *r*Program illustrated in Figure 7(b). Figure 7(c) illustrates how the *r*Program is mapped to each memory layer of a device. Specifically, each time it loads a $[4,4]$ data tile in A and a $[4,8]$ tile in B from memory L2 to L1; and then it loads the data tiles from memory L1 to memory L0 (i.e., registers) in shapes of $[2,1]$ and $[1,2]$. After each `Compute`, the resulting $[2,2]$ tile will be directly stored from L0 to L2.

Given a data processing pipeline, the optimization goal of the corresponding *r*Program is to maximize the throughput of the pipeline. The goal can be translated into three conditions: 1) the computation and memory movement should fully leverage the hardware features; 2) the throughput should saturate the bottleneck stage; and 3) there needs to be sufficient parallelism to leverage all the parallel execution units. Thus, ROLLER proposes the following *r*Program construction policy: first scale-up on one core by constructing a single-core *r*Program to saturate the core's hardware utilization and then

```
1  Func ConstructProg(expr:TensorExpr, dev:Device):
2      T = rTile(expr);
3      Results = [];
4      EnlargeTile(T, dev.MemLayer(0), rProg());
5  Func EnlargeTile(T:rTile, mem:MemLayer, P:rProg):
6      if mem.IsLowestLayer()
7          Results.append(P);
8          if (Results.Size() > TopK) Exit();
9          Return();
10     for T' : GetNextRTileShapes(T, mem) do
11         if Visited(T')
12             Return();
13         if MemFootprint(T') > mem.Capacity()
14             P.Add(mem, T);
15             EnlargeTile(T, mem.Next(), P);
16         else
17             if MemPerf(T') > MaxComputePerf(T'.expr)
18                 P.Add(mem, T');
19                 EnlargeTile(T', mem.Next(), P);
20             EnlargeTile(T', mem, P);
21 Func GetNextRTileShapes(T:rTile, mem:MemLayer)
22     alignedSizes = GetNextAlignedAxisSize(T, mem);
23     SortedRTiles = OrderedMap();
24     for d : T.Dimensions() do
25         T' = T.Replace(d, alignedSizes[d]);
26         SortedRTiles.Insert({T', DataReuseScore(T')});
27     Return SortedRTiles;
```

Figure 8: ROLLER's *r*Program constructing algorithm for a single core (e.g., an SM).

scale-out to leverage the multi-core parallelism by replicating the constructed single-core *r*Program.

Scaling up an *r*Program. Since the alignment properties of *r*Tile ensure hardware efficiency, ROLLER can just focus on maximizing the throughput at each memory layer by constructing the right *r*Tile shape. By leveraging the data reuse score defined in §3.1, the single-core *r*Program construction algorithm starts from an initial *r*Tile and gradually enlarges it towards the most cost-effective axis in the *r*Tile (i.e., with the maximum data reuse score). Note that the construction algorithm does not require an absolute data reuse score, it just picks the largest one to maximize the throughput. During the process, the memory performance improves until it hits the computational bound or the maximal memory capacity. The above process repeats for each memory layer from top to bottom, until a desired *r*Program is constructed. Note that if the data reuse score remains constant for some tensor expressions, e.g., element-wise operators, ROLLER will just construct *r*Tiles for the top layer and loads them directly from the bottom layer memory.

Figure 8 shows the detailed construction algorithm. Given a tensor expression `expr` and a target device `dev`, the algo-

rithm constructs an initial *r*Tile T at the top memory layer and enlarges T recursively (`EnlargeTile` in line 4). At each step, it enumerates the next larger *r*Tile T' that improves the data reuse score most (`GetNextRTileShapes` in line 10). If T' hits the memory capacity (line 13) or the data tile loading throughput $MemPerf(T')$ exceeds the peak computing throughput $MaxComputePerf(T')$ (line 17), the algorithm records the current *r*Tile T and goes on to `EnlargeTile` at the next memory layer. Otherwise, it continues to enlarge T' at the current layer (line 20). The construction finishes at the lowest memory layer (line 6), producing one result and repeating, until it obtains K (e.g., 5-20) *r*Programs (to tolerate the hidden factors affected by the device compiler). Note that $MemPerf(T')$ and $MaxComputePerf(T')$ are derived based on `dev`, based on the micro-performance model (§3.3).

Scaling out an *r*Program. Given the homogeneity of both the computation pattern of most DNN operators and the parallel execution units in an accelerator, ROLLER simply replicates the *r*Program constructed on one execution unit to other units, by uniformly partitioning the computation into *r*Tiles of the size equals to the lowest layer *r*Tile. We achieve this by distributing all the partitions evenly to all execution units. Note that ROLLER prefers to assign the partitions split along a reduction axis on the same execution unit, as they can share the reduction results in the higher memory layers. Note that ROLLER does not assume an *r*Program will exclusively occupy all computing units, the system can explicitly control the parallelism of a *r*Program when scaling out.

Small operator and irregular tensor shape. The scale-out algorithm inherently favors operators with sufficient parallelism, e.g., where the partition number is significantly larger than the number of execution units. For a small operator, the overall performance of the algorithm could suffer from the low utilization of parallel execution units. In general, this can be addressed by co-scheduling with other operators in compilers like Rammer [26], if there exists sufficient inter-operator parallelism. Otherwise, for each *r*Program, ROLLER will try to shrink its *r*Tiles along the axis that has the smallest data reuse score to achieve sufficient parallelism. Note that this enumerating process returns the next aligned tile size each time just like other alignment rules, which is an efficient process and incurs negligible costs compared to the overall construction process.

In addition, a large operator may contain irregular tensor shapes with small dimensions, whereas ROLLER might not generate a sufficient number of *r*Programs due to the alignment requirements. To address this issue, ROLLER transforms a tensor expression into a canonical form by an axis fusion pass. Specifically, for all the involved tensors, if there exist two adjacent axes in one tensor, which are either both existing and still adjacent or both missing in all other tensors, ROLLER can safely merge these two axes. For example, an element-wise operator with the tensor shape $[17, 11, 3]$ in both input and output tensors, ROLLER will transform it into the tensor shape $[561](17 \times 11 \times 3)$ by fusing the three axes. Besides axis fusion, ROLLER will also try to greedily increase the parameter ε in the tensor padding mechanism (§3.1) until K *r*Programs have been constructed.

```
// compute interface
int Load(T* src, rTile st, T* dst, rTile dt);
int Store(T* dst, rTile dt, T* src, rTile st);
int Compute(TensorExpr e, rTile t, T** args);

Spec GetDeviceSpec(); // Spec query interface

// interfaces of the micro-performance model
size_t MemFootprint(rTile t);
size_t MemTraffic(rTile t);
double MaxComputePerf(TensorExpr expr);
double MemPerf(rTile t);
```

Figure 9: The interface of ROLLER's hardware abstraction

3.3 Efficient Evaluation of an *r*Program

In the construction algorithm, ROLLER needs to evaluate the performance of *r*Program. Instead of evaluating the end-to-end *r*Program in a real hardware device, ROLLER only needs to evaluates the performance of the corresponding *r*Tile, e.g., `MemPerf` and `MaxComputePerf` in Figure 8.

To this end, ROLLER builds a micro-performance model against a device described in a hardware abstraction layer (HAL). The HAL models an accelerator as multiple parallel execution units with a hierarchical memory layer. The HAL exposes three *r*Tile-based interfaces: `Load`, `Compute`, and `Store` (Figure 9). An execution unit is abstracted as an *rTile Execution Unit (TEU)*, which computes the data tiles through the `Compute` interface. Multiple TEUs can be organized as a group, which `Load` and `Store` tiles cooperatively. The HAL treats different memory layers, e.g., register, shared memory, DRAM, as an unified type exposing the hardware specifications that affect the performance of tile movement. The specifications include memory capacity, transaction lengths, cache line size, and number of memory banks, which can be obtained by the `GetDeviceSpec` interface in Figure 9.

Micro performance model. With the hardware abstraction layer, ROLLER can easily derive the performance of a *r*Tile (and hence the *r*Program). First, given an *r*Tile, the incurred memory footprint (including padding) and the memory traffic volume across different layer can be statically inferred from the *r*Tile's tensor expression `expr` and the `shape`, i.e., the `MemFootprint` and `MemTraffic` interfaces in Figure 9. They are used to calculate the data reuse scores and check if an *r*Tile exceeds the memory capacity. Second, to calculate `MaxComputePerf` of an *r*Tile, ROLLER conducts a one-time profiling to measure the peak compute throughput by aggressively enlarging the compute tiles (e.g., multiple of warp size in an SM) to saturate the TEU. This performance data is cached in ROLLER for future query in the construction al-

gorithm. Finally, for a given *r*Tile, ROLLER also estimates `MemPerf`, the performance on loading data tiles from a memory layer to a higher layer. Given the aligned memory access in *r*Tile, the latency of loading a regular chunk of data can be simply modeled by the division of the total traffic to the memory bandwidth. For the memory layer shared by all TEUs, we split the bandwidth evenly. For the smaller accessing sizes, ROLLER also conducts a one-time offline profiling for each device type and cache the results. It is worth noting that the micro-performance model only *needs* to be accurate when the tile shapes are fully aligned, a key requirement of ROLLER.

4 Implementation

Our implementation of ROLLER is based on TVM [15] and Rammer [26], two open-source DNN compilers. ROLLER's core mechanisms, including expression optimization, construction algorithm, micro-performance model, etc., are implemented with 8K lines of code. ROLLER's compilation pipeline is as follows. Its input is an ONNX graph [9] or a TensorFlow frozen graph [13]. ROLLER first leverages Rammer to conduct graph level optimizations (e.g., inter- and intra-operator co-scheduling). Next ROLLER derives the TVM tensor expressions for each (fused) operator extracted from the optimized graph, and generates corresponding *r*Program by ROLLER's construction algorithm, and performs kernel generation. Finally, the generated kernels are injected to Rammer's runtime and generate the end-to-end model code.

Code generation. Given the fixed code structure in an *r*Program (in Figure 6), ROLLER generates the kernel code through a predefined template, implemented as a TVM schedule with its built-in scheduling primitives. Loading and storing data tiles at each memory layer are implemented by TVM's `cache_read` and `cache_write` primitives. Partitioning on *r*Tile is done through `split` and `fuse`. Some primitive *r*Tile computation is implemented with TVM's intrinsic API. With the template, a given *r*Program can be directly generated into device codes, e.g., CUDA kernels.

Tensor padding. ROLLER relies on tensor padding to align *r*Tiles with tensor shape. In practice, most tensors in the lowest memory (e.g., DRAM) are allocated by external program (e.g., DNN framework), thus we just apply padding in the upper layer memory (e.g., shared memory). Our tensor padding currently requires the input tensor expression to specify whether it allows to pad, as well as the default padding value (e.g., 0 for MatMul operator). For the storage padding for memory bank alignment, we leverage TVM's `storage_align` primitive to add padding.

Performance profiling. ROLLER implements two profilers: a *micro-performance profiler* and a *kernel profiler*. The former generates device specifications, e.g., memory bandwidth, computing throughput, etc., through a set of micro-benchmarks, which is a one-time offline profiling for each device type and tensor expression types (regardless of the tensor shapes). The latter profiles the fastest kernels among the top K *r*Programs and is used for each compilation result if the K is larger than 1. In practice, the performance of a specific kernel code is also slightly affected by some device-compiler and hardware related hidden factors, which ROLLER can hardly control. These factors include instruction density of different instruction types, register allocation behaviors, device compiler optimizations, warp scheduling overhead, etc. Particularly, on NVIDIA GPUs, ROLLER relies on nvcc [3] to compile the generated CUDA codes into machine code. However, nvcc's proprietary optimizations might undesirably affect the program execution behaviors. Thus, ROLLER leverages the kernel profiler to quickly evaluate top performing *r*Programs and select the best one. A larger K could generally increase kernel quality. After evaluating the top 10, 20, and 50 results, our experiences show that top 10 could recall the optimal results for most cases. Note that ROLLER's kernel profiler differs from the evaluation process driven by a machine learning algorithm in previous compilers [15,33,35]. The ML-based approach usually requires hundreds even thousands of *sequential* evaluation steps, while ROLLER only profiles tens of candidates *in parallel*. In future, we plan to implement assembly-level code generation to alleviate the hidden issues in a high-level device compiler.

ROLLER's HAL allows us to support different accelerators easily. User can configure the corresponding HAL for each device type. ROLLER also provides built-in configurations for most common device types. Some detailed configurations, e.g., memory bandwidth, rely on micro-benchmark profiling or derive from published device specifications. Next, we share our experiences in implementing the HAL on several popular DNN accelerators, including NVIDIA GPUs, AMD GPUs and Graphcore IPU.

ROLLER on NVIDIA CUDA GPUs. An NVIDIA GPU usually employs a centralized memory architecture. We implement ROLLER on V100 and K80, two CUDA GPUs with different architectures on the *streaming multi-processors (SMs)*. Their memory architecture contains global memory, L2 cache, L1 cache, shared memory, and register. In ROLLER's HAL, we abstract them into 3 memory layers: L2 layer for global memory and L2 cache, L1 layer for only the shared memory, and the L0 layer for register. We ignore L1 cache because it shares the space with shared memory and cannot be controlled by user programs. The memory bandwidths of all levels are measured by our micro-benchmarks. The transaction length at the global memory layer is set to 32 Bytes, i.e., 8 float elements, for both GPUs. For V100 GPUs, the bank number and the bank length of the shared memory is 32 and 4 Bytes respectively. For K80 GPUs, the bank length is 8 Bytes. The shared memory capacities are set as 48KB for both GPUs (based on `deviceQuery`).

We implement the TEU on CUDA GPUs as a warp of 32 threads, which is also the basic unit to execute the TensorCore WMMA instructions. The size of a TEU Group on a HAL

(e.g., a SM) is set to the warp scheduler number, which is 4 for both GPUs. The SM number is 80 for V100 [21] and 13 for K80. On CUDA GPUs, each thread has a limited register capacity, e.g., 255 registers for V100. Exceeding this limit will lead to register spilling, causing significant performance degradation. This sets a limit to the size of an *r*Tile at register layer. We notice that the *nvcc* compiler will implicitly declare more registers (for loop variables or other purposes). Given that this behaviour is hard to predict, we reduce the register limit empirically to only 96 registers for both V100 and K80 per thread to avoid unexpected performance impacts.

ROLLER on AMD ROCm GPUs. We also implement ROLLER on MI50 [12], AMD's second-generation Vega series GPU. MI50 shares a similar memory architecture as V100: the centralized global memory can be accessed by all *compute units (CUs)*. Like SMs in NVIDIA GPU, each CU has its own scratchpad memory, registers, and computation cores. The data movement of a ROCm [1] kernel program is also similar. The memory transaction size for the global memory is set as 64 Bytes. The memory bank number is 32 and bank length is also 4 Bytes. We also implement the TEU as a warp of threads, which is 64 threads on MI50 GPUs. The maximal register size is empirically limited to 70 registers per thread. All other specifications such as the memory bandwidths at each layer, peak computing throughput, etc., are measured with our micro-benchmark.

ROLLER on Graphcore IPUs The Graphcore IPU [22] is a massive parallel MIMD processor with 1216 parallel processing cores. Distinct from NVIDIA and AMD GPUs, an IPU employs a distributed memory architecture. There is only 256KB on-chip local memory attached per core, and no unified global memory. When the local memory is unable to hold all the input data, by default, the initial data of a kernel program is stashed in the on-chip local memory and evenly distributed across the nodes. Thus, ROLLER's HAL for IPUs also abstracts three memory layers: L2 for all the remote memories across all cores, L2 for the local memory on each core, and L0 for the register. We take advantage of prior benchmarking work [22], which has successfully measured peak memory bandwidth and computation throughput. The size of the register files per IPU core is not publicly available. Considering that we have no prediction for behaviours of the IPU program compiler, we allow each upper-level *r*Tile to use only 10 registers, which safely guarantee that the tiling algorithm does not emit invalid tiling configurations.

5 Evaluation

We evaluate ROLLER on both DNN operator benchmarks and end-to-end models by comparing with state-of-the-art DNN compilers and frameworks. We first summarize our findings: 1) ROLLER achieves *three orders of magnitude speedup* on compilation time, compared to TVM and Ansor. On V100 GPU, the most expensive operator takes 43 seconds, while all other operators take only around 13 seconds to compile. 2) ROLLER matches the state-of-the-art performance of vendor libraries and other compilers on a wide range of operators. It even outperforms others for more than 50% of operators. 3) For operators with smaller sizes and irregular shapes, ROLLER's results are sub-optimal because of the difficulty in aligning with the hardware. However, their kernel execution time is usually small (around or below 1ms). 4) We have conducted the most extensive evaluations (119 ops in total) covering different operator types over different accelerators.

Operator	Configuration	Note
MatMul	M=65536,K=2,N=1024	M0
MatMul	M=128,K=4032,N=1000	M1
MatMul	M=65536,K=1024,N=4096	M2
Conv2D	D=(128,128,28,28), K=(128,128,3,3),S=1	C0
Conv2D	D=(128,128,58,58), K=(128,128,3,3),S=2	C1
Conv2D	D=(128,256,30,30), K=(256,256,3,3),S=2	C2
DepthwiseConv	D=(128,84,83,83), K=(84,84,5,5),S=2	D0
DepthwiseConv	D=(128,42,83,83), K=(42,42,5,5),S=1	D1
DepthwiseConv	D=(128,84,21,21), K=(336,336,1,1),S=1	D2
Element(Relu)	I=(128,1008,42,42)	E0
Element(Relu)	I=(128,256,14,14)	E1
Element(Relu)	I=(128,1024,14,14)	E2
Avgpool	D=(128,168,83,83),K=1,S=2,VALID	P0
Avgpool	D=(128,617,21,21),K=3,S=2,SAME	P1
Avgpool	D=(128,42,83,83),K=3,S=1,SAME	P2
ReduceMean	I=(128, 512, 1024), axis=[2]	R0
ReduceMean	I=(65536, 1024),axis=[1]	R1
ReduceMean	I=(128, 4032, 11, 11), axis=[2,3]	R2

Table 1: A subset of operator configurations in our benchmark.

Experimental setup. ROLLER is evaluated on four types of servers equipped with different accelerators. The CUDA GPU evaluations use two types of servers: an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB) GPUs and an Azure NC24_v1 VM with 24 Intel(R) Xeon(R) CPU E5-2690v3 CPUs and 4 NVIDIA Tesla K80 GPUs. Both running on Ubuntu 16.04 with CUDA 10.2 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 4 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 4.0.1 [1]. The IPU evaluations use an Azure ND40s_v3 VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare ROLLER against other tensor compilers, vendor libraries and DNN frameworks, including TVM [15] (v0.8) and Ansor [33] (v0.8), two state-of-the-art tensor compilers; cuDNN, cuBLAS, rocBLAS (ROCm GPUs), POPLAR library (Graphcore IPU), which are vendor libraries; TensorFlow (v1.15), a state-of-the-art DNN framework; TensorFlow-XLA a state-of-the-art DNN full-model compilers; and TensorRT (v7.0) (with TensorFlow integration version), a vendor-specific inference library for NVIDIA GPUs. We validate our compilation results by comparing them against Ansor's.

Benchmarks. Our evaluation benchmark uses four typical DNN models, including ResNet-50 [19] (CNN), LSTM [20] (RNN), NASNet [36] (a state-of-the-art CNN model obtained

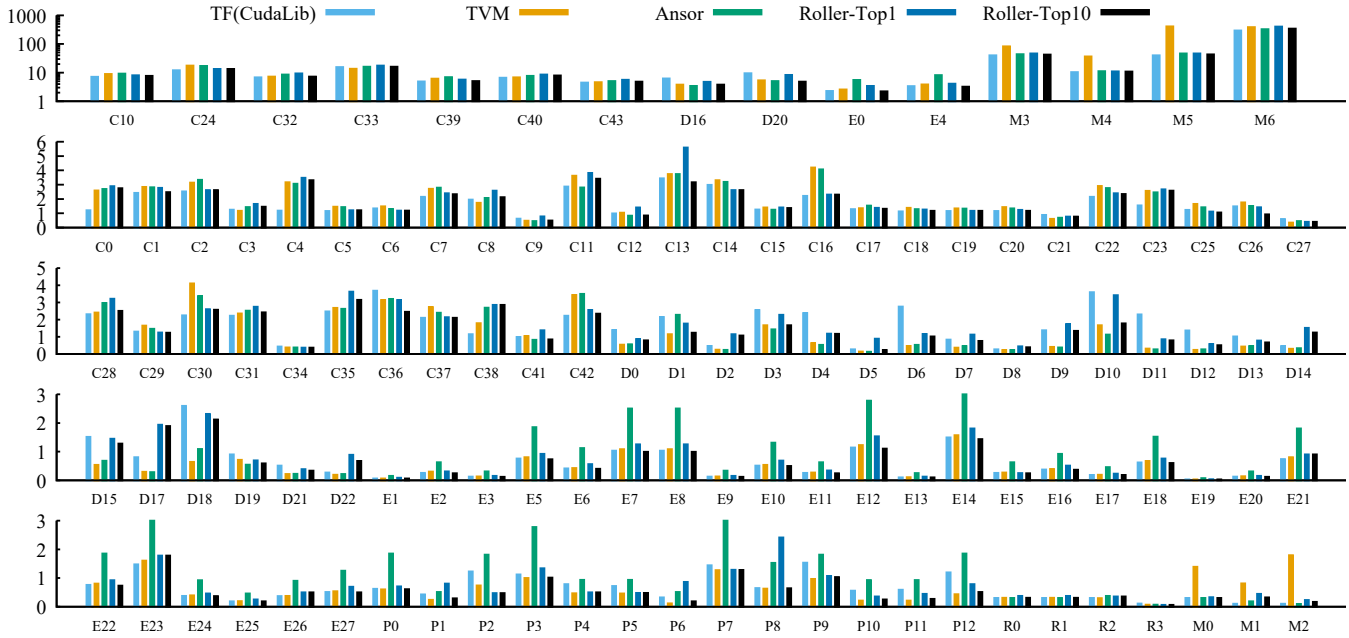

Figure 10: Operator performance on V100 GPUs (y-axis: average kernel execution time in ms).

by the neural architecture search), and BERT-Large [17] (transformer-based). We set the default batch size of each model to 128. From each model, we choose the most-frequently used operators to construct our operator benchmark. It contains 6 classes of operator type with total 119 operator instances with different configurations (7 MatMul operators, 44 Conv2D operators, 23 DepthwiseConv operators, 28 element-wise operators, 13 pooling operators, and 4 reduction operators). Table 1 lists a representative subset of operators as well as their configurations. The last column lists the corresponding abbreviation of each operator. The full list of the operator configurations is omitted due to page limit.

5.1 Evaluation on NVIDIA GPUs

This section first evaluates ROLLER's operator performance, compilation time, and scalability on large operators by comparing against the state-of-the-art tensor compilers and vendor libraries. We also evaluate the performance of ROLLER on TensorCore. Finally, we show the end-to-end model performance compared to existing DNN compilers and framework.

Operator performance. We first evaluate the performance of ROLLER generated kernels by comparing against TVM (i.e., AutoTVM with XGBoost tuning algorithm [16]), Ansor, cuBLAS (for matrix multiplication operators) and cuDNN (for convolution operators). Vendor libraries like cuBLAS and cuDNN are wrapped in TensorFlow to evaluate the performance. For the rest of operators (e.g., element-wise, reduce), we use TensorFlow's built-in kernel implementations. To amortize the overhead of data feeds/fetches in TensorFlow's session, we repeat the kernel running for 1,000 times in each session and calculate the average. We set the tuning steps for TVM and Ansor to 1,000 for each operator, same as Ansor's evaluation setup [33], and report the best results. We compare both the top-1 and the best from the top-10 kernels constructed by ROLLER, the latter can tolerate some hidden performance impacts from device compilers.

Figure 10 plots the average kernel performance for all the 119 operators in our benchmark, ordered by the operator type and ID. We plot the large operators (e.g., kernel time is larger than 5ms) in the top sub-figure in a log-scale for y-axis, and the other medium and small operators in the bottom 4 sub-figures [2]. First, compared to CUDA libraries (CudaLib), ROLLER could get comparable performance (i.e., within 10% performance) for 81.5% of the total operators, and can be even faster for 59.7% of them. We observe that the majority of operators that ROLLER performs worse are convolution operators with 3×3 or larger filters, which are usually implemented with a more efficient numerical algorithm (e.g., Winograd [23]) in cuDNN and hard to be expressed by the tensor expression. This is the reason Ansor and TVM are also slower than CudaLib in these cases. Second, compared to TVM and Ansor, ROLLER could also get comparable performance for 72.3% and 80.7% of the total operators respectively. The rest 27.7% and 19.3% of them are mainly small operators or with irregular tensor shapes, which are by natural hard to align with the hardware. However, these operators usually have relatively short kernel time, e.g., only 1.65ms and 1.16ms on average. Among 54.6% and 65.5% of the total

[2] Please find the complete results in our artifact.

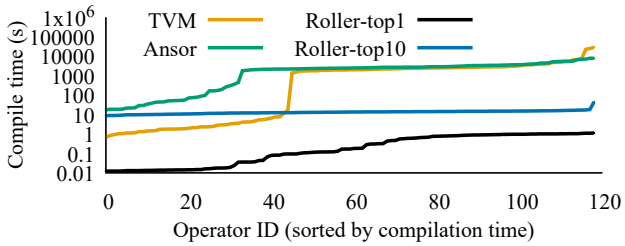

Figure 11: Compilation time for each operator.

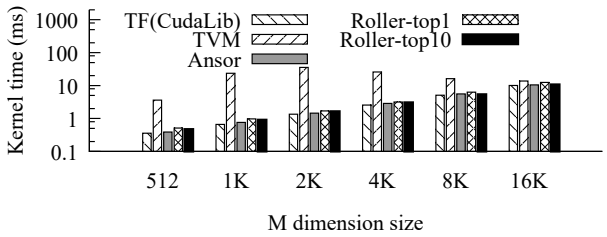

Figure 12: Kernel time for MatMul operator with different sizes of M in BERT-Large model, K=1024, N=4096.

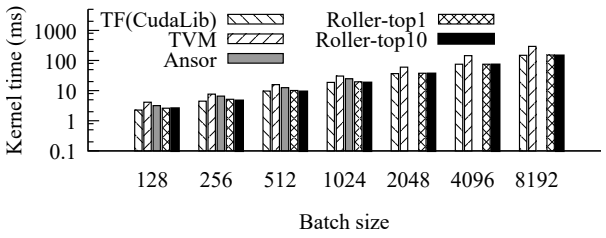

Figure 13: Kernel time for Conv2d operator with different batch sizes of N, where C=1024, H=14, F=2048, K=1, S=2.

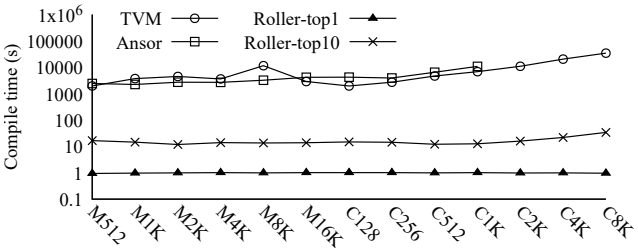

Figure 14: Compilation time for both MatMul and Conv2d operator with different batch sizes.

operators, ROLLER can even produce faster kernels than TVM and Ansor, respectively. We observe that the majority of these operators are large and time-consuming ones. As it shows in the top sub-figure where operators are larger than 5ms (up to 343ms), ROLLER could achieve better performance for most of these operators, e.g., by 1.85× and 1.27 × speedup over TVM and Ansor on average.

Compilation time. Given the comparable kernel performance, the major advantage of ROLLER is its fast compilation. Figure 11 compares ROLLER's compilation time against TVM and Ansor for all the operators. The operator ID is sorted by the compilation time for each line. The average operator compilation time for TVM is 0.65 hours and up to 7.89 hours. For the first 40 operators, which are mainly the element-wise, reduction, and pooling operators, TVM's compilation takes less than 10 seconds. This is because TVM's manually-written code templates for these operators can directly emit code without searching. However, Ansor generates search spaces for all the operators. Its compilation time takes 0.66 hours on average and up to 2.17 hours. In contrast, ROLLER's top-1 kernel results can be generated in 1 second for most operators and in 0.43s on average, which is *more than three orders of magnitude* faster. The major time is spent on the recursive constructing algorithm, which increases slightly with the growth of operator size, but quickly stabilizes as the recursive depth (to enlarge the *r*Tiles) is bounded by the limited memory capacity. To get the optimal kernels from the top-10 candidates, ROLLER's average compilation time is only 13.3 seconds. The major cost comes from the kernel code compilation with the device compiler and the evaluation on target devices.

Scale-out with operator size. We evaluate the scalability of ROLLER on larger operators by comparing with both CUDA libraries, TVM, and Ansor. We select a MatMul operator from the BERT model and a Conv2D operator from the ResNet mode, and scale them by setting different batch sizes. Figure 12 and Figure 13 show the performance comparisons. For the MatMul operator, both Ansor and ROLLER have a linear scalability over the batch sizes and comparable performance with CudaLib (i.e., cuBLAS). However, TVM's performance is relatively non-stable. For example, ROLLER can outperform TVM by average 11.2× and up to 36.1× for the batch size of 1024. For Conv2D operators, ROLLER can still achieve linear scalability over the batch size, and get slightly better performance than Ansor and TVM (by 1.25 and 1.54× on average). Note that Anosr is unable to search for a valid kernel for the batch size over 2048 using its default configurations. TVM can generate valid kernels, but the performance is scaled sublinearly for the larger batch sizes, e.g., ROLLER can achieve more than 1.9 × speedup for batch sizes greater than 2048.

Finally, Figure 14 compares the compilation time for the two operators with different batch sizes. The average compilation time of TVM and Ansor is 2.36 (up to 9.55) hours and 1.19 (up to 3.0) hours respectively. Moreover, their compilation time grows constantly with the growing of batch size. This is because that they are both based on ML-based search approach, whose search space usually increases exponentially with the operator size. In contrast, ROLLER produces the top-1 kernel in 1 second, and 16 seconds (up to 34 seconds) on average for the top-10 kernel.

Compile on TensorCore. ROLLER could easily support hardware tensor ISAs (e.g., TensorCore) by aligning the *r*Tile shape with the hardware instruction shape. We use the 16×16×16 WMMA instruction in ROLLER. We remove An-

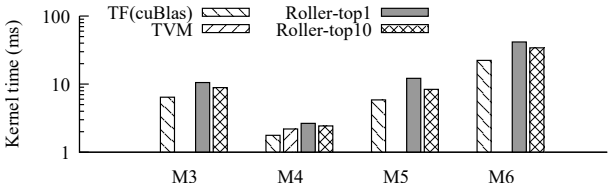

Figure 15: Matmul kernel time on TensorCore.

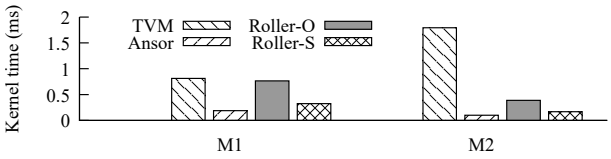

Figure 16: Performance for small operators.

sor in this experiment as it does not support TensorCore to our best knowledge. We select 4 large MatMul operators that are friendly to TensorCore in this experiment. Figure 15 shows the performance comparisons. As it shows, by constructing from the aligned *r*Tile shape, ROLLER can quickly produce good kernels on TensorCores, e.g., within a 43% performance gap to cuBLAS. Note that cuBLAS is highly optimized with a lot of hand-crafted optimizations on TensorCore. As a comparison, TVM fails to generate valid kernels for 3 of the 4 total operators with the default configurations. We try to increase the tuning steps from 1,000 to 10,000, it is still unable to find a legitimated kernel due to its poorly-defined search space.

Small operators and irregular tensor shape. ROLLER optimizes performance for small operators by shrinking the *r*Tile when there is insufficient parallelism. We demonstrate the performance of this optimization for the two small MatMul operators. Figure 16 compares the performance of the original *r*Tile configuration without sufficient parallelism (Roller-O), and the shrunken *r*Tile configuration (Roller-S) which matches the SM parallelism. As it shows, shrinking *r*Tile could significantly improve performance than the original kernel, e.g., by 2.3× on average. However, ROLLER is still slower than Ansor, e.g., by 50% on average, on small operators, even it is significantly faster than TVM by 6.6×. For such operators, we can further leverage search-based approach to fine-tune the configurations to obtain a better performance.

ROLLER compiles operators with irregular tensor shapes with two optimizations: i.e., *axis fusion* and *tensor padding* with bound parameter ε. We demonstrate their benefits on a representative set of irregular convolution operators, as shown in Figure 17. We compare the performance of ROLLER without any optimizations (Roller-B), with axis fusion (Roller-F), and further with tensor padding of ε from 0.4 to 1.0 (Roller-P0.4 and Roller-P1.0). All ROLLER's performances are the best one selected from the top-10 candidates. First, with axis fusion optimization, ROLLER is able to have more *r*Tiles that aligns with the tensor shapes, which improves the kernel performance by 1.5× on average. Moreover, with the tensor

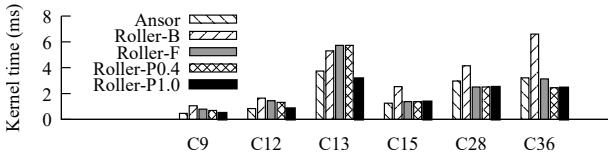

Figure 17: Performance for operators with irregular shapes.

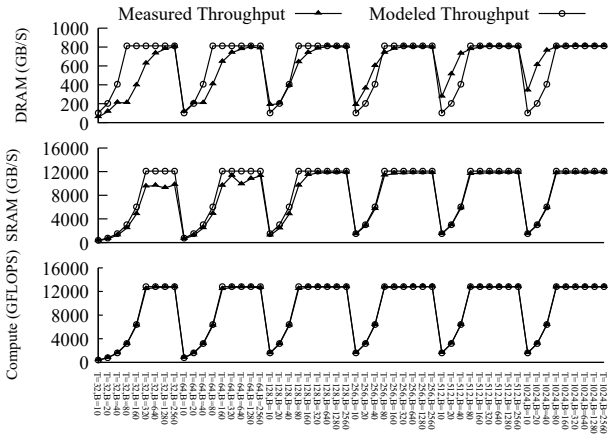

Figure 18: Memory throughput (DRAM and shared memory) and compute throughput from our micro-performance model and real measurement (X-axis: kernel configurations with different number of threads per block (T) and blocks (B)).

padding optimizations (e.g., at ε of 1.0), ROLLER can further improve performance than Roller-F by 1.4×. This is mainly because the number of legitimated kernels is very limited with smaller ε for irregular shapes. Increasing the ε allows ROLLER to have chance to select from more candidate kernels.

Micro-performance model. We conduct extensive experiments to validate the micro-performance model, including global memory throughput, shared memory throughput, and compute throughput, under different kernel configurations (i.e., different thread block and grid size). Figure 18 compares the performance estimated by our micro-performance model with that measured on real device. As shown, when the configuration is not aligned with the parallelism of execution units, i.e., thread number per block is less than 128 (4 warps), our model produces a relatively estimation error, especially for the DRAM throughput. This is also the case when there is insufficient parallelism (i.e., block number is less than 80). Thus, we can see our micro-performance model is accurate only for those shape-aligned configurations (i.e., *r*Tiles), as they fully exploit hardware efficiency. This also motivates us to choose only the aligned *r*Tiles, which greatly reduces the complexity of micro-performance model.

Kernel performance. We further study how close the performance of ROLLER generated kernels can approach the optimal. Since ROLLER's data pipeline model can naturally identify the bottleneck layer, e.g., DRAM, shared memory, or

Resource utilization	60-70%	70-80%	80-90%	90-100%
Operator #	6	13	22	78
Percentage	5%	11%	18%	66%

Table 2: The distribution of resource utilization at the saturated layer for different kernels.

Baseline	MemAlignn	EUAlign	ShapeAlign	BankAlign
1.0x	1.42x	1.88x	1.92x	1.94x

Table 3: Average accumulated performance improvement with different alignment optimization.

computation, we profile each generated kernel and compare the corresponding resource utilization at the saturated layer with the theoretical hardware limit. Table 2 lists the distribution of the resource utilization for the total 119 operators. The table shows most kernels saturate hardware resources, e.g., 66% of them utilize more than 90% of the theoretical limit. For the few under-utilized kernels, especially whose utilization is less than 80%, our investigation shows that they are mostly small operators with insufficient parallelisms.

To understand the impact of different alignment rules, we incrementally turn on each alignment optimization and evaluate its performance improvement. Table 3 shows the average speedup compared with the baseline (without any optimization). For example, EUAlign shows the kernels with the alignment on execution units and memory transaction alignment (MemAlign) can together improve the performance by 1.88x than the baseline. Bank alignment (BankAlign) has relatively small improvement because most kernels are already bank conflict free.

End-to-end model performance. We evaluate the end-to-end model performance of ROLLER by comparing against TensorFlow (TF), TensorFlow-XLA (TF-XLA), TensorRT (TF-TRT), and Ansor, which represent the state-of-the-art DNN framework, graph-level compiler, vendor-provided DNN engine, and DNN compiler with tensor compilation, respectively. Note that TensorRT is also the core engine in NVIDIA Triton inference server [8]. We omit TVM in this experiment as it usually requires an order of magnitude longer compilation time on tuning end-to-end models than Ansor [33]. ROLLER's end-to-end model compilation is implemented in Rammer (i.e., Rammer+Roller) by feeding the generated kernels into it. To create a fair baseline, we manually feed both the TVM and Ansor generated kernels for the same set of operators into Rammer, which are denoted as Rammer+TVM and Rammer+Ansor.

Table 4 lists the model execution time for each model compiled or executed by each compiler and framework. Note that TF-XLA fails to compile the BERT-Large and NASNet model (out-of-memory). TF-TRT also fails to run the BERT-Large model due to exceeding the maximum protobuf size limit (2GB) in its graph loading stage. For Ansor, we set the total tuning steps as 1,000 multiplied with the number of subgraphs for each model. However, Ansor also fails to produce

	BERT-Large	ResNet	NASNet	LSTM
TF	5,186	131	1,041	141
TF-XLA	OOM	112	OOM	98
TF-TRT	N/A	137	883	31
Ansor	46,847 (TVM)	122	927	84
Rammer+TVM	17,730	143	1,168	43
Rammer+Ansor	5466	137	1036	48
Rammer+Roller	4,850	142	1,005	20
Ansor compile-time	30.9h (TVM)	33.4 h	41.8h	11.3 h
Roller compile-time	371s	352s	668s	298s

Table 4: End-to-end model execution time (in milliseconds) and compilation time on V100 GPUs.

	TF(CudaLib)	TVM	Ansor
Better Performance	82.4%	65.5%	71.4%
Perf. within 5%	82.4%	67.2%	75.6%
Perf. within 10%	83.2%	73.1%	79.0%
Perf. within 50%	99.2%	93.3%	94.1%
Perf. within 90%	100.0%	100.0%	100.0%

Table 5: The percentage of better and comparable performant operators on NVIDIA K80 GPUs.

a legitimate program for BERT-Large models. Thus, for this case, we use TVM to compile the model. Note that, the performance of TVM for BERT-Large is about 2.6× slower than Rammer+TVM, as the default layout of the dense operator in TVM (i.e., NT) is different from that in Rammer (i.e., NN). First, for the ResNet and NASNet models, ROLLER can only achieve comparable and mostly slower performance than TF, TF-XLA, and TF-TRT (up to 26.7% slower compared to TF-XLA for ResNet). This major overhead in ROLLER is caused by the less efficient convolution kernels compared to cuDNN as explained before. However, for the BERT-Large and LSTM models, ROLLER can outperform all other frameworks and compilers, e.g., by 1.07× and 1.55× faster than the state-of-the-arts, i.e., TF for BERT-Large and TensorRT for LSTM. This mainly due to ROLLER's kernel construction favors large and regular operator shape, which are heavily used in the BERT-Large model. For both the BERT and LSTM models, since ROLLER can control to generate resource-efficient kernels by the scaling-up policy, it provides more opportunities for Rammer to co-schedule parallel kernels on the parallel SMs on GPUs. They together produce an efficient end-to-end program, which can even outperform TF-TRT by 1.55× for LSTM. Among all the implementations, Ansor can also produce very efficient programs for all the rest 3 models except for the BERT. However, it requires a long compilation time (29.3 hours on average). For the NASNet model, it reaches only 32% of the overall searching progress after tuning for 41.8 hours. In contrast, ROLLER only takes 422s on average to compile these models. This includes the graph-level optimization and the full-model compilation time in Rammer, which occupies about 41% of the total time on average.

Operator performance on K80 GPUs. We also evaluate ROLLER on the K80 GPUs. Table 5 shows the percentage of better or comparable performing operators (e.g., within 10%

	TF(RocLib)	TVM	Ansor
Better Performance	73.1%	58.8%	70.6%
Perf. within 5%	79.0%	62.2%	72.3%
Perf. within 10%	81.5%	62.2%	73.9%
Perf. within 50%	94.1%	84.0%	86.6%
Perf. within 90%	100%	100%	100%

Table 6: The percentage of better and comparable performant operators on AMD ROCm MI50 GPUs.

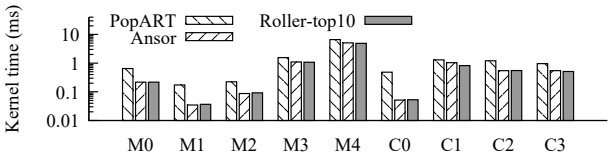

Figure 19: Operator performance on Graphcore IPU (y-axis in log-scale).

differences or 1.1× slow down) ROLLER generates for our operator benchmarks. Compared to CUDA libraries, TVM, and Ansor, ROLLER produces 82.4%, 65.5% and 71.4% better kernels for the whole operator benchmark. The percentage is relatively low for TVM mainly because the manual-crafted element-wise kernel templates in TVM are already highly-optimized. Finally, the average compilation time for all operators is 0.65 hours for TVM and 0.95 hours for Ansor respectively. In contrast, ROLLER's average compilation time is only 5.24 milliseconds for top-1 kernel and 12.3 seconds for top-10 kernel.

5.2 Evaluation on Other Accelerators.

Operator performance on AMD ROCm GPUs. We evaluate ROLLER on AMD ROCm GPUs by comparing it against ROCm libraries, TVM, and Ansor. Table 6 shows the percentage of operators that ROLLER can produce better or comparable performance (e.g., within 5% and 10% differences) in our operator benchmarks. Compared to the ROCm libraries (e.g., rocBlas), 73.1% of the total operators ROLLER can produce better kernels. This percentage is much higher than that on CUDA GPUs (59.7% and 54.6% for V100 and K80 GPUs). This is mainly because the libraries on CUDA GPUs are more mature than the ROCm GPUs, where ROLLER can help significantly. Compared to TVM and Ansor, ROLLER can also produce 58.8% and 70.6% better kernels. Similar to CUDA GPUs, the kernels that are slower by more than 10% are mostly small operator and those with irregular tensor shapes: the average execution time of these kernels are only 1.69ms and 1.57ms for TVM and Ansor, respectively. Finally, the average compilation time for all operators is 0.85 (up to 4.2) hours for TVM and 0.99 (up to 3.4) hours for Ansor, respectively. In contrast, ROLLER's average compilation time is 0.24 (up to 0.63) seconds for top-1 kernel and 7.69 (up to 49.0) seconds for top-10 kernel.

Operator performance on Graphcore IPU. We evaluate ROLLER on Graphcore IPUs. Due to the limited on-chip memory capacity, we only evaluate a set of small MatMul and Conv2D operators with different configurations. Figure 19 shows the average kernel time of each operator in log-scale, comparing against the Poplar-sdk library (i.e., PopART) provided by Graphcore and Ansor. Since TVM and Ansor do not have Graphcore backends, we use a modified version of Ansor in this experiment. As it shows, ROLLER can generate faster kernels than PopART for all operators, with an average of 3.1× and up to 9.2× speedup. Even comparing to Ansor, ROLLER can still construct comparable or even better kernels in most of operators, i.e., 2.9% average improvement. Note that Ansor still requires hours of tuning for each operator, as the device compiler on IPUs could take up to minutes to compile a program. However, ROLLER usually produce good kernels from the top-10 constructed candidates in several minutes. This time is mainly bottle-necked by the less-matured device compiler. It also brings more challenges to adopt the ML-based tensor compilers on these devices.

6 Discussion and Future Work

Optimization space compared with loop-based compiler. The abstraction of *r*Tile and data processing pipeline allows ROLLER to construct an optimization space overlapped with, but different from, existing DNN compilers (e.g., Ansor) [15, 33, 35]. As mentioned previously, these compilers view tensor compilation as nested loop optimizations. For example, Ansor allows only divisible tiling sizes along a tensor dimension to partition a loop axis evenly. This makes it usually perform worse for tensor shapes with prime dimensions. ROLLER instead focuses on maximizing hardware efficiency from the data-processing-pipeline view, allowing more aggressive optimizations, e.g., exploring non-divisible but hardware-aligned tiling sizes with fused adjacent axis and padded tensor shapes. Driven by our observation that most DNN operators are memory-bound, ROLLER fundamentally differs from existing DNN compilers by first optimizing data-tile throughput, i.e., maximizing reuse score rewards and aligning with hardware features, and then for parallelism. Such a trade-off inherently leads to fast compilation and good performance for operators with sufficient parallelism.

Optimization trade-off. ROLLER's design philosophy is based on an observation: large and dense operators tend to be major contributors to the execution time. This leads to a design trade-off: optimizing data reuse (i.e., maximizing pipeline throughput) as the primary optimization goal, and turning other hardware related optimizations into alignment constraints. Such trade-off results in fast compilation and high kernel quality for a majority of operators in mainstream workloads. For small operators, ROLLER further employs some adaptive mechanisms to trade-off among different optimiza-

tion goals, e.g., using a threshold to limit redundant work (§3.1) when there are insufficient results, employing an adapting *r*Tile shrinking process to increase parallelism (§3.2), etc.

Future work. ROLLER currently relies on high level device compiler, e.g., nvcc, to compile kernel code to executable. This sometimes introduces undesirable performance impacts and forces ROLLER to allocate registers conservatively. This is because the device compiler will implicitly allocate registers for intermediate values (e.g., loop variables). ROLLER cannot detect implicit register allocation beforehand, hence it is difficult to estimate and decide the precise register usage. One of our future work is to generate assembly (e.g., PTX for NVIDIA GPUs) code directly to avoid the side effects from the high level device compiler.

Moreover, although the key hardware information that affects performance, including memory bandwidth, capacity, and transaction length, is often available in the hardware specification, there are still some devices (e.g., mobile GPUs) lacking such information. Another future work is to leverage some profiling techniques [25] to disclose and quantify those hardware features.

ROLLER's HAL assumes hardware contains homogeneous computing units and symmetric memory accessing. However, we also observe that some devices have NUMA architecture. This makes it difficult for the micro-performance model to estimate *r*Tile performance, as the same tile will perform differently at different locality under NUMA architecture. We leave this issue as future work.

Finally, the optimization for sparse kernel may also violate the assumption of homogeneous workload in a DNN kernel and make the micro-performance model inaccurate. Some tiles with a larger degree of sparsity may perform differently from dense tiles. ROLLER assumes a higher level, sparsity-aware compiler (e.g., SparTA [34]) will address this issue.

7 Related Work

Most tensor compilers treat DNN operators as nested multi-level loop computation, which essentially defines a large space with a combinatorial complexity. TVM [15] inherits the insight from Halide [27] and describes DNN operators as loop optimization schedule primitives. Later, AutoTVM [16] extends TVM to apply an ML-method to search for the best configurations from manually written code templates. FlexTensor [35] proposes to automatically explore the space without manual templates. Ansor [33] further advances such automation. It generates an even larger search space considering a hierarchical code structure and adopts an evolution algorithm to find performant kernels. Compilers like Tiramisu [14], AKG [32], and Tensor Comprehensions [29] apply polyhedral-based techniques to loop optimization, which transforms the loop into an integer programming problem and finds a good configuration with a solver. All these approaches rely on a huge search space to provide good kernel, which leads to long compilation/solving time. ROLLER explores a different approach to construct *r*Tiles that align with hardware features.

Tensor Processing Primitives (TPPs) [18] define a set of 2D-tensor operators to compose complex operators on high-dimensional tensors, providing limited expressiveness. In contrast, ROLLER does not limit the dimension of tile shape and can be applied to general tensor expressions. The OpenAI Triton [28] is a programming framework and compiler for developing block-based GPU kernels. Triton relies on programmers to decide the block size and block scheduling, while this is the key problem ROLLER addressed by considering both hardware features and tensor shapes. MLIR [5] and Tensor IR [10] plan to support block-level (i.e., tile) computation representation in their IRs. ROLLER's *r*Tile abstraction and the *r*Program construction are compatible with these initiatives.

Graph-level DNN compilers like XLA [11], TVM [15], and Rammer [26] focus on cross-operator optimizations, e.g., operator fusion/co-scheduling. ROLLER's kernel generation is compatible with these compilers. ROLLER's *r*Tile abstraction complements the rTask concept in Rammer [26] as it provides an efficient way to construct an rTask.

Finally, some works focus on operator-specific optimizations. CUTLASS [7] is a template for implementing matrix-multiplication. An analytical model [24] is proposed to find the best loop-level optimization configuration only for convolution operators on multi-core CPUs. And DREW [30] proposes a new way to optimize Winograd convolution using data compression [31]. ROLLER's optimization approach is general for DNN operators on various devices.

8 Conclusion

ROLLER takes an unconventional approach to deep learning compiler. Instead of relying on costly machine learning algorithms to find a good solution in a large search space, ROLLER generates efficient kernels using a recursive construction-based algorithm that leverages the new *r*Tile abstraction with much fewer shapes that align with multiple hardware features. The constructed program can be evaluated by a micro performance model, without running on a real device every time. As a result, ROLLER can compile high-performance kernels in seconds, even in less mature accelerators. More importantly, ROLLER offers a unique and significantly more efficient approach for new AI hardware vendors to build competent vendor-specific DNN libraries, bridging the ecosystem gap to market leaders and thereby facilitating innovations in AI accelerators.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof.Yufei Ding, for their extensive suggestions.

References

[1] AMD ROCm Platform. https://github.com/RadeonOpenCompute/ROCm.

[2] CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html.

[3] CUDA NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[4] IPU PROGRAMMER'S GUIDE. https://www.graphcore.ai/docs/ipu-programmers-guide.

[5] MLIR. https://mlir.llvm.org/.

[6] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[7] NVIDIA cutlass. https://github.com/NVIDIA/cutlass.

[8] NVIDIA TRITON INFERENCE SERVER. https://developer.nvidia.com/nvidia-triton-inference-server.

[9] ONNX. https://onnx.ai/.

[10] TensorIR. https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872.

[11] XLA. https://www.tensorflow.org/xla.

[12] AMD Radeon Instinct™ MI50 Accelerator, accessed 2018 Nov. https://www.amd.com/en/products/professional-graphics/instinct-mi50.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[14] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3389–3400. Curran Associates, Inc., 2018.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[18] Evangelos Georganas, Dhiraj D. Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Narendra Chaudhary, Abhisek Kundu, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. *CoRR*, abs/2104.05755, 2021.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[22] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[23] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[24] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization

and design space exploration for optimization of cnns. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 928–942, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *The 28th Annual International Conference On Mobile Computing And Networking (MobiCom 2022)*. ACM, February 2022.

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[28] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[29] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[30] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1807–1816, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Poclib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):459–475, 2022.

[32] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[34] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. Deep-learning model sparsity via tensor-with-sparsity-attribute. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, 2022.

[35] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. pages 859–873, 03 2020.

[36] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.