Scaling Deep Learning Computation over the Inter-core Connected Intelligence Processor

Yiqi Liu
yiqiliu2@illinois.edu
UIUC

Yuqi Xue
yuqixue2@illinois.edu
UIUC

Yu Cheng
yu.cheng@pku.edu.cn
Microsoft Research

Lingxiao Ma
lingxiao.ma@microsoft.com
Microsoft Research

Ziming Miao
ziming.miao@microsoft.com
Microsoft Research

Jilong Xue
jxue@microsoft.com
Microsoft Research

Jian Huang
jianh@illinois.edu
UIUC

Abstract

As AI chips incorporate numerous parallelized cores to scale deep learning (DL) computing, chips like Graphcore IPU enable high-bandwidth and low-latency inter-core links. They allow each core to directly access other cores' scratchpad memory, which enables new parallel computing paradigms. However, without proper support for the inter-core connections in current DL compilers, it is hard to exploit the benefits of this new architecture. We present T10, the first DL compiler to exploit the inter-core bandwidth and distributed on-chip memory on AI chips. To formulate the computation and communication patterns of tensor operators in the new architecture, T10 introduces a distributed tensor abstraction *r*Tensor. T10 maps a DNN model to execution plans with a compute-shift pattern, by partitioning DNN computation into sub-operators and mapping them to cores, so that the cores can exchange data following predictable patterns. T10 alleviates unnecessary inter-core communications, makes globally optimized trade-offs between on-chip memory usage and inter-core communication overhead, and selects the best execution plan from a vast optimization space.

1 Background and Motivation

To meet the ever-increasing compute demand of deep learning (DL) workloads, various AI chips or intelligence processors have been developed [4, 8, 9]. Typically, an AI chip employs numerous cores to provide high compute throughput. Each core has a small SRAM as its local scratchpad memory. To exploit the parallelism across cores, DL compilers partition the computation into multiple pieces. To synchronize data across cores, all cores share a global memory backed by a high-bandwidth off-chip memory (e.g., the HBM).

1.1 Inter-core Connected Intelligence Processor

Unfortunately, the global memory bandwidth growth gradually lags behind the fast growth of computing performance. Instead of fetching all data from the global memory, inter-core links allow cores to directly reuse data from each other, enabling more on-chip data reuse. For example, unlike the TPU [3] and GPU [9] architectures shown in Figure 1, the Graphcore IPU [5] enables a 624KB local memory on each core and allows each core to access another's local memory at 5.5GB/s. The 1,472 cores per chip yield a total on-chip capacity of 896MB and an all-to-all transfer bandwidth of 8TB/s, much higher than the HBM bandwidth (1.94TB/s on an A100 GPU). The aggregated capacity and bandwidth can further scale when future technology fits more cores into one chip, making it a promising approach for breaking the memory wall. Thus, inter-core links are employed in many emerging accelerators, including Graphcore IPU [5], SambaNova SN10 [10], Cerebras WSE [6], and Tenstorrent Grayskull [12].

1.2 Inefficiency of Existing Approaches

However, this new architecture makes executing DL models more complex. In the traditional *global shared-memory architecture*, programs access all data from a unified memory, so compilers only need to focus on partitioning computation among cores. In contrast, the on-chip inter-core connections enable a *distributed on-chip memory* architecture from programmers' perspective, which requires compilers to coordinate the computation partitioning, data placement, and inter-core communication. Simply employing existing compiler techniques causes unnecessary inter-core communication and data duplication in the precious on-chip memory.

To support inter-core connected AI chips, existing compilers [15, 16] and libraries [2] mimic a shared memory for all cores by reserving a portion of local memory in each core and abstracting them as a "virtual global memory" (VGM), as shown in Figure 2 (a). By default, all tensors used by the operators from a DL model, including persistent weights and temporary activations, are placed in the VGM. During execution, the active operator (i.e., currently running operator) is partitioned into sub-operators, each running on one core. Tensors of the idle operators (i.e., operators stored on-chip but not currently running) are unused in the VGM. To execute a sub-operator, each core retrieves data from VGM to its local "sub-operator" memory (shaded in green in Figure 2), performs computation locally, and stores the result back to VGM. We define this as a "*load-compute-store*" paradigm.

Inefficient inter-core communications. VGM introduces high inter-core communication overheads. First, accessing

The 2nd Workshop on Hot Topics in System Infrastructure (HotInfra'24), co-located with SOSP'24, November 3, 2024, Austin, TX, USA

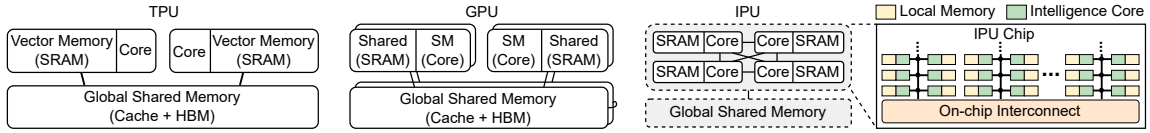
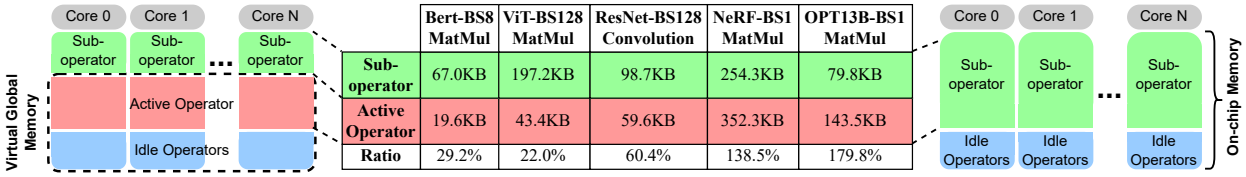


Figure 1. System architecture of TPU (left), GPU (middle), and IPU (right) chips. This paper focuses on a single IPU chip.



	Bert-BS8 MatMul	ViT-BS128 MatMul	ResNet-BS128 Convolution	NeRF-BS1 MatMul	OPT13B-BS1 MatMul
Sub-operator	67.0KB	197.2KB	98.7KB	254.3KB	79.8KB
Active Operator	19.6KB	43.4KB	59.6KB	352.3KB	143.5KB
Ratio	29.2%	22.0%	60.4%	138.5%	179.8%

(a) Load-Compute-Store (b) Memory Footprint Per-core when Using Load-Compute-Store (c) Compute-Shift

Figure 2. A comparison of the conventional load-compute-store (a) vs. our compute-shift (c) style execution. (b) shows the per-core memory footprint of representative operators when running DNN models on IPU using VGM. **Ratio** is the potential increase in sub-operator size by removing VGM. The result of OPT13B [14] comes from profiling one of its layers on IPU.

tensor data from VGM forces some cores to issue or serve more data accesses than others. As each tensor partition is stored in one core but is usually used by multiple cores for computation, some cores can obtain needed partitions from their local memory, while other cores must remotely fetch the partitions from peers. Thus, the execution is bottlenecked by cores that access remotely, which leads to *imbalanced memory accesses*. Second, to store a tensor using the VGM, as shown by the red "Active Operator" boxes in Figure 2 (a), we split it into small pieces across multiple cores. To retrieve a complete tensor, a core must fetch each piece from a different core, requiring it to communicate with multiple cores. This leads to *redundant inter-core communications*.

Inefficient use of on-chip memory. The VGM uses the on-chip memory capacity inefficiently. As shown in Figure 2 (a), each sub-operator of the currently active operator loads required data from the VGM to its local memory, which duplicates the data in both memory spaces. To host the duplicated data, VGM reserves memory space on each core, as shown by the active operator region in Figure 2 (a). This leaves less free on-chip memory, restricts each core to accommodate a smaller sub-operator, and results in *low compute intensity*. With less data reuse inside a core, higher data transfer volume is required for performing the same computation. We quantify the storage overhead of VGM for representative operators in Figure 2 (b). By removing the VGM (i.e., merging the active operator region into the sub-operator region) in Figure 2 (c), we can increase sub-operator size by 22%–180%.

2 Core Idea of T10

To eliminate the excessive memory footprint and redundant inter-core communications of VGM, we map the DNN computation to a *compute-shift* pattern. In each step, each core independently computes a sub-task with data received from its upstream neighbors and shifts the data to its downstream. The feasibility of this approach for general DNNs comes from this observation: most DNN operators can be divided into regular computation tasks, which load and produce consecutive data tiles of the input and output tensors, respectively.

We show an example that maps a MatMul operator to two cores in Figure 3 (a). We first partition it along dimension m onto two cores in Figure 3 (b). By default, both cores hold a copy of the weight tensor, incurring memory capacity overhead. To reduce memory footprint, in Figure 3 (c), we further split the weight tensor along dimension n into two parts and place each part on one of the cores. Then, the computation must be conducted in two steps, as each core holds half of the weight tensor and performs half of its computation per step. Between computation steps, each core circularly shifts its partition to the next core, forming a shift ring of two cores.

The compute-shift pattern avoids the inefficiencies of VGM. First, each part of the shared tensor is stored in at least one core at any time, which no longer needs a global memory. Second, by circularly shifting tensors across cores, the communication volume is evenly distributed across the inter-core connections. Third, each core only needs to communicate with one other core to shift each tensor at each step, which avoids redundant communications to many cores.

To find the best execution plan with the compute-shift pattern, we exploit *the tradeoff between memory footprint and communication overhead*. For example, both Figure 3 (b) and (c) show valid plans. Plan (b) finishes computation in one step without inter-core communication, but it has a higher memory footprint. Plan (c) has less memory footprint but incurs more communication overhead. Furthermore, given limited inter-core bandwidth and on-chip capacity, we must also holistically tradeoff among multiple operators on the chip to derive an optimized end-to-end execution plan.

3 System Design of T10

We introduce T10, a compiler that optimizes end-to-end DNN model execution on an inter-core connected DL processor. We present the design overview of T10 in Figure 4.

3.1 *r*Tensor: A New Tensor Abstraction

To map each tensor onto the distributed on-chip memory and shift the partitioned sub-tensors across cores, T10 introduces a tensor abstraction called RotatingTensor (*r*Tensor). Besides

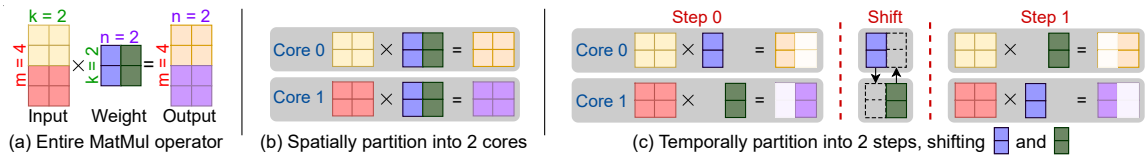


Figure 3. An example that maps a MatMul operator to two cores with the compute-shift style execution. Both (b) and (c) are valid compute-shift execution plans, but with different tradeoffs between memory footprint and communication overhead.

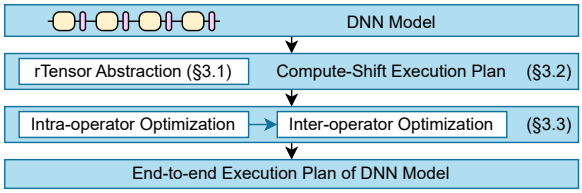


Figure 4. System overview of T10.

tensor shape and data type, *r*Tensor also defines how each tensor is partitioned, mapped, and shifted on multiple cores.

First, T10 partitions the computation of an operator onto multiple cores. Based on data dependency, the computation partitioning will imply how each of its input/output tensors is partitioned. This gives a *spatial partition factor* (f_s), which splits a tensor into ***sub-tensors***. Second, each sub-tensor may be shared by many cores, so we specify how the sub-tensor is further partitioned among the cores using a *temporal partition factor* (f_t). Third, we specify how the partitions of a sub-tensor are circularly shifted among the cores using the *rotating pace* (***rp***), so we can align the data shifting with computation. Altogether, a set of *r*Tensors of an operator defines a compute-shift execution plan. The numerous possible *r*Tensor configurations of an operator generate a combinatorial optimization space of execution plans.

3.2 *Compute-Shift* Execution Plan

Using *r*Tensor, T10 organizes the computation of a DNN operator into a compute-shift pattern, where the operator's computation and tensors are partitioned to individual cores and their local memories. The entire computation involves multiple compute-shift steps until each tensor has been shifted across all cores. Each compute step is defined as a ***sub-task***. In each compute-shift step, each core computes a sub-task and shifts local tensors to its neighbors.

To represent an operator's computation, T10 uses tensor expression [1, 11, 13, 15, 16] to define how each output tensor value is computed from the input values. For example, a matrix multiplication of tensors A and B into C is defined as

$$C[m, n] \mathrel{+}= A[m, k] * B[k, n], \tag{1}$$

where m, n, and k are axes to index the elements in each tensor. Equation (1) indicates that any value in C indexed by m and n (i.e., $C[m, n]$) is computed by summing $A[m, k] * B[k, n]$ over all possible indices k. T10 supports all common operators that can be represented by tensor expressions.

To map an operator to interconnected cores, T10 first partitions it into parallel ***sub-operators*** along all axes in its tensor expression, using an *operator partition factor* ($\boldsymbol{F_{op}}$). T10 then uses F_{op} to derive the f_s for each tensor, following the data dependencies in tensor expression. If a tensor's dimensions do not include some axis in F_{op}, each sliced sub-tensor is required by multiple sub-operators along the missing axis. After the spatial factor determines the number of cores that will share a sub-tensor, the temporal factor can specify how the sub-tensor is split and rotated across these cores.

For each operator, each set of factors can map its computation into a compute-shift execution plan. In each step, each sub-operator computes a sub-task partitioned by F_{op} and *rp* along each axis. The sub-operator iterates all its sub-tasks by nested-looping through all rotating axes. Between sub-tasks, an *r*Tensor is rotated along the currently iterating axis.

3.3 Intra-operator and Inter-operator Trade-off

For each operator, there could be a vast number of execution plans involving different spatial and temporal partition factors and rotating paces. Moreover, an end-to-end model consists of numerous operators, creating a substantial combinatorial optimization space. T10 defines a *two-level trade-off space between execution time and memory consumption.*

Intra-operator trade-off. When determining each operator's execution plan, we can trade memory space for execution efficiency by specifying a smaller temporal partitioning factor. This can reduce communication costs by reducing the hops in the rotation loop, while at the cost of using more memory to hold duplicated tensors. To efficiently tradeoff among numerous plans, T10 enables a cost model to predict each plan's execution time. Thus, T10 can quickly examine each plan and choose the best candidates that sit on the Pareto optimal trade-off curve, where each plan either runs faster than any other plans with the same or less memory footprint or uses less memory than any others with the same or lower execution time. T10 also employs search constraints to reduce the number of plans to be examined.

Inter-operator trade-off. We can tradeoff between memory space and execution time across all operators holistically when deciding the end-to-end model execution plan. To execute a DNN, T10 fits multiple operators in the on-chip memory, so adjacent operators can reuse intermediate data on chip. To start tradeoff, T10 assumes that all operators execute using the plans with the minimum memory footprint. As different operators have different memory-latency trade-offs, T10 iteratively allocates more memory to the operator with the highest memory-cost efficiency (i.e., reducing the most execution time while incurring the same memory footprint overhead), in each step. The tradeoff stops when all on-chip

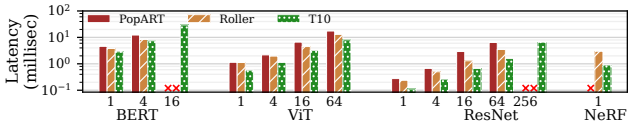


Figure 5. Inference latency of DNN models for various batch sizes. "✖" indicates the program cannot fit into an IPU chip.

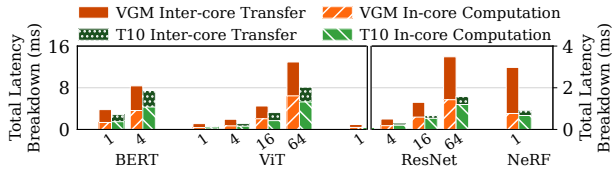


Figure 6. Data transfer overhead of executing various DNN models with different batch sizes on IPU.

memories are allocated to the operators, where T10 obtains an end-to-end plan with optimized total execution time.

4 Performance of T10

We use T10 to generate end-to-end execution plans for DNNs of different types and sizes, including CNNs (ResNet), Transformers (BERT and ViT), and fully-connected networks (NeRF). We execute the models on a Graphcore IPU MK2 chip [5], and compare T10 with Graphcore's official PopART library [2] and a VGM-based DL compiler, `Roller` [16]. More detailed analyses are presented in our full paper[1] [7].

End-to-end performance. In Figure 5, while Roller outperforms PopART by 1.38× on average by using appropriate sub-operator sizes for each operator, T10 further achieves 1.69× end-to-end inference latency improvement on average than Roller by globally optimizing the compute-shift execution. Also, T10 supports larger batch sizes and models by saving the on-chip memory space, while other baselines fail to execute the models as the batch size gets larger.

Inference latency breakdown. We break down the compute time and inter-core data transfer time of Roller and T10 in Figure 6. Compared with Roller, T10 reduces the inter-core communication overhead from 50%–74% to only 8%–43%. For most models, while large operators like MatMul intrinsically incur significant inter-core data sharing, T10 reduces the overhead by smartly shifting the shared tensor data. For ResNet and NeRF, T10 minimizes the inter-core movements of their large input activation tensors, by sharing the smaller convolution kernels or model weights across the cores, enabling even lower communication overhead.

T10 compilation time. Figure 7 shows the compilation time of T10 for different models and batch sizes. T10 finishes compilation in a few hours for most DNN inference programs (tested with AMD Ryzen 7950X3D). As T10 exploits the predictability of the hardware architecture with the cost model and search constraints (§3.3), it avoids the expensive profiling for tuning each operator (e.g., Ansor [15]).

[1]Our full paper will appear in the proceedings of the 30th Symposium on Operating Systems Principles (SOSP'24).

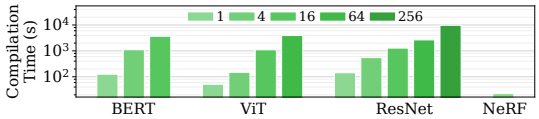


Figure 7. T10 compilation time of different batch sizes.

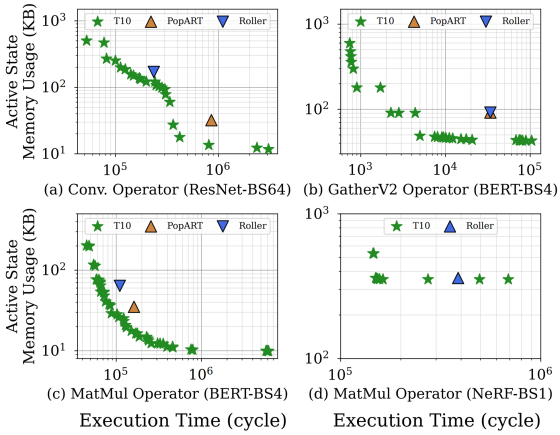


Figure 8. Candidate execution plans of representative operators (e.g., Figure (a) is a convolution operator from ResNet with batch size 32). Stars are optimal execution plans found by T10. Triangles are the plans used by PopART and Roller, respectively. PopART fails to execute NeRF in Figure (d).

Intra-operator trade-off space. Figure 8 shows the set of optimal execution plans found by T10 for representative operators compared to the plans used by PopART and Roller. For most operators, T10 finds a plan that is both faster and more memory efficient than PopART. Roller always tries to find the fastest plan that uses the most per-core memory. However, Roller's maximum memory usage is limited due to the VGM region (Figure 2). By removing VGM, T10 allows larger active memories and faster plans. By maintaining a set of Pareto-optimal plans for each operator, T10 can trade-off among all operators for the best end-to-end performance.

5 Conclusion and Future Work

We present T10, an end-to-end deep learning compiler for inter-core connected intelligence processors. To best utilize the IPU hardware, T10 enables the compute-shift computing paradigm, the tradeoff between memory and communication, and the cost-aware operator partitioning and scheduling. As future work, we will extend T10 to optimize the IPU with off-chip HBM, which requires us to mitigate new hardware resource contentions on the memory and interconnect.

Acknowledgments

We thank the members in the Systems Platform Research Group at University of Illinois Urbana-Champaign for proofreading our paper. They include Benjamin Reidys, Haoyang Zhang, Shaobo Li, Daixuan Li, Eric Zhou, and Noelle Crawford. We thank Chen Jin, Chengshun Xia, and Han Zhao for the technical support on Graphcore IPU. This work was partially supported by NSF CAREER CNS-2144796.

References

[1] 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. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*.

[2] Graphcore. 2022. PopART User Guide. https://docs.graphcore.ai/projects/popart-user-guide/en/latest/intro.html.

[3] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten lessons from three generations shaped Google's TPUv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*.

[4] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* (2020).

[5] Simon Knowles. 2021. Graphcore Colossus Mk2 IPU. In *2021 IEEE Hot Chips 33 Symposium (HCS '21)*.

[6] Sean Lie. 2021. Multi-Million Core, Multi-Wafer AI Cluster. In *2021 IEEE Hot Chips 33 Symposium (HCS '21)*.

[7] Yiqi Liu, Yuqi Xue, Yu Cheng, Lingxiao Ma, Ziming Miao, Jilong Xue, and Jian Huang. 2024. Scaling Deep Learning Computation over the Inter-Core Connected Intelligence Processor with T10. *arXiv preprint arXiv:2408.04808* (2024).

[8] Nvidia. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper.

[9] NVIDIA. 2022. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[10] Raghu Prabhakar and Sumti Jairath. 2021. SambaNova SN10 RDU: Accelerating Software 2.0 with Dataflow. In *2021 IEEE Hot Chips 33 Symposium (HCS '21)*.

[11] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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)*.

[12] Tenstorrent. 2023. Meet Grayskull. https://tenstorrent.com/grayskull/.

[13] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[14] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).

[15] 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. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*.

[16] 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, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*.