AutoGTCO: Graph and Tensor Co-Optimize for Image Recognition with Transformers on GPU

Yang Bai[1], Xufeng Yao[2], Qi Sun[1], Bei Yu[1]
[1]The Chinese University of Hong Kong [2]SmartMore
{ybai,qsun,byu}@cse.cuhk.edu.hk,xufeng.yao@smartmore.com

***Abstract*—Performance optimization is the art of continuously seeking an effective mapping between algorithm and hardware. Existing deep learning compilers or frameworks optimize the computation graph via adapting transformations manually designed by expert efforts. We argue that these methods ignore some possible graph-level optimizations, thus it is difficult to generalize to emerging deep learning models or new operators. In this work, we propose AutoGTCO, a tensor program generation system for vision tasks with the transformer architecture on GPU. Compared with existing fusion strategies, AutoGTCO explores the optimization of operator fusion in the transformer model through a novel dynamic programming algorithm. Specifically, to construct an effective search space of the sampled programs, new sketch generation rules and a search policy are proposed for the batch matrix multiplication and softmax operators in each subgraph, which are capable of fusing them into large computation units, it can then map and transform them into efficient CUDA kernels. Overall, our evaluation on three real-world transformer-based vision tasks shows that AutoGTCO improves the execution performance relative to deep learning engine TensorRT by up to 1.38×.**

I. Introduction

Recent years have witnessed a surge of industry-scale application of deep learning models, ranging from autonomous driving, augmented reality, human pose estimation, language translation, to billion-scale search and recommendation systems. In computer vision, the most popular line of work is based on convolutional neural networks, which has led to a plethora of methods for CNN-based network architectures [1]–[4]. Despite its huge success, increasing attention has been paid to combining CNN-based architectures with self-attention mechanisms [5]. The design ethos is inspired by the successful application of transformer-based architecture on neural language processing. Many methods have successfully replaced the CNN with a standard transformer entirely [6]. The great successes of using transformer have been proved in various tasks [7]–[10].

The deep learning models can be expressed as a directed acyclic computation graph (DAG), in which nodes verbalize the operators and edges represent the relationship between adjacent operators. These computation graphs are mapped to hardware accelerators (*e.g.*, GPUs [11], [12], FPGAs [13]–[15], ASICs [16]) through existing deep learning frameworks (*e.g.*, TensorFlow [17], PyTorch [18], Caffe [19]) by vendor-provided kernel libraries (*e.g.*, cuDNN [20], MKL-DNN [21]) to achieve high performance computing. These libraries require significant engineering effort to manually tune for different operators on a diversity of hardware platforms.

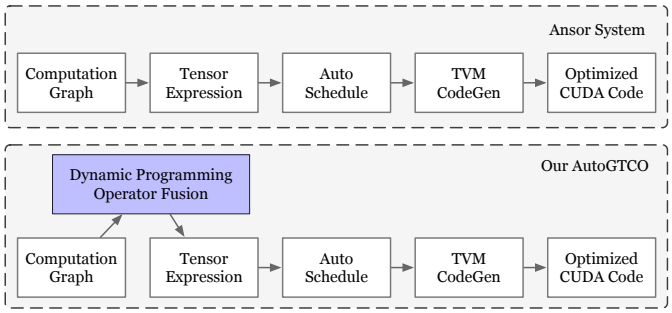

Fig. 1 The system overview of Ansor [22] and AutoGTCO.

While previous arts focus on optimizing CNN-based models, the transformer-based vision models have not yet been scaled effectively on accelerators due to the use of specialized attention mechanisms. Though the existing deep learning libraries (*e.g.*, cuDNN/cuBLAS) are capable of fusing the element-wise layers into large compute-intensive kernels, these techniques do not work well for workloads that are dominated by memory-intensive structures (*e.g.*, transformer). In addition, the transformer-based vision models have a large number of fine-grained operators, causing notable runtime launch overheads when being executed on GPU platforms. For these reasons, traditional optimization methods, (*e.g*, Halide [23]) optimizing compute-intensive operators alone, fail to unlock the full potential of execution efficiency of transformer models.

To alleviate these burdens of optimizing, some works propose to optimize model from graph-level. TensorRT [24] uses two-steps fusion strategy to optimize the computation graph. Some operators (*e.g.,* convolution, batch normalization [25], and ReLU [26]) are fused vertically via the specific rules in the first step. In the second step, all of the operators in the same stage are fused horizontally. This optimization can achieve good performance for CNN-based architecture with multiple branches. Some popular frameworks, *e.g.*, TensorFlow, PyTorch, TVM [27], and Ansor [22], optimize the computation graph by performing greedy rule-based subgraph substitutions. Although manually designed substitutions improve the performance, they fall short in maintenance respect. The maintenance problem is aggravated by the fact that new operators or models are continuously introduced because hand-written graph substitution rules require significant engi-

neering effort. To find performant graph substitutions, search-based approaches are proposed by TASO [28] and IOS [29] to explore a large search space to cover the useful graph optimizations. TASO automatically generates graph substitutions and employs formal verification to ensure the correctness of the generated graph substitutions. IOS combines intra-operator and inter-operator parallelism to find an efficient schedule that better utilizes the hardware accelerator. However, these approaches fail to capture effective code generation techniques, since they rely on deep learning library cuDNN to implement each kernel in their runtime engine, which prevents them from covering a comprehensive search space for operator optimization.

To address these challenges, we propose AutoGTCO, a framework based on a novel dynamic programming algorithm to explore operator fusion strategy automatically for generating high-performance tensor programs on GPUs. Our work is based on Ansor, shown in Fig. 1. The dynamic programming algorithm is to find possible fusion relationships between adjacent operators in the transformer model. After getting different kinds of subgraphs, AutoGTCO can dynamically prioritize subgraphs of transformer that are more likely to improve the end-to-end performance. Then AutoGTCO employs a hierarchical representation to cover a large search space, which is created automatically for a given subgraph. Complete tensor programs are sampled from the search space and a learned cost model is introduced to fine-tune these programs. In summary, our paper makes the following contributions:

- We introduce a novel dynamic programming algorithm to solve the operator fusion problem for transformer models. This technique can automatically generate the optimal combination for adjacent operators in the graph-level, which explores a large combination space than the rule-based method in other libraries.
- We propose new sketch generation rules and a search policy for the batch matrix multiplication and softmax operators in subgraphs. This mechanism constructs an effective search space for the kernel generation. To get high-performance and end-to-end compilation flow, a learned cost model is used to fine-tune the performance of each kernel.
- We apply our method to currently popular image recognition tasks with transformer models including DETR [7], SETR [9], and ViT [10]. Our method can automatically generate corresponding CUDA code on GPU under different inference configurations. Empirical studies show the superiority of our optimized CUDA code which outperforms TensorRT with 1.01 to 1.38× measured speedup in the inference stage.

II. Related Work and Background

A. Image Recognition

Image recognition has been significantly boosted with the development of deep neural networks. There are three particularly important tasks in image recognition: image classification, object detection, and semantic segmentation. Image classification [30] can classify what is contained in an image. Object detection [31] is a combination of image location and classification. Image localization specifies the location of a single object in an image whereas object detection specifies the location of multiple objects with their labels in the image. Image segmentation [32] creates a pixel-wise mask of each object in the image.

B. Transformers

The transformer model, originally developed for a new attention-based building block for machine translation. Transformer models make two main contributions. First, it popularizes attention mechanisms to a particular module named multi-head attention (MHA) [9]. Second, it does not rely on recurrent or convolutional algorithms. Besides, transformer models contain many similar subgraph structures which can be executed in parallel with the same configuration.

Encoder. The encoder module is composed of a stack of attention-based layers with identical structures. There are two sub-layers in each layer. The first sub-layer is a multi-head attention mechanism, and the second sub-layer is a simple, position-wise fully dense layer. A residual connection layer is used between each of the two sub-layer with a layer normalization.

Decoder. The structure of the decoder is akin to the encoder. In addition to the two sub-layers mentioned in the encoder module, a third sub-layer is inserted into the decoder module. The function of it is to perform multi-head attention over the output of the encoder part. Besides, some modifications about masking mechanisms in the self-attention sub-layer are to prevent positions from attending to subsequent positions.

Multi-Head Attention. Multi-head Attention generalizes attention mechanisms and employs *h* attention heads parallelly to get different learned projections of a given sequence. Each attention head is an instance of scaled dot-product attention, and takes queries (q), keys (k), values (v) as its input. The function of attention is to find values corresponding to the keys closest to the input queries. The functions of heads are also augmented with linear layers that project their inputs into a lower-dimensional space. The three inputs are first multiplied by weight tensors *wq*, *wk*, *wv*, respectively, as a learned input projection. The query and key tensors are subsequently multiplied together and scaled, followed by applying the softmax operation to weight and select the most relevant results. This is then multiplied with *vv* to produce the per-head output. The outputs of all the heads are finally concatenated and linearly projected back to the input dimensional size, as depicted in Figure 2.

C. Deep Learning Compiler

Ansor [22] is a framework for automated tensor program generation, which is equipped with a hierarchical search space that decouples high-level structures and low-level details. Ansor constructs the search space for a computation graph automatically, eliminating the need for manually developing high-performance computing templates by experienced

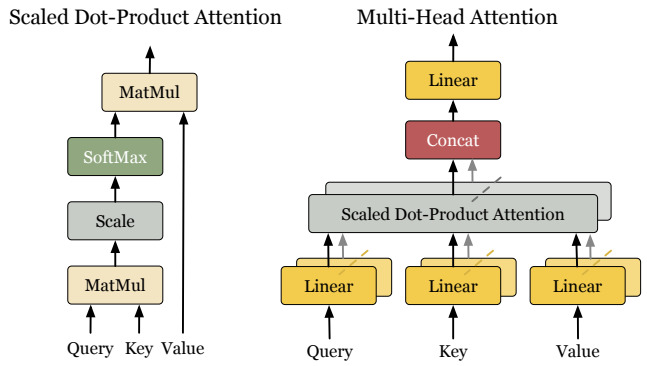

Fig. 2 Scaled Dot-Product and Multi-Head Attention (MHA).

engineers. Then, it uses an auto-tuner to sample complete programs from the search space and implements fine-tuning on complete programs under the XGBoost cost model [33]. The brief flow of Ansor is shown in Fig. 1. Tensor Comprehensions (TC) [34] has its unique design, which combines the Halide and polyhedral model [35]. It uses Halide-based IR to represent the computation and adopts the polyhedral-based IR to represent the loop structures.

D. Hierarchy of 2080 Ti GPUs

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model for GPUs, which exposes high-performance computing programmers to the concepts of memory hierarchy and threads hierarchy. Under the hood, accelerating deep learning models on complex memory hierarchy needs to make good use of memory units and computation units. As shown in Fig. 3, there are many programmable units at different levels of GPU devices. RTX 2080 Ti GPU follows the Turing microarchitecture and contains 68 parallel streaming multiprocessors (SMs). SM is partitioned into 4 processing blocks. Each processing block possesses a 64 KB register file, and the 4 processing blocks share a combined 96 KB shared memory. And each thread block contains a group of threads that can execute the same code on different data, following the Single Instruction Multiple Thread (SIMT) mechanism.

III. Problem Formulation

Definition 1 (Computation Graph)**.** *A transformer model is defined by a computation graph $G = (V, E)$, where V is the set of vertices and E is the edge set. Each vertex can represent an operator such as GEMM and softmax operation in the computation graph. Each edge $(u, v) \in E$ is to describe the dependencies between node u and v.*

Operator Pattern. Operator fusion combines multiple adjacent operators into a single kernel rather than storing the intermediate results into the global memory for data movement. This optimization can greatly improve performance, particularly in throughput oriented architectures such as GPUs. In order to combine operators efficiently, we define the pattern of each operator in the computation graph. In Ansor, it has five patterns for each operator: (1) *injective*, (2) *reduction*, (3) *complex-out-fusable*, (4) *element-wise*, and (5) *opaque*. And it provides generic rules to fuse these operators [27]. From the Section II, we know that transpose of a matrix, batch matrix multiplication, layer normalization, softmax, and dense layer occur frequently in the transformer computation graph. In the meantime, the default configuration of these operators is defined as follow: Softmax is tagged as the opaque pattern. Batch matrix multiplication and dense are tagged as the complex-out-fusable pattern. Layer normalization [36] can be decomposed to be a set of add/ subtract/ multiple operators which are all tagged as the element-wise pattern.

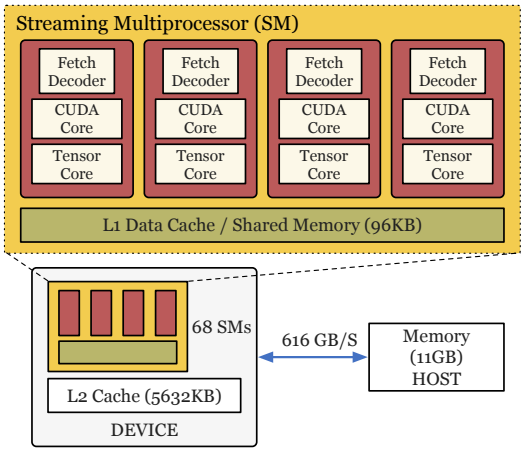

Fig. 3 A streaming multiprocessor and the memory architecture of GeForce RTX 2080 Ti GPU.

Fusion Strategy and Schedule. We define a schedule S of a computation graph G as follow:

$$S = \{(V_1, F_1), (V_2, F_2), ..., (V_k, F_k)\}, \tag{1}$$

where V_i represents a group of operators in the i-th phase and F_i is a pair to describe the fusion relationship between two nodes. Finally, computation graph can be executed under the schedule S from the first phase (V_1, F_1) to the last phase (V_k, F_k) consecutively.

Problem 1. *Given a computation graph G and fusion schedule S on GPU, our goal is to search for a schedule S^*:*

$$S^* = \underset{S}{\operatorname{argmin}} \ Cost(G, S), \tag{2}$$

where $Cost$ is the latency of executing G according to the schedule S.

IV. Details of AutoGTCO

A. Overview

Our tensor generation framework is shown in Fig. 4, composed of four important modules, *i.e.*, dynamic programming-based operator fusion (DPOF), subgraph scheduler, CUDA program sampler, and performance tuner. The input of DPOF is a transformer model without any operator fusion. Each operator is tagged with a type label to determine whether it is fused with other adjacent operators or not. After running through the DPOF module, each operator is given a new tag, and adjacent operators are possibly merged into subgraphs

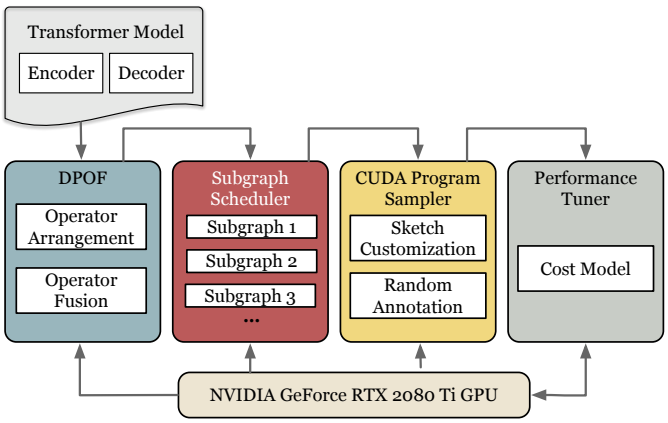

Fig. 4 Overview of our system. The arrows show the flow of the optimized subgraphs from transformer model and tensor programs generation on GPU platform.

according to the relationship of the predicted tags. It then generates the high-performance tensor programs for each subgraph. Our framework is summarized as follows:

- A DPOF that finds an optimized operator fusion schedule for the transformer model.
- A subgraph scheduler that allocates time resources for optimizing multiple subgraphs generated by the DPOF.
- A program sampler that delineates a large search space and randomly samples various programs from it.
- A performance tuner that trains a cost model to measure the performance of sampled tensor programs.

B. Dynamic Programming-based Operator Fusion (DPOF)

Operator Arrangement. To find an optimized schedule for a transformer model, we first use topological sort to obtain the execution order of the computation operators. Second, we build a computation Queue to store these operators. It is convenient for us to find a good schedule based on the Queue rather than the graph data structure. For the placeholder variables, we do not consider them because they only store the input and output and do not mitigate the performance of the whole graph. As mentioned in the section (III), each operator has its own type and the same operator with different types make the difference in the inference stage. Third, we set all of the operators as opaque type and assume that there is no fusion relationship between them. The size of the Queue is the maximum number of phase in the scheduling algorithm which is defined in the section (III).

Operator Fusion. Getting the execution order of the computation operators and the maximum number phase of our schedule, we partition the original computation graph $G = (V, E)$ into $V - V^{'}$ and $V^{'}$. The edges in set of $V - V^{'}$ have the pointing relationship with the edges in set of V'. That is, all of the edges start from $V - V^{'}$ and end up with V'. We call V' the segmentation set. The relationship between the $V^{'}$ and $V - V^{'}$ is illustrated in Fig. 5. Following the dynamic programming, we can enumerate the segmentation sets V' of V and convert the problem into a sub-problem that attains the

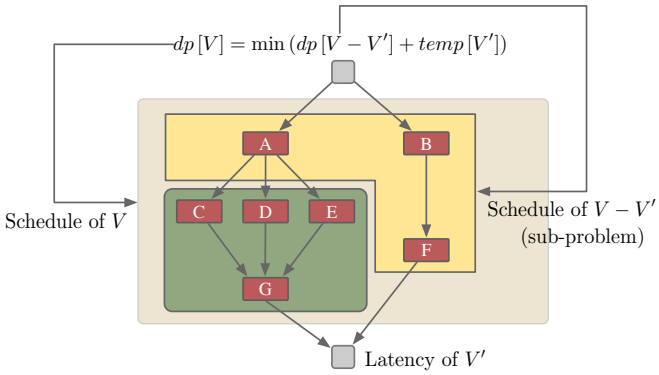

Fig. 5 Dynamic Programming Operator Fusion.

optimal schedule for $V - V^{'}$. Therefore, the whole graph can be solved by employing the segmentation set recursively.

We define $dp[V]$ as the latency of the graph with the node set V under an optimal schedule S. And then we define $temp[V^{'}]$ as the latency of phase $(V^{'}, F)$. F is the better fusion strategy for the segmentation set $V^{'}$. Consequently, we can get the state transition equation as follow:

$$dp[V] = \min_{v \in V^{'}}(dp[V - V^{'}] + \sum_{v} temp[v]), \tag{3}$$

where v is the node in segmentation set $V^{'}$ and the boundary value of the state transition equation is $dp[\varnothing] = 0$. In order to get the optimal schedule, we store the each node v in a segmentaion set $V^{'}$ that can make the latency of each V in $action[V]$. Following the above information, we implement the operator fusion scheduling as shown in Algorithm 1.

C. Subgraph Scheduler

A transformer model can be partitioned into kinds of independent subgraphs. In the process of optimization, it is meaningless to spend much time to tune some subgraphs that can not improve the execution performance. There are two reasons: (1) the subgraph is not a performance bottleneck, or (2) spending much time tuning only brings minimal improvement. Therefore, we should dynamically allocate different amounts of time resources to different kinds of subgraphs. It means that obtaining a well-optimized transformer needs completing lots of scheduling tasks and we define a scheduler to optimize this part.

When tuning a set of subgraphs, we combine three types of goals: (1) reducing the whole latency of transformer, (2) meeting latency requirements for a set of subgraphs, or (3) minimizing tuning time when tuning no longer improves the performance of transformer significantly. We define t as the allocation vector, where t_i is the number of time units spent on i-th task and the initial value of t is $(1, 1, ..., 1)$. We also define $g_i(t)$ as the minimum subgraph latency under the i-th task with t_i time units. Therefore, $f(g_1(t), g_2(t), ..., g_n(t))$ can describe the end-to-end latency of the model and our goal is to minimize this function. We define the following objective

Algorithm 1 Operator Fusion Scheduling

Input: A computation graph $G = (V, E)$ with the **opaque** type for $\forall v \in V, pattern(v) = 0$;
Output: A operator fusion strategy with the type of each operator $v \in V, pattern(v)$;

Defining $dp[\varnothing] \leftarrow 0$, $dp[V] \leftarrow +\infty$, $action[V] \leftarrow \varnothing$;
Defining $S \leftarrow [\varnothing]$ (A Stack data structure to store the phase of optimial schedule for operator fusion);

function `SelectSchedule`(G)
  V = all operators in computation graph G;
  `Scheduler`(V);
  while $V \neq \varnothing$ **do**
    $V^{'}, F = action[V]$;
    Put phase $(V^{'}, F)$ into the stack S;
    $V = V - V^{'}$
  end while
  return the Fusion Strategy S
end function

function `Scheduler`(V)
  if $dp[V] \neq +\infty$ **then**
    return dp[V]
  end if
  for all $v \in V^{'}$ **do**
    $T_{V'}, F_{V'} =$ `PhasePartition`$(V^{'})$
    $T_V =$ `Scheduler`$(V - V^{'}) + \sum_{v_i \in V'} T_{V'}$
    if $T_V \leq dp[V]$ **then**
      $dp[V] = T_V$
      $action[V] = (V^{'}, F_{V'})$
    end if
  end for;
  return $dp[V]$
end function

function `PhasePartition`($V^{'}$)
  for all operators $v_i \in V^{'}$ **do**
    if $pattern(v_i, v_j) \neq opaque$ **then**
      $T_{\text{fused}(i,j)} = \text{Runtime}(pair(v_i, v_j))$
    else
      $T_{\text{fused}(i,j)} = +\infty$
    end if
  end for
  return $T_{\text{fused}(i,j)}, pattern(v_i, v_j)$
end function

function:

$$f = \max\left[\sum_{i=1}^{n} w_i \times \max(g_i(t), ES(g_i, t)), L_j\right]. \quad (4)$$

In the above function, w_i is the number of appearances of task i. We define L_j as the latency requirement of subgraph j, meaning that we do not want to spend tuning time on a subgraph if its latency has already met the requirement. In order to achieve the effect of early stopping, we also define a function named $ES(g_i, t)$ by looking at the history of latency of i-th task. Unlike the objective functions defined in Ansor, we first make a comparison between meeting the requirement and early stopping, and then optimize each task sequentially. Finally, we use a scheduling algorithm based on gradient descent to efficiently optimize the objective function.

D. Program Sampler

To sample the tensor program effectively, we also define a hierarchical search space with two levels like Anosr: **sketch** and **annotation**. We define the high-level structures of tensor programs as our sketches and set millions of low-level choices (*e.g.*, blocking size, virtual thread tiling, cooperative fetching) as our annotations. The generated tensor program is composed of two levels. The component of the first level are sketches generated by the derivation rules which are designed specifically on the GPU platforms. The detailed information of the second level is randomly annotated from the annotation space. To generate sketches for each subgraph, we visit all of he computation nodes in a topological order and iteratively build a generation structure. For compute-intensive or the nodes with a high chance of data reuse, we build a classic tile and fusion structures for them as the sketch. It is worth noting that some new nodes for caching will also be introduced to increase memory usage during the generation of sketch.

Fig. 6 (left) shows an example of the generated sketches for a subgraph that contains matrix multiplication ($[1050, 8, 32] \times [32, 8, 1050]$) and softmax operators in MHA. For the example subgraph, the sorted order of the five nodes in the DAG is $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{M}, \boldsymbol{S}, \boldsymbol{D})$. To get the sketches for the subgraph, we start from the output node **D** and apply the generation rules to the computation node one by one. We can get the generated sketch 1 by Anosr. From the generated sketch 1, we find that the matrix multiplication and softmax operators are performed separately, and they are not integrated into a computation kernel. In order to optimize multiple operators as a computation kernel to unlock the full potential of execution efficiency, we design new derivation rules for batch matrix multiplication and softmax operators in transformer architecture. We integrate them seamlessly with existing rules to optimize the execution performance for tensor programs.

Sketch Customization. The default sketch configuration of Anosr for the multi-level tiling structure to match the GPU backend is "SSSRRSRS". The first three "S" corresponds to BlockIdx, Virtual thread, and ThreadIdx in the GPU programming model, respectively. The "SSSRRSRS" tile structure for matrix multiplication expands the original 3-level for-loop into a 19-level for-loop. Even if we do not enumerate the loop order, this multi-level tiling structure can take loop order into consideration. As shown in Fig. 6, we design an effective operator fusion strategy for the sketch generation. The "SSSR-RSRS" tile structure is for batch matrix multiplication and softmax operators. To fuse more operators and make full use of computation resources on GPUs, we insert a caching node with the "SS-S" tile structure to store the sketch generation of the matrix multiplication in it, and then send the result to the sketch generation of softmax to get the final sketch of the subgraph. From Fig. 6, we can find that the generation sketch of matrix multiplication and softmax operators are fused into the same computation kernel.

Random Annotation. The sketches generated by our customization are incomplete programs because they only have thread parallel structures without the specific value. Therefore, we should turn them into complete programs and then evaluate the performance of tensor programs. We randomly pick one sketch from a list of generated sketches by our customizations. For the outer loops, we use parallelize intrinstics to generate

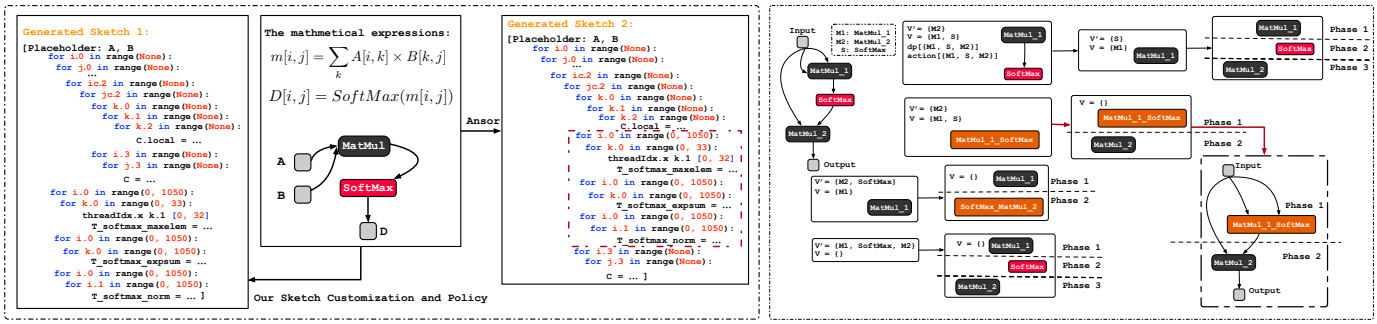

Fig. 6 An example to illustrate how to do operator fusion in MHA (right) and sketch generation for the subgraph (left).

complete tensor programs and optimize them. And for the inner loops, we use vectorize and unroll intrinstics to optimize them. All valid parameters for the random values are sampled from a uniform distribution.

Performance Tuner. Auto-tuning [37] is to find the best schedule of the tensor programs from a search space. Therefore, a cost model is particularly important in the process of evaluating the performance. The extracted features include arithmetic and memory access features, which includes the number of float operations, the number of integer operations, vectorization related features, unrolling related features, parallelization related features, GPU thread binding related features, buffer access feature, and allocation related feature. More specifically, we use weighted squared error as the loss function, and the loss function of the model f on a sampled program P with throughput y can be defined as follow:

$$loss(f, P, y) = w_p \left[\sum_{s \in S(P)} f(s) - y \right]^2 , \qquad (5)$$

where $S(P)$ is the set of innermost non-loop statements in P and we train a gradient boosting decision tree as the underlying model f. In the actual calculation, we directly make y approximately equal to w.

A search policy is also necessary for the performance tuner. The evolutionary search leverages mutation and crossover mechanism to generate a new set of candidates repeatedly for several rounds and outputs a small set of programs with the highest scores. The generated programs will be compiled and measured on the GPU to obtain the real runing time. In the meantime, the collected data from the training is then used to improve the performance of the cost model. Therefore, we adopt the search policy by designing corresponding evolution operations to rewrite and fine-tune the sampled programs.

V. Evaluation Results

A. Experimental Setup

We evaluate the fusion optimization and kernel generation mechanisms on three modern vision tasks with transformers: DETR [7] for object detection, SETR [9] for semantic segmentation, and ViT [10] for image classification. We use PyTorch 1.7.1, cuDNN V7.6.5, CUDA 10.0, NVIDIA driver 460.67, and adopt TensorRT V7.0.0.11 [24] and TVM 0.8.dev0 [27] as baselines for comparisons. All evaluation results are collected on a NVIDIA GeForce RTX 2080Ti GPU.

Workflow. The pipeline of our workflow can be summarized in the following two patterns: 1) For high-performance computing library TensorRT, we first use PyTorch to build models with different parameters, and then use the ONNX export interface to obtain ONNX model. We use ONNX-Simplifier to simplify the ONNX model, and then convert it into an executable engine defined in the TensorRT environment; 2) In terms of the compilation flow like Ansor and AutoGTCO, we first compile the model into the TorchScript format, and then use the PyTorch interface defined in Relay to read it. For the subgraphs, the corresponding tensor programs are generated by TVM code generation.

Workloads. Details of the models tested in this work are listed in TABLE I, including the number of encoders, the number of decoders, *etc*. We report all of experiment results for batch size 1.

B. End-to-End Performance

Baselines and Configurations. We use PyTorch JIT [18], TensorRT [24], TVM [27], and Ansor [22] as baseline frameworks. Generally, there are two ways to improve the runtime speedup on GPUs. One way is to optimize operators by the vendor-provided library cuDNN/cuBLAS such as PyTorch and TensorRT. Another strategy is to use machine learning algorithm to search the schedule of tensor program for each computation kernel such as TVM and Ansor. Like the definition in Section IV, we expect that the end-to-end execution time can be represented as the sum of the latency of all subgraphs in the computation graph. We let Ansor and AutoGTCO run auto-tuning 10000 measurement trails unless runtime converges to a stable value. The objective of the subgraph scheduler is set to minimize the total execution time of the model and then generate efficient tensor programs.

Results.TABLE II shows the results. Compared with vendor-specific high-performance computing library TensorRT, AutoGTCO consistently outperforms all benchmark models except for ViT-Base vision model with 1.01 to 1.38× speedup. The reason for the drop in performance on ViT-Base is that ViT-Base is composed of a small number of encoder layers and the input shape $(197, 1, 768)$ of the encoder in ViT-Base is relatively limited compared with ViT-Large and ViT-

TABLE I Architecture of the Benchmark Model and Configurations of all Experiments

model	ec	dc	width	mlp-dim	nh	input shape	patch	mha input	encoder input	decoder input	Params
DETR-ResNet50-E3	3	6	256	2048	8	[1,3,800,1333]	N/A	query[1050,1,256] key[1050,1,256] value[1050,1,256]	src[1050,1,256]	tgt[100,1,256] mem[1050,1,256]	37.40M
DETR-ResNet50-E6	6	6	256	2048	8	[1,3,800,1333]	N/A	query[1050,1,256] key[1050,1,256] value[1050,1,256]	src[1050,1,256]	tgt[100,1,256] mem[1050,1,256]	41.30M
DETR-ResNet50-E12	12	6	256	2048	8	[1,3,800,1333]	N/A	query[1050,1,256] key[1050,1,256] value[1050,1,256]	src[1050,1,256]	tgt[100,1,256] mem[1050,1,256]	49.20M
SETR-Naive-Base	12	1	768	4096	12	[1,3,384,384]	16	query[576,1,768] key[576,1,768] value[576,1,768]	src[576,1,768]	tgt[576,1,768]	87.69M
SETR-Naive	24	1	1024	4096	16	[1,3,384,384]	16	query[576,1,1024] key[576,1,1024] value[576,1,1024]	src[576,1,1024]	tgt[576,1,1024]	305.67M
SETR-PUP	24	1	1024	4096	16	[1,3,384,384]	16	query[576,1,1024] key[576,1,1024] value[576,1,1024]	src[576,1,1024]	tgt[576,1,1024]	310.57M
ViT-Base-16	12	0	768	3072	12	[1,3,224,224]	16	query[197,1,768] key[197,1,768] value[197,1,768]	src[197,1,768]	N/A	86.00M
ViT-Large-16	24	0	1024	4096	16	[1,3,224,224]	16	query[197,1,1024] key[197,1,1024] value[197,1,1024]	src[197,1,1024]	N/A	307.00M
ViT-Huge-14	32	0	1280	5120	16	[1,3,224,224]	14	query[257,1,1280] key[257,1,1280] value[257,1,1280]	src[257,1,1280]	N/A	632.00M

TABLE II End-to-End Exeuction Performance on the Benchmark (ms)

	PyTorch JIT [18]	TVM-CUDA [27]	TVM-cuDNN/BLAS [27]	TensorRT [24]	Ansor [22]	AutoGTCO
DETR-ResNet50-E3	18.62	54.73	54.43	6.97	5.85	**5.32**
DETR-ResNet50-E6	23.67	93.59	88.25	7.73	6.78	**5.60**
DETR-ResNet50-E12	33.01	171.96	157.97	15.79	14.29	**13.18**
SETR-Naive	68.26	753.25	742.21	33.71	34.22	**28.65**
SETR-Naive-Base	31.06	186.13	187.39	16.97	15.44	**14.21**
SETR-PUP	37.62	199.42	189.21	18.61	17.89	**16.01**
ViT-Base-16	24.92	91.86	96.31	**5.87**	8.57	8.43
ViT-Large-16	52.96	329.74	334.38	18.45	18.99	**18.41**
ViT-Huge-14	76.07	846.87	846.27	34.14	32.53	**29.89**

TABLE III The Number of Subgraphs and Scheduling Weights for Graph Partition

	n_1	Weight-Encoder	n_2	Weight-Decoder	n_3	Weight-Transformer
Ansor [22]	9	$\{[6*7],[12*2]\}$	13	$\{[6*10],[18*3]\}$	22	$\{[6*17],[13*2],[19*2],[18*1]\}$
AutoGTCO	6	$\{[8*4],[10*2]\}$	11	$\{[8*6],[20*5]\}$	17	$\{[9*12],[20*3],[16*2]\}$

Huge, which limits the search space of the specific fusion operator (such as batch matrix multiplication and softmax) in MHA. Compared with automatical search-based Ansor [22], AutoGTCO outperforms it in all benchmark models with 1.01 to 1.21× speedup. It proves that our operator fusion strategy and sketch generation rules have achieved good performance on image recognition with transformer models on GPU. TVM-cuDNN/BLAS is the TVM compiler that implements operators in computation graphs with cuDNN and cuBLAS library. Compared with TVM-cuDNN/BLAS [27], AutoGTCO consistently outperforms all benchmark models with significant speedup. The reason for this phenomenon is that TVM-cuDNN/BLAS only uses the operator fusion rules defined in Relay to partition the entire computation graph into some kinds of subgraphs, and then replace each operator in the subgraph with the template in cuDNN or cuBLAS library. In the process of optimization, it does not involve searching the tensor program schedules and fine-tuning the performance of each kernel by cost model. Therefore, AutoGTCO has more advantages on uncommon operator fusion patterns in the emerging image recognition with transformer models, because it is not easy for vendor-specific static libraries to manually optimize for all the cases. The only difference between TVM-CUDA and TVM-cuDNN/BLAS is that the operators in subgraphs are implemented by the default schedule template.

Ablation study.We run variants of AutoGTCO on DETR-

TABLE IV Performance on DETR with different settings

	AutoGTCO			
Setting	(a)	(b)	(c)	(d)
DPOF		✓	✓	✓
Sketch Customization			✓	✓
Subgraph Scheduler				✓
Speedup	1.00×	1.19×	1.34×	1.38×

ResNet50-E6 benchmark. "DPOF ✓" means we use the dynamic programming algorithm to do operator fusion rather than the rule-based operator fusion strategy defined in Relay. "Sketch Customization ✓ " means we use the sketch generation rules and search policy defined in AutoGTCO rather than default configurations in Ansor. "subgraph scheduler ✓" means we use the object function defined in Equation (4). *Design (a)* is the Ansor system and we set the execution time of *Design (b)-(d)* to be the speedup against Ansor. From TABLE IV, we can find that *Design (d)* performs the best in speedup performance among all of the designs. It can prove that the graph and tensor co-optimize is very important for the transformer model acceleration in our system. In graph-level, AutoGTCO employs the DPOF module to optimize the operator fusion and then uses a subgraph scheduler designed for transformer architecture to schedule different tasks. In tensor-level, the tensor programs are optimized and generated by our sketch generation rules and search policy.

C. Subgraph Benchmark

Baselines and Configurations. We perform the subgraph benchmark on three common subgraphs in DETR-ResNet-50-E6: MHA, Encoder, and Decoder. We run auto-tuning with up to 20,000 measurement trails per test case, and report the execution time. We use the same set of baseline frameworks and run three benchmarks on one NVIDIA GeForce RTX 2080Ti GPU.

Results. Fig. 7 shows that AutoGTCO outperforms PyTorch JIT on the Encoder and Decoder by 2.47× and 11.67× speedup. For high-performance computing library TensorRT, AutoGTCO can achieve 2.47×, 1.08×, and 4.19× speedup on MHA, Encoder, and Decoder. For the compiler-based search algorithm Ansor, AutoGTCO can achieve 1.29×, 1.17×, and 1.17× speedup on MHA, Encoder, and Decoder. It can prove that AutoGTCO can generate efficient tensor programs for these subgraphs on the NVIDIA GPU platform.

D. Graph Partition and Tuning Time

TABLE III shows the graph partition of our method on the DETR-ResNet50-E6 benchmark. "n_1", "n_2" and "n_3" means the number of subgraphs in encoder, decoder and transformer model respectively. The first value in "Weight-*" means the weight of the subgraphs and the second value means the number of subgraphs. For example, $\{[6*7],[12*2]\}$ means there are 7 subgraphs with weights 6 and 2 subgraphs with weights 12. The total number of subgraphs in encoder is 9. As shown in the table, our graph partition and subgraph scheduler methods can achieve a more effective operation fusion strategy

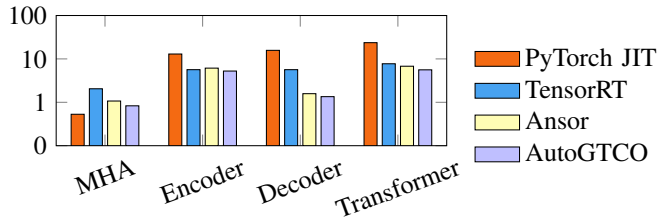

Fig. 7 Sugraph performance benchmark. The y-axis is the throughput based log 10 and then plus 1.

for the number of subgraphs and weights compared to the rule-based method in Ansor.

TABLE V shows the search time needed for AutoGTCO to match the performance of Ansor on the subgraph benchmark. We use "number of measurement trails" to evaluate the search time. As shown in the table, AutoGTCO can match the performance of Ansor with fewer measurement trails. It can prove that the saving in search time comes from the design of subgraph scheduler for transformer models, the operator fusion strategy based on dynamic programming in graph-level and the sketch generation rules for tensor programs generation.

TABLE V The Number of Measurement Trails

	Ansor [22]	AutoGTCO	Speedup
Multi-Head Attention	1,600	1,408	1.13×
Encoder-3-Layer	3,008	2,816	1.07×
Encoder-6-Layer	4,992	4,096	1.22×
Encoder-12-Layer	6,528	5760	1.13×
Decoder-6-Layer	2,688	2,432	1.11×
DETR-ResNet-50-E6	8,640	6,784	1.27×

VI. Conclusions

Existing frameworks do graph-level optimization through greedy methods designed by human experts, which are strictly the improvement of execution performance. These approaches miss the potential performance gains from more effective operators fusion strategies. This work tackles the problem from two aspects. First, we introduce a novel dynamic programming algorithm to explore operator fusion strategies. Combined with operator fusion optimizations, we propose new sketch generation rules and a search policy for the batch matrix multiplication and softmax operators in transformer subgraphs, which are capable of fusing them into large computation units, then mapping and transforming them into efficient CUDA kernels. To get a high-performance and end-to-end compilation flow, a learned cost model is used to fine-tune the performance of each kernel in the code generation stage. Overall, AutoGTCO can reach up to 1.38× execution performance speedups compared to the current state-of-the-art deep learning library TensorRT.

Acknowledgment

This work is partially supported by SmartMore and ITF Partnership Research Programme (No. PRP/65/20FX).

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015, pp. 1–14.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[4] Y. Bai and W. Wang, "ACPNet: Anchor-Center Based Person Network for Human Pose Estimation and Instance Segmentation," in *IEEE International Conference on Multimedia and Expo (ICME)*, 2019, pp. 1072–1077.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 5998–6008.

[6] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *arXiv preprint arXiv:2101.01169*, 2021.

[7] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European Conference on Computer Vision (ECCV)*, 2020, pp. 213–229.

[8] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, "Emerging properties in self-supervised vision Transformers," *arXiv preprint arXiv:2104.14294*, 2021.

[9] S. Zheng, J. Lu, H. Zhao, X. Zhu, Z. Luo, Y. Wang, Y. Fu, J. Feng, T. Xiang, P. H. Torr, and L. Zhang, "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers," in *CVPR*, 2021.

[10] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[11] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang, "A history-based auto-tuning framework for fast and high-performance dnn design on GPU," in *ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[12] Q. Sun, C. Bai, H. Geng, and B. Yu, "Deep neural network hardware deployment optimization via advanced active learning," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2021.

[13] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 859–873.

[14] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[15] T. Chen, B. Duan, Q. Sun, M. Zhang, G. Li, H. Geng, Q. Zhang, and B. Yu, "An efficient sharing grouped convolution via bayesian learning," in *IEEE Transactions on Neural Networks and Learning Systems*, 2021, pp. 1–13.

[16] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 9, no. 2, pp. 292–308, 2019.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean *et al.*, "TensorFlow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Workshop*, 2017.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *ACM International Multimedia Conference (MM)*, 2014, pp. 675–678.

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

[21] "oneAPI Deep Neural Network Library," https://github.com/oneapi-src/oneDNN#documentation.

[22] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 863–879.

[23] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 4, pp. 32:1–32:12, 2012.

[24] "NVIDIA TensorRT," https://docs.nvidia.com/deeplearning/tensorrt/index.html.

[25] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *International Conference on Machine Learning (ICML)*, 2015, pp. 448–456.

[26] A. F. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," *CoRR*, vol. abs/1803.08375, 2018.

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

[28] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: optimizing deep learning computation with automatic generation of graph substitutions," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 47–62.

[29] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "IOS: Inter-operator scheduler for cnn acceleration," *Machine Learning and Systems (MLSys)*, vol. 3, 2021.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[31] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2015, pp. 91–99.

[32] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[33] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 785–794.

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

[35] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *International Conference on Compiler Construction*, A. Mycroft and A. Zeller, Eds., 2006, pp. 185–201.

[36] L. J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016.

[37] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to Optimize Tensor Programs," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2018, pp. 3393–3404.