GTCO: Graph and Tensor Co-Design for Transformer-Based Image Recognition on Tensor Cores

Yang Bai, Xufeng Yao, Qi Sun, *Member, IEEE*, Wenqian Zhao, Shixin Chen, Zixiao Wang, and Bei Yu, *Senior Member, IEEE*

***Abstract*—Deep learning frameworks or compilers optimize the operators in computation graph using fixed templates via significant engineering efforts, which may miss potential optimizations such as operator fusion. Therefore, automatically implementing and optimizing the emerging new combinations of operators on a specific hardware accelerator is of importance. In this article, we introduce GTCO, a tensor compilation system designed to accelerate transformer-based vision models' inference on GPUs. GTCO tackles the operator fusion techniques in the transformer-based model using a novel dynamic programming algorithm and proposes a search policy with new sketch generation rules for the fused batch matrix multiplication and softmax operators. Tensor programs are sampled from an effective search space, and a hardware abstraction with hierarchical mapping from tensor computation to domain-specific accelerators (Tensor Cores) is formally defined. Finally, our framework can map and transform tensor expression into efficient CUDA kernels with hardware intrinsics on GPU. Our experimental results demonstrate that GTCO improves the end-to-end execution performance by up to 1.73× relative to the cutting-edge deep learning library TensorRT on NVIDIA GPUs with Tensor Cores.**

***Index Terms*—Compilation, GPU acceleration, operator fusion, tensor core, transformer.**

I. Introduction

RECENT years have witnessed the success of deep learning in the industry-scale application, ranging from language translation, virtual reality, and recommendation systems to computer vision and autonomous driving. In particular, convolutional neural networks (CNNs) remain dominant in computer vision [2], [3], [4], [5]. Despite their significant success, attention has been increasingly focused on integrating self-attention techniques with CNN-based models [6], inspired by their success in neural language processing (NLP). Many model architectures have entirely replaced CNNs with transformer-based models [7], promoting the further application of these models in various vision tasks [8], [9], [10], [11]. This trend paves the way for using a unified transformer architecture in future research developments.

Deep learning models, whether they are traditional CNNs or transformer architectures, can be represented as directed acyclic computation graphs (DAGs). Each node in the DAG represents an operator, and edges denote the relationship between two connected operators. DAGs are currently scheduled to hardware accelerators, such as GPUs [12], [13], [14], [15], [16], [17], FPGAs [18], and ASICs [19] through popular frameworks, such as Caffe [20], TensorFlow [21], and PyTorch [22], using vendor-provided libraries, such as MKL-deep neural network (DNN) [23], ARM-Compute Library [24], and cuDNN [25] to achieve efficient model deployment.

Optimizing the performance of different operators with diverse hardware platforms requires a significant engineering effort when using these vendor-provided libraries. As a result, researchers and engineers often concentrate on enhancing the performance of compute-intensive primitives, such as GEMM and convolution, which are frequently employed in CNN architecture. Alternatively, they turn to a search-based compilation approach [26], [27] that involves separating kernel definition from computation scheduling to automatically generate tensor programs. While prior research has primarily focused on optimizing the deployment of CNN models, the potential of transformer-based vision models on modern accelerators has not been effectively leveraged due to the specialized self-attention modules. Although existing libraries have the capability to fuse element-wise operators into compute-intensive kernels, they are not as effective in optimizing memory-intensive workloads, such as matrix multiplication with softmax operators. Furthermore, transformer-based models typically involve a large number of fine-grained operators, which can result in significant overheads when implemented on specific hardware, such as GPUs. Meanwhile, due to the predefined rules for optimizing compute-intensive operators, traditional techniques, such as Halide [26], [28], cannot effectively utilize the inference efficiency of specialized modules in transformer models.

Numerous techniques have been proposed to enhance the efficiency of models by optimizing them at the graph-level. Lots of work are proposed to facilitate the optimization from the graph-level to improve the efficiency of the model. TensorRT [29] utilizes a two-step fusion policy to optimize the computation graph. First, specific operators, such as fully connected, convolution, batch normalization [30], and ReLU [31], are fused vertically using rules designed by high-performance computing engineers. Second, the operators are fused horizontally within the same stage. This two-step

Manuscript received 26 October 2022; revised 23 April 2023; accepted 2 September 2023. Date of publication 19 September 2023; date of current version 22 January 2024. This work was supported in part by SmartMore and in part by The Research Grants Council of Hong Kong, SAR, under Project CUHK24209017. The preliminary version has been presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2021 [1] [DOI: 10.1109/ICCAD51958.2021.9643487]. This article was recommended by Associate Editor L. C. Wang. *(Corresponding author: Bei Yu.)*

The authors are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, SAR (e-mail: byu@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2023.3317169

optimization method works well for CNN architectures with multiple parallel branches and kernels of the same size. Greedy rule-based subgraph substitutions are employed to optimize the computation graph in classical frameworks, such as TensorFlow [21], PyTorch [22], TVM [27], and Ansor [28]. While template-based substitutions may enhance the efficiency of computations, they are not suitable for long-term maintenance. With new operators continuously being proposed from a high-level model perspective, the rule-based graph substitution approach becomes increasingly unsuitable for real-world production, as it requires significant engineering effort. To search potential substitutions, search-based methods are introduced by TASO [32] and IOS [33] to exploit a large enough search space. TASO generates graph substitutions with a formal verification to verify the correctness of the optimized graph substitutions automatically. Search-based methods have been introduced to tackle the issue of finding potential substitutions for new operators. TASO [32] and IOS [33] are two examples of search-based methods that are capable of exploring a vast search space. TASO generates graph substitutions and verifies the correctness of the optimized graph substitutions through formal verification techniques. While IOS is able to leverage interoperator and intraoperator parallelism to schedule operators with compute unified device architecture (CUDA) stream, thus maximizing the benefits of both software and hardware accelerators, it is not able to fully exploit the code generation capabilities from a compilation perspective for the implementation of each operator. This is due to the IOS using the vendor-provided library cuDNN to do the runtime, which provides a fixed template for each operator with limited runtime performance. As a result, searching for optimal solutions with operator fusion from a comprehensive search space is not possible using this paradigm.

The NVIDIA GPUs' domain-specific accelerators (Tensor Cores) are programmed using instruction set architecture, which allows for algorithmic specification to be separated from hardware architectural details. These instructions are commonly referred to as intrinsic, and using them for tensor computation is known as tensorization. While intrinsics provide programmability, mapping specific intrinsics remains a challenging task. For example, there are 35 different ways to map the seven for-loops of a 2-D convolution implementation on the Tensor Cores. Mapping performance is crucial to configurations that can impact data locality and parallelism on GPUs. However, current compilers [27], [28] rely heavily on manual programming with hardware intrinsic to develop high-performance implementation, which may overlook optimal mapping choices between the complex memory hierarchy. To efficiently support algorithms on domain-specific accelerators, an automated mapping solution is necessary to explore software and hardware co-design. While the baseline framework is presented in [1] as depicted in Fig. 1, our framework is called GTCO.

This article presents a solution to the automatic mapping problem on domain-specific accelerators (Tensor Cores) by introducing a novel abstraction above the hardware intrinsics. The abstraction comprises two components: *computation* and *memory* abstraction, which describe the computation and data movement behaviors within an intrinsic. Based on this proposed abstraction, the authors develop a two-step mapping generation method that can map software computations to a virtual hardware accelerator without hardware constraints and then modify the mapping configurations based on actual physical constraints. Additionally, the authors efficiently explore the search space to achieve low inference latency on domain-specific accelerators (Tensor Cores). Finally, the authors implement all of the techniques and integrate them into an end-to-end tensor compilation named GTCO. In summary, this article makes the following contributions.

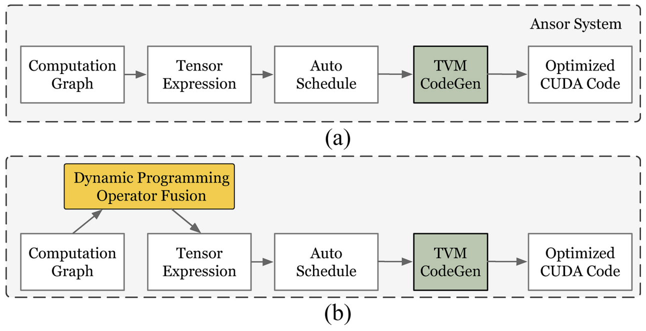


Fig. 1. Overview of (a) Ansor [28] and (b) [1]. Initially, the input is a computation graph, which is converted into tensor expression language. Subsequently, the auto-schedule module is capable of automatically searching for the optimal schedule for each operator. Finally, TVM code generation is performed to generate optimized CUDA code on GPU. However, the primary distinguishing factor between the two systems is the DPOF module.

1) A dynamic programming algorithm is introduced to tackle the operator fusion for transformer-based vision models. The algorithm is capable of automatically generating the optimal combination for operator fusion in transformer-based vision models, which exploits a comprehensive combination search space than the template-based techniques devised by high-performance experts.
2) Novel sketch generation rules incorporating a search policy are introduced for the purpose of fusing batch matrix multiplication and softmax operators in an effective search space. Furthermore, a regression-based learned cost model is employed to optimize the performance of kernels via an end-to-end automatic compilation flow.
3) A hardware abstraction at the register-level is introduced to formally define the computation and memory behavior of the operators on Tensor Cores. Additionally, a fully automatic mapping from the tensor expressions to hardware execution is developed to accelerate the transformer-based vision models on Tensor Cores.
4) We analyze transformer-based image recognition models, specially DETR [8], SETR [10], and vision transformer (ViT) [11], using the GTCO. Our framework can generate corresponding CUDA code with WMMA instructions on GPU under different inference settings automatically. Experiment results demonstrate the superiority of the optimized code, which outperforms TensorRT with measured inference speedup ranging from 1.01× to 1.38× with CUDA Cores and 1.15× to 1.73× with Tensor Cores.

II. Background and Related Works

A. Transformer Architecture

Initially, transformer-based models were developed for machine translation, using novel attention-based building blocks. These models are constructed for processing long sequences in natural language processing. The attention mechanism is the primary component in transformer architecture, comprising neural network layers that aggregate information from the entire input sequence and enable the model to learn

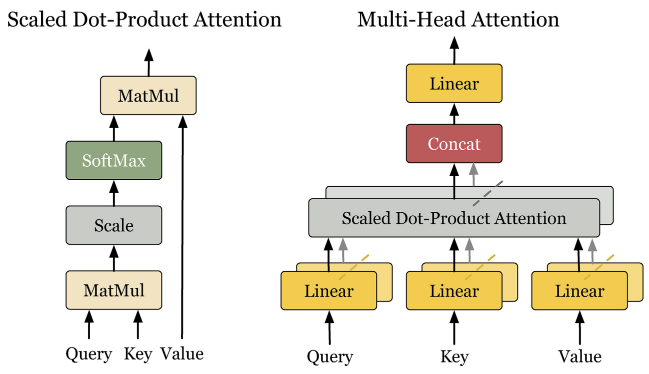


Fig. 2. Details of scaled dot-product attention and MHA in the transformer architecture.

how to focus on specific parts of the sequence. Transformers have made two significant contributions. First, they popularized the attention technique as a dedicated module known as multihead attention (MHA) [10], which is widely used in many models. Second, transformers do not rely on the convolutional or recurrent network, which sets them apart from previous models. Moreover, an essential characteristic of the transformer model is that it is composed of many similar subgraph structures. This attribute provides ample opportunities for constructing novel systems that can be executed in parallel.

MHA: MHA employs h attention heads to simultaneously obtain diverse learned projections from the input sequence. Each head operates on an instance of scaled dot-product attention with queries (q), keys (k), and values (v) as input variables. Linear layers are utilized to enhance the function of attention heads by projecting the input variables into a lower-dimensional learning subspace. Weight tensors (wq, wk, wv) are crucial matrix variables in the transformer-based models. A learned input projection function, equipped with query and key variables, is then scaled by their product, followed by multiplication with the vv matrix to obtain the output. The query and key variables are multiplied and scaled, and the softmax operation is applied to extract the most relevant results. Finally, the per-head result is obtained by multiplying with vv. The outputs from all attention heads are concatenated and linearly projected back to the input dimensional size. More details can be found in Fig. 2. This work centers on transformer models that are based on encoders and decoders, with a particular focus on the ViT architecture family. ViT comprises a series of uniform encoder blocks, wherein each block comprises an MHA layer followed by a pointwise feedforward network (FFN). Additionally, a residual connection layer is developed between the two submodules with a layer normalization operator. The MHA layer is made up of multiple attention heads that are independently parameterized

$$\mathrm{MHA}(\boldsymbol{x}) = \sum_{i=1}^{H} \mathrm{Att}_i(\boldsymbol{x}), \quad \boldsymbol{x}_{\mathrm{MHA}} = \mathrm{LN}\big(\boldsymbol{x} + \mathrm{MHA}(\boldsymbol{x})\big)$$

where Att is a dot product attention head, LN is layer normalization, and $\boldsymbol{x}$ is the input vector. The output of the MHA layer is then entered into the FFN layer, which consists of N filters

$$\mathrm{FFN}(\boldsymbol{x}) = \left(\sum_{i=1}^{N} \mathcal{W}^{(2)}_{:,i}\sigma\left(\mathcal{W}^{(1)}_{i,:}\boldsymbol{x} + \boldsymbol{b}^{(1)}_i\right)\right) + \boldsymbol{b}^{(2)}$$
$$\boldsymbol{x}_{\mathrm{out}} = \mathrm{LN}\big(\boldsymbol{x}_{\mathrm{MHA}} + \mathrm{FFN}(\boldsymbol{x}_{\mathrm{MHA}})\big)$$

where $\mathcal{W}^{(1)}$, $\mathcal{W}^{(2)}$, $\boldsymbol{b}^{(1)}$, and $\boldsymbol{b}^{(2)}$ are the FFN parameters, and σ is the activation function, typically GELU [34].

Decoder: The decoder layer is very similar to the structure of the encoder layer. Besides the two submodules introduced in the encoder layer, a third submodule is inserted into the decoder layer seamlessly. The function of the third sublayer is to perform MHA over the output of the encoder part. Besides, some improvements in masking techniques [35] are designed to prevent positions from attending to subsequent positions in the self-attention module.

B. Machine Learning Compilation

This work focuses exclusively on the application of deep learning compilers. Specifically, we take Ansor [28] as an example, a widely used deep learning compiler for generating tensor programs across various hardware platformsThe compiler leverages a hierarchical search space to optimize and separate the high-level generation structures from the low-level sampling details. This approach enables Ansor to automatically construct the search space for each operator or subgraph, eliminating the need for experienced engineers to manually develop computing templates, which can be a time-consuming and engineering-heavy process. Subsequently, Ansor incorporates an automatic performance tuner that utilizes a comprehensive search space to obtain complete tensor programs, which are then fine-tuned with a regression-based model [36].

C. Efficient Transformers

In order to enhance the inference performance and decrease the memory consumption of Transformer-based models, a variety of techniques have been proposed. These techniques can be classified into the following categories: 1) the design of efficient model architecture [37]; 2) quantization [38]; and 3) pruning [39] and knowledge distillation [40]. This work focuses on compilation techniques for accelerating the execution time of transformer-based vision models through the utilization of specific hardware units, such as CUDA Cores or Tensor Cores on NVIDIA GPUs.

D. Hardware Details of 2080Ti GPU

CUDA is a high-performance programming language developed by NVIDIA to expose software programmers to concepts, such as computation parallelism and memory hierarchy. However, effectively leveraging the memory and computation units of GPUs to accelerate the execution time of DNNs requires significant engineering efforts. As depicted in Fig. 3, the GPU device has many programmable units at different levels. For our experiments, we utilize the NVIDIA RTX 2080Ti GPU, which follows the Turing microarchitecture and has 68 parallel streaming multiprocessors (SMs) within the processing elements. Each SM has its local shared memory that can be accessed by threads within the same SM. An SM is further divided into four processing blocks, with each block possessing a 64-kB register file, and every four blocks sharing a combined 96-kB L1 data cache/shared memory. Thread blocks are scheduled on each processing block, and each thread block contains a group of threads that can execute the same code on different data using the single-instruction–multiple-threads (SIMT) technique. Typically, prior to launching the computation kernel, data is first copied from the host

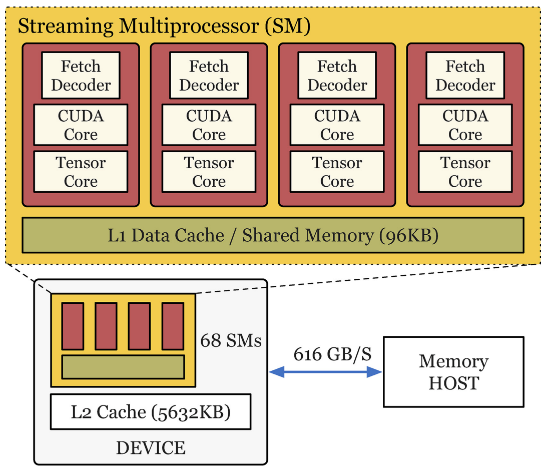


Fig. 3. Hardware details of the SM and the memory hierarchy of NVIDIA RTX 2080 Ti GPU.

memory to the device memory. On-chip memory, which is close to the computing elements, has a faster access speed than off-chip memory, which is slower and further away from the computing elements. Each type of memory has its own unique access patterns. Registers and local memory are private to the threads within a block and are located on-chip with low latency. Shared memory is composed of a set of full-sized banks, and multiple threads accessing the same bank simultaneously may lead to conflicts and increased latency.

III. Problem Formulation

Definition 1 (Computation Graph): The model is defined using a computation graph $G = (V, E)$, where V is the set of vertices and E is the edge set. Each vertex in the graph represents an operator, such as GEMM or softmax. As for the edge $(u, v) \in E$, it represents a tensor that can store the input of operator v and the output of operator u.

Computation graphs are a common way to represent deep learning models in frameworks or compilers. Fig. 2 shows the computation graph in the transformer-based model.

Operator Pattern: Operator fusion is a very efficient technique to optimize memory-bound workloads. To circumvent the need for storing intermediate results in global memory, operator fusion merges multiple connected operators into a single computation kernel. This optimization technique can substantially enhance the inference speed-up, particularly in throughput-oriented architectures like GPUs. In order to effectively do operator fusion, the operator patterns must first be defined. We categorize operators into five distinct patterns: 1) *injective*; 2) *element-wise*; 3) *opaque*; 4) *reduction*; and 5) *complex-out-fusable*. As mentioned in Section II, we observe that matrix transposition, layer normalization, batch matrix multiplication, softmax, and fully connected layers frequently occur in transformer-based models. Furthermore, the default configuration of them adheres to these guidelines: 1) softmax is marked as the *opaque* pattern; 2) dense and batch matrix multiplication are identified as the *complex-out-fusible* pattern; and 3) layer normalization [41] can be decomposed into a set of fundamental operators (multiple, add, and subtract), which are labeled as the *element-wise* pattern.

Fusion Scheduling: Based on each computation graph G extracted from the transformer-based model, a corresponding schedule S is defined to optimize the inference latency on the

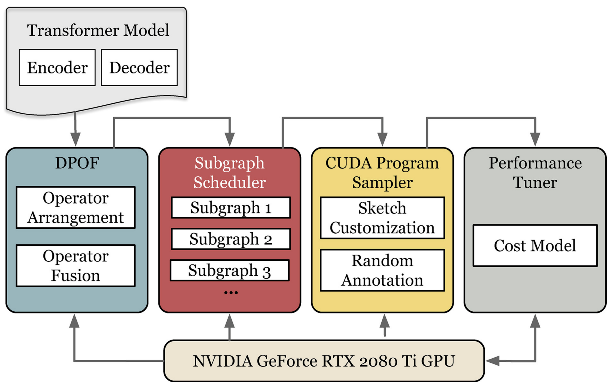


Fig. 4. Workflow and components of our framework. The input is the transformer-based vision models and the output is the tensor programs generated on the GPU platform.

GPU platform as follows:

$$S = \{(V_1, F_1), (V_2, F_2), \dots, (V_k, F_k)\}$$

where V_i denotes a set of computation operators in the ith stage, and F_i denotes the paired variable that describes the fusion relationship between any two nodes. The execution of G with the schedule S is carried out consecutively from the first stage (V_1, F_1) to the last stage (V_k, F_k).

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

$$S^* = \underset{S}{\operatorname{argmin}} \ \operatorname{Cost}(G, S) \tag{1}$$

where Cost is the execution time of G with the schedule S.

IV. Details of GTCO

A. Overview

Fig. 4 illustrates the overall architecture of our tensor compilation system, which comprises four essential modules: dynamic programming operator fusion (DPOF), subgraph scheduler, program sampler, and performance tuner. Starting with the input of the transformer-based model lacking the operator fusion technique, each operator is initially labeled with a pattern that denotes the relationship between the connected operators. After applying the operator fusion technique, each operator is assigned a new pattern, and the types of the connected operators are also altered based on the predicted labels that arise from the operator fusion. High-performance tensor programs with hardware intrinsic are generated for the fused operators on the GPU platform. All in all, our framework includes the following components.

1) A DPOF module that can find an optimized operator fusion schedule for the transformer-based vision model.
2) A subgraph scheduler module assigns time slots for optimizing fused subgraphs optimized by the DPOF.
3) A program sampler module that constructs a comprehensive search space and samples different kinds of tensor programs from it randomly to ensure diversity.
4) A performance tuner module trains a regression-based model to predict the performance of sampled programs.
5) An automatic mapping flow with compilation techniques that can find the optimal implementation of the fused operators in transformer-based models on Tensor Cores with floating-point 16 (`FP16`) datatype.

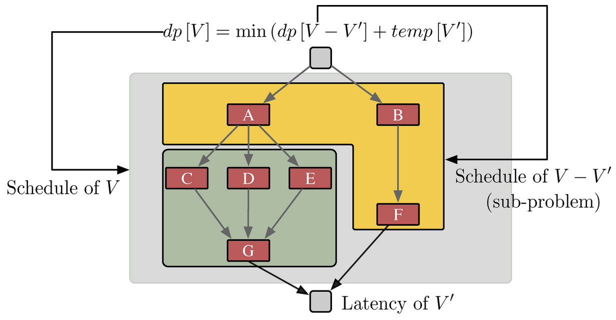

$dp[V] = \min(dp[V - V'] + temp[V'])$
Schedule of V
A
B
C
D
E
F
G
Schedule of $V - V'$ (sub-problem)
Latency of V'

Fig. 5. The design of dynamic programming operator fusion.

B. Operator Fusion With Dynamic Programming (DPOF)

Operator Arrangement: Initially, the execution order of computation operators in the original graph is determined by a topological sorting algorithm, which serves as our starting point for finding an optimized schedule. Next, we use a computation queue to store the selected operators from the topological sorting algorithm. Rather than using a more complex graph data structure, we employ a queue data structure to identify and store the optimal schedule. The variables in the queue can be divided into two categories: placeholder and computation variables. Placeholder variables are used to store input and output results, which do not affect the execution time of the entire computation graph and are therefore not taken into account. As introduced in Section III, the same operator with various labels makes the difference in the execution stage. Third, we assume that no fusion relationship exists between the operators at the beginning stage, and all operators are assigned to an opaque state. The size of the queue is determined by the maximum number of stage defined in the scheduling Section III. As discussed in Section III, the same operator with different patterns can have a significant impact on the execution stage.

Operator Fusion: The computation graph $G = (V, E)$ is initially partitioned into two sets: V' and $V - V'$ based on the execution order of the computation variables and the maximum number of stages during the scheduling. In a set of V', the edges in a set of $V - V'$ have a pointing relationship with the directed edges. Specifically, the start points of the edges are in $V - V'$ and the end points are in V'. The set of vertices V' is defined as the segmentation set. The interplay between V' and $V - V'$ is illustrated in Fig. 5. We observe that the computation graph exhibits numerous segmentation sets. According to the dynamic programming algorithm, we can systematically enumerate the elements in the segmentation sets V' of V. This technique transforms the original problem into a subproblem that aims to determine the optimal schedule for $V - V'$. Consequently, the computation graph G can be recursively optimized for each element in the segmentation set. The dynamic programming approach defines $dp[V]$ as the execution time of the computation graph G with an optimal schedule S in the nodes set V. Additionally, temp$[V']$ represents the execution time of the subgraph composed of the nodes in the stage (V', F). Here, F represents the optimal fusion strategy in the segmentation set V'. The state transition equation can be defined as follows:

$$dp[V] = \min_{v \in V'}\left(dp\left[V - V'\right] + \sum_{v} \text{temp}[v]\right). \tag{2}$$

Algorithm 1 Operator Fusion Strategy

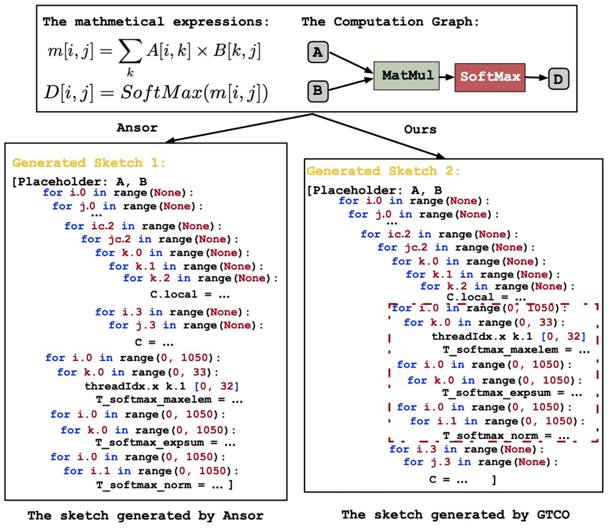
Input: A computation graph $G = (V, E)$ with the **opaque** type for $\forall v \in V, pattern(v) = 0$;
Output: A operator fusion strategy with the type of each operator $v \in V, pattern(v)$;

```
1:
2: Defining $dp[\varnothing] \leftarrow 0$, $dp[V] \leftarrow +\infty$, $action[V] \leftarrow \varnothing$;
3: Defining $S \leftarrow [\varnothing]$ (A Stack data structure to store the phase of optimal schedule for operator fusion);
4:
5: function SelectSchedule($G$)
6:     $V$ = all operators in computation graph $G$;
7:     Scheduler(V);
8:     while $V \neq \varnothing$ do
9:         $V', F = action[V]$;
10:        Put phase $(V', F)$ into the stack $S$;
11:        $V = V - V'$;
12:    end while
13:    return the Fusion Strategy $S$;
14: end function
15:
16: function Scheduler(V)
17:    if $dp[V] \neq +\infty$ then
18:        return dp[V];
19:    end if
20:    for all $v \in V'$ do
21:        $T_{V'}, F_{V'}$ = PhasePartition($V'$);
22:        $T_V$ = Scheduler($V - V'$) + $\sum_{v_i \in V'} T_{V'}$;
23:        if $T_V \leq dp[V]$ then
24:            $dp[V] = T_V$;
25:            $action[V] = (V', F_{V'})$;
26:        end if
27:    end for;
28:    return $dp[V]$;
29: end function
30:
31: function PhasePartition($V'$)
32:    for all operators $v_i \in V'$ do
33:        if $pattern(v_i, v_j) \neq opaque$ then
34:            $T_{\text{fused}(i,j)}$ = Runtime($pair(v_i, v_j)$);
35:        else
36:            $T_{\text{fused}(i,j)} = +\infty$;
37:        end if
38:    end for
39:    return $T_{\text{fused}(i,j)}, pattern(v_i, v_j)$;
40: end function
```

In Algorithm 1, v represents a node in the segmentation set V', and $dp[\varnothing]$ is the boundary value of the state transition equation, set to 0. The optimal solution is obtained by storing each node v in the segmentation set V' and measuring the execution time of each V through action$[V]$.

C. Subgraph Scheduler

It is evident that partitioning a model into different subgraphs is a crucial step before performance optimization. However, it is futile to invest significant time in tuning subgraphs without the possibility of enhancing the execution performance of the model during optimization. Therefore, we opt to dynamically assign varying amounts of time slots to different types of subgraphs. For transformer models, a subgraph may occur multiple times. As a result, achieving a well-optimized transformer-based model requires resolving numerous scheduling tasks during compilation.


The mathmetical expressions:
$m[i,j] = \sum_k A[i,k] \times B[k,j]$
$D[i,j] = SoftMax(m[i,j])$
The Computation Graph:
A
B
MatMul
SoftMax
D
Ansor
Ours
The sketch generated by Ansor
The sketch generated by GTCO

```
Generated Sketch 1:
[Placeholder: A, B
    for i.0 in range(None):
      for j.0 in range(None):
        ...
        for ic.2 in range(None):
          for jc.2 in range(None):
            for k.0 in range(None):
              for k.1 in range(None):
                for k.2 in range(None):
                    C.local = ...
            for i.3 in range(None):
              for j.3 in range(None):
                  C = ...
    for i.0 in range(0, 1050):
      for k.0 in range(0, 33):
          threadIdx.x k.1 [0, 32]
            T_softmax_maxelem = ...
     for i.0 in range(0, 1050):
       for k.0 in range(0, 1050):
            T_softmax_expsum = ...
     for i.0 in range(0, 1050):
       for i.1 in range(0, 1050):
            T_softmax_norm = ... ]
```

```
Generated Sketch 2:
[Placeholder: A, B
    for i.0 in range(None):
      for j.0 in range(None):
        ...
        for ic.2 in range(None):
          for jc.2 in range(None):
            for k.0 in range(None):
              for k.1 in range(None):
                for k.2 in range(None):
                  C.local = ...
            for i.0 in range(0, 1050):
              for k.0 in range(0, 33):
                  threadIdx.x k.1 [0, 32]
                    T_softmax_maxelem = ...
             for i.0 in range(0, 1050):
               for k.0 in range(0, 1050):
                    T_softmax_expsum = ...
             for i.0 in range(0, 1050):
               for i.1 in range(0, 1050):
                    T_softmax_norm = ...
            for i.3 in range(None):
              for j.3 in range(None):
                  C = ...    ]
```

Fig. 6. Sketch generation for the subgraph of MHA. This figure shows two generated sketches. The left one is generated by the default Ansor [28] and the right one is generated by [1]. The difference between these two sketches is that the operator fusion occurs in GTCO with "red-dotted." The code example is pseudocode in a Python-like syntax.

We integrate three objectives during the tuning process: 1) reducing the total execution time of the transformer-based models; 2) meeting requirements of execution time for various subgraph; and 3) decreasing the overall tuning time when certain subgraphs already meet the requirement and cannot be significantly improved. To achieve this, we define an assignment vector as t, where t_i denotes the time slots assigned to the ith task. Initially, all t values are set to $(1, 1, \ldots, 1)$. We then define $g_i(t)$ as the minimum execution time required for the ith subgraph under task t_i. The subgraph execution times $f(g_1(t), g_2(t), \ldots, g_n(t))$ represent the end-to-end execution time of a transformer-based model, and our objective is to minimize the following function. To minimize the end-to-end execution time of the transformer, the objective function is defined as follows:

$$f = \max\left[\sum_{i=1}^{n} w_i \times \max(g_i(t), \mathrm{ES}(g_i, t)), L_j\right]. \tag{3}$$

The number of search task occurrences is denoted as w_i, where i is the task index. It is important to note that if the latency requirement is already satisfied, no tuning time slots will be allocated for a subgraph i. Thus, the latency requirement of subgraph j is represented as L_j. Additionally, we define a function $\mathrm{ES}(g_i, t)$ to enable early stopping by utilizing the historical log information of the ith task. Our framework differs from other frameworks in that it compares the execution and early stopping configurations. Furthermore, we optimize each search task sequentially. Finally, a scheduling design based on the gradient descent method is developed to efficiently solve the objective function.

D. Program Sampler

A hierarchical search space is defined to sample the tensor program, which is based on two techniques: high-level sketch generation and low-level annotation sampling. The high-level information is encapsulated in the sketches, and millions of low-level choices are made to obtain specific optimization, such as blocking size, virtual thread tiling, and cooperative fetching, as the final annotations. To generate high-level sketches for each subgraph, computation nodes are visited in topological order, and a generation structure is built iteratively with multilevel for-loop nests. For computation-intensive nodes with a higher likelihood of data reuse, such as batch matrix multiplication, standard loop tiling, and fusion strategies are implemented as the sketch generation technique.

As illustrated in Fig. 6, a running example is presented to elucidate the process of generating high-level sketches for a common subgraph that comprises matrix multiplication ([1050, 8, 32] × [32, 8, 1050]) and softmax operators in the MHA mechanism. The computation nodes are sorted in the order of $(\boldsymbol{A}, \boldsymbol{B}, \mathit{MatMul}, \mathit{Soft}, \boldsymbol{D})$. Starting from the output node $\mathbf{D}$, we utilize the generation rules to obtain the sketches of the subgraph. The generated sketch 1 depicts that the softmax operators and matrix multiplication are implemented separately, and are not integrated into a single CUDA kernel. To fully exploit the potential of execution efficiency, batch matrix multiplication and softmax operators with new derivation rules are designed in the transformer-based model to optimize numerous operators as a unified computation kernel. In the end, we integrate all of the techniques with existing optimization rules seamlessly to improve execution efficiency.

Sketch Customization: The default sketch generation rules for the GPU backend with a multilevel tiling structure in Ansor are denoted by the string "SSSRRSRS." Here, the letters "*S*" and "*R*" indicate the spatial and reduction dimensions, respectively. The first three "*S*" correspond to BlockIdx,Virtual Thread, and ThreadIdx in GPU programming. The consecutive "*S*" in the tiling structure "SSSRRSRS" describes the matrix multiplication process, which transforms the original 3-level for-loop into a 19-level for-loop, as illustrated in Fig. 6. Additionally, the special multilevel tiling structure can take loop order into consideration during the tensor transformation.

Therefore, we have designed an effective operator fusion strategy, denoted as "SSSR-RSRS." For more details, please refer to Fig. 7. This customized tile structure is specifically designed for batch-matrix multiplication and softmax operators in transformer-based models. By incorporating a caching node with the optimized loop tiling structure, we are able to fully utilize the computation resources on GPUs and fuse multiple operators together. Finally, the computation structure is sent to the sketch of the softmax operator to obtain the fused subgraph implementation. In Fig. 7, the optimized fusion strategy for a subgraph containing batch-matrix multiplication and softmax operators is shown. This strategy was discovered using Algorithm 1 in DPOF. There are four states ❶, ❷, ❸, and ❹ in the process. The initial state has three operators in the subgraph $V = \{M_1, M_2, S\}$. For each state, we can get the ending of V' of V. Thus, the execution efficiency of V can be composed of the latency of V' and $V - V'$. The latency of V' can be measured on the GPU directly and the optimized latency of $V - V'$ can be solved with dynamic programming. ❶ to ❹ show the four different states during the dynamic programming. ❹ is the final state with the best fusion strategy and it also has the optimal inference latency compared with other states. In this state, M_1 and S are fused into a single computation kernel in the first phase and M_2 is executed after that in the second phase.

Annotation Sampling: The sketches generated by the customization rules are incomplete tensor programs because they only have parallel thread structures without the specific value. In order to generate the complete tensor programs

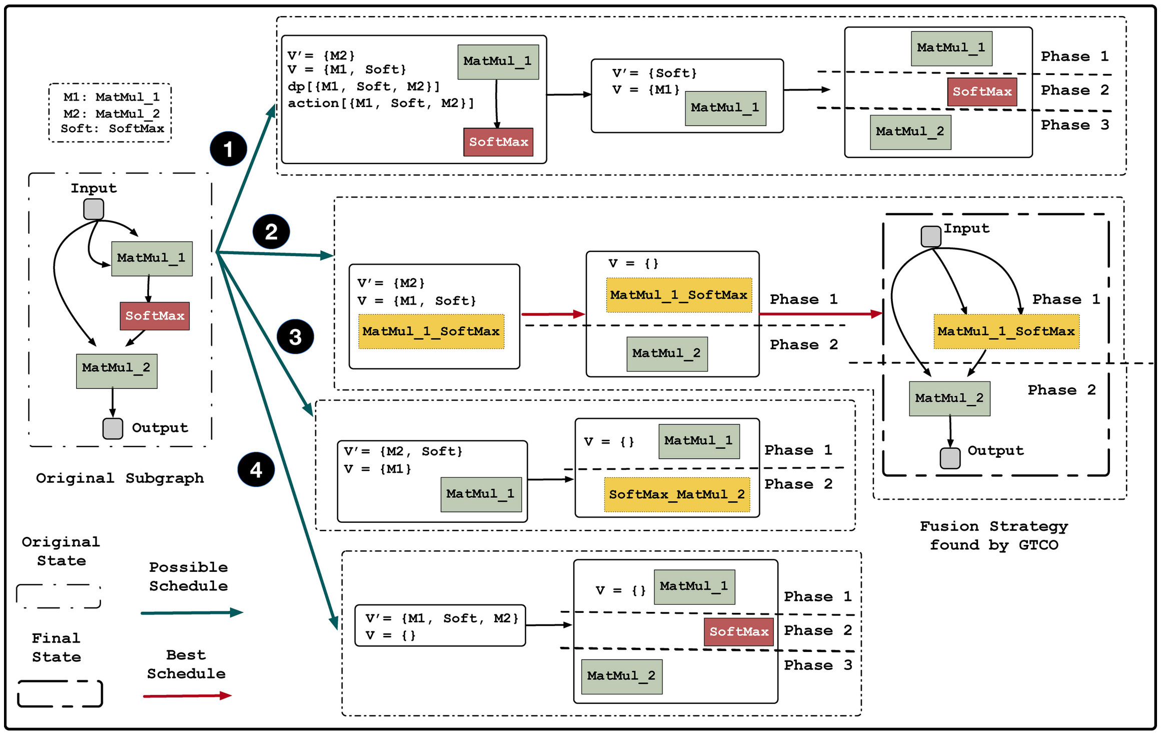
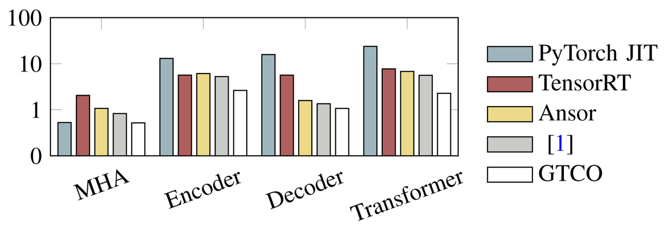


Fig. 7. Example to illustrate how DPOF finds the fusion strategy. The original computation graph is shown on the left. It has three operators, $M1$, $M2$, and *Soft*. There are four states during the dynamic programming algorithm and each transition is shown in the figure. Any transition starts from the state $V = \{M1, Soft, M2\}$ to $V = \{\}$. The best fusion strategy can be obtained by the dynamic programming process.



Listing 1. Basic WMMA instructions for tensor cores.

which can be successfully executed on the GPU, an auto-tuning technique is employed to find optimal parameters for these optimized parallel thread structures. To achieve this, a performance tuner is developed. a performance tuner is developed to make up for the incomplete tensor programs with optimal values. To illustrate this, we randomly select a sketch from a list of generated sketches using our customization rules. Parallel intrinsic functions are used to generate complete tensor programs for the outer for-loop optimization, while vectorize and unroll intrinsic functions are employed to optimize the inner for-loops. It is worth noting that all valid hyperparameters are sampled from a uniform distribution and assigned random values during the tuning process.

E. Performance Tuner

ML-Based Cost Model: Auto-tuners [42] provide a means to search for the optimal scheduling of a tensor program from a vast search space. A crucial component of this process is the use of a learned cost model to evaluate the performance of all sampled tensor programs. The cost model is trained on a wide range of extracted features, including arithmetic and memory access features that represent the number of floating-point and integer operations, vectorization, unrolling, parallelization, buffer access, allocation, and GPU thread binding-related features. We adopt the same feature extraction scheme as that used in Ansor [28]. The loss function of the prediction model f on a set of sampled programs P with throughput y is defined as the weighted squared error. Specifically, the loss function is given as

$$\text{loss}(f, P, y) = w_p \left(\sum_{s \in S(P)} f(s) - y \right)^2 \tag{4}$$

where $S(P)$ denotes a set of innermost nonloop statements in P. To predict the performance of the sampled tensor programs, we train a gradient-boosting decision tree [36] as the underlying prediction model f. In the training process, we set y to be approximately equal to w for the actual calculation, which is consistent with Ansor's approach.

Evolutionary Search: To collect training data, a search policy is required for the performance tuner. Evolutionary search is utilized, which repeatedly generates a new set of candidates through `mutation` and `crossover` mechanisms for multiple iteration rounds during the search. The objective of the performance tuner is to select a set of tensor programs with the highest prediction scores for optimization. If a generated tensor program has a higher prediction score, it indicates that it will run faster on the platform. The generated tensor programs are compiled and measured on the actual GPU platform to obtain the execution time as the training labels. In addition, the collected data with the highest prediction scores from the previous training is incorporated into the training dataset to enhance the quality of the cost model.

V. Hardware Abstraction and Mapping Exploration on Tensor Cores

A. *Domain Specific Accelerators on GPU*

The recent advancements of GPU hardware technology have resulted in a significant increase in computing power, particularly with the introduction of the Tensor Cores on NVIDIA GPUs. Unlike the scalar-to-scalar primitives found in CPUs or the general CUDA Cores in GPUs, Tensor Cores provide specialized tensor computation capacities, which can deliver over 10× higher throughput. Notably, the initial version of Tensor Core is designed for handling the GEMM with half-precision input and full-precision output. Recently, new features supporting different datatypes, such as `int8`, `int4`, and `int1` input variables, have been introduced in the latest architecture (Truing and Ampere). Listing 1 demonstrates several essential instructions utilized in NVIDIA GPU Tensor Cores. It is worth noting that Tensor Cores are capable of executing fused-multiply–add (FMA) operations in each instruction cycle. These FMA operations process input values in half-precision, while the output values can be in either half-precision (`FP16`) or full-precision (`FP32`).

Tensor Cores enable the computation primitive of $\vec{D} = \alpha(\vec{A} \times \vec{B}) + \beta\vec{C}$, where tiling of matrix $\vec{A}$ and $\vec{B}$ is required to be a certain type of precision, while the type of matrix $\vec{C}$ and $\vec{D}$ are also be determined. The shape of tiling $\vec{A}(M \times K)$ and $\vec{B}(N \times K)$ may have multiple configurations which depend on the input data precision and GPU architecture. Unlike CUDA Cores, which require users to define the execution flow of each thread, Tensor Cores only require the collaboration of a warp of threads. Using Tensor Cores for computation involves the following steps.

1) Before calling Tensor Cores, all registers of a warp of threads collaboratively store the tiling into a memory component called `fragment`, which allows for data sharing across all registers. The intrawarp sharing mechanism provides opportunities for fragment-based memory optimizations.
2) The loaded matrix fragment components serve as the input variables of the Tensor Cores to generate the output fragment, which also consists of the registers from each thread in a warp and data movements among these registers are also managed collaboratively by a warp of threads.

In this section, we present a compilation-based approach to optimize tensorization programs and introduce the abstractions utilized in our design. Our primary aim is to convert high-level tensor expressions into low-level hardware intrinsics with optimal inference performance. We define the register-level abstraction using hierarchical mapping, which is divided into two categories: computation and data movement (memory) abstractions. The purpose of these abstractions is to formalize the behavior of domain-specific accelerators, which enables our system to automatically analyze and optimize different compute-intensive workloads. In this work, we focus solely on the NVIDIA Turing 2080Ti GPU with the `FP16` datatype via Tensor Cores. Turing architecture supports two common matrix multiplication shapes from the instruction-level parallelism, with 8 × 8 × 4 and 16 × 8 × 8. An overview of our framework can be found in Fig. 8. We illustrate the whole process by using the matrix multiplication operator in Fig. 9 and demonstrate how to define the abstractions, generate tensorization candidates, and explore the optimal mapping step-by-step on the Tensor Cores. Tensor Cores are supported by different architectures, such as Volta, Turing, and Ampere, and they can handle various datatypes, such as `TF32`, `FP16`, `INT8`, `INT4`, and `INT1`.

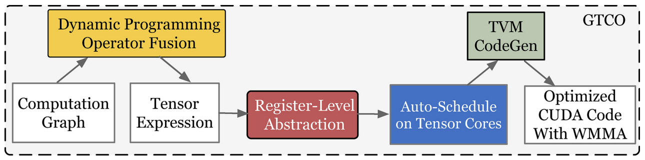
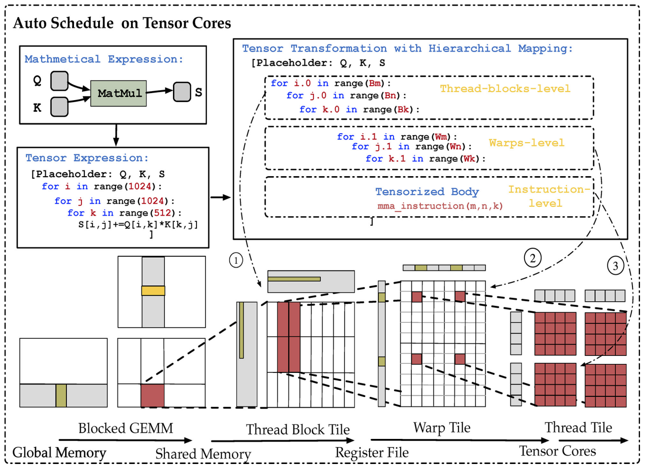

GTCO
Dynamic Programming Operator Fusion
TVM CodeGen
Computation Graph
Tensor Expression
Register-Level Abstraction
Auto-Schedule on Tensor Cores
Optimized CUDA Code With WMMA

Fig. 8. *Register-level abstraction* is defined to enable optimal configuration of tensor computation with WMMA instructions on Tensor Cores. The Input of the GTCO is the computation graph extracted from a deep learning framework. The *DPOF* technique, as introduced in [1] with Section IV-B, is utilized for graph-level optimization. With *register-level abstraction*, the original tensor expressions can be encoded with hardware intrinsic. Subsequently, a tensorization-aware auto-schedule, which includes code generation is developed to generate high-performance tensor programs on Tensor Cores.



```
[Placeholder: Q, K, S
  for i in range(1024):
    for j in range(1024):
      for k in range(512):
        S[i,j]+=Q[i,k]*K[k,j]
                          ]
```

```
[Placeholder: Q, K, S
  for i.0 in range(Bm):
    for j.0 in range(Bn):
      for k.0 in range(Bk):
            for i.1 in range(Wm):
              for j.1 in range(Wn):
                for k.1 in range(Wk):
                  mma_instruction(m,n,k)
  ]
```

Fig. 9. During auto-tuning, the matrix multiplication is mapped to Tensor Cores with hardware intrinsic via a hierarchical mapping described in Algorithm 3. It involves three levels of optimization, namely, thread-blocks, warps, and instruction-level. It employs six parameters (B_m, B_n, B_k, W_m, W_n, and W_k) and two sets of WMMA instruction for tensor transformation. Additionally, the double buffering technique is employed during kernel execution. It is worth noting that the primary difference between GTCO and [1] is that the former utilizes more comprehensive optimization techniques that span across all three levels of parallel programming models, while the latter only uses thread-blocks-level optimization with shared memory on CUDA cores.

B. *Standard Attention Implementation on CUDA Cores*

In the previous Section II, we presented an overview of the transformer-based model. To compute the attention output for input sequences $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, where N is the number of tokens in the input and d is the head dimensions in the MHA, the following three steps are performed: 1) the matrix multiplication $\boldsymbol{S} = \boldsymbol{Q}\boldsymbol{K}^\top \in \mathbb{R}^{N\times N}$ is computed; 2) the intermediate result $\vec{P} = \mathrm{softmax}(\boldsymbol{S}) \in \mathbb{R}^{N\times N}$ is calculated; and 3) the final attention output $\vec{O} = \vec{P}\vec{V} \in \mathbb{R}^{N\times d}$ is obtained. Note that the softmax operation is performed row-wise. During the computation, the matrices $\boldsymbol{S}$ and $\vec{P}$ are stored in the global memory of the GPU. However, since some or most of the operations in the MHA are memory-intensive, such as the softmax operation, the large number of memory accesses can lead to slow execution times. To illustrate this point, we provide Algorithm 2, which describes the standard attention implementation using the load-compute-store pipeline in the GPU programming model.

Algorithm 2 Standard Attention Implementation in the MHA

```
Input: Input Matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N\times d}$ in the global memory.
Output: Output Matrics $\boldsymbol{O} \in \mathbb{R}^{N\times d}$
1: Load $\boldsymbol{Q}$, K with thread blocks from the global memory;
2: Compute $\boldsymbol{S} = \boldsymbol{Q}\boldsymbol{K}^\top$ with CUDA cores;
3: Store $\boldsymbol{S}$ back to the global memory;
4: Read $\boldsymbol{S}$ from the global memory;
5: Compute $\boldsymbol{P} = \mathit{softmax}(\boldsymbol{S})$ with the CUDA cores;
6: Write $\boldsymbol{P}$ back to the global memory;
7: Load $\boldsymbol{P}$ and $\boldsymbol{V}$ with thread blocks form the global memory;
8: Compute $\boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$ with the CUDA cores;
9: Store $\boldsymbol{O}$ back to the global memory;
```

Memory Access Overheads: Memory-aware design involves carefully managing the movement of data between the different levels of hierarchical memory on a GPU. Since most operations in transformer-based models are memory-bound [43], it is crucial to implement memory-aware optimizations to achieve an efficient performance. However, popular frameworks like TensorFlow, PyTorch, and TVM lack the necessary fine-grained control for memory-bound optimization. A computation paradigm that can compute exact operators in transformer-based models with fewer data movements would be highly beneficial. Such a paradigm would reduce the number of required memory accesses and improve overall performance. Operation fusion, which involves using the output values from one operation, such as a matrix multiplication layer as the input directly to the following operation such as a softmax layer, without writing intermediate values to off-chip memory, is one way to achieve this optimization.

Opportunities: Given the inherent limitations of the standard attention mechanism, it is imperative to minimize the number of memory accesses to improve performance on GPUs. However, a naive approach may lead to increased data movement overheads between on-chip and off-chip memory, even with the register files. Fortunately, the MHA computation process can be deconstructed into distinct steps, allowing the softmax reduction to be computed incrementally without accessing the entire input. As such, the attention computation can be restructured by partitioning the input into blocks and performing several passes over these blocks. Our approach leverages a compiler to enable precise control over memory access and combines the matrix multiplication and softmax operations into a single kernel using specialized hardware intrinsic to accelerate inference.

C. Register-Level Abstraction

Regarding computation abstraction, we utilize the function to represent arithmetic operations, such as addition and multiplication operators. Specifically, we represent Tensor Core computations as `mma_sync` intrinsic, which is capable of computing a matrix multiplication for a special shape. The abstraction can be defined as follows:

$$\boldsymbol{dst}[m, n] = \texttt{multiply}(\boldsymbol{src_1}[m, k]\boldsymbol{src_2}[k, n]). \tag{5}$$

Register-level abstraction is a list of statements that specify the scope, operands, and memory access indices. The notation used for this abstraction includes $\boldsymbol{dst}$ to represent the output tensor, and $\vec{src}_{1,2}$ for the input tensors. The terms "global," "shared," and "register" are used to indicate the different memory hierarchies, while $\vec{i}$, $\vec{j}$, and $\vec{k}$ denote the indices for tensor storage in each hierarchy. Specifically, $\vec{i}$ refers to the indices in the global memory, $\vec{j}$ in the shared memory, and $\vec{k}$ in the register files. It is important to note that the same operand may have different indices within different memory scopes. For example, the intrinsic `load_matrix_sync` is used to load data from shared memory to register files, while the intrinsic for storing data from registers to global memory can be expressed in the same way. The whole process of the data movement can be formulated as follows:

$$\mathit{shared.Src}\left[\vec{j}\right] = \mathit{global.Src}\left[\vec{i}\right] \tag{6}$$

$$\mathit{reg.Src}\left[\vec{k}\right] = \mathit{shared.Src}\left[\vec{j}\right] \tag{7}$$

$$\mathit{global.Src}\left[\vec{i}\right] = \mathit{reg.Src}\left[\vec{k}\right]. \tag{8}$$

Both the computation and memory abstractions are designed to capture the behavior of hardware intrinsics with the behavior of Tensor Cores.

Algorithm 3 Optimized Implementation With Tensor Cores

```
Input: Input Matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N\times d}$ in the global memory, shared
    memory of size $mem_1$ and register files of size $mem_2$.
Output: Output Matrics $\boldsymbol{O} \in \mathbb{R}^{N\times d}$.
1: Divide $\boldsymbol{Q}$ into a numbers of blocks $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_i, \ldots, \boldsymbol{Q}_{\frac{mem_1}{4d}}$;
2: Divide $\boldsymbol{K}$ into a numbers of blocks $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_i, \ldots, \boldsymbol{K}_{\frac{mem_1}{4d}}$;
3: Load $\boldsymbol{Q}_i$ and $\boldsymbol{K}_i$ with blocks from global memory to shared
    memory and store in the fragment;
4: Divide $\boldsymbol{Q}_i$ into blocks $\boldsymbol{Q}_{i1}, \boldsymbol{Q}_{i2}, \ldots, \boldsymbol{Q}_{ij}, \ldots, \boldsymbol{Q}_{i\frac{mem_2}{4d}}$;
5: Divide $\boldsymbol{K}_i$ into blocks $\boldsymbol{K}_{i1}, \boldsymbol{K}_{i2}, \ldots, \boldsymbol{K}_{ij}, \ldots, \boldsymbol{K}_{i\frac{mem_2}{4d}}$;
6: Load $\boldsymbol{Q}_{ij}$ and $\boldsymbol{K}_{ij}$ from the fragment to the register files;
7: Compute $\boldsymbol{S}_{ij} = \boldsymbol{Q}_{ij}\, \boldsymbol{K}_{ij}^\top$ with the Tensor cores;
8: Compute the max value of row in $\boldsymbol{S}_{ij}$ via $\boldsymbol{m}_{ij}$;
9: Compute the $\boldsymbol{p}_{ij} = \exp(\boldsymbol{S}_{ij} - \boldsymbol{m}_{ij})$ for softmax;
10: Compute the sum value of row via $\boldsymbol{l}_{ij} = \sum \boldsymbol{p}_{ij}$;
11: Compute $\boldsymbol{P} = \mathit{softmax}(S)$ with $\boldsymbol{m}_{ij}$, $\boldsymbol{p}_{ij}$ and $\boldsymbol{l}_{ij}$;
12: Write $\boldsymbol{P}$ back to the shared memory and store in the fragment;
13: Load $\boldsymbol{V}$ to the shared memory and store in the fragment;
14: Compute $\boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$ with the tensor cores;
15: Store $\boldsymbol{O}$ back to the global memory;
```

D. Auto-Schedule on Tensor Cores

To generate the abstraction with hardware intrinsics, we propose a hierarchical mapping approach. First, we map the software iterations to a virtual accelerator component composed of the load/store unit and computation units, without considering memory allocation or computation resource assignment. Next, we consider constraints, such as memory capacity and hardware intrinsic with six parameters, as shown in Fig. 9. These parameters can represent three-level parallel optimization on the GPU with Tensor Cores. The first three parameters B_m, B_n, and B_k can map the computation on the thread-block-level concurrency. The last three parameters, W_m, W_n, and W_k represent warp-level optimization and parallelism in the implementation of the main loop. In the Tensorized Body component, the shapes of the WMMA instruction and datatype differ between GPU architectures. For this work, we focus solely on the Turing architecture with `FP16` datatype, with the option of using $8 \times 8 \times 4$

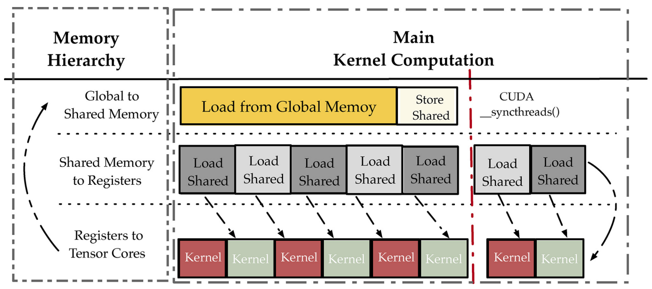


Fig. 10. Fusion of softmax and matrix multiplication kernel computation with data movement across a complex memory hierarchy is implemented with the double buffering technique to improve overall execution efficiency.

and 16 × 8 × 8 shapes. In order to obtain a valid physical mapping from software iterations to the domain-specific accelerators with a more complex memory hierarchy, we propose modifications to the previous auto-schedule introduced in [1]. We present a practical example to illustrate our hierarchical mapping approach for matrix multiplication on Tensor Cores. Specifically, we consider the computation $\boldsymbol{S}[m, n] +\!= \boldsymbol{Q}[m, k] \times \boldsymbol{K}[k, n]$, which can be easily extended to the batch matrix multiplication used in ViT models on Tensor Cores.

The initial step assumes that GTCO is capable of loading data of arbitrary volume into on-chip memory (register files and shared memory), and has sufficient hardware resources to directly perform all the tensor computations. The primary challenge at this stage is to map software-defined tensor operations to their corresponding hardware intrinsic.

The second step involves considering two types of constraints: memory capacity and intrinsic size. Hardware accelerators have a fixed capacity for computing results at any given time, which is limited by the problem size of an intrinsic extracted from its indices range represented in the computation abstraction. In our running example, the matched software iterations are limited by a factor of 4 due to the problem size of the example Tensor Core shapes being 8 × 8 × 4 and 16 × 8 × 8. ①, ②, and ③ show the proposed mapping across different memory hierarchies from high-level mathematical expression to low-level hardware intrinsic. Step ① represents that the original matrix is divided into tiling data, which are loaded from the global memory to the shared memory and then stored in the fragment with thread block. It corresponds to the operation defined in (6). Step ② describes the process of moving tiling data from the fragment to the register files for the computation, which corresponds to (7). Step ③ indicates that performing matrix multiplication within one clock cycle, using the tiling data stored in the register and scaling it according to the dimensions ($m \times n \times k$) specified by the WMMA instruction. It corresponds to the data movement operations defined in (8). To alleviate the impact of memory latency, software pipelining is utilized to overlap memory accesses with other computations within a thread. The pipelining stage is set to 2 throughout the process. Further details on the pipelining of various kernel fusions, such as softmax and matrix multiplication operators in transformer-based models, can be found in Fig. 10.

Thus, in order to identify the optimal implementation, valid software–hardware mappings need to be selected from the design search space. However, when optimizing these schedules with tiling or parallelization primitives on domain-specific accelerators, it is challenging to determine the performance of these mappings. The reason is that these optimizations, combined with different software–hardware mappings, exhibit varying performance due to differences in computation and memory utilization. Moreover, the search space for performance tuning is too vast. Therefore, to address this challenge, we employ a combination of tuning techniques and an ML-based cost model to explore the mapping and schedule space. For a more detailed description of the performance tuner design, please refer to Section IV-E. It enables our computation and data movement in one kernel, loading input from global memory, performing all the computation operations, such as softmax and matrix multiplication on the register-level files, then writing the result back to global memory. It can avoid repeatedly reading and writing inputs and outputs with the memory access overhead. More details about the proposed optimization technique can be found in Section V-C.

VI. Experimental Results

A. Experimental Setup

Our framework is implemented on top of PyTorch and the HuggingFace Transformers library. We evaluate the effectiveness of our approach, such as operator fusion optimization, kernel generation mechanisms, and Tensor Cores acceleration on three modern vision transformers: 1) ViT [11] for image classification; 2) DETR [8] for object detection; and 3) SETR [10] for semantic segmentation. The DETR and ViT pretrained models are downloaded from the Hugging Face datasets hub. The SETR models are downloaded from the original GitHub repo [44]. PyTorch 1.7.1, CUDA 10.0, cuDNN V7.6.5, NVIDIA driver 460.67, TVM 0.8.dev0 [27], and TensorRT V7.0.0.11 [29] are set as baselines for fair comparisons. Note that all evaluation results are collected on 2080Ti GPU by different batch sizes.

Workflow: Our workflow can be categorized into the following two patterns.

1) For the inference engine like TensorRT, PyTorch is used to build the models and then the ONNX export interface is used to export ONNX models. To avoid some redundant operators caused by conversion, ONNX-Simplifier [45] is used to simplify the ONNX model. Finally, the model is converted into an executable file with the CUDA runtime environment.
2) As for the compiler flow like Ansor, [1] and GTCO, the model is converted into the TorchScript format first and then is imported into the Relay interface to read the TorchScript model into our compilation system. In terms of the subgraphs, the corresponding tensor programs are generated by the code generation part. As for the Tensor Cores settings, we only use the basic instructions provided by CUDA, such as `fragment`, `load_matrix_sync`, `mma_sync`, `store_matrix_sync` without extensions of customized instructions.

B. End-to-End Performance

Workloads: The configurations of the models are described in Table I, including the number of encoders, decoders, attention heads, the shape of inputs, and the outputs. All of the results are reported with batch size 1 on an NVIDIA 2080Ti GPU.

Baselines and Configurations: PyTorch JIT [22], TensorRT [29], TVM [27], and Ansor [28] are used as our

TABLE I
DETAILED INFORMATION OF THE BENCHMARK AND EXPERIMENT MODELS

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 NETWORK EXECUTION PERFORMANCE BENCHMARK (MS)

	PyTorch JIT [22]	TVM-CUDA [27]	TVM-cuDNN [27]	TensorRT [29]	Ansor [28]	[1]	GTCO
DETR-ResNet50-E3	18.62	54.73	54.43	6.97	5.85	5.32	**4.18**
DETR-ResNet50-E6	23.67	93.59	88.25	7.73	6.78	5.60	**4.46**
DETR-ResNet50-E12	33.01	171.96	157.97	15.79	14.29	13.18	**11.08**
SETR-Naive	68.26	753.25	742.21	33.71	34.22	28.65	**22.34**
SETR-Naive-Base	31.06	186.13	187.39	16.97	15.44	14.21	**11.45**
SETR-PUP	37.62	199.42	189.21	18.61	17.89	16.01	**12.88**
ViT-Base-16	24.92	91.86	96.31	5.87	8.57	8.43	**5.08**
ViT-Large-16	52.96	329.74	334.38	18.45	18.99	18.41	**14.85**
ViT-Huge-14	76.07	846.87	846.27	34.14	32.53	29.89	**24.08**

baseline frameworks. More specifically, there are two common ways to improve the efficiency of execution time on GPUs. Optimizing operators via the vendor-provided libraries, such as PyTorch and TensorRT, is the first way. On the other hand, another strategy is to use a regression-based model to search the schedule for each kernel such as TVM. In the meantime, TVM also supports calling external libraries such as cuBLAS/cuDNN to optimize some kernels in the computational graph. PyTorch JIT is a just-in-time compiler in PyTorch, which is a way to create serializable and optimizable models from PyTorch code to a production environment where Python programs may be disadvantageous for performance and multithreading reasons. As mentioned in Section IV, the end-to-end execution time can be described as the sum of the latency of all subgraphs in the computation graph. Auto-tuning trails are set to 10 000 measurement unless execution time converges to a stable value. The goal of the subgraph scheduler is to minimize the total execution time. Finally, optimized tensor programs for subgraphs are generated for measurement. In the Turing architecture GPUs, they have two recommended types of GEMM shapes with $8 \times 8 \times 4$ and $16 \times 8 \times 8$. We use $8 \times 8 \times 4$ in all of the experiments and we find that $16 \times 8 \times 8$ is not better than the former after a certain number of experiments.

Results: Table II shows the results on a NVIDIA RTX 2080Ti GPU. In general, [1] surpasses all of the baseline frameworks except the ViT-Base vision model. Compared with vendor-specific engine TensorRT, [1] consistently outperforms all benchmarks with 1.01× to 1.38× speedup except for the ViT-Base vision model. The reason for the drop in execution time is that ViT-Base is composed of a number of encoders, and the input shape $(197, 1, 768)$ of the encoder in ViT-Base is relatively limited compared with ViT-Large and ViT-Huge. Thus, it limits the search space of specific operators in transformer-based models, such as batch matrix multiplication and softmax fusion in MHA. Compared with the tuning-based method [1], [28] outperforms all benchmarks with 1.01× to 1.21× speedup. That is, our framework equipped with operator fusion technique and sketch customization rules has achieved good performance on transformer-based vision models. TVM-cuDNN/BLAS means we use the TVM as the compiler and then call the operators defined in the cuDNN and cuBLAS library to optimize the execution time of each kernel. Compared with TVM-cuDNN/BLAS [1], [27] consistently outperforms all benchmarks with significant speedup. Obviously, TVM-cuDNN/BLAS uses the operator fusion patterns defined in Relay to partition the graph into lots of subgraphs. Each operator in the subgraph is replaced with the implementations in cuDNN/cuBLAS. Neither the search-based optimization for the tensor program nor fine-tuning the performance of each kernel by a regression-based cost model is implemented during the optimization. Therefore, [1] has more advantages on the emerging new operators or uncommon operator fusion patterns because it is not easy

TABLE III
ViT-Base-16 With Different Optimization Settings

Setting	GTCO (a)	(b)	(c)	(d)	(e)	(f)
TC		✓	✓	✓	✓	✓
DPOF			✓	✓	✓	✓
Sketch Customization				✓	✓	✓
Subgraph Scheduler					✓	✓
Auto-Schedule Register						✓
Speedup	1.00×	1.22×	1.44×	1.56×	1.59×	1.68×

for vendor-specific static libraries such as cuDNN/cuBLAS to optimize for all the cases manually. The only difference between TVM-CUDA and TVM-cuDNN/BLAS is that the implementation of each operator in the subgraph is done by the default scheduling template defined in the deep learning compiler. Note that the GTCO means all of the transformer-based vision models are conducted on the Tensor Core with floating-point `16` data type. From the last column in Table II, we can find that our abstraction design and mapping strategies on Tensor Cores for GTCO perform the best in the end-to-end benchmark. In the DETR-ResNet series of vision models, GTCO can achieve up to 1.27 × speedup compared with [1] on 2080Ti GPU. In the SETR series of vision models, GTCO can achieve up to 1.28 × speedup compared with [1]. All in all, our design can fully take advantage of the Tensor Cores on the modern GPU to accelerate the ViT execution with specific datatype `FP16`.

Ablation Study: To explore the function of each module, we run variants of GTCO on the ViT-Base-16 benchmark. "DPOF ✓" means DPOF technique is used to optimize the computation graph from graph-level rather than the template-based method designed in Relay. "Sketch Customization ✓ " means sketch generation rules and search policy defined in GTCO is used to generate the tensor program rather than default configurations defined in Ansor. "Subgraph scheduler ✓" means we use the object function defined in (3) to optimize our auto-tuning. Obviously, *Design (a)* is the native Ansor with Tensor Core support. We set the execution time of *Design (b)–(d)* to be the speedup against Ansor equipped with Tensor Core. As shown in Table III, *Design (e)* performs the best in speedup performance among all of the designs. "TC" means that we only use the Tensor Cores to accelerate the operators in the models without register-level abstraction and hierarchical mapping. "Auto-Schedule Register" represents the proposed register-level abstraction and auto-schedule on Tensor Cores in Section V. It can verify that the graph and tensor co-design is very significant for the transformer-based model execution. At the graph-level, GTCO utilizes the "DPOF" to optimize the operator fusion and then uses a subgraph scheduler designed for the transformer-based model to assign different search tasks with various time slots. At the tensor-level, the tensor programs are generated by our sketch generation rules with search policy. The design of register-level abstraction of computation and memory can boost the final performance on Tensor Cores with hierarchical mapping.

C. Subgraph Benchmark

Baselines and Configurations: Three common subgraphs in DETR-ResNet-50-E6 are conducted to verify the subgraph benchmark, including MHA, Encoder, and Decoder. The measurement trails per test case are set to 20 000 during the

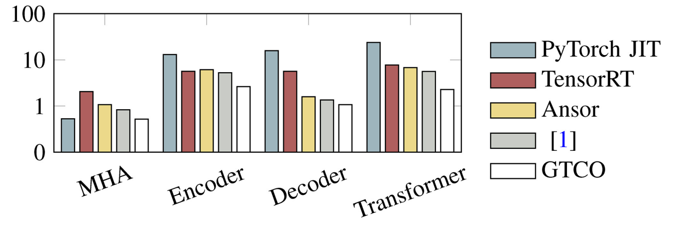


Fig. 11. Subgraph performance benchmark. The y-axis is the throughput-based log 10 and then plus 1.

TABLE V
Number of Measurement Trails

	Ansor [28]	[1]	GTCO
Multi-Head Attention	1,600	1,408	1,264
Encoder-3-Layer	3,008	2,816	2,416
Encoder-6-Layer	4,992	4,096	3,464
Encoder-12-Layer	6,528	5,760	5,022
Decoder-6-Layer	2,688	2,432	1,986
DETR-ResNet-50-E6	8,640	6,784	6,112
Average	100%	87%	**75%**

auto-tuning for Ansor and we use the consumed time in the whole process to demonstrate the final performance. We use the same set of baseline frameworks and run benchmarks with the approximate converged latency.

Results: Fig. 11 shows that [1] outperforms PyTorch JIT on the Encoder and Decoder by 2.47× and 11.67× speedup. For the high-performance computing library TensorRT, [1] can achieve 2.47×, 1.08×, and 4.19× speedup on MHA, Encoder, and Decoder. For the compiler-based search algorithm Ansor, GTCO can achieve 1.29×, 1.17×, and 1.17× speedup on MHA, Encoder, and Decoder. It can prove that [1] can generate efficient tensor programs for these subgraphs on the NVIDIA GPU platform. Meanwhile, GTCO performs best in all subgraph benchmarks compared with Pytorch-JIT, TensorRT, Ansor, and [1] under the same inference configuration. GTCO can achieve 1.83×, 1.44×, and 1.29× speedup on MHA, Encoder, and Decoder compared to [1].

D. Graph Partition and Tuning Time

The graph partition on the DETR-ResNet50-E6 benchmark is shown in Table IV. "n_i" means the number of subgraphs in the encoder, decoder, and transformer models, respectively. The meaning of "Weight-*" can be explained with two important values. For example, $\{[6 * 7], [12 * 2]\}$ means there are seven subgraphs with weight value 6 and 2 subgraphs with weight value 12. Therefore, the total number of subgraphs in the encoder is 9. Compared to the rule-based method in Ansor, we can find that our graph partition and subgraph scheduler methods can achieve a more effective operation fusion strategy for the number of subgraphs and weight values. In addition, GTCO can not change the partition methods defined in our optimization on encoders and decoders compared with [1] from Table IV, and it only accelerates the runtime on the Tensor Cores.

Table V shows the search time needed for [1] and GTCO to match the execution time of Ansor on the same benchmark."number of measurement trails" are used to evaluate the search time. From the table, GTCO can match the performance of Ansor with fewer measurement trails. It can prove that the efforts saving in search time come from the techniques we introduced before including a subgraph scheduler, the DPOF at

TABLE IV
INFORMATION OF SUBGRAPHS AND SCHEDULING WEIGHTS WITH GRAPH PARTITION

	n_1	Weight-Encoder	n_2	Weight-Decoder	n_3	Weight-Transformer
Ansor [28]	9	$\{[6*7],[12*2]\}$	13	$\{[6*10],[18*3]\}$	22	$\{[6*17],[13*2],[19*2],[18*1]\}$
[1]	6	$\{[8*4],[10*2]\}$	11	$\{[8*6],[20*5]\}$	17	$\{[9*12],[20*3],[16*2]\}$
GTCO	6	$\{[8*4],[10*2]\}$	11	$\{[8*6],[20*5]\}$	17	$\{[9*12],[20*3],[16*2]\}$

TABLE VI
TIME USED IN THE TOTAL COMPILATION PHASE

Network	batch	[1]	GTCO	Ratio
DETR-ResNet50-E3	1	1,625	1,430	88%
	4	3,652	3,324	91%
	8	6,322	5,436	86%
DETR-ResNet50-E6	1	1,825	1,624	89%
	4	4,314	3,796	88%
	8	6,792	6,112	90%
DETR-ResNet50-E12	1	1,997	1,698	85%
	4	4,798	4,220	87%
	8	7,001	5,880	84%
SETR-Naive	1	2,158	1,770	82%
	4	4,334	3,640	84%
	8	6,698	5,626	84%
SETR-PUP	1	4,160	3,452	83%
	4	5,026	4,272	84%
	8	8,298	6,804	82%
ViT-Base-16	1	1,682	1,312	78%
	4	4,146	3,358	81%
	8	7,019	5,826	83%
ViT-Large-16	1	2,767	2,214	80%
	4	5,310	4,354	82%
	8	7,631	6,410	84%
ViT-Huge-14	1	3,158	2,748	87%
	4	6,620	5,564	84%
	8	9,012	7,388	82%

the graph level, the sketch generation rules for tensor programs generation, and register-level abstraction with hierarchical mapping on Tensor Cores. From the average performance of six benchmarks, we can find that GTCO outperforms the Ansor and [1]. CTCO can generate high-performance tensor programs on Tensor Cores with less effort.

Table VI shows the total compilation time needed for GTCO to match the nearly uniform latency performance of [1] on the end-to-end performance. All the data used in this table are recorded in seconds. We use "Ratio" to demonstrate the efficiency of our compilation technique. Compared with the time `1,625` s, the total compilation time used in GTCO with DETR-ResNet-E3 is `1,430` s. If we use the GTCO with `FP16` datatype on Tensor Cores, we can get the final latency performance compared with the dense CUDA Core version of GTCO in 88% of the total time. From all of the experimental results in Table VI, we can find that the techniques used in the hardware abstraction and automatic mapping on Tensor Cores can accelerate the optimization time during the total compilation phase among all of the ViT models.

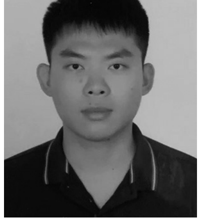
VII. CONCLUSION

Existing deep learning compilers optimize operator fusion based on the rule designed by experts, which is strictly improving execution performance for the new operators on hardware platforms. However, they fail to consider the potential performance improvements that more effective operator fusion strategies could provide. This article addresses this issue by tackling the problem from two perspectives. First, a dynamic programming algorithm is introduced to explore operator fusion patterns. Second, a search policy is proposed that includes new sketch generation rules and a novel hardware abstraction with register-level optimization, enabling more flexible mapping for tensor computation and better performance. This approach is applied to optimize fused matrix multiplication and softmax operators with WMMA instructions. To achieve an end-to-end flow automatically, a regression-based learned model is used to fine-tune the performance of each kernel. Overall, GTCO achieved up to 1.73× inference speedups compared to the high-performance inference engine TensorRT with Tensor Cores, and 1.38× speedups with CUDA Cores.

REFERENCES

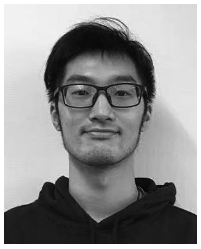
[1] Y. Bai, X. Yao, Q. Sun, and B. Yu, "Autogtco: Graph and tensor co-optimize for image recognition with transformers on GPU," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, 2021, pp. 1–9.
[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016, pp. 770–778.
[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015, pp. 1–14.
[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. CVPR*, 2016, pp. 2818–2826.
[5] Y. Bai and W. Wang, "ACPNet: Anchor-center based person network for human pose estimation and instance segmentation," in *Proc. ICME*, 2019, pp. 1072–1077.
[6] A. Vaswani et al., "Attention is all you need," in *Proc. NIPS*, 2017, pp. 1–15.
[7] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," 2021, *arXiv:2101.01169*.
[8] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *Proc. ECCV*, 2020, pp. 213–229.
[9] M. Caron et al., "Emerging properties in self-supervised vision Transformers," 2021, *arXiv:2104.14294*.
[10] S. Zheng et al., "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers," in *Proc. CVPR*, 2021, pp. 1–10.
[11] A. Dosovitskiy et al., "An image is worth 16×16 words: Transformers for image recognition at scale," 2020, *arXiv:2010.11929*.
[12] 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 *Proc. DAC*, 2020, pp. 1–6.
[13] G. Huang et al., "ALCOP: Automatic load-compute pipelining in deep learning compiler for AI-GPUs," in *Proc. Mach. Learn. Syst.*, 2023, pp. 1–15.
[14] W. Zhao et al., "A high-performance accelerator for super-resolution processing on embedded GPU," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 10, pp. 3210–3223, Oct. 2023.
[15] Q. Sun et al., "GTuner: Tuning DNN computations on GPU via graph attention network," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, 2022, pp. 1045–1050.
[16] Y. Zhao, Q. Sun, Z. He, Y. Bai, and B. Yu, "AutoGraph: Optimizing DNN computation graph for parallel GPU kernel execution," in *Proc. AAAI Conf. Artif. Intell.*, vol. 37, 2023, pp. 11354–11362.
[17] S. Zheng et al., "AMOS: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *Proc. ISCA*, 2022, pp. 874–887.
[18] C. Hao et al., "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proc. DAC*, 2019, pp. 1–6.

[19] 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 J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.
[20] Y. Jia et al., “Caffe: Convolutional architecture for fast feature embedding,” in *Proc. Multimedia*, 2014, pp. 675–678.
[21] M. Abadi et al., “TensorFlow: A system for large-scale machine learning,” in *Proc. OSDI*, 2016, pp. 265–283.
[22] A. Paszke et al., “Automatic differentiation in PyTorch,” in *Proc. NIPS Workshop*, 2017, pp. 1–4.
[23] “OneAPI deep neural network library.” 2023. [Online]. Available: https://github.com/oneapi-src/oneDNN#documentation
[24] “ARM-compute-library.” 2023. [Online]. Available: https://github.com/arm-software/ComputeLibrary
[25] S. Chetlur et al., “cuDNN: Efficient primitives for deep learning,” 2014, *arXiv:1410.0759*.
[26] 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 Trans. Graph.*, vol. 31, no. 4, p. 32, 2012.
[27] T. Chen et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. OSDI*, 2018, pp. 579–594.
[28] L. Zheng et al., “Ansor: Generating high-performance tensor programs for deep learning,” in *Proc. OSDI*, 2020, pp. 863–879.
[29] “NVIDIA TensorRT,” 2023. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/index.html
[30] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proc. ICML*, 2015, pp. 448–456.
[31] A. F. Agarap, “Deep learning using rectified linear units (ReLU),” 2018, *arXiv:1803.08375*.
[32] 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 *Proc. SOSP*, 2019, pp. 47–62.
[33] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, “IOS: Inter-operator scheduler for CNN acceleration,” in *Proc. MLSys*, vol. 3, 2021, pp. 1–14.
[34] D. Hendrycks and K. Gimpel, “Gaussian error linear units (GELUs),” 2016, *arXiv:1606.08415*.
[35] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” 2019, *arXiv:1810.04805*.
[36] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proc. KDD*, 2016, pp. 785–794.
[37] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-attention with linear complexity,” 2020, *arXiv:2006.04768*.
[38] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao, “Post-training quantization for vision transformer,” in *Proc. NIPS*, 2021, pp. 1–12.
[39] F. Yu, K. Huang, M. Wang, Y. Cheng, W. Chu, and L. Cui, “Width & depth pruning for vision transformers,” in *Proc. AAAI*, 2022, pp. 3143–3151.
[40] S. Lin et al., “Knowledge distillation via the target-aware transformer,” 2022, *arXiv:2205.10793*.
[41] L. J. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016, *arXiv:1607.06450*.
[42] T. Chen et al., “Learning to optimize tensor programs,” in *Proc. NIPS*, 2018, pp. 3393–3404.
[43] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data movement is all you need: A case study on optimizing transformers,” 2021, *arXiv:2007.00072*.
[44] “SEgmentation TRansformers—SETR.” 2023. [Online]. Available: https://github.com/fudan-zvg/SETR
[45] “ONNX simplifier.” 2023. [Online]. Available: https://github.com/daquexian/onnx-simplifier



Yang Bai received the B.S. degree in telecommunications engineering from Xidian University, Xi’an, China, in 2017, and the master’s degree in computer science from the Chinese Academy of Sciences, Beijing, China, in 2020. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests focus on the optimization for deep neural network training and inference via compilation techniques.



Xufeng Yao received the B.Eng. degree in information system and information management from Fudan University, Shanghai, China, in 2016, and the M.Sc. degree in computer science from The Chinese University of Hong Kong, Hong Kong, in 2020, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering.

His research interests include computer vision and machine learning.

Qi Sun (Member, IEEE) received the Doctoral degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, in 2022.

In 2022, he was a Postdoctoral Researcher with Cornell University, Ithaca, NY, USA. He is currently a ZJU100 Young Professor with Zhejiang University, Hangzhou, China. His research areas include ML for EDA and DNN acceleration.

Dr. Sun has been awarded the ICCAD 2021 Best Paper Award and the DATE 2021 Best Paper Nomination Award.

Wenqian Zhao received the B.Sc. degree in computer science and engineering from The Chinese University of Hong Kong, Hong Kong, in 2019, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering.

His research interests include machine learning for VLSI design automation and hardware-aware deep-learning acceleration.

Shixin Chen received the B.Eng. degree in VLSI design and system integration from Nanjing University, Nanjing, China, in 2022. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests include agile hardware design, hardware accelerator, 3-D IC and chiplet, and machine learning in EDA.

Zixiao Wang received the B.Eng. degree in automation and the M.Sc. degree in computer science from Tsinghua University, Beijing, China, in 2019 and 2022, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests include generative AI $\times$ EDA.

Bei Yu (Senior Member, IEEE) received the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 2014.

He is currently an Associate Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

Dr. Yu received nine Best Paper Awards from DATE 2022, ICCAD 2021 and 2013, ASPDAC 2021 and 2012, ICTAI 2019, *Integration, the VLSI Journal* in 2018, ISPD 2017, and SPIE Advanced Lithography Conference 2016, and six ICCAD/ISPD contest awards. He has served as the TPC Chair of ACM/IEEE Workshop on Machine Learning for CAD, and in many journal editorial boards and conference committees. He is an Editor of IEEE TCCPS Newsletter.