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

Yang Bai\textsuperscript{1}, Xufeng Yao\textsuperscript{2}, Qi Sun\textsuperscript{1}, Bei Yu\textsuperscript{1}

\textsuperscript{1}The Chinese University of Hong Kong
\textsuperscript{2}SmartMore
\{ybai,byu\}@cse.cuhk.edu.hk

Nov. 1, 2021
Introduction
• Deep Learning Models

LSTM

NasNet

Normal Cell

graph neural network
• Modern Accelerators

NVIDIA GPU

AMD GPU

Google TPU

Graphcore IPU
• Fuses kernels – Vertically (Conv, BN, ReLU) and Horizontally (Reuse Inputs)
Frameworks
High Level Graph Rewriting
Machine Learning Based
Automated Optimizer
Optimized Computational Graph

Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation

Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives

Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module
Related Work and Background
Image Recognition in Computer Vision Tasks (CS231n)
Image Classification

The architecture of Vision-Transformer (ViT, ICLR 2021)
The architecture of DEtection-TRansformer (DETR, ECCV 2020)
The architecture of SEgmentation-TRansformer (SETR, CVPR 2021)
Scaled Dot-Product Attention

- MatMul
- SoftMax
- Scale
- MatMul

Query → Key → Value

Multi-Head Attention

- Linear
- Concat
- Scaled Dot-Product Attention
- Linear
- Linear
- Linear

Query → Key → Value

Scaled Dot-Product and Multi-Head Attention (MHA).
A streaming multiprocessor and the memory architecture of GeForce RTX 2080 Ti GPU.
Problem Formulation
Computate 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
- injective
- reduction
- complex-out-fusable
- element-wise
- opaque

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

\[
S = \{(V_1, F_1), (V_2, F_2), \ldots, (V_k, F_k)\},
\]

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

$$S^* = \arg\min_S \text{Cost}(G, S),$$

(2)

where $\text{Cost}$ is the latency of executing $G$ according to the schedule $S$.

- Multi-Head Attention Function:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

(3)

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

(4)

- Transformers have lots of softmax operators in Multi-Head Attention and can be fused with batch matrix multiplication operators
Overview of our system
The Proposed System

The arrows show the flow of the optimized subgraphs from transformer model and tensor programs generation on GPU platform.
Our tensor generation framework is composed of four important modules:

1. Dynamic Programming-based Operator Fusion (DPOF)
2. Subgraph Scheduler
3. CUDA Program sampler
4. Performance Tuner
• Input: A transformer-based model without any operator fusion
• Output: Operators with new tags
• Function: A DPOF that finds an optimized operator fusion schedule for the transformer model
The structure of DPOF

1. **Operator Arrangement**
   - topological sort to get operators
   - queue to store operators
   - compute-type, no placeholder-type operators
   - size of queue = maximum number of queue

2. **Operator Fusion**

   \[
   dp[V] = \min(dp[V - V'] + temp[V'])
   \]

   Schedule of \(V\)
   Schedule of \(V - V'\) (sub-problem)

   Latency of \(V'\)
• Input: A transformer-based model with operator fusion
• Output: Lots of subgraphs decomposed by compute-intensive operators
• Function: A subgraph scheduler that allocates time resources for optimizing multiple subgraphs generated by the DPOF
• **Input:** Subgraph with fused operators
• **Output:** CUDA kernel code for these operators
• **Function:** A program sampler that delineates a large search space and randomly samples various programs from it
1 Sketch Generation

2 Annotation

Generated Sketch 1:
[Placeholder: A, B
for i0 in range(None):
for j0 in range(None):
  ...
for ic2 in range(None):
for jc2 in range(None):
  ...
for k.0 in range(None):
  for k.1 in range(None):
    for k.2 in range(None):
      for k.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.1 in range(0, 1050):
    T_softmax_exprsum = ...
  for i.2 in range(0, 1050):
    T_softmax_norm = ...
for i.3 in range(0, 1050):
  for j.0 in range(None):
    for k.0 in range(0, 1050):
      T_softmax_maxelem = ...
    for k.1 in range(None):
      ...
    for k.2 in range(None):
      ...
    for k.3 in range(None):
      ...
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.1 in range(0, 1050):
    T_softmax_exprsum = ...
  for i.2 in range(0, 1050):
    T_softmax_norm = ...
for i.3 in range(0, 1050):
  C = ...
]

The mathematical expressions:
\[ m[i, j] = \sum_k A[i, k] \times B[k, j] \]
\[ D[i, j] = SoftMax(m[i, j]) \]

Generated Sketch 2:
[Placeholder: A, B
for i.0 in range(None):
for j.0 in range(None):
  ...
for ic2 in range(None):
for jc2 in range(None):
  ...
for k.0 in range(None):
  for k.1 in range(None):
    for k.2 in range(None):
      for k.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.1 in range(0, 1050):
    T_softmax_exprsum = ...
  for i.2 in range(0, 1050):
    T_softmax_norm = ...
for i.3 in range(0, 1050):
  C = ...
]

Our Sketch Customization and Policy

CUDA Program sampler
• Input: Sampled CUDA kernel codes
• Output: The performance of the generated code
• Function: A performance tuner that trains a cost model to measure the performance of sampled tensor programs
Evaluation Results
1 Image Recognition Models
   - DETR for Object Detection
   - SETR for Semantic Segmentation
   - ViT for Image Classification

2 WorkFlow
   - TensorRT: PyTorch $\rightarrow$ ONNX $\rightarrow$ ONNX-Simplifier $\rightarrow$ TensorRT Engine
   - AutoGTCO: PyTorch $\rightarrow$ TorchScript $\rightarrow$ Relay $\rightarrow$ Code Generation

3 WorkLoads
   - Batch Size=1
### Architecture of the Model and Configurations

<table>
<thead>
<tr>
<th>Model</th>
<th>ec</th>
<th>dc</th>
<th>Width</th>
<th>Mlp-dim</th>
<th>Nh</th>
<th>Input Shape</th>
<th>Patch</th>
<th>Mha Input</th>
<th>Encoder Input</th>
<th>Decoder Input</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETR-ResNet50-E3</td>
<td>3</td>
<td>6</td>
<td>256</td>
<td>2048</td>
<td>8</td>
<td>[1,3,800,1333]</td>
<td>N/A</td>
<td>query[1050,1,256] &lt;br&gt; key[1050,1,256] &lt;br&gt; value[1050,1,256]</td>
<td>src[1050,1,256]</td>
<td>tgt[100,1,256] &lt;br&gt; mem[1050,1,256]</td>
<td>37.40M</td>
</tr>
<tr>
<td>DETR-ResNet50-E6</td>
<td>6</td>
<td>6</td>
<td>256</td>
<td>2048</td>
<td>8</td>
<td>[1,3,800,1333]</td>
<td>N/A</td>
<td>query[1050,1,256] &lt;br&gt; key[1050,1,256] &lt;br&gt; value[1050,1,256]</td>
<td>src[1050,1,256]</td>
<td>tgt[100,1,256] &lt;br&gt; mem[1050,1,256]</td>
<td>41.30M</td>
</tr>
<tr>
<td>DETR-ResNet50-E12</td>
<td>12</td>
<td>6</td>
<td>256</td>
<td>2048</td>
<td>8</td>
<td>[1,3,800,1333]</td>
<td>N/A</td>
<td>query[1050,1,256] &lt;br&gt; key[1050,1,256] &lt;br&gt; value[1050,1,256]</td>
<td>src[1050,1,256]</td>
<td>tgt[100,1,256] &lt;br&gt; mem[1050,1,256]</td>
<td>49.20M</td>
</tr>
<tr>
<td>ViT-Base-16</td>
<td>12</td>
<td>0</td>
<td>768</td>
<td>3072</td>
<td>12</td>
<td>[1,3,224,224]</td>
<td>16</td>
<td>query[197,1,768] &lt;br&gt; key[197,1,768] &lt;br&gt; value[197,1,768]</td>
<td>src[197,1,768]</td>
<td>N/A</td>
<td>86.00M</td>
</tr>
<tr>
<td>ViT-Large-16</td>
<td>24</td>
<td>0</td>
<td>1024</td>
<td>4096</td>
<td>16</td>
<td>[1,3,224,224]</td>
<td>16</td>
<td>query[197,1,1024] &lt;br&gt; key[197,1,1024] &lt;br&gt; value[197,1,1024]</td>
<td>src[197,1,1024]</td>
<td>N/A</td>
<td>307.00M</td>
</tr>
<tr>
<td>ViT-Huge-14</td>
<td>32</td>
<td>0</td>
<td>1280</td>
<td>5120</td>
<td>16</td>
<td>[1,3,224,224]</td>
<td>14</td>
<td>query[257,1,1280] &lt;br&gt; key[257,1,1280] &lt;br&gt; value[257,1,1280]</td>
<td>src[257,1,1280]</td>
<td>N/A</td>
<td>632.00M</td>
</tr>
</tbody>
</table>
Experimental Results on E2E Performance

- Baseline: PyTorch JIT, TVM-cuDNN, TensorRT, Ansor
- Pytorch 1.7.1, cuDNN V7.6.5, CUDA 10.0, TensorRT V7.0.0.11, TVM 0.8

Table: End-to-End Execution Performance on the Benchmark (ms)

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

- Compared with TensorRT: 1.01-1.38× speedup
- Compared with Ansor: 1.01-1.21× speedup
Experimental Results on Subgraph Benchmark

- Baseline: MHA, Encoder, and Decoder of DETR-ResNet-50-E6

The y-axis is the throughput based log 10 and then plus 1.

Compared with:

- PyTorch JIT: $2.47 \times$ on Encoder and $11.67 \times$ on Decoder
- TensorRT: $2.47 \times$ speedup on MHA, $1.08 \times$ on Encoder, and $4.19 \times$ on Decoder
- Ansor: $1.29 \times$ on MHA, $1.17 \times$ on Encoder, and $1.17 \times$ on Decoder
Conclusions
Conclusions

• Graph-Level optimization designed by human experts miss the potential performance.

• Graph and Tensor Co-Optimize (AutoGTCO):
  • A novel dynamic programming algorithm to explore operator fusion strategies.
  • new sketch generation rules and a search policy for CUDA kernel generation.

• Key Results: 1.01 - 1.38× speedup on diverse Transformer-based vision models.
THANK YOU!