Presented by: Bojian Zheng
EcoSystem Research Group, University of Toronto, Toronto

Presenting on behalf of:

Jie Zhao¹  Xiong Gao²  Ruijie Xia²
Zhaochuang Zhang²  Deshi Chen²  Lei Chen³
Renwei Zhang²  Zhen Geng²†  Bin Cheng²  Xuefeng Jin²

¹State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou
²Huawei Technologies Co. Ltd., Hangzhou, Beijing and Shenzhen
³Hong Kong University of Science and Technology, Hong Kong

†Now is with the Parallel Computing Software Team at Alibaba, Hangzhou

Fifth Conference on Machine Learning and Systems (MLSys’22)

2022.08.29, Santa Clara, CA, USA
Outline

1. Introduction
2. Partition Phase
3. Fusion Phase
4. Putting It All Together
5. Results
6. Conclusion
Fusion in Deep Learning Compilers

- *fusion* is an important transformation for making use of faster local memory, but it was **NOT** exploited by deep learning frameworks like TensorFlow [1] and Pytorch [10].
Fusion in Deep Learning Compilers

- **fusion** is an important transformation for making use of faster local memory, but it was **NOT** exploited by deep learning frameworks like TensorFlow [1] and Pytorch [10].
- In recent years, fusion has fascinated massive attentions in deep learning compilers.
fusion is an important transformation for making use of faster local memory, but it was NOT exploited by deep learning frameworks like TensorFlow [1] and Pytorch [10].

In recent years, fusion has fascinated massive attentions in deep learning compilers.

While extensively investigated, fusion in these deep learning compilers can be inspected in different ways.
A primitive/compound operator is denoted using a circle or a box. A compound operator is composed of multiple primitive operators. These operators constitute two sub-graphs.
A primitive/compound operator is denoted using a circle or a box. A compound operator is composed of multiple primitive operators. These operators constitute two sub-graphs.

- Graph compilers like XLA [5] and DLVM [12] did not consider compute-intensive operators (\(op_3\) or \(op_5\)), isolating each of the two sub-graphs into multiple components.
A primitive/compound operator is denoted using a circle or a box. A compound operator is composed of multiple primitive operators. These operators constitute two sub-graphs.

- Graph compilers like XLA [5] and DLVM [12] did not consider compute-intensive operators ($op_3$ or $op_5$), isolating each of the two sub-graphs into multiple components.
A primitive/compound operator is denoted using a circle or a box. A compound operator is composed of multiple primitive operators. These operators constitute two sub-graphs.

- Graph compilers like XLA [5] and DLVM [12] did not consider compute-intensive operators (\(op_3\) or \(op_5\)), isolating each of the two sub-graphs into multiple components.
- More recent works [6, 8, 17] investigated horizontal fusion between independent operators, e.g., \((op_1\) and \(op_2\)), but training workloads and dedicated chips were rarely considered.
Architecture of APOLLO

The partition phase considers compute-intensive operators (missed by XLA [5]); defines rules with the awareness of its loop optimizer's requirements (not investigated by TVM [3]).

The fusion phase addresses the scalability issue of existing polyhedral compilers [16, 11]; goes beyond the recent works [8, 17] by enabling memory and parallelism stitching for training workloads on a dedicated accelerator.
Architecture of Apollo

The partition phase considers compute-intensive operators (missed by XLA [5]) and defines rules with the awareness of its loop optimizer’s requirements (not investigated by TVM [3]).

The fusion phase addresses the scalability issue of existing polyhedral compilers [16, 11] and goes beyond the recent works [8, 17] by enabling memory and parallelism stitching for training workloads on a dedicated accelerator.

- The partition phase
- The fusion phase
The partition phase
- considers compute-intensive operators (missed by XLA [5]);

The fusion phase
The partition phase
- considers compute-intensive operators (missed by XLA [5]);
- defines rules with the awareness of its loop optimizer’s requirements (not investigated by TVM [3]).

The fusion phase
The partition phase
- considers compute-intensive operators (missed by XLA [5]);
- defines rules with the awareness of its loop optimizer’s requirements (not investigated by TVM [3]).

The fusion phase
- addresses the scalability issue of existing polyhedral compilers [16, 11];
The partition phase
- considers compute-intensive operators (missed by XLA [5]);
- defines rules with the awareness of its loop optimizer’s requirements (not investigated by TVM [3]).

The fusion phase
- addresses the scalability issue of existing polyhedral compilers [16, 11];
- goes beyond the recent works [8, 17] by enabling memory and parallelism stitching for training workloads on a dedicated accelerator.
A graph is first simplified through algebraic simplification (also used by [5]) data-flow optimization (also considered by nGraph [4]) control-flow optimization data layout transformation.

A sub-graph cluster (i.e., the set of colored operators) is next extracted with two kinds of operators excluded: user-defined and/or extraordinary operators with complex computational logic, e.g., all-reduce used in training speech recognition. Control flow operators like TensorFlow's RefSwitch.
A graph is first simplified through algebraic simplification (also used by [5]) data-flow optimization (also considered by nGraph [4]) control-flow optimization data layout transformation.

A sub-graph cluster (i.e., the set of colored operators) is next extracted with two kinds of operators excluded: user-defined and/or extraordinary operators with complex computational logic, e.g., all-reduce used in training speech recognition, control flow operators like TensorFlow’s RefSwitch.
A graph is first simplified through

- algebraic simplification (also used by [5])
- data-flow optimization (also considered by nGraph [4])
- control-flow optimization
- data layout transformation
A graph is first simplified through
- algebraic simplification (also used by [5])
- data-flow optimization (also considered by nGraph [4])
- control-flow optimization
- data layout transformation

A sub-graph cluster $\mathcal{P}$ (i.e., the set of colored operators) is next extracted with two kinds of operators excluded
A graph is first simplified through
- algebraic simplification (also used by [5])
- data-flow optimization (also considered by nGraph [4])
- control-flow optimization
- data layout transformation

A sub-graph cluster $\mathcal{P}$ (i.e., the set of colored operators) is next extracted with two kinds of operators excluded
- user-defined and/or extraordinary operators with complex computational logic, e.g., all-reduce used in training speech recognition.
- control flow operators like TensorFlow’s RefSwitch.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$. 

$$S(t_i) = t_i - \ln(\sum_{j=1}^{N} e^{t_j})$$

It requires two operations, one computing the logarithm and the other performing the subtraction. When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

\[
S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)
\]
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $F_x$.

$$S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)$$

It requires two operations, one computing the logarithm and the other performing the subtraction.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

\[ S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right) \]

- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$. 

$$S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)$$

- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
- We open a compound operator by removing its operator boundary.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

$$S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)$$

- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
- We open a compound operator by removing its operator boundary.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_X$.

$$S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)$$

- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
- We open a compound operator by removing its operator boundary.
The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an \( \mathcal{F}_x \).

\[
S(t_i) = t_i - \ln\left(\sum_{j=1}^{N} e^{t_j}\right)
\]

- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
- We open a compound operator by removing its operator boundary.
A micro-graph $\mathcal{G}_y$ is built by merging primitive operators using rules.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$\mathcal{G}_p$</th>
<th>$\mathcal{G}_c$</th>
<th>$\mathcal{G}_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>element-wise</td>
<td>element-wise</td>
<td>element-wise</td>
</tr>
<tr>
<td>2</td>
<td>broadcast</td>
<td>element-wise</td>
<td>broadcast</td>
</tr>
<tr>
<td>3</td>
<td>broadcast</td>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>4</td>
<td>element-wise</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>5</td>
<td>broadcast</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>6-transpose</td>
<td>element-wise/broadcast</td>
<td>transpose</td>
<td>transpose</td>
</tr>
<tr>
<td>6-matmul</td>
<td>matmul</td>
<td>element-wise</td>
<td>matmul</td>
</tr>
<tr>
<td>6-matmul</td>
<td>element-wise</td>
<td>matmul</td>
<td>matmul</td>
</tr>
<tr>
<td>6-conv</td>
<td>conv</td>
<td>element-wise</td>
<td>conv</td>
</tr>
<tr>
<td>6-conv</td>
<td>element-wise</td>
<td>conv</td>
<td>conv</td>
</tr>
</tbody>
</table>

$\mathcal{G}_p$ and $\mathcal{G}_c$ hold a producer-consumer relation; $\mathcal{G}_a$ is the merged micro-graph.

How $\mathcal{G}_p$ and $\mathcal{G}_c$ are classified are defined in the paper.
A micro-graph $G_y$ is built by merging primitive operators using rules.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$G_p$</th>
<th>$G_c$</th>
<th>$G_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>element-wise</td>
<td>element-wise</td>
<td>element-wise</td>
</tr>
<tr>
<td>2</td>
<td>broadcast</td>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>3</td>
<td>broadcast</td>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>4</td>
<td>element-wise</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>5</td>
<td>broadcast</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>6-transpose</td>
<td>element-wise/broadcast</td>
<td>transpose</td>
<td>transpose</td>
</tr>
<tr>
<td>6-matmul</td>
<td>matmul</td>
<td>element-wise</td>
<td>matmul</td>
</tr>
<tr>
<td>6-matmul</td>
<td>element-wise</td>
<td>matmul</td>
<td>matmul</td>
</tr>
<tr>
<td>6-conv</td>
<td>conv</td>
<td>element-wise</td>
<td>conv</td>
</tr>
<tr>
<td>6-conv</td>
<td>element-wise</td>
<td>conv</td>
<td>conv</td>
</tr>
</tbody>
</table>

- $G_p$ and $G_c$ hold a producer-consumer relation; $G_a$ is the merged micro-graph.
- How $G_p$ and $G_c$ are classified are defined in the paper.
- In particular, the definition classifies reshaping operations, (batched) matrix multiplication and convolution as opaque operators.
A micro-graph $G_y$ is built by merging primitive operators using rules.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$G_p$</th>
<th>$G_c$</th>
<th>$G_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>element-wise</td>
<td>element-wise</td>
<td>element-wise</td>
</tr>
<tr>
<td>2</td>
<td>broadcast</td>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>3</td>
<td>broadcast</td>
<td>broadcast</td>
<td>broadcast</td>
</tr>
<tr>
<td>4</td>
<td>element-wise</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>5</td>
<td>broadcast</td>
<td>reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>6-transpose</td>
<td>element-wise/broadcast</td>
<td>transpose</td>
<td>transpose</td>
</tr>
<tr>
<td>6-matmul</td>
<td>matmul</td>
<td>element-wise</td>
<td>element-wise</td>
</tr>
<tr>
<td>6-matmul</td>
<td>element-wise</td>
<td>matmul</td>
<td>matmul</td>
</tr>
<tr>
<td>6-conv</td>
<td>conv</td>
<td>element-wise</td>
<td>conv</td>
</tr>
<tr>
<td>6-conv</td>
<td>element-wise</td>
<td>conv</td>
<td>conv</td>
</tr>
</tbody>
</table>

$G_p$ and $G_c$ hold a producer-consumer relation; $G_a$ is the merged micro-graph.

How $G_p$ and $G_c$ are classified are defined in the paper.

In particular, the definition classifies reshaping operations, (batched) matrix multiplication and convolution as opaque operators.

These rules do not need to cover all composition patterns of operators, since some pair of operators should not be fused.
A $G_y$ is converted into a sequence of loop nests and used to produce a single kernel through our prior polyhedral loop optimizer AKG [16].
Polyhedral Loop Fusion and its Scalability within a $G_y$

- A $G_y$ is converted into a sequence of loop nests and used to produce a single kernel through our prior polyhedral loop optimizer AKG [16].

```c
for i in [0,M)
    for j in [0,N)
        a(i,j)=a(i,j)+bias; //S_1
for i in [0,M/2)
    for j in [0,N/2)
        pool(i,j)=max(a(2i,2j),
                     a(2i,2j+1), a(2i+1,2j),
                     a(2i+1,2j+1)); //S_2
```

- But AKG’s fusion algorithm still suffers from the scalability issue caused by (automatically computed) large loop shifting factors.
Polyhedral Loop Fusion and its Scalability within a $\mathcal{G}_y$

- A $\mathcal{G}_y$ is converted into a sequence of loop nests and used to produce a single kernel through our prior polyhedral loop optimizer AKG [16].

```plaintext
for i in [0,M)
    for j in [0,N)
        a(i,j)=a(i,j)+bias;  // $S_1$
    for i in [0,M/2)
        for j in [0,N/2)
            pool(i,j)=max(a(2i,2j),
                        a(2i,2j+1), a(2i+1,2j),
                        a(2i+1,2j+1));  // $S_2$
```  

- But AKG’s fusion algorithm still suffers from the scalability issue caused by (automatically computed) large loop shifting factors.
- **APOLLO** considers this when defining rules for building a $\mathcal{G}_y$.  

```plaintext
for i in [0,M)
    for j in [0,N){
        a(i,j)=a(i,j)+bias;  // $S_1$
        if(i+1) mod 2 = 0 and (j+1) mod 2 = 0
            pool((i-1)/2,(j-1)/2)=
                max(a(i-1,j-1),a(i,j-1),
                    a(i-1,j),a(i,j));  // $S_2$
```
A $G_y$ is converted into a sequence of loop nests and used to produce a single kernel through our prior polyhedral loop optimizer AKG [16].

```
for i in [0,M)
    for j in [0,N)
        a(i,j)=a(i,j)+bias;  // $S_1$
for i in [0,M/2)
    for j in [0,N/2)
        pool(i,j)=max(a(2i,2j),
            a(2i,2j+1), a(2i+1,2j),
            a(2i+1,2j+1));  // $S_2$
```

But AKG’s fusion algorithm still suffers from the scalability issue caused by (automatically computed) large loop shifting factors.

**APOLLO** considers this when defining rules for building a $G_y$.

As the loop nest composition of a $G_y$ is always predictable thanks to our aggregation rules, polyhedral loop fusion heurstics are not challenged by the scalability issue in our framework.

```
for i in [0,M)
    for j in [0,N){
        a(i,j)=a(i,j)+bias;  // $S_1$
        if(i+1) mod 2 = 0 and (j+1) mod 2 = 0
            pool((i-1)/2,(j-1)/2)=
                max(a(i-1,j-1),a(i,j-1),
                    a(i-1,j),a(i,j));  // $S_2$
    }
```
A $G_y$ is converted into a sequence of loop nests and used to produce a single kernel through our prior polyhedral loop optimizer AKG [16].

```plaintext
for i in [0, M)
  for j in [0, N)
    a(i,j) = a(i,j) + bias; // $S_1$

for i in [0, M/2)
  for j in [0, N/2)
    pool(i,j) = max(a(2i, 2j),
                  a(2i, 2j+1),
                  a(2i+1, 2j),
                  a(2i+1, 2j+1)); // $S_2$
```

But AKG’s fusion algorithm still suffers from the scalability issue caused by (automatically computed) large loop shifting factors.

**APOLLO** considers this when defining rules for building a $G_y$.

As the loop nest composition of a $G_y$ is always predictable thanks to our aggregation rules, polyhedral loop fusion heurstics are not challenged by the scalability issue in our framework.

We design and implement a framework called **PANAMERA** [14] to optimize a reduction not fused with its follow-up elementwise operators.
Memory Stitching between multiple $\mathcal{G}_y$’s

- Micro-graphs often end with reductions. We define complementary rules to exploit the stitching possibilities between them.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$\mathcal{G}_p$</th>
<th>$\mathcal{G}_c$</th>
<th>$\mathcal{G}_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>reduction</td>
<td>element-wise/broadcast reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>8</td>
<td>reduction</td>
<td>reduction</td>
<td>reduction</td>
</tr>
</tbody>
</table>

When performing memory stitching between $\mathcal{G}_y$’s, the complexity of an ending reduction can complicate Layer II. We rely on Panamera [14] to convert all reductions into three canonical forms: all-reduce, $x$-reduce and $y$-reduce.

For $i$ in $[0, M)$ and $j$ in $[0, N)$ and $k$ in $[0, P)$ and $l$ in $[0, Q)$

$$a(i,j,k,l) = a(i,j,k,l) + bias$$

For $i$ in $[0, M)$ and $j$ in $[0, N)$ and $k$ in $[0, P)$ and $l$ in $[0, Q)$

$$b(i,k) += a(i,j,k,l)$$

For $x$ in $[0, M \times P)$ and $y$ in $[0, N \times Q)$

$$a(x/P,y/Q,x\%P,y\%Q) = a(x/P,y/Q,x\%P,y\%Q) + bias$$

For $x$ in $[0, M \times P)$ and $y$ in $[0, N \times Q)$

$$b(x/P,x\%P) += a(x/P,y/Q,x\%P,y\%Q)$$

Canonicalizing reductions guarantees the matching between the loop dimensions of two $\mathcal{G}_y$’s that are to be stitched in faster memory.
Memory Stitching between multiple $G_y$'s

- Micro-graphs often end with reductions. We define complementary rules to exploit the stitching possibilities between them.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$G_p$</th>
<th>$G_c$</th>
<th>$G_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>reduction</td>
<td>element-wise/broadcast reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>8</td>
<td>reduction</td>
<td>reduction</td>
<td>reduction</td>
</tr>
</tbody>
</table>

- When performing memory stitching between $G_y$'s, the complexity of an ending reduction can complicate Layer II.
Memory Stitching between multiple $G_y$'s

- Micro-graphs often end with reductions. We define complementary rules to exploit the stitching possibilities between them.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$G_p$</th>
<th>$G_c$</th>
<th>$G_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>reduction</td>
<td>element-wise/broadcast</td>
<td>reduction</td>
</tr>
<tr>
<td>8</td>
<td>reduction</td>
<td>reduction</td>
<td>reduction</td>
</tr>
</tbody>
</table>

- When performing memory stitching between $G_y$'s, the complexity of an ending reduction can complicate Layer II.

- We rely on Panamera [14] to convert all reductions into three canonical forms all-reduce, $x$-reduce and $y$-reduce.

```python
for i in [0,M) and j in [0,N) and k in [0,P) and l in [0,Q)
    a(i,j,k,l) = a(i,j,k,l) + bias
for i in [0,M) and j in [0,N) and k in [0,P) and l in [0,Q)
    b(i,k) += a(i,j,k,l)
for x in [0,M*P) and y in [0,N*Q)
    a(x/P,y/Q,x%P,y%Q) = a(x/P,y/Q,x%P,y%Q) + bias
for x in [0,M*P) and y in [0,N*Q)
    a(x/P,y/Q,x%P,y%Q) = ...
for x in [0,M*P) and y in [0,N*Q)
    b(x/P,x%P) += ...
for x in [0,M*P) and y in [0,N*Q)
    b(x/P,x%P) += ...
```
Memory Stitching between multiple $G_y$’s

- Micro-graphs often end with reductions. We define complementary rules to exploit the stitching possibilities between them.

<table>
<thead>
<tr>
<th>Rules</th>
<th>$G_p$</th>
<th>$G_c$</th>
<th>$G_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>reduction</td>
<td>element-wise/broadcast reduction</td>
<td>reduction</td>
</tr>
<tr>
<td>8</td>
<td>reduction</td>
<td>reduction</td>
<td>reduction</td>
</tr>
</tbody>
</table>

- When performing memory stitching between $G_y$’s, the complexity of an ending reduction can complicate Layer II.
- We rely on Panamera [14] to convert all reductions into three canonical forms all-reduce, $x$-reduce and $y$-reduce.

```plaintext
for i in [0, M) and j in [0, N) and k in [0, P) and l in [0, Q)
    a(i,j,k,l) = a(i,j,k,l) + bias
for i in [0, M) and j in [0, N) and k in [0, P) and l in [0, Q)
    b(i,k) += a(i,j,k,l)
for x in [0, M*P) and y in [0, N*Q)
    a(x/P,y/Q,x%P,y%Q) = a(x/P,y/Q,x%P,y%Q) + bias
    a(x/P,y/Q,x%P,y%Q) = ...  
    b(x/P,x%P) += ...  
    b(x/P,x%P) += a(x/P,y/Q,x%P,y%Q) 
}
```

- Canonicalizing reductions guarantees the matching between the loop dimensions of two $G_y$’s that are to be stitched in faster memory.
Parallelism Stitching independent $G_y$’s or $F_x$’s

- Layer I & II did not consider the parallelism between $G_y$’s or $F_x$’s.
Layer I & II did not consider the parallelism between $G_y$’s or $F_x$’s. Such parallelism mainly exists between the branches of a multi-head/-tail operator.
Parallelism Stitching independent $g_y$’s or $f_x$’s

- Layer I & II did not consider the parallelism between $g_y$’s or $f_x$’s.
- Such parallelism mainly exists between the branches of a multi-head/-tail operator.
- Layer III detects such parallelism by traversing backward/forward along a branch and terminating until another multi-head/-tail operator is reached.

The independent operators that belong to different branches can be stitched, with a cost model $\text{gain} = \sum_{\text{op}} (\text{cost}_{\text{op}} - \text{max}_{\text{m} \leq \text{op}} (\text{cost}_{\text{op}}))$ used to determine the number of stitched operators. A compute-intensive operator is excluded in such a traverse, since its huge amount of data usually consumes up the hardware resources.
Layer I & II did not consider the parallelism between $G_y$’s or $F_x$’s.

Such parallelism mainly exists between the branches of a multi-head/-tail operator.

Layer III detects such parallelism by traversing backward/forward along a branch and terminating until another multi-head/-tail operator is reached.

The independent operators that belong to different branches can be stitched, with a cost model

$$gain = \sum_{op=m}^{k} cost_{op} - \max_{m \leq op \leq k} (cost_{op})$$

used to determine the number of stitched operators.
Parallelism Stitching independent $G_y$’s or $F_x$’s

- Layer I & II did not consider the parallelism between $G_y$’s or $F_x$’s.
- Such parallelism mainly exists between the branches of a multi-head/-tail operator.
- Layer III detects such parallelism by traversing backward/forward along a branch and terminating until another multi-head/-tail operator is reached.
- The independent operators that belong to different branches can be stitched, with a cost model

$$gain = \sum_{op=m}^{k} cost_{op} - \max_{m \leq op \leq k} (cost_{op})$$

used to determine the number of stitched operators.
- A compute-intensive operator is excluded in such a traverse, since its huge amount of data usually consumes up the hardware resources.
To make **APOLLO** applicable to both training and inference workloads, **APOLLO** is also complemented through the following steps.
To make Apollo applicable to both training and inference workloads, Apollo is also complemented through the following steps.

**Auto-tuning**: Apollo captures the loop composition of a $G_x$ that are prevented from parallelization by Cost Model (3) of the paper.
To make Apollo applicable to both training and inference workloads, Apollo is also complemented through the following steps.

**Auto-tuning**: Apollo captures the loop composition of a $G_x$ that are prevented from parallelization by Cost Model (3) of the paper.

**Piecewise compilation** is performed along the red/violet arrows, further reducing compilation overhead.
To make **Apollo** applicable to both training and inference workloads, **Apollo** is also complemented through the following steps.

- **Auto-tuning**: **Apollo** captures the loop composition of a $G_x$ that are prevented from parallelization by Cost Model (3) of the paper.
- **Piecewise compilation** is performed along the red/violet arrows, further reducing compilation overhead.
- **Code generation** supports both GPUs and Huawei Ascend 910 chips [7].
Experiments are conducted using five training workloads.

Generated CUDA code on GPUs is executed using CUDA Toolkit 10.1 with -O3 enabled.

Generated CCE code on Ascend 910 is executed using the later’s native compiler.

The geometric mean of 10 executions is reported.

Case study on sub-graphs, results of inference workloads and compilation overhead are reported in the paper.
## Results on Single GPU

BT: BERT; TR: Transformer; WD: Wide&Deep; YO: Yolo-v3; FM: DeepFM; b.s.: batch sizes; TF: TensorFlow; MS: MindSpore; imp.: improvements

<table>
<thead>
<tr>
<th>models</th>
<th>b.s.</th>
<th>TF</th>
<th>XLA/TF</th>
<th>MS</th>
<th>APOLLO/MS</th>
<th>imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT-base</td>
<td>32</td>
<td>167</td>
<td>105%</td>
<td>135</td>
<td>252%</td>
<td>39%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>200.8</td>
<td>129%</td>
<td>183.6</td>
<td>212%</td>
<td>23%</td>
</tr>
<tr>
<td>TR</td>
<td>8</td>
<td>6750</td>
<td>16%</td>
<td>5122</td>
<td>84%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9500</td>
<td>11%</td>
<td>10868</td>
<td>59%</td>
<td>64%</td>
</tr>
<tr>
<td>WD</td>
<td>16000</td>
<td>1133696</td>
<td>15%</td>
<td>762086</td>
<td>123%</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>1470221</td>
<td>5%</td>
<td>836820</td>
<td>121%</td>
<td>20%</td>
</tr>
<tr>
<td>YO</td>
<td>4</td>
<td>33.11</td>
<td>15%</td>
<td>39.48</td>
<td>46%</td>
<td>51%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>56.00</td>
<td>12%</td>
<td>75.01</td>
<td>10%</td>
<td>31%</td>
</tr>
<tr>
<td>FM</td>
<td>8192</td>
<td>26117</td>
<td>-1%</td>
<td>479744</td>
<td>151%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>30279</td>
<td>-2%</td>
<td>543024</td>
<td>167%</td>
<td>-</td>
</tr>
</tbody>
</table>
Results on Single GPU

BT: BERT; TR: Transformer; WD: Wide&Deep; YO: Yolo-v3; FM: DeepFM; b.s.: batch sizes; TF: TensorFlow; MS: MindSpore; imp.: improvements

<table>
<thead>
<tr>
<th>models</th>
<th>b.s.</th>
<th>TF</th>
<th>XLA/TF</th>
<th>MS</th>
<th>APOLLO/MS</th>
<th>imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT-base</td>
<td>32</td>
<td>167</td>
<td>105%</td>
<td>135</td>
<td>252%</td>
<td>39%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>200.8</td>
<td>129%</td>
<td>183.6</td>
<td>212%</td>
<td>23%</td>
</tr>
<tr>
<td>TR</td>
<td>8</td>
<td>6750</td>
<td>16%</td>
<td>5122</td>
<td>84%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9500</td>
<td>11%</td>
<td>10868</td>
<td>59%</td>
<td>64%</td>
</tr>
<tr>
<td>WD</td>
<td>16000</td>
<td>1133696</td>
<td>15%</td>
<td>762086</td>
<td>123%</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>1470221</td>
<td>5%</td>
<td>836820</td>
<td>121%</td>
<td>20%</td>
</tr>
<tr>
<td>YO</td>
<td>4</td>
<td>33.11</td>
<td>15%</td>
<td>39.48</td>
<td>46%</td>
<td>51%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>56.00</td>
<td>12%</td>
<td>75.01</td>
<td>10%</td>
<td>31%</td>
</tr>
<tr>
<td>FM</td>
<td>8192</td>
<td>26117</td>
<td>-1%</td>
<td>479744</td>
<td>151%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>30279</td>
<td>-2%</td>
<td>543024</td>
<td>167%</td>
<td>-</td>
</tr>
</tbody>
</table>

- APOLLO helps MindSpore outperforms TensorFlow and XLA by $1.86 \times$ and $1.37 \times$, respectively.
### Results on Single GPU

BT: BERT; TR: Transformer; WD: Wide&Deep; YO: Yolo-v3; FM: DeepFM; b.s.: batch sizes; TF: TensorFlow; MS: MindSpore; imp.: improvements

<table>
<thead>
<tr>
<th>models</th>
<th>b.s.</th>
<th>TF</th>
<th>XLA/TF</th>
<th>MS</th>
<th>APOLLO/MS</th>
<th>imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT-base</td>
<td>32</td>
<td>167</td>
<td>105%</td>
<td>135</td>
<td>252%</td>
<td>39%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>200.8</td>
<td>129%</td>
<td>183.6</td>
<td>212%</td>
<td>23%</td>
</tr>
<tr>
<td>TR</td>
<td>8</td>
<td>6750</td>
<td>16%</td>
<td>5122</td>
<td>84%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9500</td>
<td>11%</td>
<td>10868</td>
<td>59%</td>
<td>64%</td>
</tr>
<tr>
<td>WD</td>
<td>16000</td>
<td>1133696</td>
<td>15%</td>
<td>762086</td>
<td>123%</td>
<td>48%</td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>1470221</td>
<td>5%</td>
<td>836820</td>
<td>121%</td>
<td>20%</td>
</tr>
<tr>
<td>YO</td>
<td>4</td>
<td>33.11</td>
<td>15%</td>
<td>39.48</td>
<td>46%</td>
<td>51%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>56.00</td>
<td>12%</td>
<td>75.01</td>
<td>10%</td>
<td>31%</td>
</tr>
<tr>
<td>FM</td>
<td>8192</td>
<td>26117</td>
<td>-1%</td>
<td>479744</td>
<td>151%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>30279</td>
<td>-2%</td>
<td>543024</td>
<td>167%</td>
<td>-</td>
</tr>
</tbody>
</table>

**Apollo helps MindSpore outperforms TensorFlow and XLA by 1.86\times and 1.37\times, respectively.**

Execution times of MindSpore’s model zoo (y axis: log scaled time in ms; lower is better). On average, Apollo improves MindSpore by 29.6%.
### Results on Multiple GPUs

BT: BERT; WD: Wide&Deep; FM: DeepFM; batch sizes in parenthesis; TF: TensorFlow; MS: MindSpore; imp.: improvements

<table>
<thead>
<tr>
<th>models</th>
<th>GPUs</th>
<th>TF</th>
<th>XLA/TF</th>
<th>MS</th>
<th>APOLLO/MS</th>
<th>imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT-base(32)</td>
<td>8</td>
<td>1244.9</td>
<td>96%</td>
<td>944.4</td>
<td>247%</td>
<td>34%</td>
</tr>
<tr>
<td>BT-base(64)</td>
<td>8</td>
<td>1555.4</td>
<td>117%</td>
<td>1333.1</td>
<td>222%</td>
<td>27%</td>
</tr>
<tr>
<td>BT-large(4)</td>
<td>4</td>
<td>66.94</td>
<td>33%</td>
<td>37.62</td>
<td>133%</td>
<td>-2%</td>
</tr>
<tr>
<td>WD(16000)</td>
<td>8</td>
<td>8086178</td>
<td>1%</td>
<td>4964319</td>
<td>87%</td>
<td>13%</td>
</tr>
<tr>
<td>FM(16384)</td>
<td>4</td>
<td>31767</td>
<td>-7%</td>
<td>2117685</td>
<td>130%</td>
<td>-</td>
</tr>
</tbody>
</table>

The throughput of MindSpore falls behind, sometimes significantly, than those of TensorFlow and XLA, but it outperforms the latter two by $1.96 \times$ and $1.18 \times$, respectively.
Results on Multiple GPUs

BT: BERT; WD: Wide&Deep; FM: DeepFM; batch sizes in parenthesis; TF: TensorFlow; MS: MindSpore; imp.: improvements

<table>
<thead>
<tr>
<th>models</th>
<th>GPUs</th>
<th>TF</th>
<th>XLA/TF</th>
<th>MS</th>
<th>APOLLO/MS</th>
<th>imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT-base(32)</td>
<td>8</td>
<td>1244.9</td>
<td>96%</td>
<td>944.4</td>
<td>247%</td>
<td>34%</td>
</tr>
<tr>
<td>BT-base(64)</td>
<td>8</td>
<td>1555.4</td>
<td>117%</td>
<td>1333.1</td>
<td>222%</td>
<td>27%</td>
</tr>
<tr>
<td>BT-large(4)</td>
<td>4</td>
<td>66.94</td>
<td>33%</td>
<td>37.62</td>
<td>133%</td>
<td>-2%</td>
</tr>
<tr>
<td>WD(16000)</td>
<td>8</td>
<td>8086178</td>
<td>1%</td>
<td>4964319</td>
<td>87%</td>
<td>13%</td>
</tr>
<tr>
<td>FM(16384)</td>
<td>4</td>
<td>31767</td>
<td>-7%</td>
<td>2117685</td>
<td>130%</td>
<td>-</td>
</tr>
</tbody>
</table>

- The throughput of MindSpore falls behind, sometimes significantly, than those of TensorFlow and XLA, but it outperforms the latter two by $1.96 \times$ and $1.18 \times$, respectively.
Throughput of BERT and PanGu-α [13] (examples/s) on Ascend. Batch sizes are in parentheses. Higher is better.
Throughput of BERT and PanGu-α [13] (examples/s) on Ascend. Batch sizes are in parentheses. Higher is better.

- **APOLLO** brings about a mean improvement of 19.7% over MindSpore when targeting Ascend 910 chips.
Summary of the Contributions

Apollo extends the search space of fusion by considering more operator types, generating more profitable across-layer schedules originally hindered by operator boundaries;
Summary of the Contributions

- **APOLLO** extends the search space of fusion by considering more operator types, generating more profitable across-layer schedules originally hindered by operator boundaries;

- **APOLLO** addresses the scalability issue of fusion by allowing reverse feedback from the operator-level optimizer, achieving a fully automatic fusion framework;
Summary of the Contributions

- **Apollo** extends the search space of fusion by considering more operator types, generating more profitable across-layer schedules originally hindered by operator boundaries;

- **Apollo** addresses the scalability issue of fusion by allowing reverse feedback from the operator-level optimizer, achieving a fully automatic fusion framework;

- **Apollo** enhances the performance of deep learning workloads by modeling both data locality and parallelism, producing more efficient code than the state of the art;
Summary of the Contributions

- **APOLLO** extends the search space of fusion by considering more operator types, generating more profitable across-layer schedules originally hindered by operator boundaries;

- **APOLLO** addresses the scalability issue of fusion by allowing reverse feedback from the operator-level optimizer, achieving a fully automatic fusion framework;

- **APOLLO** enhances the performance of deep learning workloads by modeling both data locality and parallelism, producing more efficient code than the state of the art;

- **APOLLO** exhibits reasonable JIT compilation overhead, demonstrating its effectiveness using rather difficult real-life training workloads.
References

Tensorflow: A system for large-scale machine learning.

Tiramisu: A polyhedral compiler for expressing fast and portable code.

Tvm: An automated end-to-end optimizing compiler for deep learning.


Taso: Optimizing deep learning computation with automatic generation of graph substitutions.

Rammer: Enabling holistic deep learning compiler optimizations with rtasks.

Revisiting loop fusion in the polyhedral framework.

Pytorch: An imperative style, high-performance deep learning library.

The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically.


Zhao, J., Bastoul, C., Yi, Y., Hu, J., Nie, W., Zhang, R., Geng, Z., Li, C., Tachon, T., and Gan, Z. 

Parallelizing neural network models effectively on GPU by implementing reductions atomically. 

In *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques* (2022), PACT’22, ACM.

Zhao, J., and Di, P. 

Optimizing the memory hierarchy by compositing automatic transformations on computations and data. 


Akg: Automatic kernel generation for neural processing units using polyhedral transformations. 


Scan the QR code to obtain the paper/code repository/artifact.

Thank you!

Any Questions?