APOLLO: Automatic Partition-based Operator Fusion through Layer by Layer Optimization

Presenter: Bojian Zheng

EcoSystem Research Group, University of Toronto, Toronto

presenting on behalf of

Jie Zhao[1] Xiong Gao[2] Ruijie Xia[2]
Zhaochuang Zhang[2] Deshi Chen[2] Lei Chen[3]
Renwei Zhang[2] Zhen Geng[2†] Bin Cheng[2] Xuefeng Jin[2]

[1]State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou
[2]Huawei Technologies Co. Ltd., Hangzhou, Beijing and Shenzhen
[3]Hong Kong University of Science and Technology, Hong Kong
[†]Now is with the Parallel Computing Software Team at Alibaba, Hangzhou

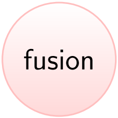
Fifth Conference on Machine Learning and Systems **(MLSys'22)**

2022.08.29, Santa Clara, CA, USA

Outline

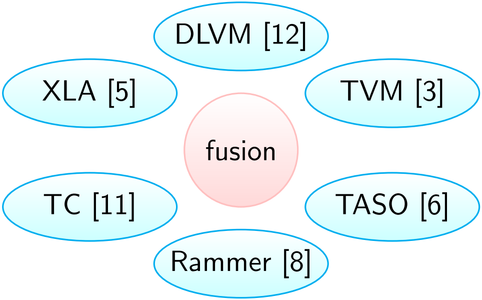
Fusion in Deep Learning Compilers

- *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].



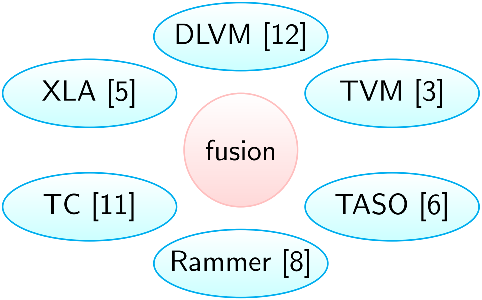
Fusion in Deep Learning Compilers

- *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.



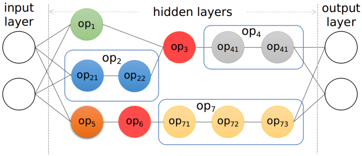
Fusion in Deep Learning Compilers

- *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.



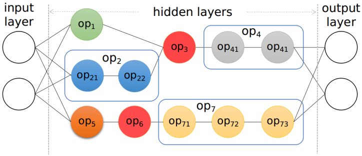
- While extensively investigated, fusion in these deep learning compilers can be inspected in different ways.

Limitations of Prior Fusion Compilers



A primitive/compound operator is denoted using a circle or a box. A compond operator is composed of multiple primitive operators. These operators consititute two sub-graphs.

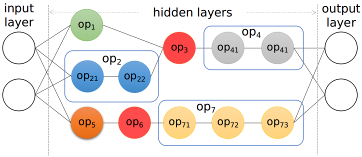
Limitations of Prior Fusion Compilers



A primitive/compound operator is denoted using a circle or a box. A compond operator is composed of multiple primitive operators. These operators consititute 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.

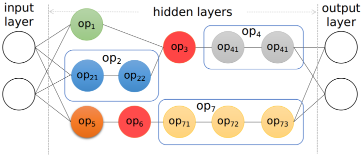
Limitations of Prior Fusion Compilers



A primitive/compound operator is denoted using a circle or a box. A compond operator is composed of multiple primitive operators. These operators consititute 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.
- Tensor compilers including TVM [3], TC [11] and Tiramisu [2] execute a routine transformation orchestration (first node grouping and next loop fusion), subject to the scalability issue [15, 9] caused by the constraints from the upstream graph engine.

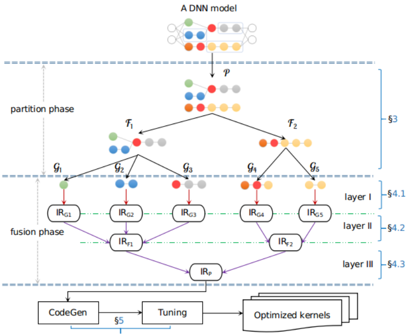
Limitations of Prior Fusion Compilers



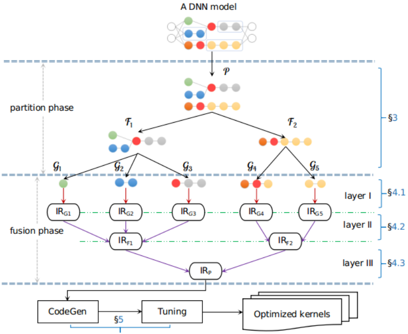
A primitive/compound operator is denoted using a circle or a box. A compond operator is composed of multiple primitive operators. These operators consititute two sub-graphs.

- 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.
- Tensor compilers including TVM [3], TC [11] and Tiramisu [2] execute a routine transformation orchestration (first node grouping and next loop fusion), subject to the scalability issue [15, 9] caused by the constraints from the upstream graph engine.
- 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



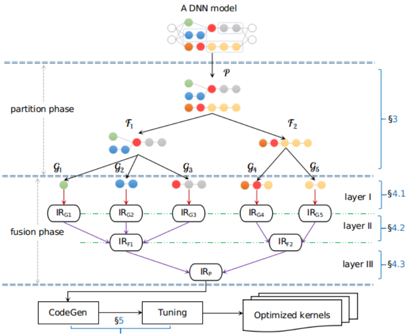
Architecture of APOLLO



- The partition phase

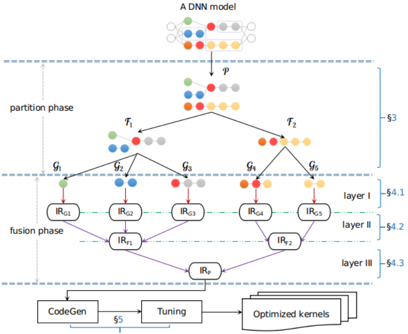
- The fusion phase

Architecture of APOLLO



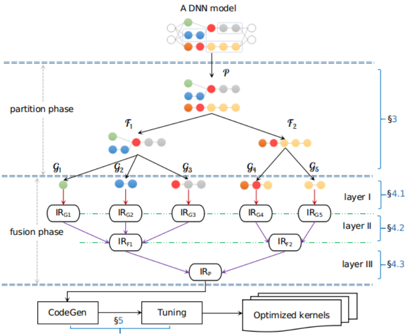
- The partition phase
  - considers compute-intensive operators (missed by XLA [5]);
- The fusion phase

Architecture of Apollo



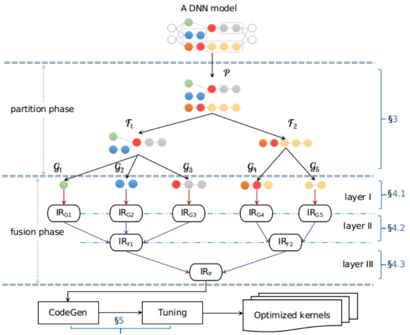
- The partition phase
  - considers compute-intensive operators (missed by XLA [5]);
  - defines rules with the awareness of its loop optimizer's requirements (not investigated by TVM [3]).
- The fusion phase

Architecture of APOLLO



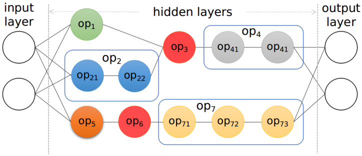
- The partition phase
  - considers compute-intensive operators (missed by XLA [5]);
  - defines rules with the awareness of its loop optimizer's requirements (not investigated by TVM [3]).
- The fusion phase
  - addresses the scalability issue of existing polyhedral compilers [16, 11];

Architecture of APOLLO

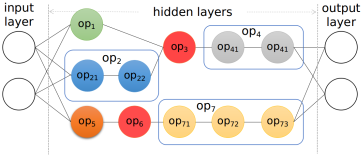


- The partition phase
  - considers compute-intensive operators (missed by XLA [5]);
  - defines rules with the awareness of its loop optimizer's requirements (not investigated by TVM [3]).
- The fusion phase
  - addresses the scalability issue of existing polyhedral compilers [16, 11];
  - goes beyond the recent works [8, 17] by enabling memory and parallelism stitching for training workloads on a dedicated accelerator.

Graph Simplification and Extracting Sub-graph Cluster $\mathcal{P}$

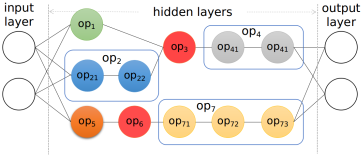


Graph Simplification and Extracting Sub-graph Cluster $\mathcal{P}$

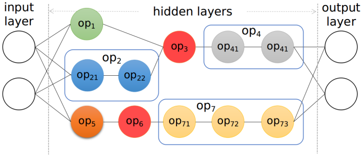


- A graph is first simplified through

Graph Simplification and Extracting Sub-graph Cluster $\mathcal{P}$

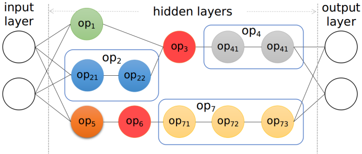


- A graph is first simplified through
  - algebraic simplification (also used by [5])
  - data-flow optimization (also considered by nGraph [4])
  - control-flow optimization
  - data layout transformation



- 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

Graph Simplification and Extracting Sub-graph Cluster $\mathcal{P}$



- 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`.

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

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

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- 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.

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- 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.

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- The use of activation functions is one of the major reasons that result in the complex dependence patterns of compound operators in an $\mathcal{F}_x$.

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

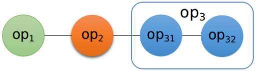
- It requires two operations, one computing the logarithm and the other performing the subtraction.
- When tiled, the subtraction must wait for the completion of all simultaneously executed tiles of the reduction, preventing the fusion between the two tiled operations.
- We open a compound operator by removing its operator boundary.

Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- 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.
- We open a compound operator by removing its operator boundary.

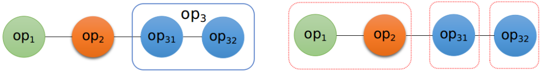


Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- 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.
- We open a compound operator by removing its operator boundary.

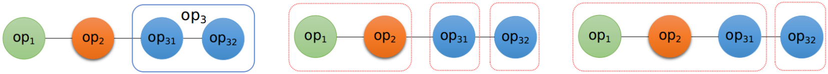


Opening Compound Operators within each Sub-graph $\mathcal{F}_x$

- 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.
- We open a compound operator by removing its operator boundary.



Merging Primitive Operators within each Micro-graph $\mathcal{G}_y$

- A micro-graph $\mathcal{G}_y$ is built by merging primitive operators using rules.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
❶	element-wise	element-wise	element-wise
❷	broadcast	element-wise	broadcast
❸	broadcast	broadcast	broadcast
❹	element-wise	reduction	reduction
❺	broadcast	reduction	reduction
❻-transpose	element-wise/broadcast	transpose	transpose
❻-matmul	matmul	element-wise	matmul
❻-matmul	element-wise	matmul	matmul
❻-conv	conv	element-wise	conv
❻-conv	element-wise	conv	conv

- $\mathcal{G}_p$ 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.

Merging Primitive Operators within each Micro-graph $\mathcal{G}_y$

- A micro-graph $\mathcal{G}_y$ is built by merging primitive operators using rules.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
❶	element-wise	element-wise	element-wise
❷	broadcast	element-wise	broadcast
❸	broadcast	broadcast	broadcast
❹	element-wise	reduction	reduction
❺	broadcast	reduction	reduction
❻-transpose	element-wise/broadcast	transpose	transpose
❻-matmul	matmul	element-wise	matmul
❻-matmul	element-wise	matmul	matmul
❻-conv	conv	element-wise	conv
❻-conv	element-wise	conv	conv

- $\mathcal{G}_p$ 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.
- In particular, the definition classifies reshaping operations, (batched) matrix multiplication and convolution as opaque operators.

Merging Primitive Operators within each Micro-graph $\mathcal{G}_y$

- A micro-graph $\mathcal{G}_y$ is built by merging primitive operators using rules.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
❶	element-wise	element-wise	element-wise
❷	broadcast	element-wise	broadcast
❸	broadcast	broadcast	broadcast
❹	element-wise	reduction	reduction
❺	broadcast	reduction	reduction
❻-transpose	element-wise/broadcast	transpose	transpose
❻-matmul	matmul	element-wise	matmul
❻-matmul	element-wise	matmul	matmul
❻-conv	conv	element-wise	conv
❻-conv	element-wise	conv	conv

- $\mathcal{G}_p$ 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.
- 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.

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].

```
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)); //$S_2$
```

```
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)); //$S_2$
  }
```

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].

```
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)); //$S_2$
```

```
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)); //$S_2$
  }
```

- 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].

```
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)); //$S_2$
```

```
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)); //$S_2$
  }
```

- 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 $\mathcal{G}_y$.

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].

```
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)); //$S_2$
```

```
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)); //$S_2$
  }
```

- 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 $\mathcal{G}_y$.
- As the loop nest composition of a $\mathcal{G}_y$ is always predictable thanks to our aggregation rules, polyhedral loop fusion heurstics are not challenged by the scalability issue in our framework.

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].

```
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)); //$S_2$
```

```
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)); //$S_2$
  }
```

- 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 $\mathcal{G}_y$.
- As the loop nest composition of a $\mathcal{G}_y$ 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.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
❼	reduction	element-wise/broadcast	reduction
❽	reduction	reduction	reduction

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.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
❼	reduction	element-wise/broadcast	reduction
❽	reduction	reduction	reduction

- When performing memory stitching between $\mathcal{G}_y$'s, the complexity of an ending reduction can complicate Layer II.

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.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
7	reduction	element-wise/broadcast	reduction
8	reduction	reduction	reduction

- 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*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)
  b(x/P,x%P) += a(x/P,y/Q,x%P,y%Q)
```

```
for x in [0,M*P) and y in [0,N*Q){
  a(x/P,y/Q,x%P,y%Q) = ...
  b(x/P,x%P) += ...
}
```

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.

Rules	$\mathcal{G}_p$	$\mathcal{G}_c$	$\mathcal{G}_a$
7	reduction	element-wise/broadcast	reduction
8	reduction	reduction	reduction

- 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*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)
  b(x/P,x%P) += a(x/P,y/Q,x%P,y%Q)
```

```
for x in [0,M*P) and y in [0,N*Q){
  a(x/P,y/Q,x%P,y%Q) = ...
  b(x/P,x%P) += ...
}
```

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

Parallelism Stitching independent $\mathcal{G}_y$'s or $\mathcal{F}_x$'s

- Layer I & II did not consider the parallelism between $\mathcal{G}_y$'s or $\mathcal{F}_x$'s.

Parallelism Stitching independent $\mathcal{G}_y$'s or $\mathcal{F}_x$'s

- Layer I & II did not consider the parallelism between $\mathcal{G}_y$'s or $\mathcal{F}_x$'s.
- Such parallelism mainly exists between the branches of a multi-head/-tail operator.

Parallelism Stitching independent $\mathcal{G}_y$'s or $\mathcal{F}_x$'s

- Layer I & II did not consider the parallelism between $\mathcal{G}_y$'s or $\mathcal{F}_x$'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.

Parallelism Stitching independent $\mathcal{G}_y$'s or $\mathcal{F}_x$'s

- Layer I & II did not consider the parallelism between $\mathcal{G}_y$'s or $\mathcal{F}_x$'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 $\mathcal{G}_y$'s or $\mathcal{F}_x$'s

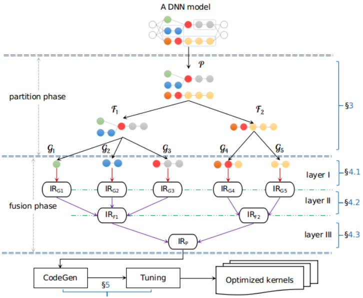
- Layer I & II did not consider the parallelism between $\mathcal{G}_y$'s or $\mathcal{F}_x$'s.
- Such parallelism mainly exists between the branches of a multi-head/-tail operator.
- Layer III detects such parallelism by traversing backward/forward along a branch and terminating until another multi-head/-tail 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.

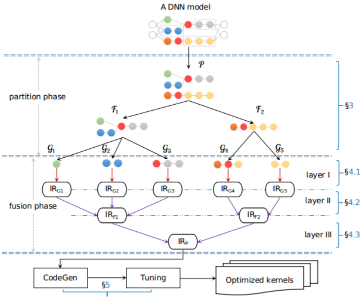
- A compute-intensive operator is excluded in such a traverse, since its huge amount of data usually consumes up the hardware resources.

Putting It All Together



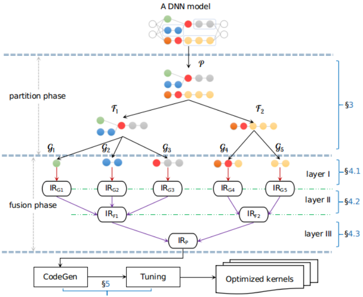
- To make APOLLO applicable to both training and inference workloads, APOLLO is also complemented through the following steps.

Putting It All Together



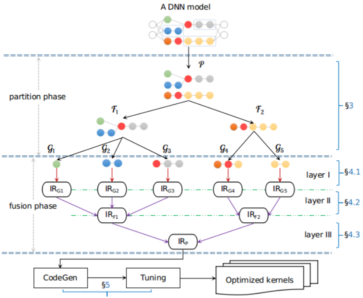
- To make APOLLO applicable to both training and inference workloads, APOLLO is also complemented through the following steps.
- **Auto-tuning**: APOLLO captures the loop composition of a $\mathcal{G}_x$ that are prevented from parallelization by Cost Model (3) of the paper.

Putting It All Together



- 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 $\mathcal{G}_x$ that are prevented from parallelization by Cost Model (3) of the paper.
- **Piecewise compilation** is performed alongthe red/violet arrows, further reducing compilation overhead.

Putting It All Together



- 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 $\mathcal{G}_x$ that are prevented from parallelization by Cost Model (3) of the paper.
- **Piecewise compilation** is performed alongthe red/violet arrows, further reducing compilation overhead.
- **Code generation** supports both GPUs and Huawei Ascend 910 chips [7].

Results on Single GPU

- 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

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

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

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

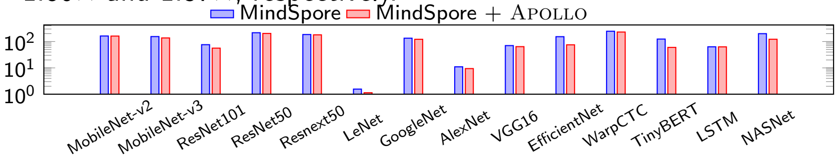
- APOLLO helps MindSpore outperforms TensorFlow and XLA by 1.86× and 1.37×, respectively.

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

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

- 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

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

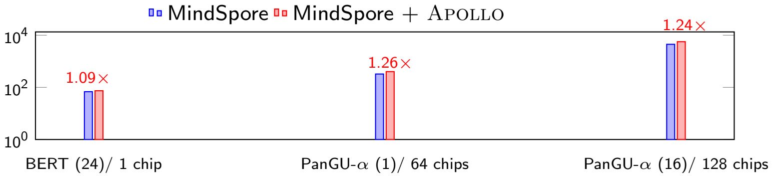
Results on Multiple GPUs

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

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

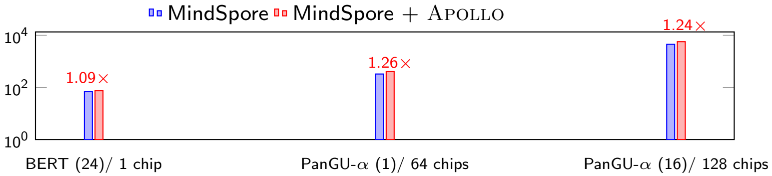
- The throughput of MindSpore falls behind, sometimes significantly, than those of TensorFlow and XLA, but it outperforms the latter two by 1.96× and 1.18×, respectively.

Results on Ascend 910 chips



Throughput of BERT and PanGu-α [13] (examples/s) on Ascend. Batch sizes are in parentheses. Higher is better.

Results on Ascend 910 chips



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

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X.
Tensorflow: A system for large-scale machine learning.
In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 265–283.

[2] Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S.
Tiramisu: A polyhedral compiler for expressing fast and portable code.
In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2019), CGO 2019, IEEE Press, pp. 193–205.

[3] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A.
Tvm: An automated end-to-end optimizing compiler for deep learning.
In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 579–594.

[4] Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., Kimball, R., Knight, J., Korovaiko, N., Kumar, V., Lao, Y., Lishka, C. R., Menon, J., Myers, J., Narayana, S. A., Procter, A., and Webb, T. J.
Intel ngraph: An intermediate representation, compiler, and executor for deep learning, 2018.

[5] Google.
Xla: Optimizing compiler for machine learning, 2017.

[6] Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A.
Taso: Optimizing deep learning computation with automatic generation of graph substitutions.
In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP'19, ACM, pp. 47–62.

References

[7] Liao, H., Tu, J., Xia, J., Liu, H., Zhou, X., Yuan, H., and Hu, Y.
Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper.
In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2021), pp. 789–801.

[8] Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L.
Rammer: Enabling holistic deep learning compiler optimizations with rtasks.
In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 881–897.

[9] Mehta, S., Lin, P.-H., and Yew, P.-C.
Revisiting loop fusion in the polyhedral framework.
In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2014), PPoPP'14, ACM, pp. 233–246.

[10] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S.
Pytorch: An imperative style, high-performance deep learning library.
In *Advances in neural information processing systems* (2019), pp. 8026–8037.

[11] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., Devito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A.
The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically.
ACM Trans. Archit. Code Optim. 16, 4 (Oct. 2019).

[12] Wei, R., Schwartz, L., and Adve, V.
Dlvm: A modern compiler infrastructure for deep learning systems, 2018.

References

[13] Zeng, W., Ren, X., Su, T., Wang, H., Liao, Y., Wang, Z., Jiang, X., Yang, Z., Wang, K., Zhang, X., Li, C., Gong, Z., Yao, Y., Huang, X., Wang, J., Yu, J., Guo, Q., Yu, Y., Zhang, Y., Wang, J., Tao, H., Yan, D., Yi, Z., Peng, F., Jiang, F., Zhang, H., Deng, L., Zhang, Y., Lin, Z., Zhang, C., Zhang, S., Guo, M., Gu, S., Fan, G., Wang, Y., Jin, X., Liu, Q., and Tian, Y.
Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation, 2021.

[14] 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.

[15] Zhao, J., and Di, P.
Optimizing the memory hierarchy by compositing automatic transformations on computations and data.
In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture* (Piscataway, NJ, USA, 2020), MICRO-53, IEEE Press, pp. 427–441.

[16] Zhao, J., Li, B., Nie, W., Geng, Z., Zhang, R., Gao, X., Cheng, B., Wu, C., Cheng, Y., Li, Z., Di, P., Zhang, K., and Jin, X.
Akg: Automatic kernel generation for neural processing units using polyhedral transformations.
In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, pp. 1233–1248.

[17] Zheng, Z., Zhao, P., Long, G., Zhu, F., Zhu, K., Zhao, W., Diao, L., Yang, J., and Lin, W.
Fusionstitching: Boosting memory intensive computations for deep learning workloads, 2020.

Questions & Answers

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