Apollo: Automatic Partition-Based Operator Fusion Through Layer By Layer Optimization

Group 12

Savini Gamage, Logan Green, Lucas Kellar, Yung-Hao Liao

November 17, 2025

Motivation

Performance Challenges in Deep Learning

- Node Grouping
  - Grouping related operations
  - Most compilers consider only memory-bound operations
    - Compute-bound ops ignored for fusion
- Loop Fusion
  - Instantiate operations as arithmetic ops in nested loops
  - Rely on downstream loop optimizers for fusion
    - At scale, challenges loop fusion heuristics
- Stitching
  - Packing independent operations into a single kernel
  - Typically requires ahead-of-time op compilation
    - Can’t handle custom ops

Apollo Architecture

Partition Phase

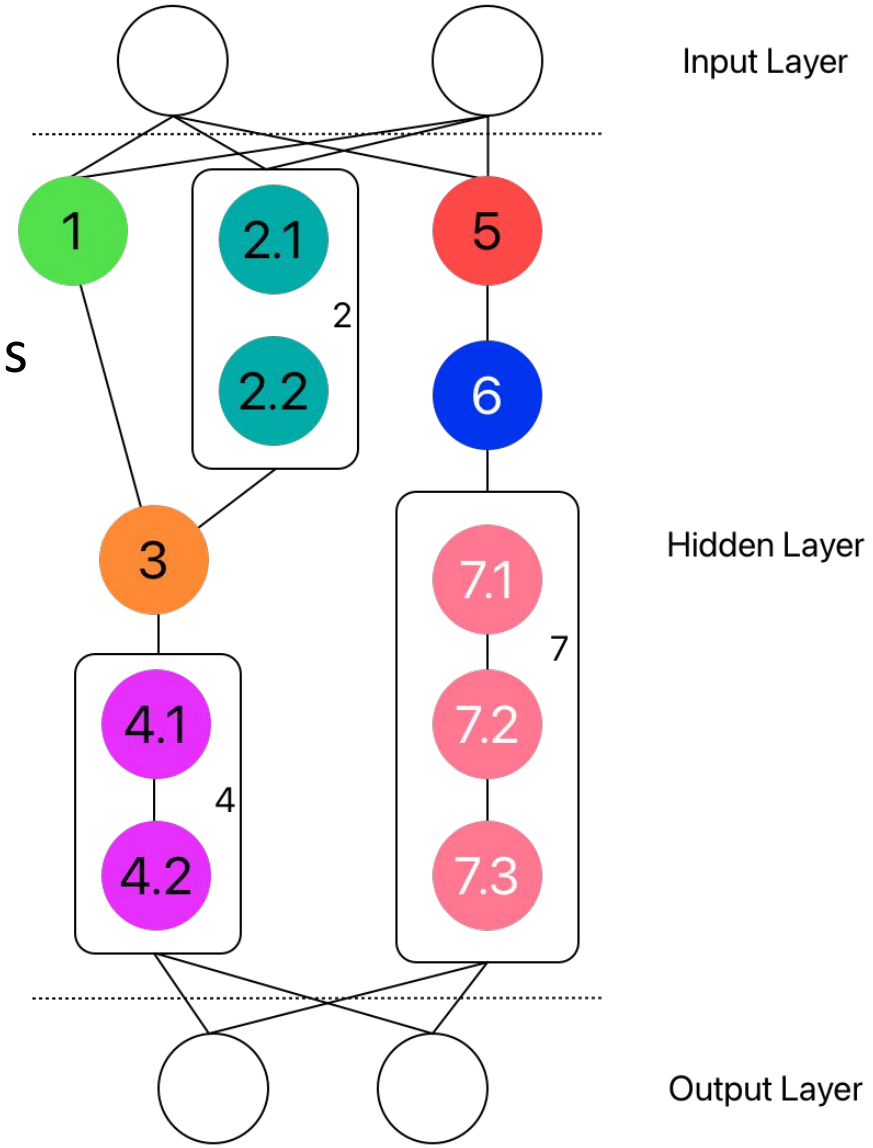
- Split ops into subgraphs
- Each subgraph split into individual micro-graphs

Fusion Phase

- Layer I: Loop Fusion
  - Fuses via polyhedral model
- Layer II: Memory Stitching
  - Fuses reduction ops with successors
- Layer III: Parallelism Stitching
  - Parallelizes independent ops

Packaging

- Auto-Tuner
- Piecewise Compilation
- Code Generation



Partition Phase

Goal: Separate DNN graphs into micro-graphs are suitable for fusion.

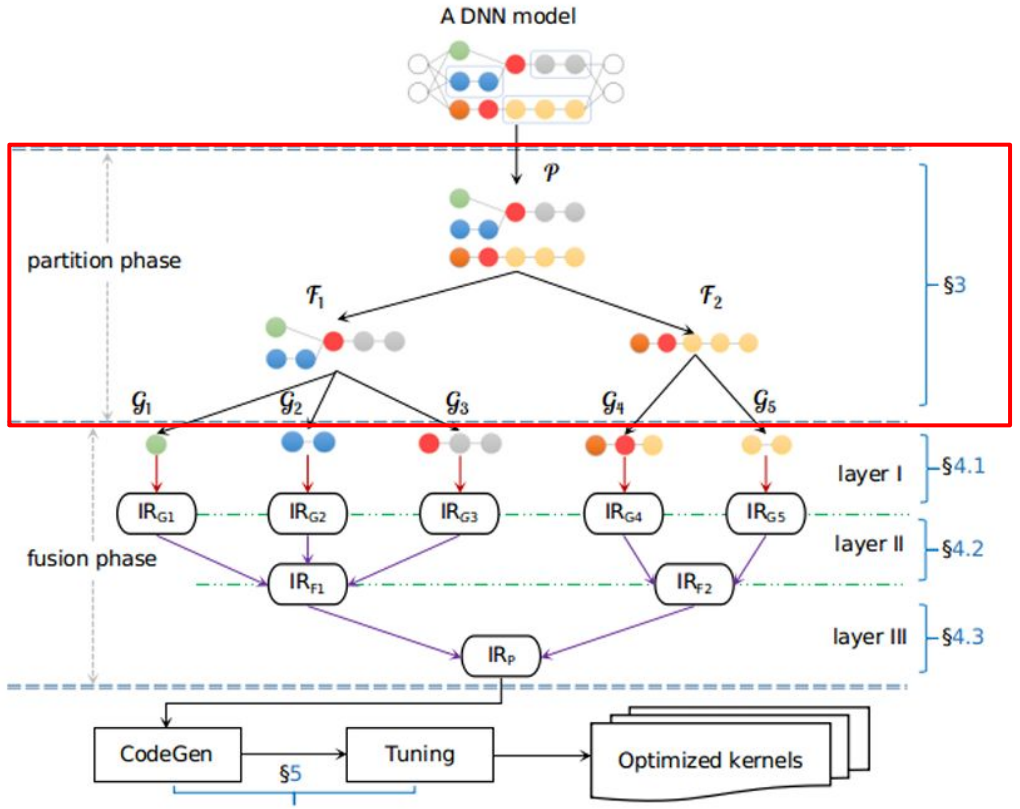


Figure 2: Architecture of APOLLO.

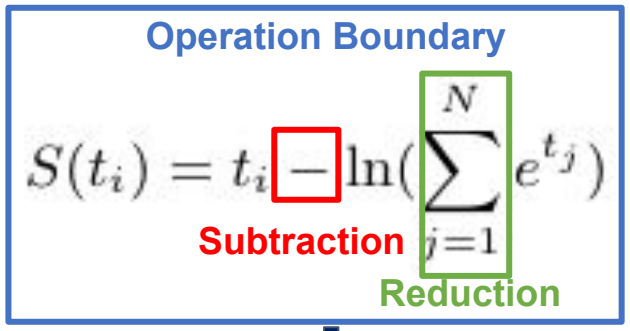
$\mathcal{P}$: maximum set of sub-graphs

$\mathcal{F}$: sub-graphs

$\mathcal{G}$: micro-graphs

Opening Compound Operators:

Compound Operator example: `LogSoftMax`



Op boundary exists
: Not fusible in traditional compilers

3 fusion groups → **2** fusion groups → **Widen the fusion search space**
More opportunities in fusion!

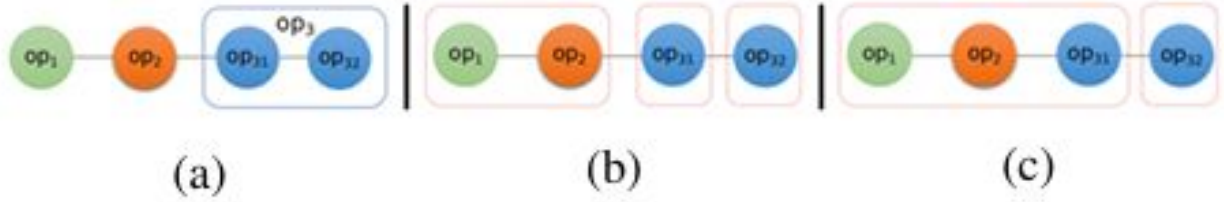


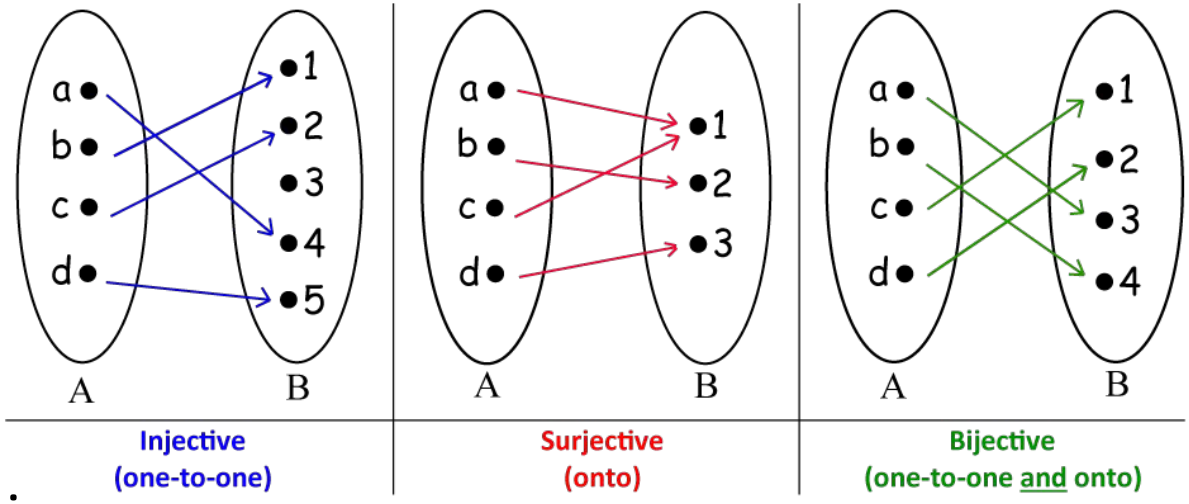
Figure 3: Effect of *op* boundaries. (a) `LogSoftMax`; and the fusion patterns when *op* boundaries (b) exist or (c) not.

Aggregating Primitive Operators

Goal: generate micro-graphs downstream optimizer can handle. (resolve scalability issue)

First: categorize the primitive operators.

How? ⟶ By Definitions



(Source: Carcworkshop.com)

Def 1. element-wise: only bijective data flow relation

Def 2. broadcast: at least one injective data flow relation

Def 3. reduction: at least one surjective data flow relation

Def 4. opaque: others, general function e.g., transpose, matmul, conv

Aggregating Primitive Operators

Rule-based algorithm

Table 1: Aggregation rules. $\mathcal{G}_p$ and $\mathcal{G}_c$ hold a producer-consumer relation; $\mathcal{G}_a$ is the merged micro-graph.

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

→ Deliverables (safe & efficient micro-graphs) to fusion phase

Some pairs are not in the table, why?

Fusion Phase

Goal: Merge micro-graphs into a single schedulable graph ensuring

1. **Optimal fast memory utilization.**
2. **Parallelism**

This is achieved through 3 Layers

- *Layer I: Polyhedral Loop Fusion*
- *Layer II: Memory Stitching*
- *Layer III: Parallelism Stitching*

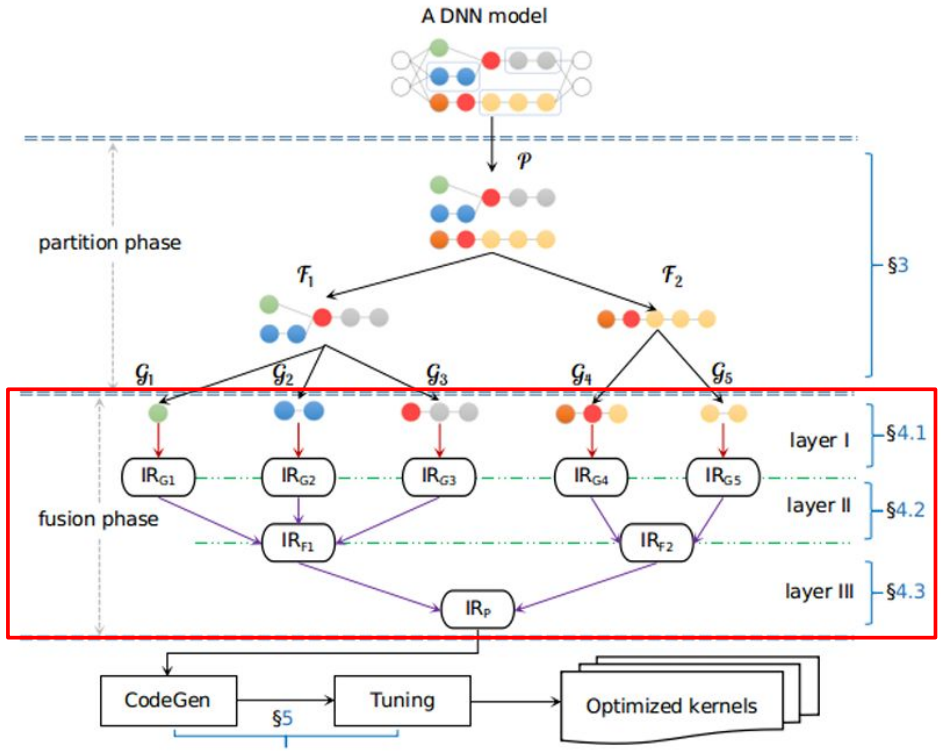


Figure 2: Architecture of APOLLO.

Fusion Phase

Layer I: Polyhedral Loop Fusion

- Micro Graph ⟶ a sequence of polyhedral loop nests ⟶ single optimized kernel via Automatic Kernel Generation (AKG).

```
for i in [0,M)
  for j in [0,N)
    a(i,j)=a(i,j)+bias; //S1
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)); //S2
```

Listing 1: Original loop nests.

```
for i in [0,M)
  for j in [0,N){
    a(i,j)=a(i,j)+bias;
    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)); //S2
  }
```

Listing 2: After fusion.

- Can this AKG apply for real networks without failing?
  - ***Real networks contain lots of reductions, which typically break polyhedral fusion***

Fusion Phase

Layer I: Polyhedral Loop Fusion

- **PANAMERA lives inside Layer I**
  - Handles **reduction-heavy micro-graphs** that break polyhedral fusion.
  - Converts reductions into **canonical forms** (all-reduce, x-reduce, y-reduce).
- **Output:**
  - A clean, canonical loop nest with **known tile sizes**, **known memory use**, and **safe dependencies**.

APOLLO sidesteps AKG's scalability issues by designing micro-graphs → eliminate pathological loop-shifting factors early in the pipeline.

```
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)
```

Listing 3: Original loop nests.

```
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)
```

Listing 4: The canonical form.

```
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) += ...
}
```

Listing 5: After fusion.

Fusion Phase

Layer II: Memory Stitching

- Fuse **across** micro-graphs when loops cannot be graph-fused.
- Reuse **fast-memory tiles** between producer/consumer micro-graphs.
- Conditions for stitching:
  - Tile dependencies safe
  - Combined tile footprint fits local memory

  Benefit: Reduce global-memory traffic; fuse patterns polyhedral model cannot.

Fusion Phase

Layer III: Parallelism Stitching

- Identify **independent / disjoint** micro-graphs (multi-head/tail patterns).
- Combine them into **parallel fused groups** for higher GPU utilization.
- Applies to:
  - Residuals
  - Attention heads
  - Inception-style branches

Benefit: Maximizes hardware throughput beyond serial fusion.

Experimental Setup

- Hardware
  - NVIDIA V100
- Benchmarks
  - BERT, Transformer, Wide & Deep, DeepFM, YOLO-v3
- Metric
  - Throughput (sentences/tokens/samples/images per second)

Single GPU Results

Table 4: Throughput on single GPU.

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%**	-

- Average gain 2.2x faster than MindSpore baseline
  - Layer I contributes 1.91x, remaining improvements from Layer II, III
- APOLLO + MindSpore outperforms TensorFlow + XLA in all workloads

Multi GPU (8) results

Table 5: Throughput on multiple GPUs.

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%**	**-**

- APOLLO avg 2.64x faster than MindSpore baseline
- APOLLO outperforms XLA by 1.18x
- BERT-large - MindSpore introduced inplace_assign ops
  - Saves memory at cost of speed

Limitations

- Incomplete aggregation rules

  Tables 1 and 2 show rule templates that are not exhaustive

- Simplified cost model: the cost model is simple, does not capture all hardware or memory behaviors, fusion decisions aren't globally optimal

- Even so, APOLLO’s rule templates were enough to handle all DNN workloads they tested

- The multilayer fusion outperforms XLA even with the simple cost model

Conclusion

- Problem: Existing fusion systems struggle to find optimal fusion plans for large / complex graphs
- Solution: APOLLO introduces a layer-by-layer partition and fusion framework that automatically identifies fusible subgraphs and applies memory and parallelism stitching
- Result: Noticeable increase in throughput over TensorFlow and XLA on multiple benchmarks

Thank you!

Questions?