ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs

Guyue Huang¹, Yang Bai², Liu Liu³, Yuke Wang¹, Bei Yu², Yufei Ding¹, Yuan Xie⁴

¹ UC Santa Barbara  ² Chinese University of Hong Kong
³ Rensselaer Polytechnic Institute  ⁴ Alibaba DAMO Academy

2023/6/7 MLSys
Outline

• Motivation - Why AI-GPU needs pipelining
• What is pipelining
  • The concept of multi-stage, multi-level pipelining for GPU
• ALCOP framework
  • Schedule transformation
  • Program transformation
  • Analytical model guided auto-tuning
• Evaluations
Motivation

Integrated accelerator (Tensor Core)

“Computation Wall”

Reduce Memory Access!
Intra-kernel: Tiling
Inter-kernel: Fusion

“Memory Wall”

Saturate Parallelism!
Intra-kernel: Pipelining
Inter-kernel: multi-stream

“Utilization Wall”

Road to Efficient DNN on GPU
The Scope of this Paper

• The goal is to overlap the data-loading and computation within a GPU kernel (GEMM, Conv2d, etc)
  • Data-loading refers to: global-mem → shared-mem → register files
  • Computation refers to: tensor core computation

• A compiler-based approach
  • As opposed to hand-written kernels which already exist in CUTLASS
  • TVM is our playground
What is Load-Compute Pipelining?

Matrix Multiplication

Without Pipelining

chunk 1
chunk 2
...
...
...
...

Data Loading

Computation

Idle
Idle
...

Idle
Idle

Time
What is Load-Compute Pipelining?

Data Loading | Computation | w/o Pipelining
--- | --- | ---
| | | ...

Data Loading | Computation | w/ Pipelining
--- | --- | ---
| | | ...

Computation utilization is improved!
Pipelining Improves Utilization

Performance of a $2048 \times 2048 \times 2048$ matrix-multiplication. Tested on NVIDIA A100.

Throughput (TFlop/s)

- No Pipeline
- 4-Stage Pipeline

High Utilization from Pipeline Parallelism

Low Utilization due to Small Inter-Tile Parallelism

Threadblock Tiling Size
Challenge-1: Multi-Stage Pipelining

Limited number of stages (e.g. double-buffering) cannot fully hide the memory latency.
Challenge-1: Multi-Stage Pipelining

We need to support *arbitrary number of stages* (auto-tuning the stage count)
Challenge-2: Multi-Level Pipelining

Knowing how to pipeline a load-use loop is not enough! (We need to handle arbitrary levels of nested load-use loop. Moreover, recursively using one-level pipelining won’t work.)
Can we design a compiler framework to automate the complex multi-stage multi-level pipelining optimization?
ALCOP Overview

Schedule Transformation (detect, annotate)

Lower to TIR

Program Transformation (transform IR)

Build

Analytical-Model Guided Tuning (choose pipeline stage count)

Assign scheduling parameters
Tensor Algorithm

# MatMul

C[i,j] = sum(A[i,k]*B[k,j],
    reduce_axis=(k,))
# MatMul
\[ C[i,j] = \text{sum}(A[i,k]*B[k,j], \text{reduce_axis}=(k,)) \]

# cache the input
A_shared = cache_read(A)
A_reg = cache_read(A_shared)

# tiling
C.tile(...); C.reorder(...)

# annotate pipeline buffers
A_shared.pipeline(stage=3)
A_reg.pipeline(stage=2)
/* Declare Buffers */
Alloc(A_shared, size_of_TB_chunk);
Alloc(A_reg, size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...] = A[k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        /* load into registers */
        A_reg[...] = A_shared[k1, ...];
        /* compute with Tensor Core */
        wmma(A_reg, ...);
ALCOP Framework

TensorIR (before transformation)

Program Transformation

TensorIR (after transformation)

Tensor Algorithm

Parse to dataflow graph

Schedule Transformation

ALCOP Pipeline Buffer Detection

Interact

Tiling, Fusion…

Lower to TensorIR

Program Transformation

ALCOP Pipeline Transformation Pass

Existing passes

1. // TransformIR
2. "// Define loop extents as variables for code brevity"
3. extent_ko, extent_ki = (C_k / TB_tile_k), (TB_tile_k / Warp_tile_k)
4. /* Declare buffer sizes */
5. alloc A_shared[3][...]
6. alloc A_reg[2][...]
7. /* Prologue for A_shared and A_reg */
8. for ko in 0..2:
9.     /* load into shared memory buffer (same as Line 15-17) */
10. for ki in 0..1:
11.     /* load into register buffer (same as Line 24-27) */
12. for ko in 0..extent_ko:
13.     /* load into shared memory buffer */
14.     /* guard data copy with producer primitives at Line 15 and Line 17 */
15.     A_shared.producer_acquire()
16.     async_memcpy(A_shared[(ko + 2)%3][...], A[...,(ko + 2)%extent_ko%3])
17.     A_shared.producer_commit()
18. /* compute with data in shared memory buffer */
19. /* guard data usage with consumer primitives at Line 22 and Line 30 */
20. for ki in 0..extent_ki:
21.     if (ki + 1)%2 = 0:
22.     A_shared.consumer_wait()
23.     /* load into register buffer */
24.     async_memcpy(A_reg[(ki + 1)%2][...], A_shared[(ko + (ki + 1)/extent_ki)%3][...,(ki + 1)%extent_ki%3])
25.     /* tensor-core computation with data in register buffer */
26.     wmma(A_reg[(ki%2)][...], ...)
27.     A_shared.consumer_release()

Existing passes

Parse to dataflow graph

Schedule Transformation

ALCOP Pipeline Buffer Detection

Interact

Tiling, Fusion…

Lower to TensorIR

Existing passes

15
ALCOP Framework

# cache the input
A_shared = cache_read(A)
A_reg = cache_read(A_shared)
# tiling
C.tile(?); C.reorder(…)

# annotate pipeline buffers
A_shared.pipeline(stage=?)
A_reg.pipeline(stage=?)

Tensor Algorithm

Schedule Transformation

Interact

Tiling, Fusion…

Lower to TensorIR

Program Transformation

Existing passes

Schedule parameters

Build and evaluate

Auto-Tuning

Optimized Tensor Program
# ALCOP Framework

## Schedule Transformation

1. **Tensor Algorithm**
2. **Schedule Transformation**
3. **ALCOP Pipeline Buffer Detection**
   - Interact
   - Tiling, Fusion...
4. **Lower to TensorIR**
5. **Program Transformation**
   - **ALCOP Pipeline Transformation Pass**
   - **Existing passes**
6. **Build and evaluate**
   - **Auto-Tuning**
   - **ALCOP Analytical Perf. Model**
   - **ML-based Perf. Model**
7. **Optimized Tensor Program**

### Control Flow

- **Schedule parameters**
- **# cache the input**
  - A_shared = cache_read(A)
  - A_reg = cache_read(A_shared)
  - # tiling
  - C.tile(?)
  - C.reorder(...)
- **# annotate pipeline buffers**
  - A_shared.pipeline(stage=?)
  - A_reg.pipeline(stage=?)
GEMM Example

Tensor Algorithm

```python
# MatMul
C[i,j] = sum(A[i,k]*B[k,j],
    reduce_axis=(k,))
```

Schedule Statements

```python
# caching
A_shared = cache_read(A)
A_reg = cache_read(A_shared)
# tiling
C.tile(...); C.reorder(...)

# pipelining
A_shared.pipeline(stage=3)
A_reg.pipeline(stage=2)
```

Unoptimized IR

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Shared-memory level load
- Register level load
- Shared-memory level compute
- Register level compute
GEMM Example

Tensor Algorithm

# MatMul
C[i,j] = sum(A[i,k]*B[k,j],
    reduce_axis=(k,))

Schedule Statements

# caching
A_shared = cache_read(A)
A_reg = cache_read(A_shared)
# tiling
C.tile(...); C.reorder(...)

# pipelining
A_shared.pipeline(stage=3)
A_reg.pipeline(stage=2)

Unoptimized IR

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...] = A[k0, ...];
    /* compute with data in shared memory */
}
GEMM Example

Tensor Algorithm

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...];
    for (k1 = 0; k1 < extent_k1; k1++) {
        A_reg[...] = A_shared[k1, ...]; // load into registers
        wmma(A_reg, ...);}} // compute with Tensor Core

Schedule Statements

/* load k0-th chunk into the shared memory */
A_shared[...] = A[k0, ...];

/* compute with data in shared memory */
for (k1 = 0; k1 < extent_k1; k1++) {
    A_reg[...] = A_shared[k1, ...]; // load into registers
    wmma(A_reg, ...);}} // compute with Tensor Core

Unoptimized IR

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...] = A[k0, ...];
    for (k1 = 0; k1 < extent_k1; k1++) {
        A_reg[...] = A_shared[k1, ...]; // load into registers
        wmma(A_reg, ...);}} // compute with Tensor Core
GEMM Example

Tensor Algorithm

```c
/* Declare Buffers */
A = Alloc(A_shared, size_of_TB_chunk);
Alloc(A_reg, size_of_warp_chunk);

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...]
        = A[k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        A_reg[...]
            = A_shared[k1, ...]; // load into registers
        // compute with Tensor Core
        wmma(A_reg, ...);} }}
```

Unoptimized IR

```
/* Declare Buffers */
Alloc(A_shared, size_of_TB_chunk);
Alloc(A_reg, size_of_warp_chunk);

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...]
        = A[k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        A_reg[...]
            = A_shared[k1, ...]; // load into registers
        wmma(A_reg, ...);} }
```

Schedule Statements

- # MatMul
  
  \[
  C[i,j] = \text{sum}(A[i,k]*B[k,j], \text{reduce_axis}=(k))
  \]

- # caching
  
  A_shared = cache_read(A)
A_reg = cache_read(A_shared)

- # tiling
  
  C.tile(...); C.reorder(...)

- # pipelining
  
  A_shared.pipeline(stage=3)
A_reg.pipeline(stage=2)
Before

```c
/* Declare Buffers */
Alloc(A_shared, size_of_TB_chunk);
Alloc(A_reg, size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...] = A[k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        // load into registers
        A_reg[...] = A_shared[k1, ...];
        wmma(A_reg, ...); // compute with TC
    }
}
```

After

```c
/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[...] = A[k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        // load into registers
        A_reg[...] = A_shared[k1, ...];
        wmma(A_reg, ...); // compute with TC
    }
```
/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[k0, …] = A[k0, …];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        // load into registers
        A_reg[k1, …] = A_shared[k0, k1, …];
        wmma(A_reg, …); // compute with TC
    }
}

/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);

/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[k0 + 2, …] = A[k0 + 2, …];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        // load into registers
        A_reg[k1 + 1, …] = A_shared[k0, k1 + 1, …];
        wmma(A_reg[k1], …); // compute with TC
/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
  /* load k0-th chunk into the shared memory */
  A_shared[(k0 + 2) % 3] = A[(k0 + 2) % extent_k0, ...];
  /* compute with data in shared memory */
  for (k1 = 0; k1 < extent_k1; k1++) {
    // load into registers
    A_reg[(k1 + 1) % 2] = A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, ...];
    wmma(A_reg[(k1 + 1) % 2], ...); // compute with TC
  }
}

/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
  /* load k0-th chunk into the shared memory */
  A_shared[(k0 + 2) % 3] = A[(k0 + 2) % extent_k0, ...];
  /* compute with data in shared memory */
  for (k1 = 0; k1 < extent_k1; k1++) {
    // load into registers
    A_reg[(k1 + 1) % 2] = A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, ...];
    wmma(A_reg[(k1 + 1) % 2], ...); // compute with TC
/* Declare Buffers */
Alloc(A_shared, 3*size_of_TB_chunk);
Alloc(A_reg, 2*size_of_warp_chunk);
/* Main loop */
for (k0 = 0; k0 < extent_k0; k0++) {
    /* load k0-th chunk into the shared memory */
    A_shared[(k0 + 2) % 3] = A[(k0 + 2) % extent_k0, ...];
    /* compute with data in shared memory */
    for (k1 = 0; k1 < extent_k1; k1++) {
        // load into registers
        A_reg[(k1 + 1) % 2] = \n            A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, ...];
        wmma(A_reg[(k1 % 2)], ...); // compute with TC
    }
}
/* Prologue for Loop k0 */
for (k0 = 0; k0 < 2; k0++) {
    A_shared[k0] = A[k0, ...];
}
/* Prologue for Loop k1 */
for (k1 = 0; k1 < 1; k1++) {
    A_reg[k1] = A_shared[0][k1];
}
/* Main loop (unchanged) */
for (k0 = 0; k0 < extent_k0; k0++) { ... }
Before

1 /* Declare Buffers */
2 Alloc(A_shared, 3*size_of_TB_chunk);
3 Alloc(A_reg, 2*size_of_warp_chunk);
4 /* Main loop */
5 for (k0 = 0; k0 < extent_k0; k0++) {
6     /* load k0-th chunk into the shared memory */
7     A_shared[(k0 + 2) % 3] = A[(k0 + 2) % extent_k0, ...];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10        // load into registers
11        A_reg[(k1 + 1) % 2][...]
12        async_memcpy(A_reg[(k1 + 1) % 2], A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1], ...);
13        wmma(A_reg[k1 % 2], ...); // compute with TC
14    }
15 }

After

1 /* Declare Buffers (omitted) */
2 /* Prologue (omitted) */
3 /* Main loop */
4 for (k0 = 0; k0 < extent_k0; k0++) {
5     /* load k0-th chunk into the shared memory */
6     A_shared.producer_acquire();
7     async_memcpy(A_shared[(k0 + 2) % 3], A[(k0 + 2) % extent_k0, ...]);
8     A_shared.producer_commit();
9     /* compute with data in shared memory */
10    for (k1 = 0; k1 < extent_k1; k1++) {
11        // load into registers
12        A_reg[(k1 + 1) % 2][...]
13        async_memcpy(A_reg[(k1 + 1) % 2], A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1], ...);
14        wmma(A_reg[k1 % 2], ...); // compute with TC
15    }
16     A_shared.consumer_release();
Single Operator Results

Settings

• Hardware: NVIDIA A100
  • A10—SMX4 40GB
• Software: cuda v11.4; TVM v0.8
• Benchmark: all FP16 kernels on Tensor Cores
• Baselines:
  • Vanilla TVM
  • TVM with double-buffering (TVM-DB)
  • *We augment our work and all baselines with shared-memory swizzling to avoid bank conflict overhead.*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Tag</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMul</td>
<td>MM_Square_1k</td>
<td>1024/1024/1024</td>
</tr>
<tr>
<td></td>
<td>MM_Square_2k</td>
<td>2048/2048/2048</td>
</tr>
<tr>
<td></td>
<td>MM_Square_4k</td>
<td>4096/4096/4096</td>
</tr>
<tr>
<td></td>
<td>MM_BERT_MHA1</td>
<td>512/768/768</td>
</tr>
<tr>
<td></td>
<td>MM_BERT_MHA2</td>
<td>512/2304/768</td>
</tr>
<tr>
<td></td>
<td>MM_BERT_FC1</td>
<td>512/3072/768</td>
</tr>
<tr>
<td></td>
<td>MM_BERT_FC2</td>
<td>512/768/3072</td>
</tr>
<tr>
<td></td>
<td>MM_Conv1x1_1</td>
<td>200704/64/256</td>
</tr>
<tr>
<td></td>
<td>MM_Conv1x1_2</td>
<td>3136/512/256</td>
</tr>
<tr>
<td></td>
<td>MM_RN50_FC</td>
<td>1024/64/2048</td>
</tr>
<tr>
<td>BMM</td>
<td>BMM_BERT_QK</td>
<td>12/512/512/64</td>
</tr>
<tr>
<td></td>
<td>BMM_BERT_SV</td>
<td>12/512/64/512</td>
</tr>
<tr>
<td></td>
<td>BMM_GPT2_QK</td>
<td>12/1024/1024/64</td>
</tr>
<tr>
<td></td>
<td>BMM_GPT2_SV</td>
<td>12/1024/64/1024</td>
</tr>
<tr>
<td>Conv2D</td>
<td>Conv2D_RN50_1</td>
<td>64/56/56/64/64/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv2D_RN50_2</td>
<td>64/28/28/128/128/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv2D_RN50_3</td>
<td>64/14/14/256/256/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv2D_RN50_4</td>
<td>64/7/7/512/512/3/3/1</td>
</tr>
<tr>
<td>Conv3D</td>
<td>Conv3D1</td>
<td>64/14/56/56/64/64/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv3D2</td>
<td>64/14/28/28/128/128/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv3D3</td>
<td>64/14/14/256/256/3/3/1</td>
</tr>
<tr>
<td></td>
<td>Conv3D4</td>
<td>64/14/7/7/512/512/3/3/1</td>
</tr>
</tbody>
</table>
**Single Operator Results**

- Average 1.23x (max 1.73x) over vanilla TVM
- TVM-DB does not bring obvious speedup over TVM
- Ablation study: only 1.01x if without multi-stage pipelining and 1.13x if without multi-level pipelining

<table>
<thead>
<tr>
<th>Operator</th>
<th>TVM Baseline</th>
<th>TVM DB</th>
<th>ALCOP w/o MS,ML</th>
<th>ALCOP w/o ML</th>
<th>ALCOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM_Square_1k</td>
<td>1.32</td>
<td>1.31</td>
<td>1.34</td>
<td>1.28</td>
<td>1.69</td>
</tr>
<tr>
<td>MM_Square_2k</td>
<td>1.23</td>
<td>1.17</td>
<td>1.09</td>
<td>1.08</td>
<td>1.73</td>
</tr>
<tr>
<td>MM_BERT_MHA1</td>
<td>1.09</td>
<td>1.19</td>
<td>1.01</td>
<td>1.08</td>
<td>1.54</td>
</tr>
<tr>
<td>MM_BERT_FC1</td>
<td>1.08</td>
<td>1.21</td>
<td>1.00</td>
<td>1.20</td>
<td>1.36</td>
</tr>
<tr>
<td>MM_Conv1_x_1</td>
<td>0.95</td>
<td>1.22</td>
<td>1.17</td>
<td>1.00</td>
<td>1.17</td>
</tr>
<tr>
<td>MM_Conv1_x_2</td>
<td>1.22</td>
<td>1.36</td>
<td>1.17</td>
<td>1.00</td>
<td>1.13</td>
</tr>
<tr>
<td>MM_RN50_FC</td>
<td>1.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMM_BERT_OK</td>
<td>1.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMM_GPT2_OK</td>
<td>1.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D_RN50_1</td>
<td>1.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D_RN50_2</td>
<td>1.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D_RN50_3</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D_RN50_4</td>
<td>1.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv2D</td>
<td>1.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv3D</td>
<td>1.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Performance Normalized to TVM Baseline**

![Bar chart showing performance normalization](chart.png)
End-to-end results

<table>
<thead>
<tr>
<th>Model</th>
<th>End-to-end speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vs. TVM</td>
</tr>
<tr>
<td>BERT</td>
<td>1.15</td>
</tr>
<tr>
<td>BERT-Large</td>
<td>1.18</td>
</tr>
<tr>
<td>GPT-2</td>
<td>1.15</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>1.02</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>1.06</td>
</tr>
<tr>
<td>VGG-16</td>
<td>1.10</td>
</tr>
</tbody>
</table>

All Operators in the Model that can Use Pipelining

Speedup over TVM vs. TVM vs. XLA

- BERT
- BERT-Large
- GPT-2
- ResNet-18
- ResNet-50
- VGG-16
Summary

• This work presents a DL-compiler based workflow for automatic pipelining targeting GPU

• Our framework has 3 steps
  • Schedule transformation to detect and annotate buffers to pipeline
  • Program transformation to implement pipelining in TensorIR
  • Auto-tuning under analytical model guidance
  • A pipelining-aware GPU analytical performance model + tuning workflow

• Key results
  • Single operator: average 1.23x (max 1.73x) over vanilla TVM
  • End-to-end: 1.02-1.18x over vanilla TVM, 1.01-1.64x over XLA
  • Using analytical model to pre-train AutoTVM’s ML model can significantly improve the search efficiency

Code: https://github.com/hgyhungry/alcop-artifact