UC SANTA BARBARA 香港中文大學 The Chinese University of Hong Kong Rensselaer Alibaba

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

Guyue Huang[1], Yang Bai[2], Liu Liu[3], Yuke Wang[1], Bei Yu[2], Yufei Ding[1], Yuan Xie[4]

[1] UC Santa Barbara [2] Chinese University of Hong Kong

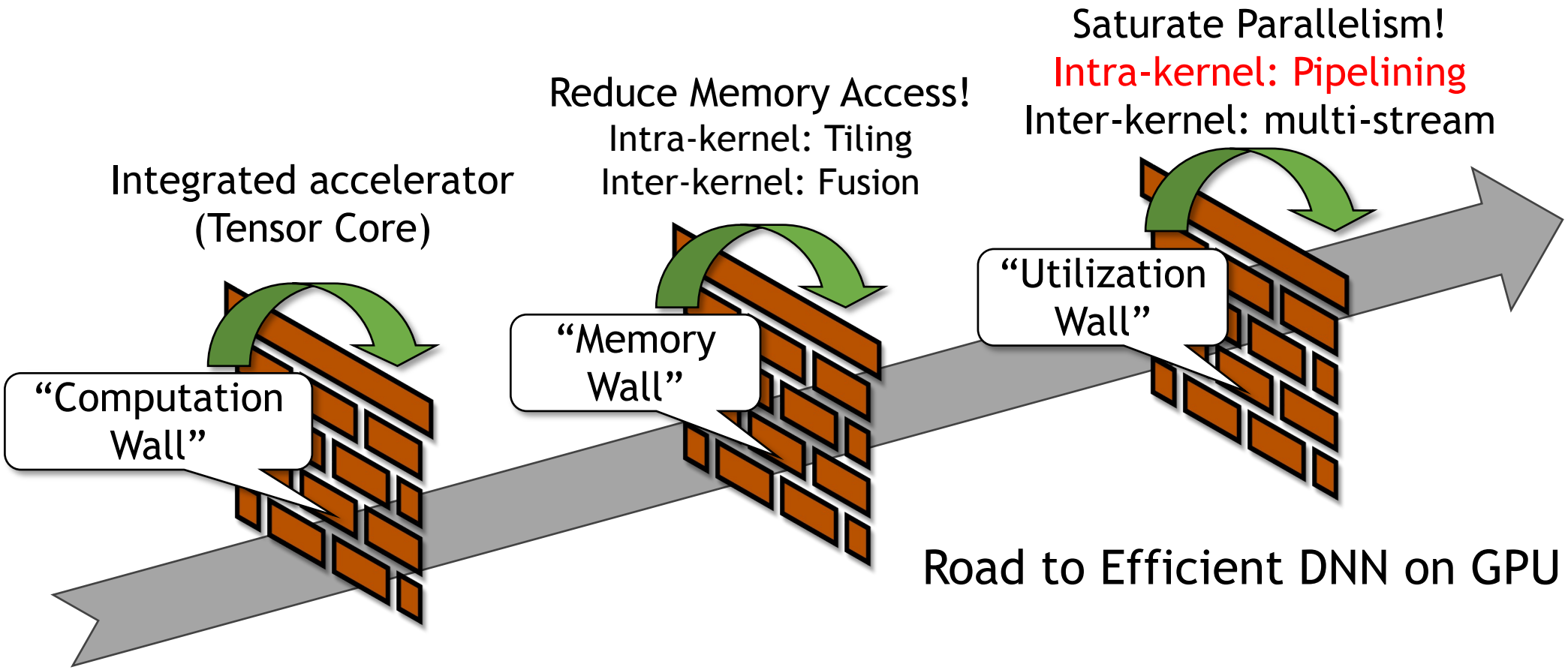
[3] Rensselaer Polytechnic Institute [4] Alibaba DAMO Academy

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

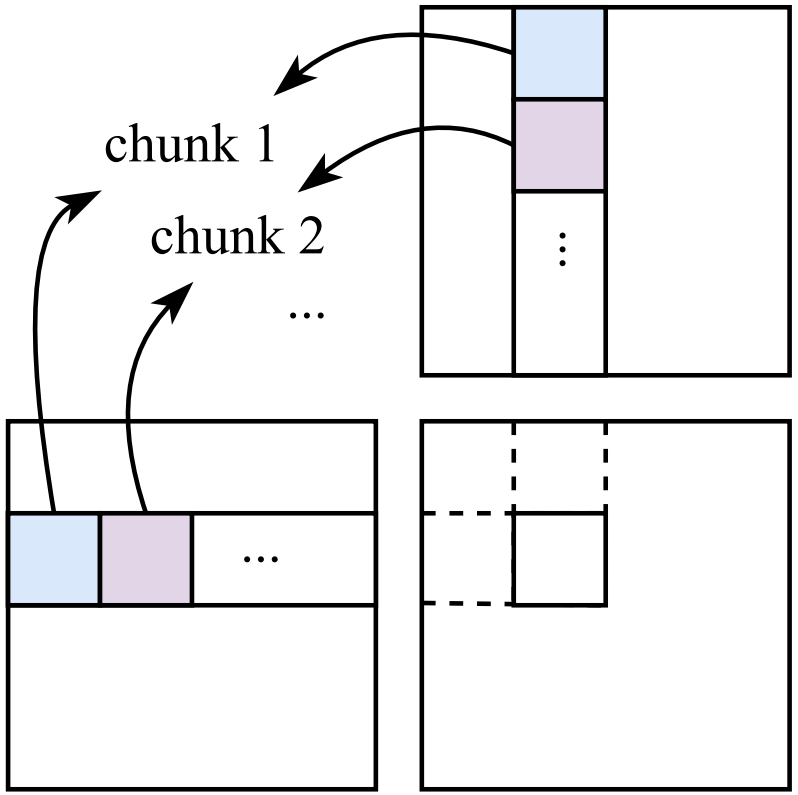

Integrated accelerator
(Tensor Core)
Reduce Memory Access!
Intra-kernel: Tiling
Inter-kernel: Fusion
Saturate Parallelism!
Intra-kernel: Pipelining
Inter-kernel: multi-stream
“Computation Wall”
“Memory Wall”
“Utilization Wall”
Road to Efficient DNN on GPU

The Scope of this Paper

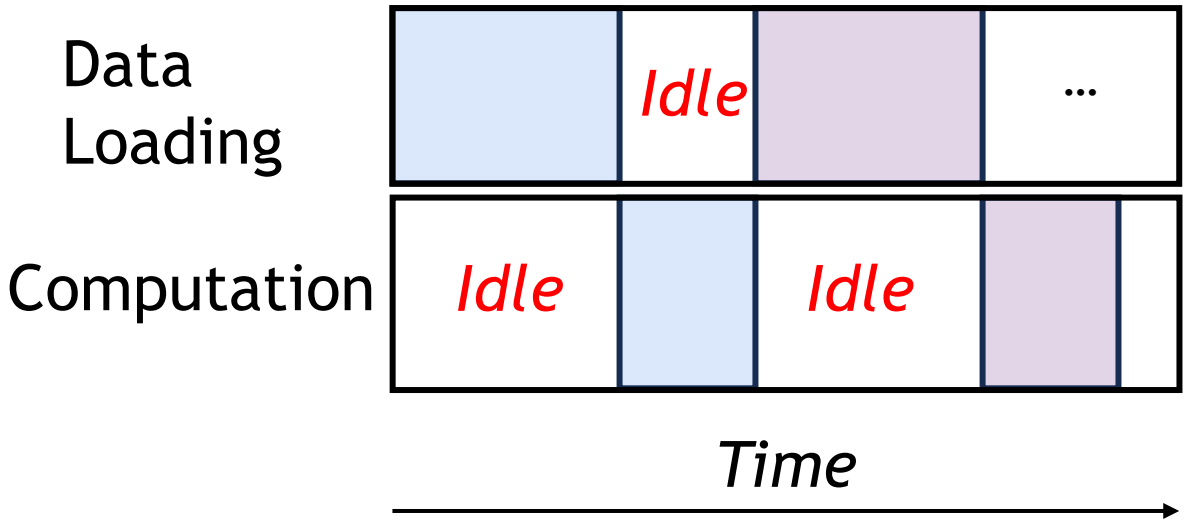
- The goal is to overlap the data-loading and computation **within a GPU kernel** (GEMM, Conv2d, etc)
  - Data-loading refers to: global-mem → shared-mem → register files
  - Computation refers to: tensor core computation
- A compiler-based approach
  - As opposed to hand-written kernels which already exist in CUTLASS
  - TVM is our playground

What is Load-Compute Pipelining?

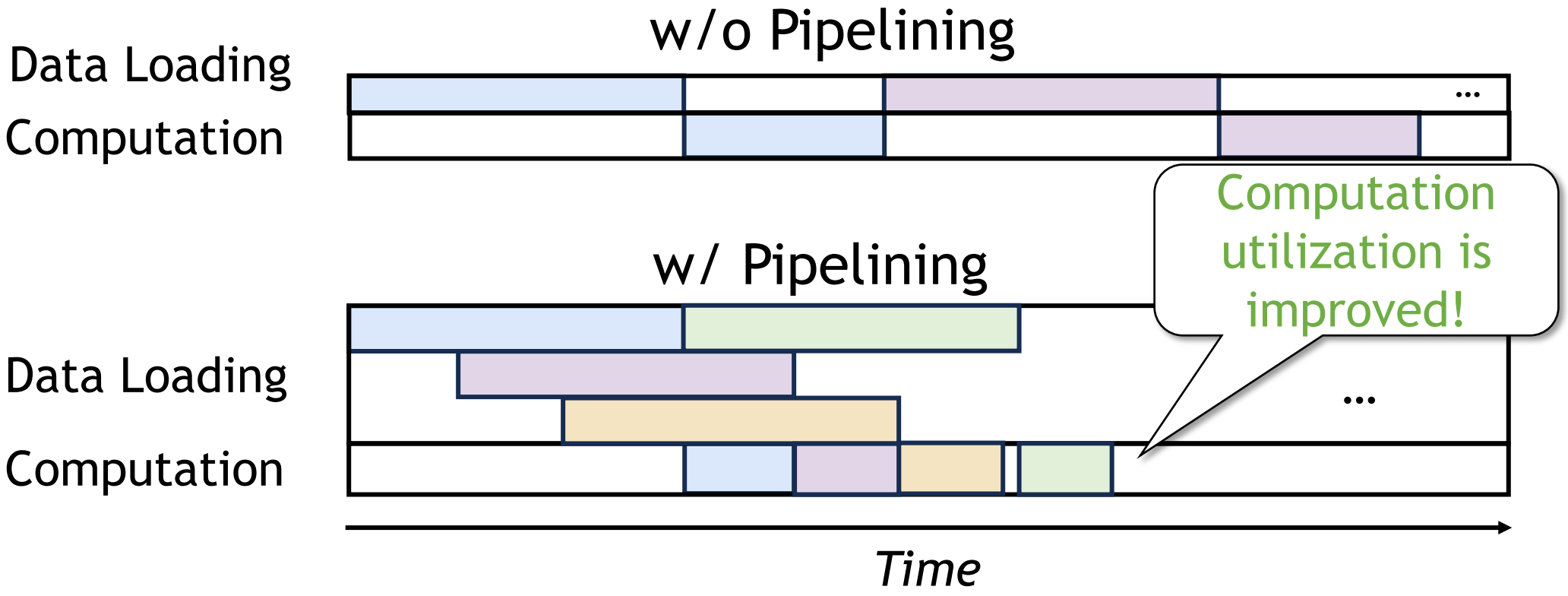
Matrix Multiplication



Without Pipelining

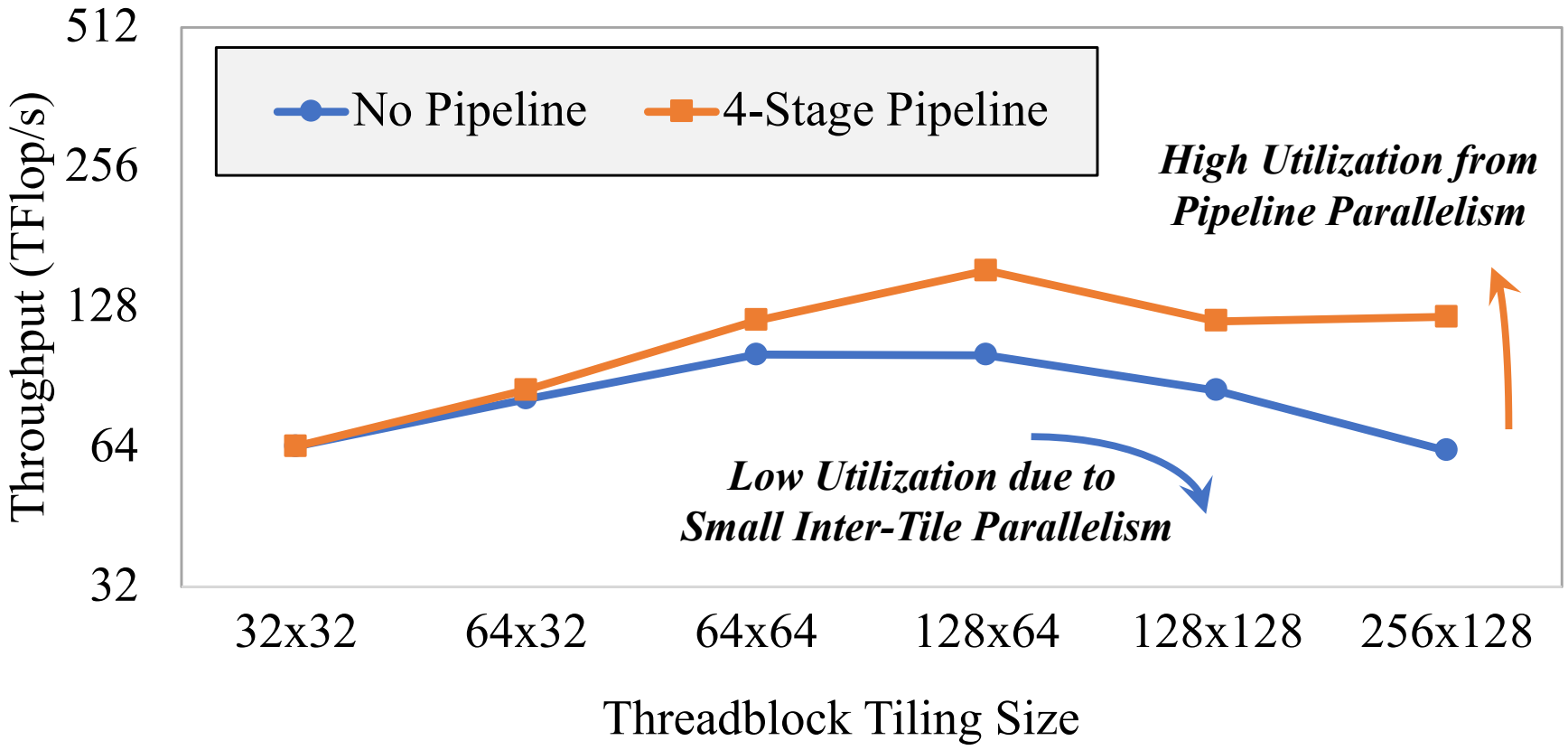


What is Load-Compute Pipelining?

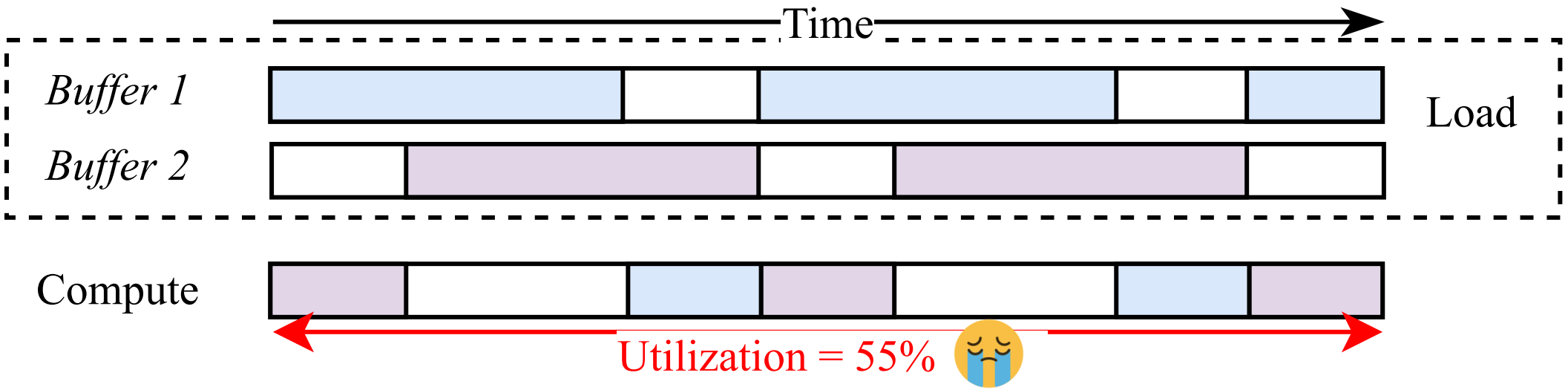


Pipelining Improves Utilization

Performance of a 2048×2048× 2048 matrix-multiplication. Tested on NVIDIA A100.

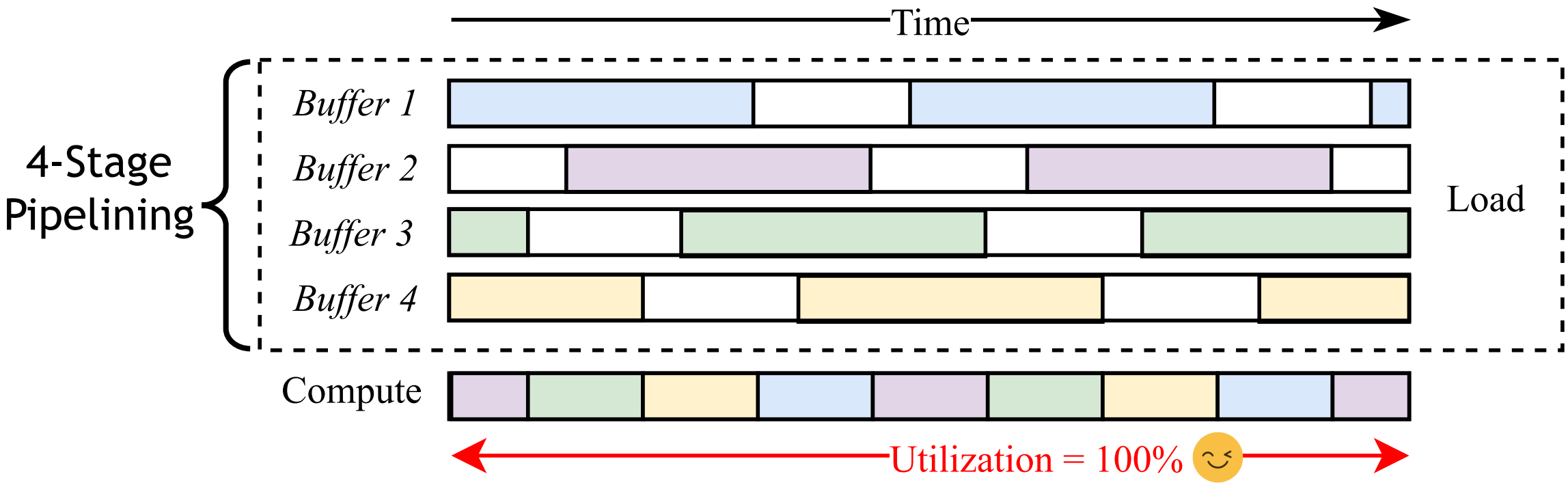


Challenge-1: Multi-Stage Pipelining



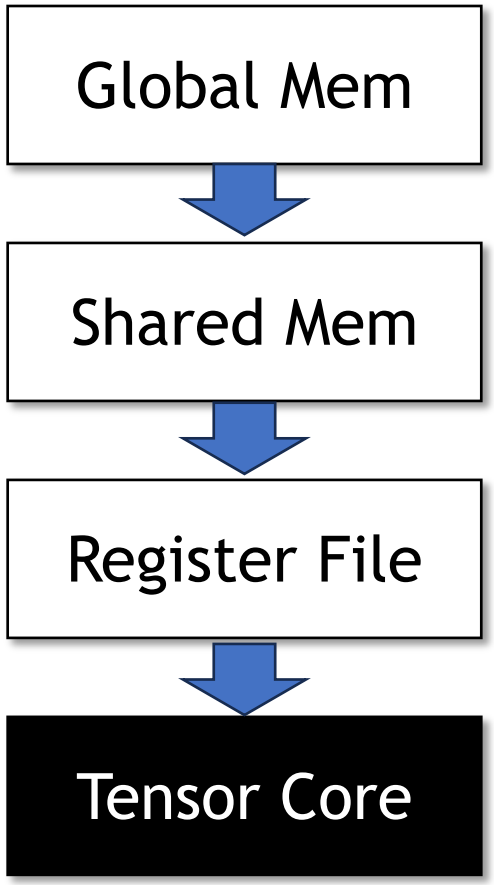
Limited number of stages (e.g. double-buffering) cannot fully hide the memory latency.

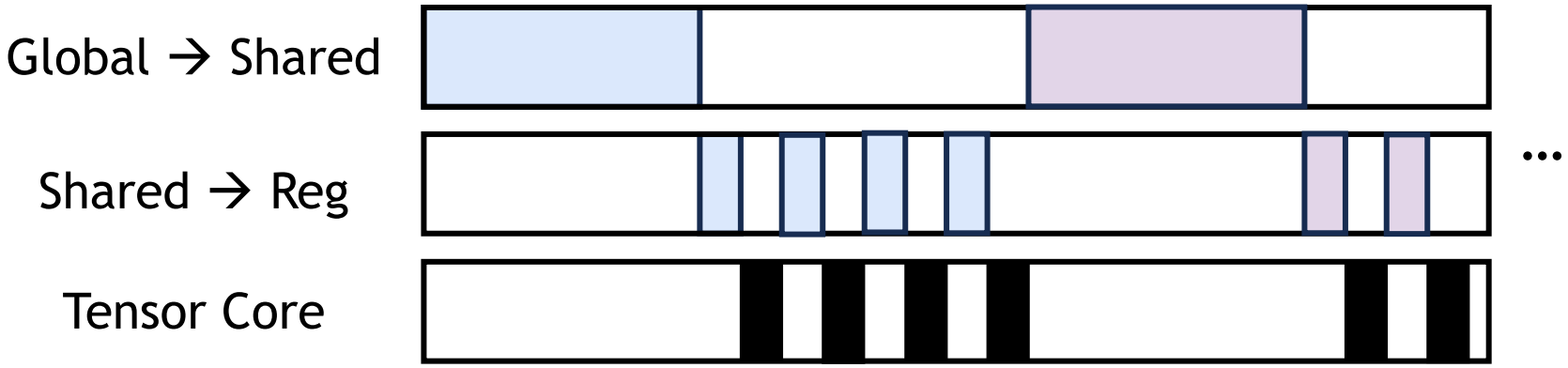
Challenge-1: Multi-Stage Pipelining



We need to support ***arbitratry number of stages*** (auto-tuning the stage count)

Challenge-2: Multi-Level Pipelining

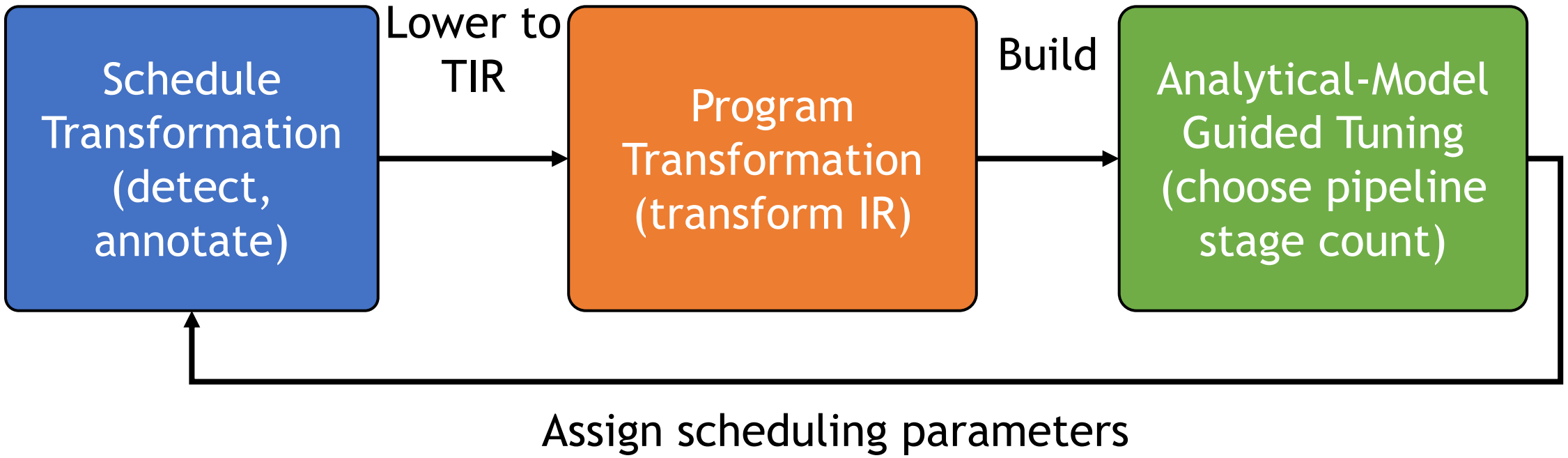




Knowing how to pipeline a load-use loop is not enough! (We need to handle arbitrary levels of nested load-use loop. Moreover, recursively using one-level pipelining won't work.)

Can we design a compiler framework to automate the complex multi-stage multi-level pipelining optimization?

ALCOP Overview



ALCOP Framework

Tensor Algorithm

Tensor Algorithm

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

ALCOP Framework

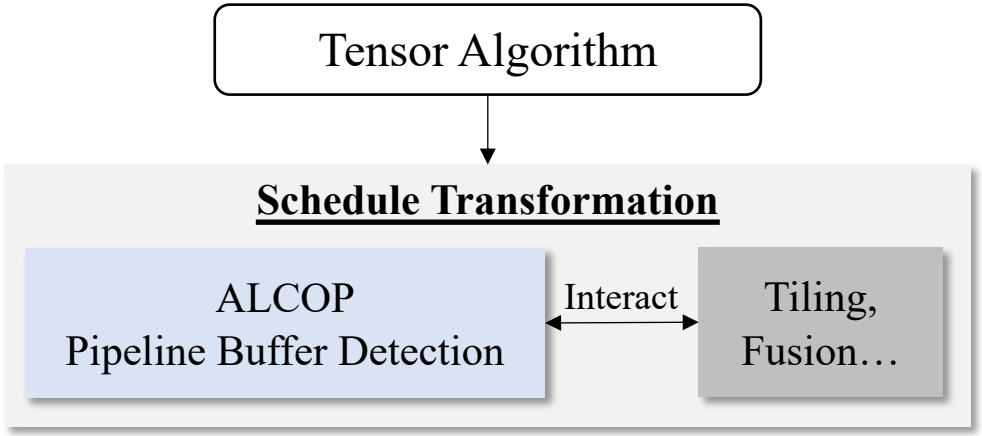
Tensor Algorithm

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

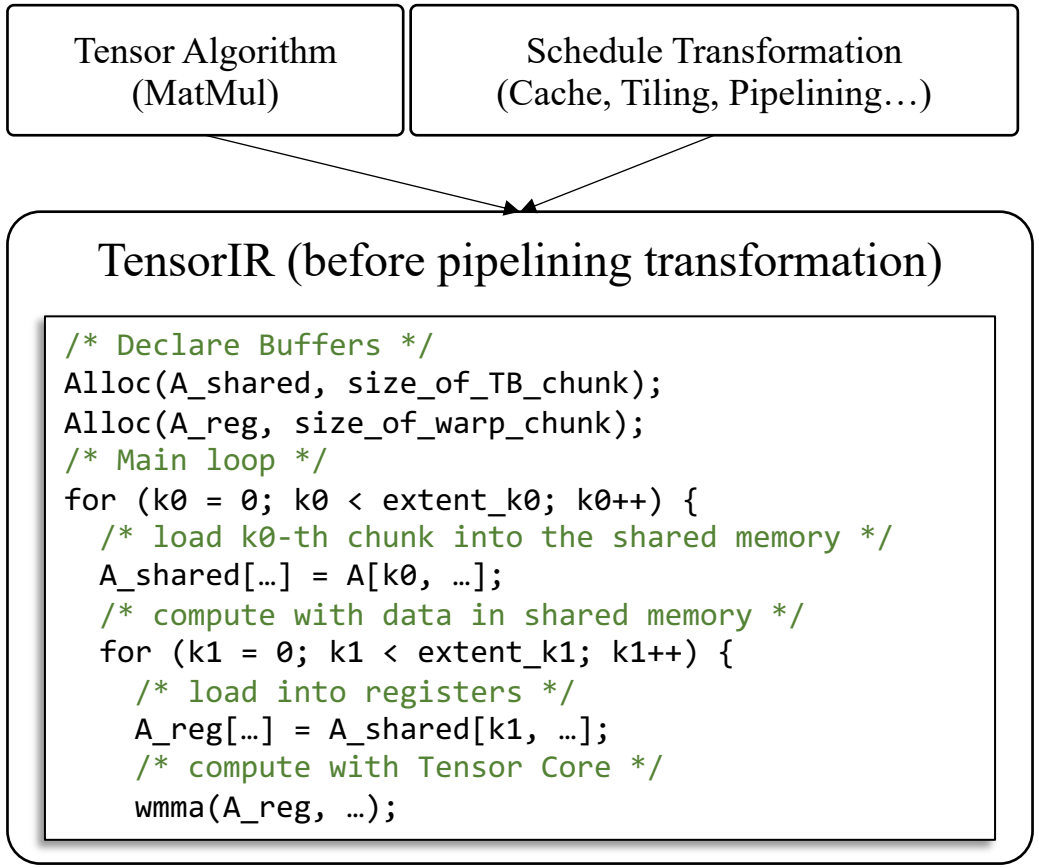
Schedule Transformation

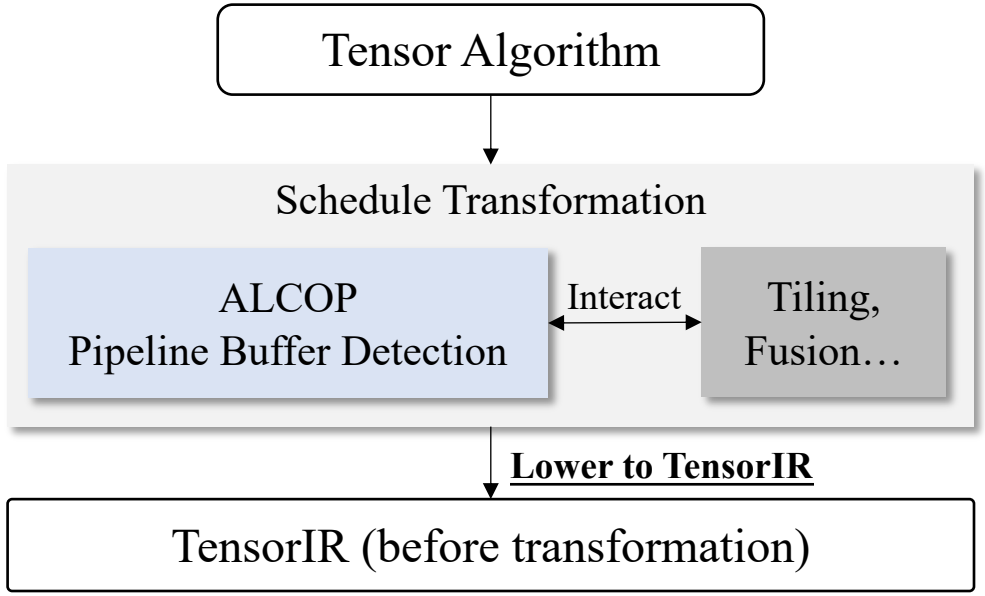
```
# cache the input
A_shared = cache_read(A)
A_reg = cache_read(A_shared)
# tiling
C.tile(…); C.reorder(…)
```

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



ALCOP Framework





ALCOP Framework

TensorIR (before transformation)

Program Transformation

TensorIR (after transformation)

```
TransformedIR:
/* define loop extents as variables for code brevity */:
extent_ko, extent_ki = (C_k / TB_tile_k), (TB_tile_k / Warp_tile_k)

/* Declare buffer size. */
alloc A_shared[3][...]
alloc A_reg[2][...]

/* Prologue for A_shared and A_reg */
for ko in 0 .. 2:
    /* load into shared memory buffer (same as Line 15-17) */
for ki in 0 .. 1:
    /* load into reg. buffer (same as Line 24-27) */

for ko in 0 .. extent_ko:

    /* load into shared memory buffer  */
    /* guard data copy with producer primitives at Line 15 and Line 17 */
    A_shared.producer_acquire()
    async_memcpy(A_shared [ (ko + 2) % 3 ][...], A[..., (ko + 2) % extent_ko ])
    A_shared.producer_commit()

    /* compute with data in shared memory buffer */
    /* guard data usage with consumer primitives at Line 22 and Line 30 */
    for ki in 0 .. extent_ki:
        if ( ki + 1 ) % 2 == 0:

            A_shared.consumer_wait()
            /* load into register buffer */
            async_memcpy(
                A_reg [ (ki + 1) % 2 ][...],
                A_shared [ (ko + ((ki+1) / extent_ki) ) % 3 ][..., (ki + 1) % extent_ki ]
            )
            /* tensor-core compute with data in register buffer */
            wmma(A_reg [ (ki % 2) ] [...], ...)
    A_shared.consumer_release()
```

Tensor Algorithm

Parse to dataflow graph

Schedule Transformation

ALCOP
Pipeline Buffer Detection

Interact

Tiling,
Fusion…

Lower to TensorIR

Program Transformation

ALCOP
Pipeline Transformation Pass

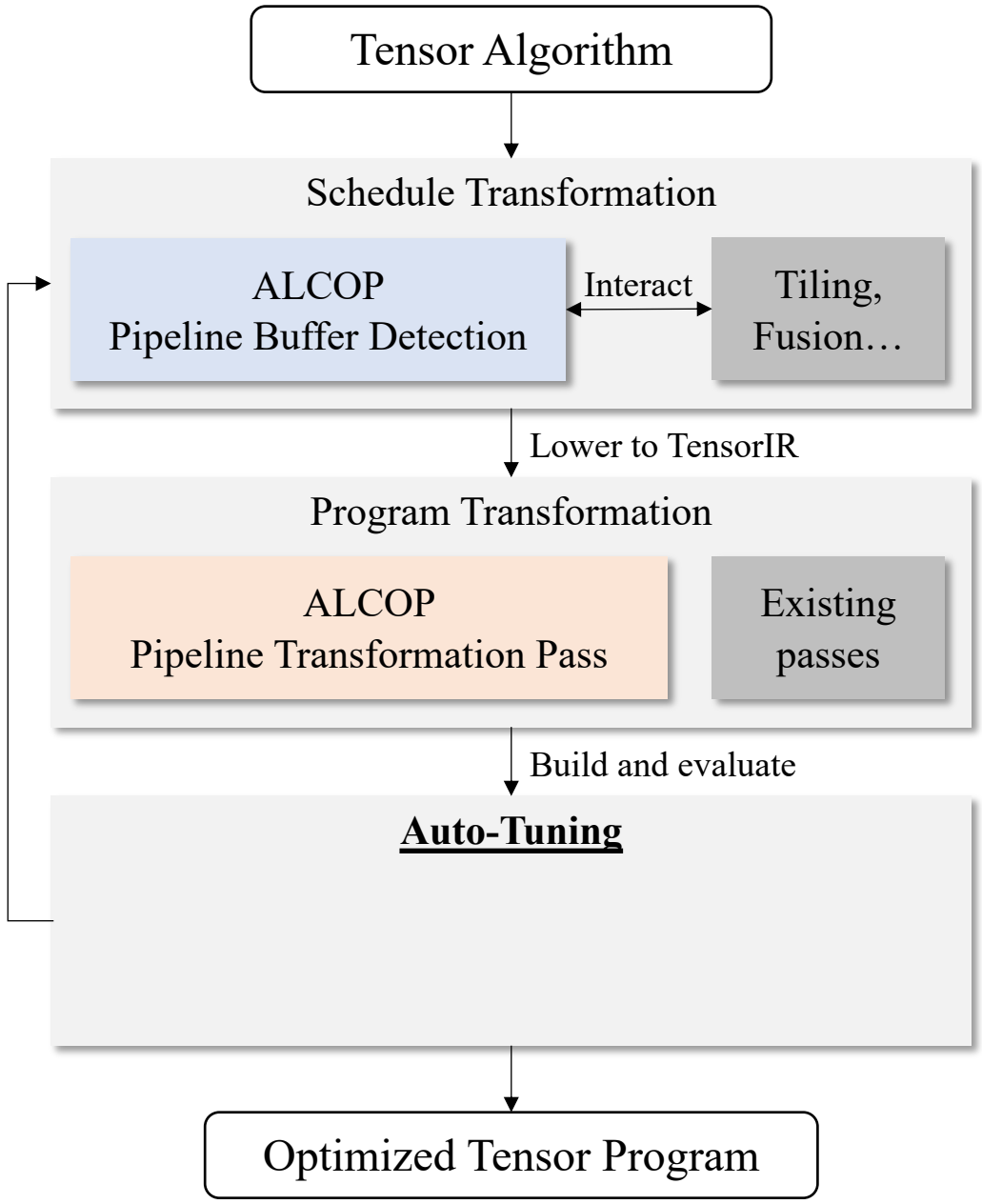
Existing
passes

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=?)
```

Schedule parameters

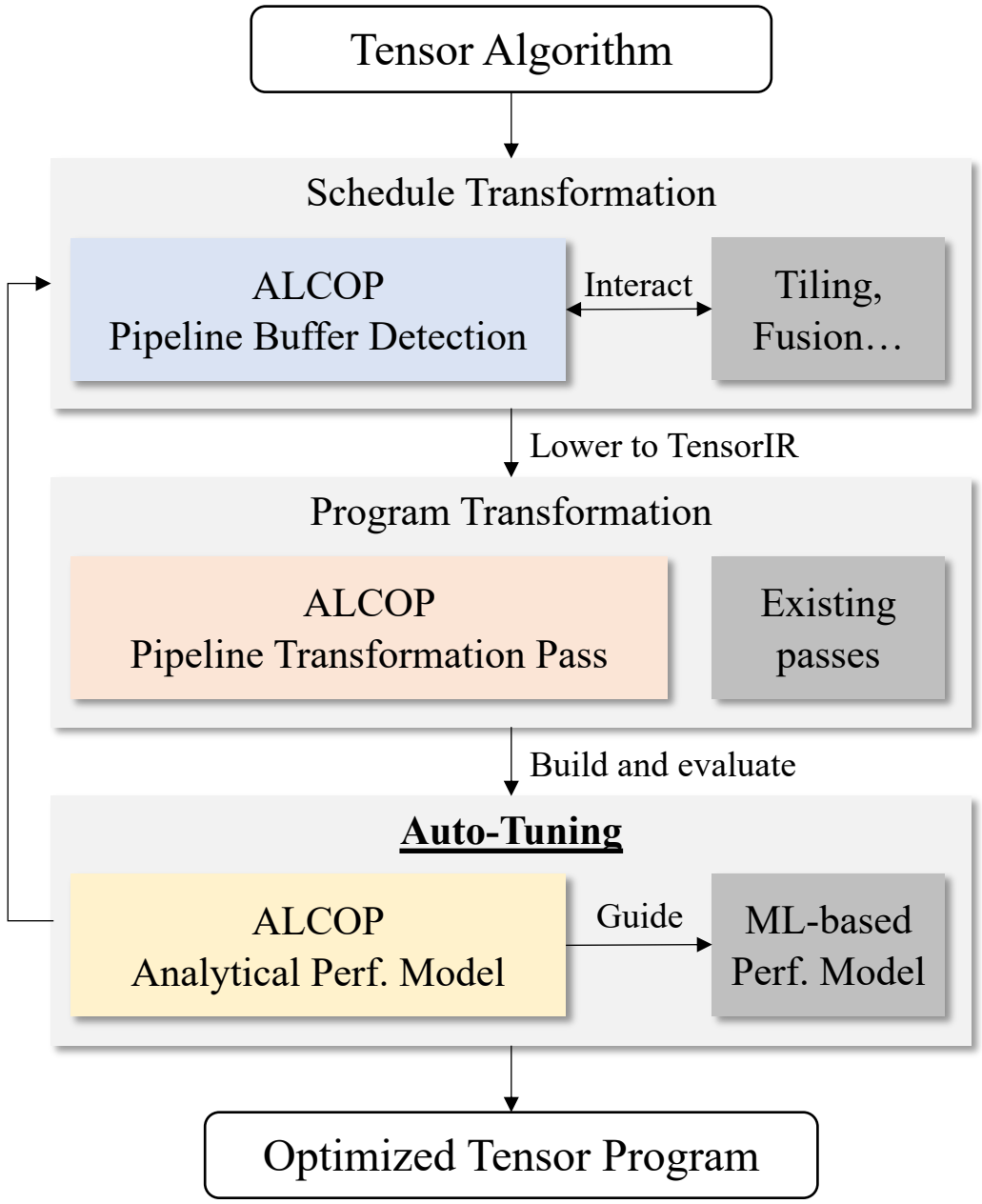


ALCOP Framework

```
# 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=?)
```

Schedule parameters



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

```
1
2
3
4
5
6
7
8
9
10
11
```

Shared-memory level load

Shared-memory level compute

Register level load

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

```
1
2
3
4   /* Main loop */
5   for (k0 = 0; k0 < extent_k0; k0++) {      load-compute level-1
6     /* load k0-th chunk into the shared memory */
7     A_shared[…] = A[k0, …];
8     /* compute with data in shared memory */
9
10
11  }
```

Shared-memory level load

Shared-memory level compute

Register level load

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

```
1
2
3
4   /* Main loop */
5   for (k0 = 0; k0 < extent_k0; k0++) {      load-compute level-1
6     /* load k0-th chunk into the shared memory */
7     A_shared[…] = A[k0, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {    load-compute level-2
10      A_reg[…] = A_shared[k1, …]; // load into registers
11      wmma(A_reg, …);}}       // compute with Tensor Core
```

Shared-memory level load

Shared-memory level compute

Register level load

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

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

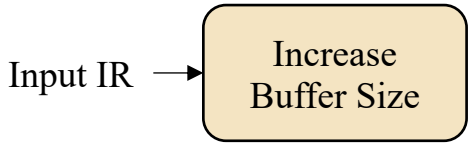
Shared-memory level load

Shared-memory level compute

Register level load

Register level compute

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



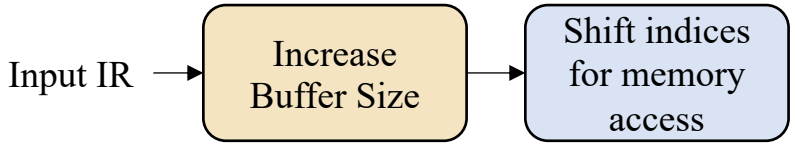
Before

```
1   /* Declare Buffers */
2   Alloc(A_shared, size_of_TB_chunk);
3   Alloc(A_reg, 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[…] = A[k0, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10      // load into registers
11      A_reg[…] = A_shared[k1, …];
12      wmma(A_reg, …); // compute with TC
```

After

```
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[…] = A[k0, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10      // load into registers
11      A_reg[…] = A_shared[k1, …];
12      wmma(A_reg, …); // compute with TC
```

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



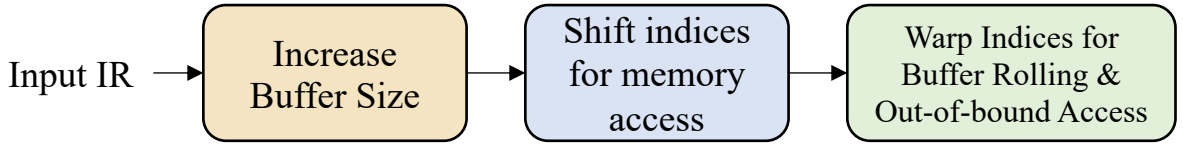
Before

```
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[…] = A[k0, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10      // load into registers
11      A_reg[…] = A_shared[k1, …];
12      wmma(A_reg, …); // compute with TC
```

After

```
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][…] = A[k0 + 2, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10      // load into registers
11      A_reg[k1 + 1][…] = A_shared[k0][k1 + 1, …];
12      wmma(A_reg[k1], …); // compute with TC
```

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



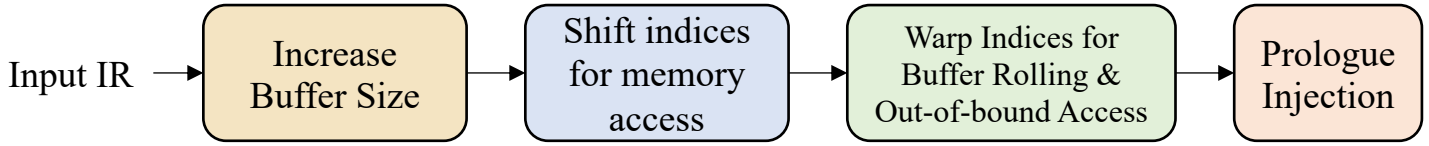
Before

```
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][…] = A[k0 + 2, …];
8     /* compute with data in shared memory */
9     for (k1 = 0; k1 < extent_k1; k1++) {
10      // load into registers
11      A_reg[k1 + 1][…] = A_shared[k0][k1 + 1, …];
12      wmma(A_reg[k1], …); // compute with TC
```

After

```
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][…] = \
        A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) %
extent_k1, …];
12      wmma(A_reg[k1 % 2], …); // compute with TC
```

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



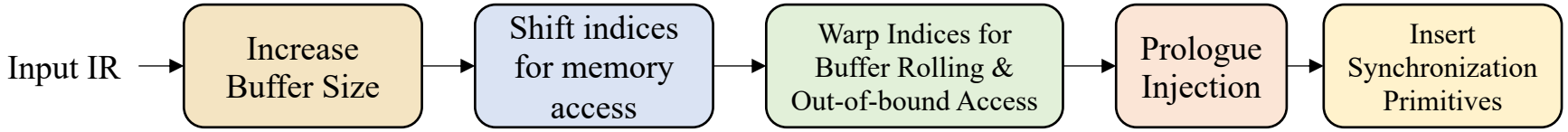
Before

```
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][…] = \
        A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, …];
12      wmma(A_reg[k1 % 2], …); // compute with TC
```

After

```
1   /* Declare Buffers */
2   Alloc(A_shared, 3*size_of_TB_chunk);
3   Alloc(A_reg, 2*size_of_warp_chunk);
4   /* Prologue for Loop k0 */
5   for (k0 = 0; k0 < 2; k0++) {
6     A_shared[k0] = A[k0, …];
7     /* Prologue for Loop k1 */
8     for (k1 = 0; k1 < 1; k1++) {
9       A_reg[k1] = A_shared[0][k1];
10  }}
11  /* Main loop (unchanged) */
    for (k0 = 0; k0 < extent_k0; k0++) { … }
```

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



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][…] = \
        A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, …];
12      wmma(A_reg[k1 % 2], …); // compute with TC
```

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      if ((k1 + 1) % 2 == 0) /* when advancing to a new k0 */
12        A_shared.consumer_wait();
13      // load into registers
14      async_memcpy(A_reg[(k1 + 1) % 2][…], \
        A_shared[(k0 + ((k1+1)/extent_k1)) % 3][(k1 + 1) % extent_k1, …]);
15      wmma(A_reg[k1 % 2], …); // compute with TC
16    A_shared.consumer_release();
```

Single Operator Results

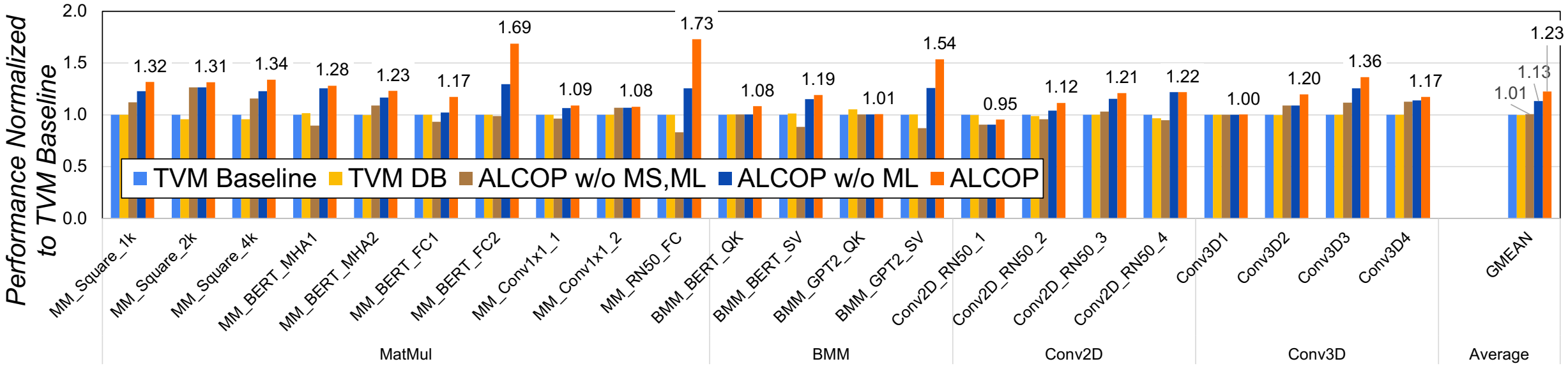
Settings

- 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.*

Benchmark Kernels

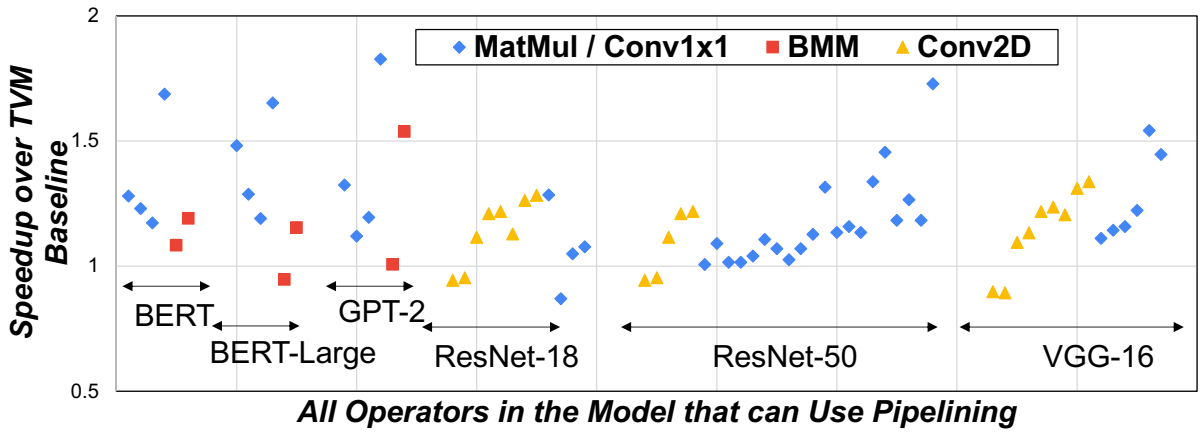
Operator	Tag	Shape
MatMul		**M/N/K**
	MM_Square_1k	1024/1024/1024
	MM_Square_2k	2048/2048/2048
	MM_Square_4k	4096/4096/4096
	MM_BERT_MHA1	512/768/768
	MM_BERT_MHA2	512/2304/768
	MM_BERT_FC1	512/3072/768
	MM_BERT_FC2	512/768/3072
	MM_Conv1x1_1	200704/64/256
	MM_Conv1x1_2	3136/512/256
	MM_RN50_FC	1024/64/2048
BMM		**Batch/M/N/K**
	BMM_BERT_QK	12/512/512/64
	BMM_BERT_SV	12/512/64/512
	BMM_GPT2_QK	12/1024/1024/64
	BMM_GPT2_SV	12/1024/64/1024
Conv2D		**N/H/W/CI/CO/KH/KW/stride**
	Conv2D_RN50_1	64/56/56/64/64/3/3/1
	Conv2D_RN50_2	64/28/28/128/128/3/3/1
	Conv2D_RN50_3	64/14/14/256/256/3/3/1
	Conv2D_RN50_4	64/7/7/512/512/3/3/1
Conv3D		**N/D/H/W/CI/CO/KD/KH/KW/stride**
	Conv3D1	64/14/56/56/64/64/3/3/3/1
	Conv3D2	64/14/28/28/128/128/3/3/3/1
	Conv3D3	64/14/14/14/256/256/3/3/3/1
	Conv3D4	64/14/7/7/512/512/3/3/3/1

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

End-to-end results



Model	End-to-end speedup	
	vs. TVM	vs. XLA
BERT	1.15	1.27
BERT-Large	1.18	1.16
GPT-2	1.15	1.34
ResNet-18	1.02	1.64
ResNet-50	1.06	1.02
VGG-16	1.10	1.01

Summary

Preprint: https://arxiv.org/abs/2210.16691

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

- 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