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** [1 4]

Abstract

Pipelining between data loading and computation is a critical tensor program optimization for GPUs. In order to unleash the high performance of latest GPUs, we must perform a synergetic optimization of multi-stage pipelining across the multi-level buffer hierarchy of GPU. Existing frameworks rely on hand-written libraries such as cuBLAS to perform pipelining optimization, which is inextensible to new operators and un-composable with prior tensor compiler optimizations. This paper presents ALCOP, the first framework that is compiler-native and fully supports multi-stage multi-level pipelining. ALCOP overcomes three critical obstacles in generating code for pipelining: detection of pipelining-applicable buffers, program transformation for multi-level multi-stage pipelining, and efficient schedule parameter search by incorporating static analysis. Experiments show that ALCOP can generate programs with $1.23\times$ speedup on average (up to $1.73\times$) over vanilla TVM. On end-to-end models, ALCOP can improve upon TVM by up to $1.18\times$, and XLA by up to $1.64\times$. Besides, our performance model significantly improves the efficiency of the schedule tuning process and can find schedules with 99% of the performance given by exhaustive search while costing $40\times$ fewer trials.

1 Introduction

Deep learning (DL) has achieved great success in a variety of application fields, spanning computer vision, natural language processing, and recommendation systems (He et al., 2016; Devlin et al., 2018; Naumov et al., 2019). The widespread use of GPUs (Nvidia, 2020b;a) to accelerate DNNs makes an indispensable contribution in this AI era.

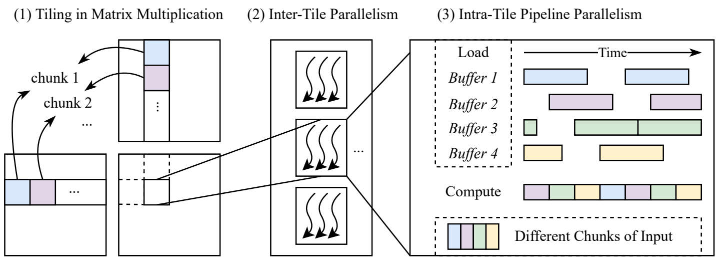
High-performance tensor programs on GPUs require complex optimization efforts. When Tensor Core was introduced to GPUs to accelerate deep learning, harnessing the power of Tensor Cores became the center of GPU software optimization, motivating the development of a number of libraries and compilers (Nvidia, 2021; Yan et al., 2020; Dakkak et al., 2019; Feng et al., 2021; Chen et al., 2018a; Katel et al., 2022). Because Tensor Core throughput continued to increase but memory bandwidth lagged, research on tiling and fusion to improve data re-use surged (Niu et al., 2021; Zhao et al., 2022; Zheng et al., 2022b; 2020). However, aggressively large tiling limits the number of tiles and hinders inter-tile parallelism, a crucial GPU mechanism to achieve high utilization. Restoring the parallelism lost due to aggressive tiling becomes an important task. **Pipelining** – the overlap of data loading and computing – is an ideal mechanism for unleashing intra-tile parallelism. Figure 1 depicts the concept of pipelining and its performance advantages. As the difficulty of capitalizing on the ever-growing parallelism in current and future GPUs increases, the study of pipelining becomes essential.

Despite the necessity of pipelining optimization, existing approaches are either limited in their design space coverage or their degree of automation. Prior work (Katel et al., 2022) has studied double-buffering, a special case of pipelining. However, double-buffering is only a two-stage, one-level instance of the entire multi-stage, multi-level pipelining design space shown in Figure 2 and Figure 3, and simplifying pipelining to double-buffering hinders a major performance gain (which will be evaluated in Sec. 5.1). Although deep learning systems can access comprehensive pipelining optimization from hand-written libraries (NVIDIA, a) or compiler-integrated libraries (Xing et al., 2022), due to their fundamental difference from the tensor program generation workflow of DL compilers, they are inextensible to new operators and not composable with prior compiler passes such as auto-fusion and auto-tiling.

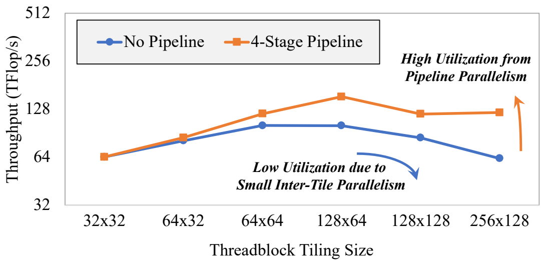
Automatic pipelining presents three distinct challenges: workload complexity (diverse DL operators), hardware complexity (multi-level memory hierarchy), and design space complexity (coherent performance tuning factors).

[1]University of California, Santa Barbara [2]Chinese University of Hong Kong [3]Rensselaer Polytechnic Institute [4]Alibaba DAMO Academy. Correspondence to: Guyue Huang <guyue@ucsb.edu>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

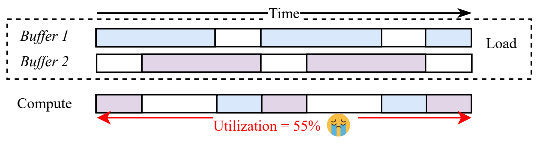

(1) Tiling in Matrix Multiplication
(2) Inter-Tile Parallelism
(3) Intra-Tile Pipeline Parallelism
chunk 1
chunk 2
Load
Time
Buffer 1
Buffer 2
Buffer 3
Buffer 4
Compute
Different Chunks of Input

(a) Concept of tiling, inter-tile parallelism and pipeline parallelism.

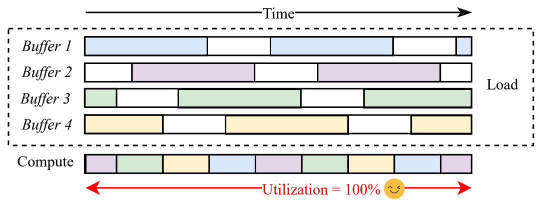


(b) Motivating example: performance of a 2048 × 2048 × 2048 matrix-multiplication tested on NVIDIA A100 with different tiling and pipelining choices.

Figure 1. Motivation of automatic pipelining. (a-3) explains the concepts of pipelining, which is overlapping data loading with computation. (b) gives a motivating example. With tiling only, the performance is always sub-optimal. Pipelining unleashes intra-tile parallelism and increases the performance under large tiling.



(a) Two-stage pipelining



(b) Multi-stage pipelining (four-stage here)

Figure 2. Concept of multi-stage pipelining. (a) two-stage pipelining (or called double-buffering) is not enough to hide the data loading latency. (b) Four-stage pipelining can hide the data loading latency and achieve full utilization of the computing units. ALCOP supports multi-stage pipelining.

Our key insight is that, instead of solving everything in a monolithic compiler pass, we should exploit the progressive lowering structure of DL compilers and the information exposed at each level. Specifically, we address the aforementioned three challenges through three decoupled and collaborative compilation modules: pipeline buffer detection, pipeline program transformation, and analytical-model guided design space search. Pipeline buffer detection addresses the workload complexity because it occurs during the scheduling phase when the entire dataflow is visible. The second module addresses the hardware complexity. During the program transformation stage, the intricate for-loop structure and data movement are revealed and modified. This module utilizes the safety check of the preceding module to execute the robust transformation. The third module addresses the design space complexity. It happens at the auto-tuning stage, where pipelining and other techniques are co-optimized. This module makes use of the preceding parameterized module. We further design an analytical hardware model to expedite the design space search.

In this paper, we propose ALCOP[1] [2](Automatic Load-COmpute Pipelining), the first DL compiler solution (auto-scheduler, program transformation, auto-tuner) for automated multi-stage, multi-level pipelining; its architecture is shown in Figure 4. Additional contributions include:

1. We design methods to examine each buffer for applying pipelining, including the ordering of pipelining and other schedule transformations to avoid mutual interference. (Sec. 2)

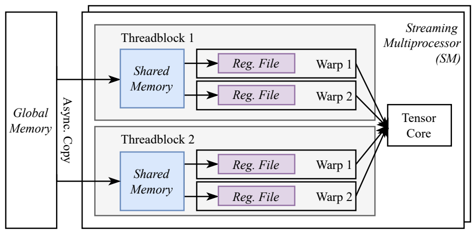
2. We design a program transformation pass that handles index manipulation, synchronization injection, and prologue injection, among other transformations. (Sec. 3)

3. We propose a pipeline-aware analytical performance model. Combining it with an existing machine-learning (ML) based tuning algorithm significantly improves the efficiency of schedule tuning. (Sec. 4)

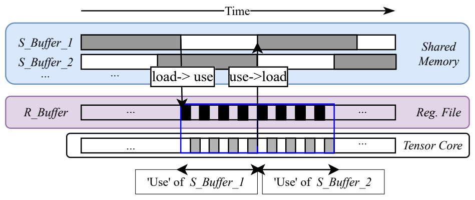
Experiments show that the ALCOP program transformation pass can bring on average 1.23×, and up to 1.73× speed-up to individual DL operators over TVM (Chen et al., 2018a). ALCOP brings 1.02-1.18× end-to-end inference speed-up for six DL models over TVM (Chen et al., 2018a), and 1.01-1.64× over XLA (Google, 2021). Through combining the analytical model with ML-based tuning, we can identify schedules with 99% performance compared to the

[1]ALCOP is a copper-based alloy with significantly-enhanced endurance and strength. We envision our ALCOP will significantly enhance the power of DL compilers for modern AI-GPUs.

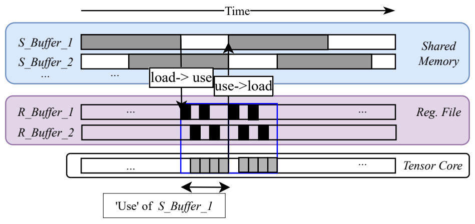
[2]The source code is released at `https://github.com/hgyhungry/alcop-artifact`.



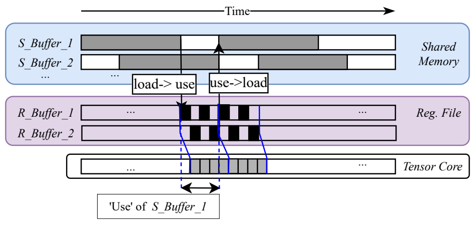
(a) GPU memory hierarchy.



(b) Single-level pipelining.



(c) Multi-level pipelining without inner-pipeline fusion.



(d) Multi-level pipelining with inner-pipeline fusion.

Figure 3. Concept of multi-level pipelining and inner-pipeline fusion. (a) shows the GPU memory hierarchy, with two levels of buffers: the shared memory and the register file. (b) shows the execution timeline of single-level (only shared memory) pipelining. (c) improves over (b) by pipelining the inner loop: register loading and computing. (d) improves over (c) via inner-pipeline fusion, which treats the repeated inner loop as a holistic loop and pipeline it. ALCOP supports optimizations in (d), which provides the best performance among (b)-(d).

best schedule in the entire design space while reducing the number of trials by $40\times$.

2 Schedule Transformation

Automatic pipelining begins by identifying potential pipelining possibilities. We implement it through a schedule transformation pass in the compiler, which attaches the pipelining primitive to buffer variables in the program.

Pipelining can be applied to a *load-and-use* loop in which the *load* step copies data into a buffer and the *use* step reads data from the buffer. The purpose of the schedule transformation is to identify and record “load-and-use” structures in a program. The pass marks the buffer variables within such load-and-use loops as *pipelined buffers*. Later on, a program transformation pass described in Section 3 will turn the load-and-use structure into its pipelined version.

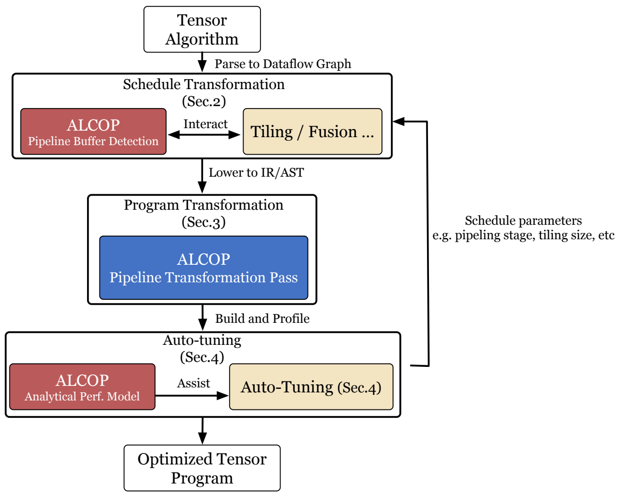

Tensor Algorithm
Parse to Dataflow Graph
Schedule Transformation (Sec.2)
ALCOP Pipeline Buffer Detection
Interact
Tiling / Fusion ...
Lower to IR/AST
Program Transformation (Sec.3)
ALCOP Pipeline Transformation Pass
Build and Profile
Auto-tuning (Sec.4)
ALCOP Analytical Perf. Model
Assist
Auto-Tuning (Sec.4)
Schedule parameters e.g. pipeling stage, tiling size, etc
Optimized Tensor Program

Figure 4. The overview of ALCOP.

Two important questions must be addressed: First, we must determine is what rules we should apply to identify the buffers that can be pipelined. The second one is determining the *ordering* if pipelining in relation to other schedule transformations, such as tiling, aware of their mutual effect.

2.1 Identification of Buffers for Pipelining

Constraints of pipelining come from not only the algorithm, *i.e.,* how the buffer is used, but also the hardware capabilities, *i.e.,* what forms of memory copy can be executed asynchronously. For each buffer variable, the following three rules are evaluated to determine whether pipelining can be applied. Firstly, we do not pipeline a buffer that is not produced by asynchronous memory copy. An asynchronous memory copy indicates that the memory copy is non-blocking, so we can initiate memory copies for future loop iterations in advance and meanwhile continue with the computation in the present iteration. Only when an explicit synchronization instruction is encountered does the program block to wait for the completion of the memory copy. If the data in a buffer is not produced by direct memory copy but rather by some compute operation, the buffer does not meet this condition.

Secondly, we require the pipelined buffer to be produced inside a sequential load-and-use loop. The purpose of pipelining is to overlap the loading of future iterations with the computation of the current iteration, hence it is critical this loop is sequential at the first place rather than unrolled or parallelized. Typically this sequential loop iterates the reduction axis in a tensor operation. As a counterexample, some stencil programs also tile input tensor and store them in buffers to enhance locality, but different tiles are often parallelized. The reflection in the tensor program is that the buffer is produces in a parallelized loop, and our schedule transformation will make sure this loop is not pipelined.

The final rule is about *synchronizing the pipeline:* If the hardware platform supports only scope-based synchronizations, we inspect all buffers within the same scope and refuse to pipeline them if their synchronization positions do not match. Synchronizing the pipeline requires special memory barriers that wait for certain loading instructions (*e.g.*, instructions issued in the third last iteration in a 4-stage pipeline). On NVIDIA Ampere GPUs, such memory barriers are provided for the shared memory scope. Hence, the hardware is incapable of resolving this conflict if two buffers are both in the shared memory scope, but their barriers must be inserted at distinct positions in the program. If this conflict occurs, our schedule transformation refuses to pipeline these buffers.

2.2 Ordering of Schedule Transformations

Pipelining is applicable to three schedule transformations already in existence: cache-reading, tiling, and fusion. We will briefly introduce these transformations and then determine whether pipelining should be applied before or after them.

Cache-reading. It means inserting a read buffer for a tensor input. Given an algorithm and computation tensor `S2` from tensor `S1`, applying cache-reading means inserting a new tensor `S1_buf` which is an identical copy of `S1` but with a buffer scope. Cache-reading should be applied before pipelining since pipelining needs to be applied to buffers generated by the former.

Tiling. It is the process of dividing the output tensor into blocks. In conjunction with cache-reading, it can cache data within buffers to improve data reuse. Tiling should also be performed before pipelining. The second condition for a buffer to qualify pipelining, *i.e.,* whether there exists a sequential load-and-use loop, must be inspected based on the for-loop sketch after tiling.

Fusion. It means avoiding writing back intermediate data between two operators. Inlining, a specific type of fusion, should come after pipelining. Inlining a tensor means producing the value of the tensor precisely where it is used; this technique is often used on lightweight element-wise operations like datatype casting. Figure 5 shows an example in which originally `S2` is produced by applying element-wise function $f(\cdot)$ to `S1`, and a buffer tensor `S2_buf` is injected

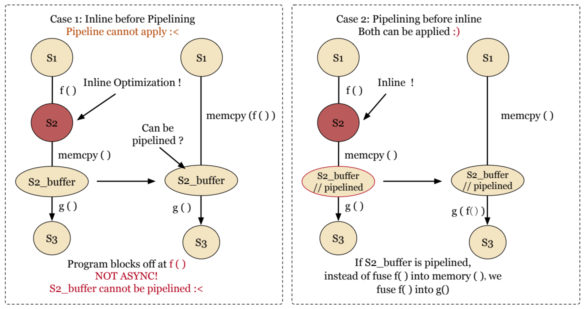


Figure 5. The effectiveness study on the optimization order of inlining and pipelining. In case 1, after inlining, `S2_buf` can no longer be pipelined because it is no longer produced by an asynchronous memory copy. In case 2, after pipelining, inlining can still be applied.

after `S2` via cache-read. Inlining `S2` is equivalent to applying $f(\cdot)$ first and then copying the data directly into `S2_buf` without writing the data back to memory. According to our first rule outlined in the previous subsection, a pipelined memory buffer should be produced from an asynchronous memory copy. However, for `S2_buf` here, the operation to produce it is no longer asynchronous, as the explicit $f(\cdot)$ forces the program to stall, waiting for data to be loaded. And since buffer `S2_buf` is not produced asynchronously, it cannot be pipelined. Here in case 1, inlining impedes the opportunity of pipelining. Nevertheless, if pipelining is applied before inlining, like in case 2, the inlining of `S2` can still be applied, but in a different manner: Instead of inlining `S2` into `S2_buf`, we cache-read `S1` and fuse the computation $f(\cdot)$ into the production of `S3`. Thus, we ensure both sides are satisfied: the buffer is produced through an asynchronous copy and can be pipelined, while computation $f(\cdot)$ is fused and we avoid explicitly generating an intermediate tensor.

3 Program Transformation

In this section, we introduce the second component of automatic pipelining: transforming the program IR (Intermediate Representation) to implement pipelining. After the schedule transformation outlined in Section 2, the program is lowered to its IR form, composed of for-loops and load/store/compute operations. Figure 7 gives a sample input and transformed IR of the pipelining pass. Figure 6 also depicts the transformation steps.

3.1 Analysis

The First Step. Given a program IR, pipelining begins with the collection of pipelining hints inserted by the schedule transformation, including the buffer to be pipelined and the number of stages for each buffer.

The Second Step. Given a set of buffers we want to apply

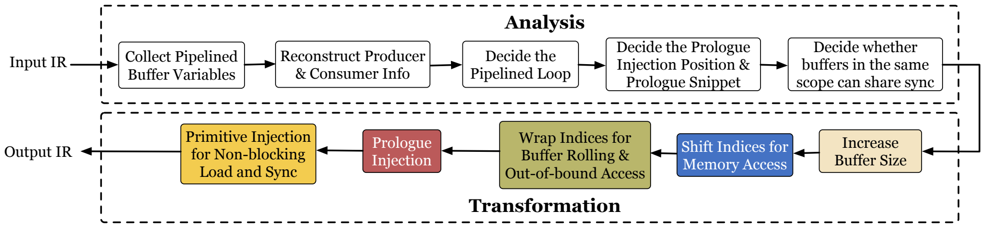

Analysis
Input IR
Collect Pipelined Buffer Variables
Reconstruct Producer & Consumer Info
Decide the Pipelined Loop
Decide the Prologue Injection Position & Prologue Snippet
Decide whether buffers in the same scope can share sync
Increase Buffer Size
Shift Indices for Memory Access
Wrap Indices for Buffer Rolling & Out-of-bound Access
Prologue Injection
Primitive Injection for Non-blocking Load and Sync
Output IR
Transformation

Figure 6. Workflow and example input and output of the pipelining program transformation

```
1  Algorithm:
2  /* MatMul */
3  C[i, j] = sum(A[i, k] * B[j, k], reduce_axis=(k,))

1  Schedule:
2  /*  cache read B is omitted for brevity */
3  A_shared = cache_read(A)
4  A_reg = cache_read(A_shared)
5
6  /* tiling */
7  C.tile((TB_tile_i, TB_tile_j, TB_tile_k),
8         (Warp_tile_i, Warp_tile_j, Warp_tile_k))
9
10 /* pipelining */
11 A_shared.pipeline(stage=3)
12 A_reg.pipeline(stage=2)

1  InputIR:
2  /* Declare buffer */
3  alloc A_shared[TB_tile_i, TB_tile_k]
4  alloc A_reg[Warp_tile_i, Warp_tile_k]
5
6  for ko in range(0, C_k / TB_tile_k):
7      /* load into shared memory buffer */
8      memcpy(A_shared[...], A[..., ko])
9
10     /* compute with data in shared memory buffer */
11     for ki in range(0, TB_tile_k / Warp_tile_k):
12         /* load into register buffer */
13         memcpy(A_reg[...], A_shared[..., ki])
14         /* compute with data in register buffer */
15         wmma(A_reg[...], ...)
```

```
1  TransformedIR:
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 size. */
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 reg. 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 ])
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 ) % extent_ki  )== 0:
22             A_shared.consumer_wait()
23             /* load into register buffer */
24         async_memcpy(
25             A_reg [  (ki + 1) % 2 ][...],
26             A_shared [ (ko  + ((ki+1) / extent_ki) ) % 3  ][...,  (ki + 1)  % extent_ki ]
27         )

28         /* tensor-core compute with data in register buffer */
29         wmma(A_reg [ (ki % 2) ] [...], ...)
30          A_shared.consumer_release()
```

Figure 7. An example to illustrate how to transform an original Tensor-IR (left) to its pipelined version (right).

pipelining, the second analysis task is to reconstruct the producer tensor and consumer tensor(s) of these buffers. Then we can derive if there are multi-level buffers by deciding if the producer of a pipeline buffer is also a pipelined buffer. Since pipelined buffers are always produced via asynchronous memory copy, to determine the producer tensor, it suffices to retrieve which tensor it copies from. The decision of consumers happens when IR traversal encounters a load operation from this buffer.

The Third Step. This step is to determine the sequential load-and-use loop for each pipelined buffer. This identifies the iteration variable to be pipelined and is required by all the index shifting operations in the transformation steps. The sequential loop can be determined as follows: starting from the instruction that copies data into the buffer, traversing all the for-loops from inside to outside, and finding the first sequential loop whose iteration variable is not used to index inside this buffer. This means the buffer is reused for each iteration of this loop, which is the loop we want to pipeline. Take Figure 7 as an example, the pipelined loop for `A_shared` is with iteration variable `ko`, and the pipelined loop for `A_reg` is with variable `ki`.

The Fourth Step. We should document the pieces of code that *loads* and *uses* this buffer. This information is required for the injection of synchronization primitives and prologues. In the *Input IR* in Figure 7, the "loading" part for `A_shared` is Line `8`, and the "using" part is the `ki` loop and everything inside. The loading part for `A_reg` is Line `13`, and the using part is Line `15`.

The Fifth Step: We also need to decide where to inject prologues. Since we transform the program to issue memory copy for future iterations while doing computation for the current iteration, we need to move the first few stages of memory copy ahead of the start of the main load-and-use loop. This pre-posed loading code block is a prologue. Typically, prologues can be injected simply before the pipelined loop. However, when a multi-level pipeline appears, the prologues of inner pipelines must be injected into the sequential loop of the outer-most pipeline, in order to build a holistic pipeline as opposed to a recursive one as shown in

Figure 3(c).

3.2 Transformations

Five steps are required to transform a *load-and-use* loop into a pipelined loop. The *Transformed IR* in Figure 7 shows the transformed version of the *Input IR* in the same figure.

The First Step. This step increases the size of the memory buffer by the number of pipeline stages. Relevant transformed code is highlighted in light yellow.

The Second Step. This step shifts the indices used in memory access. The eelevant code is highlighted in blue. In each *load-and-use* iteration, we issue asynchronous memory copy for *future* iterations rather than the present iteration. Therefore, we need to increase the pipelining loop variables in the memory access indices. If it is a 3-stage pipeline, for instance, we should load data 2 iterations ahead.

The Third Step. This step handles indices for buffer rolling (circular access) and out-of-bound wrapping. Relevant code is highlighted in green. There are two cases that we need to wrap indices: first, when we use the pipelining variable to index a chunk of the buffer, we should use the modulo of pipeline iteration variable divided by pipeline stages. Secondly, since we increase the pipelining variable, it is possible that we index out of the bound of its producer tensor. We must take the modulo of the pipelining variable divided by its own extent to avoid indexing out-of-bound. A complicated case is in a multi-level pipeline when the overflow of the inner pipeline causes the increase of the outer pipeline variable. Line `26` in the transformed IR handles this case.

The Fourth Step. This step injects prologue primitives. The contents of prologues are the memory copy of the first `n_stage -1` chunks of data, where `n_stage` is the number of the pipeline stage. We inject prologue at the positions we record in the preceding analysis pass.

The Fifth Step. The final step injects synchronization primitives. The pipeline is guarded by four primitives: `producer_acquire`, `producer_commit`, `consumer_wait`, and `consumer_release`. `producer_commit` commits a batch of asynchronous loading operations. `consumer_wait` blocks until a previous batch of loading is completed. When the pipeline is full, `producer_acquire` blocks until `consumer_release`[3]. The pairs of producer/consumer primitives are put around the loading/using part of the buffer, respectively, as shown in Line `15, 17, 22,` and `30` of the transformed IR.

[3]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#with-memcpy_async-pipeline-pattern-multi

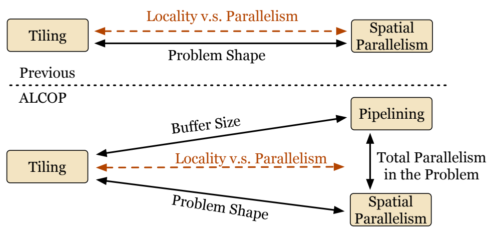


Figure 8. A high-level view of the performance model. Compared to prior work (Lym et al., 2019), our model takes into account the constraints and trade-offs among pipelining, tiling and spatial parallelism.

4 STATIC ANALYSIS GUIDED TUNING

This section introduces how we combine a static analytical performance model with existing machine-learning (ML) based auto-tuning (Chen et al., 2018b) to choose schedule parameters. The key component is a novel performance model aware of pipelining and its interaction with other optimizations, as illustrated in Figure 8.

4.1 Top-Level Model

Our analytical model is shown in Table 1. At the top level, the threadblocks are grouped into threadblock-batches ($threadblk\_batch$), and one threadblock-batch occupies all Streaming Multiprocessors (SMs) at a time. Since all threadblocks execute the same program, the latency of a kernel equals the threadblock latency multiplied by the number of batches. The number of threadblock-batches in a kernel depends on the GPU scheduling policy, which we learn through performance profiling. The maximum number of threadblocks per SM is limited by the size of shared memory and register files that each SM can provide, as well as the request of threadblock. Our simulated GPU scheduling policy considers all these factors to decide $N_{threadblk\_batch}$.

At the threadblock level, we estimate its final performance by summing the latencies of three phases: (1) the initial phase T_{init}, in which the first chunk of data is requested and the pipeline waits for it to arrive; (2) the main loop T_{main_loop}, in which the load-and-use pipeline advances at a steady rate; (3) the epilogue phase $T_{epilogue}$, in which the final results are written back into the global memory. $T_{epilogue}$ is determined using the Epilogue Model equation proposed in DELTA (Lym et al., 2019).

Let us consider T_{main_loop}. It illustrates load-and-use loop at the shared memory level, which comprises copying data from the device memory to the shared memory, reading the data into the register, and doing computations with tensor cores. We employ a Pipeline Latency Model, which is described in the next subsection, to calculate the latency of the loop. This model considers the pipelining and multiplexing factors, N_{pipe}, N_{mplx}, which means the number of

Table 1. Analytical Performance Model

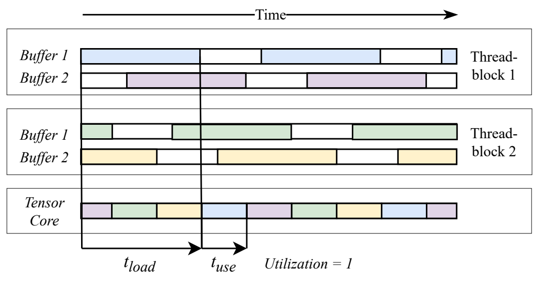
Category	Model
Kernel Latency Model	$T_{kernel} = T_{threadblk} \times N_{threadblk\_batch}$
Pipeline Latency Model	Input: $T_{load}, T_{use}, N_{loop}, N_{pipe}, N_{mplx}$ Output: $T_{load\_use\_loop}$ If $T_{load} \leq (N_{pipe} \times N_{mplx} - 1) \times T_{use}$: $T_{load\_use\_loop} = T_{use} \times N_{loop}$ Else: $T_{load\_use\_loop} = (T_{load} + T_{use}) \times N_{loop} \div N_{pipe}$
Threadblock Latency Model	$T_{threadblk} = T_{init} + T_{main\_loop} + T_{epilogue}$ $T_{init} = T_{smem\_load} + T_{reg\_load}$ $T_{main\_loop} = \texttt{PipelineLatencyModel}(T_{smem\_load}, T_{smem\_use}, N_{smem\_loop}, N_{smem\_pipe\_stage}, N_{threadblk\_per\_SM})$ $T_{smem\_use} = \texttt{PipelineLatencyModel}(T_{reg\_load}, T_{compute}, N_{reg\_loop}, N_{reg\_pipe\_stage}, N_{warp\_per\_threadblk})$
Computation Latency Model	$T_{compute} = \dfrac{FLOPs_{one\_reg\_loop}}{Throughput_{SM} \times \texttt{Util}(N_{warp\_per\_threadblk}, N_{threadblk\_per\_SM})}$
Memory Latency Model	$T_{snen\_load} = MAX(T_{LLC\_load}, T_{DRAM\_load})$ $T_{LLC\_load} = LAT_{LLC\_read} + \dfrac{Bytes_{one\_smem\_loop} \times N_{threadblk\_per\_threadblk\_batch}}{BW_{LLC}}$ $T_{DRAM\_load} = LAT_{DRAM\_read} + \dfrac{Bytes_{threadblk\_batch\_workset}}{BW_{DRAM}}$
Epilogue Model	$T_{epilogue} = LAT_{DRAM\_write} + \dfrac{Bytes_{output\_tile} \times N_{threadblk\_per\_threadblk\_batch}}{BW_{DRAM\_write}}$

stages the pipeline has, and the number of parallel workers that can be multiplexed to hide the memory copy latency. At the shared memory level, these two parameters equal to the number of stages at the outer load-and-use loop, $N_{smem\_pipe\_stage}$, and the number of parallel threadblocks in an SM, $N_{threadblk\_per\_SM}$.

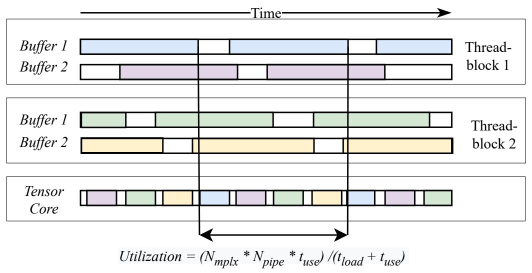
Calculation of T_{main_loop} still needs the latency of the use phase in this loop. However, the use phase is another pipeline that loads data into the register files and performs computations with tensor cores. We can calculate the latency of the use phase by estimating the stable state latency of the inner pipeline through inner-pipeline fusion. For this inner load-and-use loop, the use latency refers to the latency of performing arithmetic operations inside one loop on tensor cores. The pipeline and multiplex factors are determined by the number of stages of this inner load-and-use loop and the number of parallel warps in a threadblock.

4.2 Obtaining Detailed Latencies

Pipeline Latency Model. Now we address the core issue of estimating the latency of a load-and-use loop in its stable state. Intuitively, the prediction should differ depending on whether the bottleneck is loading or using. Line 3 in the Pipeline Latency Model in Table 1 is the criterion for determining the bottleneck. Figure 9 illustrates the two scenarios in which computation or loading is the bottleneck. The intuition is that, during the loading of one data chunk, the computation units can be used to compute other chunks of data in this pipeline (N_{pipe}), or used for other parallel workers (N_{mplx}). If the latency of data loading exceeds the latency of all computations that can overlap with it, the loading becomes the bottleneck, making the loop latency equal to the latency of one load-and-use iteration, divided by the number of overlapping streams, i.e., $(T_{load} + T_{use})/N_{pipe}$.



(a) Case 1: $t_{use} \cdot N_{mplx} \cdot N_{pipe} \geq (t_{load} + t_{use})$



(b) Case 2: $t_{use} \cdot N_{mplx} \cdot N_{pipe} < (t_{load} + t_{use})$

Figure 9. Explanation of the pipeline latency model. A *load* can be overlapped by *computing* in other threadblocks, or in other stages of the same threadblock.

Computation and Memory Latency Model. To obtain the computation latency, we can simply divide the number of float-point operations performed inside a loop by the tensor core throughput in an SM. When determining the latency of memory copies, four parameters must be considered: the amount of data transferred, the available bandwidth, the number of parallel workers (threadblocks or warps) to share with the bandwidth, and a constant round-trip latency LAT . Note that GPU LLC is shared by all SMs. Hence the DRAM traffic cannot be computed by the sum of data

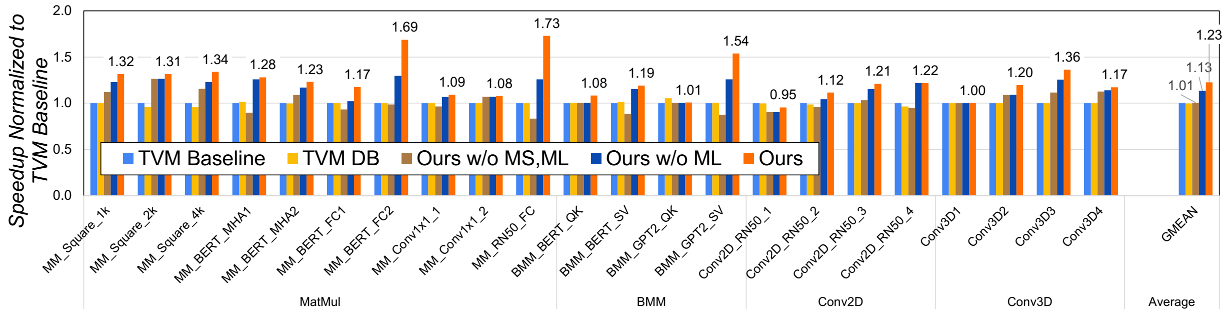


Figure 10. Single operator performance normalized to TVM on A100.

Table 2. Comparison of compiler search methods.

	Grid Search	**XGB**	**Anal. Only**	**Anal. + XGB (ours)**
Cost Model	N.A.	ML	Analytical	ML
Prior Knowledge?		No	Yes	Yes
Update Cost Model?		Yes	No	Yes
Sampling	Enumerate	Simulated Annealing	Cost-Model Ranking	Simulated Annealing

loaded by all threadblocks because the data may hit in LLC. We model DRAM traffic by deciding the working set of a threadblock-batch.

4.3 Model-Guided Auto-Tuning

Now, we will discuss how to use the analytical performance model for scheduled tuning. The workflow of auto-tuning is composed of a *cost model* to predict performance from schedule, and a *sampling method* to propose new *trials*. Unlike the analytical model we developed, TVM uses not an analytical model but a machine learning (ML)-based cost model that only learns from the profiled performance results. Analytical model and ML-based tuning offer complementary benefits: analytical model does not require the complexity of compiling and running sampled schedules but cannot be very accurate because it is difficult to capture hardware factors such as memory system thoroughly. ML-based tuning learns the cost model from measured performances that incorporate these complex factors, but it requires a large amount of sampled data, leading to a lengthy tuning process.

Finally, we leverage the analytical performance model's prediction to pre-train the ML-based model, allowing the ML model to acquire previous knowledge while still utilizing profiled data. Table 2 compares our method (Model-Assisted XGB) with other available auto-tuning approaches.

5 Evaluation

5.1 Single Operator Performance

This part evaluates pipelining speedup on single operators. Our benchmarks extracted from real DNN workloads contain four operators with a variety of shapes. All operators use half-precision and run on Tensor Cores. We run all experiments on NVIDIA Ampere GPU, as prior generations lack the asynchronous memory-copy hardware feature. Our evaluation platform is NVIDIA A100-SMX4 with 40GB device memory. The software we use is CUDA v11.4.

We implement our pipelining framework based on TVM (Chen et al., 2018a) v0.8 and compare it against the vanilla TVM. We augment both ALCOP and baselines with shared memory swizzling to avoid bank conflict limitation. We also manually insert double-buffering primitives into TVM and use it as the second baseline (TVM DB). We also compare against two downgraded versions of our compiler for ablation study: ALCOP without multi-level (ML), meaning just pipelining in shared memory level, and ALCOP without ML and multi-stage (MS), meaning only allowing two-stage pipelining. Here we exhaustively search the schedule space and give the best schedule for ours and all baselines.

Figure 10 shows the performance of different compilers normalized to TVM. Our compiler produces operators that are 0.95-1.73×, on average 1.23×, faster than TVM. Pipelining is especially effective for operators with small output shapes but long reduction axis. Take matrix-multiplication (MatMul) as an example, MM_RN50_FC, the operator that gives the largest speedup, has an output shape of 1024×64, and a reduction axis of 2048. Also, for Batched Matrix Multiplication (BMM), the operators with short reduction axis (e.g., BMM_BERT_QK) show much smaller speedup than those with long reduction axis (e.g., BMM_BERT_SV).

Insights about when pipelining works well. Problems with small output shapes (*e.g.,* MM_BERT_FC2, MM_RN50_FC) have limited spatial parallelism, so they benefit more from pipelining since pipelining uncovers extra parallelism. For problems with large output shapes (*e.g.,* MM_Conv1x1_1), or with small reduction dimensions (*e.g.,* BMM_GPT2_QK), pipelining provides limited benefit since the former already have abundant parallelism and the latter cannot amortize the latency of initial loading stages in the pipelining schedule.

Table 3. Model speedup from pipelining

Model	Speedup over TVM	Speedup over 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

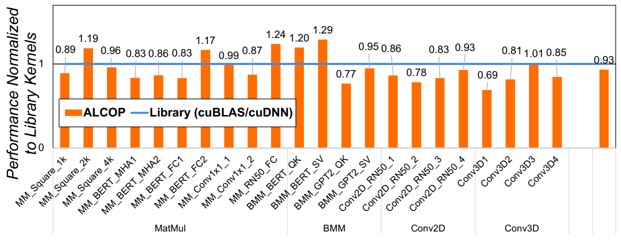


Figure 11. Single operator performance versus libraries.

Ablation study. Multi-level and multi-stage pipelining are both critical to final speedup. As shown in Figure 10, TVM DB does not bring obvious speedup over TVM. Without multi-level pipelining, ALCOP can only provide an average $1.13\times$ speedup. Without multi-level and multi-stage pipelining, ALCOP can only give $1.01\times$ speedup over TVM.

5.2 End-to-End Performance

To evaluate end-to-end model acceleration, we compare against two baselines: TVM (Chen et al., 2018a) and XLA (Google, 2021) (TF v2.9.1). XLA is a compiler integrated into the Tensorflow framework to optimize models in an end-to-end fashion. We evaluate six popular deep learning models. BERT, BERT-Large (Devlin et al., 2018) and GPT-2 (Radford et al., 2019) are popular models in Natural Language Processing (NLP). ResNet-18, ResNet-50 (He et al., 2016) and VGG-16 (Simonyan & Zisserman, 2014) are three convolution neural networks widely used in vision tasks. Pipelining can be applied to MatMuls, BMMs and Conv2Ds, which are the most computation intensive operators and consumes a great proportion of the inference latency in these models.

Table 3 shows the end-to-end speedup in real models. We achieve 1.02-$1.18\times$ end-to-end speedup over TVM and 1.01-$1.64\times$ speedup over XLA.

5.3 Comparison with Libraries

We compare with kernels in vendor libraries (cuBLAS (NVIDIA, a)/cuDNN (NVIDIA, b)), which are heavily hand-optimized for the typical problem shapes we evaluate. Note that despite their high performance, libraries take huge manual efforts due to low modularity and cannot replace com-

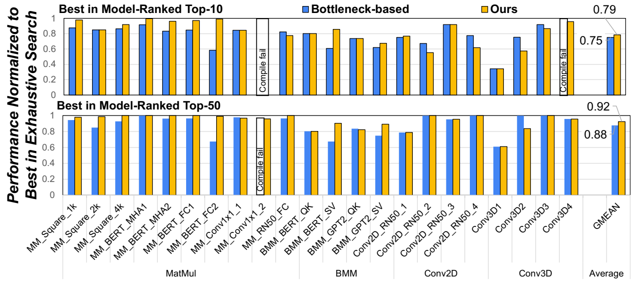


Figure 12. Best-in-top-k performance of two analytical performance models. The mark 'compile fail' means the first 10 or 50 proposed schedules fail to compile into executables.

pilers in AI-GPU optimization.

Figure 11 shows the performance of ALCOP normalized to library kernels. We can achieve on-par, on average 93% normalized, performance compared with library kernels. For some operators like BMM_BERT_QK, our compiler even generates faster kernels than cuBLAS because our compiler can search the entire schedule space and find the best schedule for input operators.

5.4 Performance Model Accuracy

The metric we use to evaluate our performance model is best performance in model-ranked top-k schedules, or *best-in-top-k* in short. It means the best performance within the top k schedules is predicted by the performance model. Compared to mean-absolute-error among the entire schedule space, best-in-top-k is more meaningful to schedule tuning because tuning cares about finding efficient schedules within a limited number of trials.

We compare against bottleneck-based analysis, a simple model that takes the maximum of computation, shared memory loading and device memory loading time, assuming full utilization of computation throughput and bandwidth. It is over-simplified in the following ways: (1) assumes an aggregated computation unit, but in GPUs the Tensor Cores are distributed in different SMs and occupancy of SMs matters. (2) agnostic to the latency hiding effect, which is what pipelining mainly benefits.

Figure 12 shows the best-in-top-k results for our analytical model and bottleneck-based analysis for $k = 10, k = 50$. All results are normalized to exhaustive search, *i.e.*, the best performance in the entire schedule space. Within the top-10 trials, our performance model achieves an average of 79% performance compared to the best in exhaustive search, but the bottleneck-based method only achieves 75%. Within the top-50 trials, which is a $40\times$ saving of trials compared to exhaustive search, our model achieves an average 92% performance, whereas the bottleneck-based method only achieves 88%. Our model also achieves $>$95% performance for all matrix-multiplication (MatMul) operators.

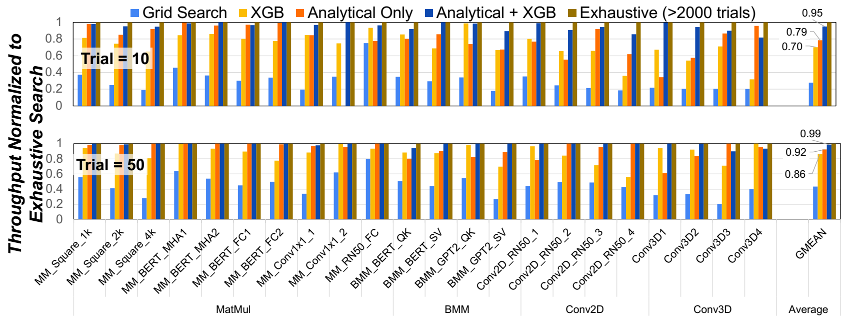


Figure 13. Search efficiency of schedule tuning methods.

5.5 Analytical-Model-Guided Schedule Tuning

This part evaluates our technique to combine the analytical model with machine learning (ML) based schedule-tuning. The metric is *best-in-k-trials* similar as in the last part. We compare our method with the other three methods, as detailed in Table 2: (1) Grid-Search, which simply grid-search all the parameter configurations and does not learn anything from the collected performance data. (2) XGB, which is the default method in TVM (Tavarageri et al., 2021), and uses XGBoost (Chen & Guestrin, 2016) as a cost model to fit the collected data and uses simulated annealing to propose new trials. (3) Analytical-only, which ranks all schedules according to their *predicted* performance via our analytical model (4) Analytical+XGB, which first pretrains the XGB model offline with pairs of schedules and their predicted performance from the analytical model, and next follows the same workflow as XGB.

Figure 13 shows the *best-in-k-trials* of the four searching methods, normalized to the best performance in exhaustive search. At a budget of 10 trials, our Model-Assisted XGB finds schedules that reach 95% of the best performance in exhaustive search, while sampling purely based on an analytical model or non-pretrained XGB gives 79% and 70% of the best possible performance accordingly. At a budget of 50 trials, which is a $> 40\times$ saving of trials compared to an exhaustive search, our method reaches 99% of the best possible performance, while Model-Ranking and XGB only obtain 92% and 86%, respectively. To sum up, we find that (1) analytical model helps ML: Model-Assisted XGB is better than XGB because it incorporates prior knowledge about the hardware, and (2) ML helps analytical model: Model-Assisted XGB is better than pure analytical model because it uses the actual profiled data to fine-tune the performance model.

6 Related Work

Pipelining. Pipelining, as a GPU kernel optimization, is frequently used in GPU libraries like CUTLASS (Nvidia, 2021). CUTLASS implements pipelining in matrix multiplication and convolution kernels. However, being a template-based kernel library, CUTLASS is unable to provide automatic pipelining for any tensor programs; this is only possible with our compiler-based solution.

The term "pipelining" in distributed DL training (Huang et al., 2019; Narayanan et al., 2019; Barham et al., 2022; Zheng et al., 2022a) refers to operator-wise parallelism. In this case, different GPUs compute different stages of a model, and multiple micro-batches are computed in a pipelined fashion. Compared to our optimization at the scope of a single kernel, those model-level pipelining work use distinct techniques and focuses mainly on stage partitioning strategy. Pipelining is also a hardware design technique widely used in accelerator designs (Liu et al., 2016; Jouppi et al., 2017; Sohrabizadeh et al., 2020; Liao et al., 2021), or hardware generation languages (Wei et al., 2017; Lai et al., 2019; Wang et al., 2021a; Parashar et al., 2019). Despite sharing the same mission of improving computation and memory system utilization, the hardware-based pipelining and our compiler-based approach are very different in that hardware technique mainly benefit accelerator design, but our technique benefits program optimization for general-purpose architectures like GPU. Software-pipelining has been studied to exploit instruction-level parallelism (Ning & Gao, 1993; Govindarajan et al., 1996) and multithread parallelism (Wei et al., 2012). Compared to those, ALCOP's task is more challenging because it must support multi-level pipelining and must automatically split code into a load- or compute-blocks using IR analysis, whereas, in other settings, the pipeline stages are straightforward.

Performance Model. There is a rich amount of work on analytical performance models for GPUs (Hong & Kim, 2009; Volkov, 2016; Wang et al., 2020; Huang et al., 2014; Zhang & Owens, 2011; Baghsorkhi et al., 2010; Lym et al., 2019). The most relevant is DELTA (Lym et al., 2019), which builds a model to predict the latency of Conv2D kernels on GPUs. However, ALCOP is the first to model how pipelining stage numbers affect performance and trade-offs between pipelining and tiling. ALCOP differs from analytical model-based search in that it combines ML- and analytical-based search as detailed in Section 4.3.

Recently, static analysis has arisen to supplement the standard ML-based schedule tuning, whose cost model lacks hardware knowledge. Tuna (Wang et al., 2021b) builds a performance model for CPU and GPU to replace the ML-based schedule tuning in AutoTVM (Chen et al., 2018b). We show that a combination of an analytical model and machine learning can achieve greater search efficiency than the Tuna technique.

7 Conclusion

This paper addresses the important need for **automatic pipelining** in deep learning compilers. Due to the large tiling size required to mitigate bandwidth constraints, inter-tile parallelism is inadequate for achieving high utilization, and intra-tile pipelining becomes essential. We propose the first compiler solution that supports *multi-stage, multi-level* pipelining. Through introducing automatic pipelining, our compiler can generate GPU programs with an average 1.23× and maximally 1.73× speedup over vanilla TVM (Chen et al., 2018a). Additionally, we develop an analytical performance model which significantly improves the search efficiency of the schedule tuning process.

References

Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 105–114, 2010.

Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., et al. Pathways: Asynchronous distributed dataflow for ml. *arXiv preprint arXiv:2203.12533*, 2022.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018a.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018b.

Dakkak, A., Li, C., Xiong, J., Gelado, I., and Hwu, W.-m. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, pp. 46–57, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Feng, B., Wang, Y., Chen, G., Zhang, W., Xie, Y., and Ding, Y. Egemm-tc: Accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 278–291, 2021.

Google. Xla: Optimizing compiler for machine learning, 2021. URL `https://www.tensorflow.org/xla`.

Govindarajan, R., Altman, E. R., and Gao, G. R. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and distributed systems*, 7(11):1133–1149, 1996.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, pp. 770–778, 2016.

Hong, S. and Kim, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 152–163, 2009.

Huang, J.-C., Lee, J. H., Kim, H., and Lee, H.-H. S. Gpumech: Gpu performance modeling technique based on interval analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 268–279. IEEE, 2014.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Katel, N., Khandelwal, V., and Bondhugula, U. Mlir-based code generation for gpu tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pp. 117–128, 2022.

Lai, Y.-H., Chi, Y., Hu, Y., Wang, J., Yu, C. H., Zhou, Y., Cong, J., and Zhang, Z. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 242–251, 2019.

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)*, pp. 789–801. IEEE, 2021.

Liu, S., Du, Z., Tao, J., Han, D., Luo, T., Xie, Y., Chen, Y., and Chen, T. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405. IEEE, 2016.

Lym, S., Lee, D., O'Connor, M., Chatterjee, N., and Erez, M. Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 293–303. IEEE, 2019.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

Ning, Q. and Gao, G. R. A novel framework of register allocation for software pipelining. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 29–42, 1993.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 883–898, 2021.

NVIDIA. cuBLAS. `https://docs.nvidia.com/cuda/cublas/index.html`, a. URL `https://docs.nvidia.com/cuda/cublas/index.html`.

NVIDIA. cuDNN. `https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html`, b. URL `https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html`.

Nvidia. Nvidia A100 tensor core GPU. *Data Sheet*, pp. 20–21, 2020a.

Nvidia. Nvidia tesla V100 GPU architecture. *Data Sheet*, pp. 20–21, 2020b.

Nvidia. Nvidia cutlass release v2.7, 2021. URL `https://github.com/NVIDIA/cutlass`.

Parashar, A., Raina, P., Shao, Y. S., Chen, Y.-H., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315. IEEE, 2019.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sohrabizadeh, A., Wang, J., and Cong, J. End-to-end optimization of deep learning applications. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 133–139, 2020.

Tavarageri, S., Heinecke, A., Avancha, S., Kaul, B., Goyal, G., and Upadrasta, R. PolyDL: Polyhedral Optimizations for Creation of High-performance DL Primitives. *ACM Transactions on Architecture and Code Optimization*, 18 (1):1–25, 2021. ISSN 15443973. doi: 10.1145/3433103.

Volkov, V. *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

Wang, J., Guo, L., and Cong, J. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 93–104, 2021a.

Wang, L., Jahre, M., Adileho, A., and Eeckhout, L. Mdm: The gpu memory divergence model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1009–1021. IEEE, 2020.

Wang, Y., Zhou, X., Wang, Y., Li, R., Wu, Y., and Sharma, V. Tuna: A static analysis approach to optimizing deep neural networks. *arXiv preprint arXiv:2104.14641*, 2021b.

Wei, H., Yu, J., Yu, H., Qin, M., and Gao, G. R. Software pipelining for stream programs on resource constrained multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2338–2350, 2012.

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., and Cong, J. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

Xing, J., Wang, L., Zhang, S., Chen, J., Chen, A., and Zhu, Y. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems*, 4:204–216, 2022.

Yan, D., Wang, W., and Chu, X. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 634–643. IEEE, 2020.

Zhang, Y. and Owens, J. D. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*, pp. 382–393. IEEE, 2011.

Zhao, J., Gao, X., Xia, R., Zhang, Z., Chen, D., Chen, L., Zhang, R., Geng, Z., Cheng, B., and Jin, X. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems*, 4:1–19, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022a.

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. *arXiv preprint arXiv:2009.10924*, 2020.

Zheng, Z., Yang, X., Zhao, P., Long, G., Zhu, K., Zhu, F., Zhao, W., Liu, X., Yang, J., Zhai, J., Song, S. L., and Lin, W. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pp. 359–373, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507723. URL https://doi.org/10.1145/3503222.3507723.

A. Artifact Appendix

A.1 Abstract

We provide the compiler transformation pass for pipelining optimization used in our paper. The functionality and technical details of the transformation pass is described in Section 3 of the paper. We provide instructions to set up evaluation environments, build the code base, and provide scripts to benchmark DNN operators with our optimizations and on baselines.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Pipelining compiler pass.
- **Program:** Benchmark scripts.
- **Run-time environment:** Docker image that we provide, or built-from-scratch environment following our instructions.
- **Hardware:** NVIDIA A100 GPU.
- **Output:** Program log files.
- **Experiments:** Experiments to reproduce the operator speedup results in the paper Section 5.A.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1-2 minutes for example test case, 5 hours for full test.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0.
- **Workflow framework used?:** Apache TVM (integrated in the docker image we provide).

A.3 Description

A.3.1 How delivered

The artifact is available through the public github repository: `https://github.com/hgyhungry/alcop-artifact`.

A.3.2 Hardware dependencies

NVIDIA A100 GPU.

A.3.3 Software dependencies

We provide two ways to set up the dependencies of this artifact.

The first way is to use the docker image we provide with all dependencies installed: hguyue1/alcop:latest.

The second way is to build our codebase from scratch. For the second way, we require

- CUDA toolkit v11.2 or higher
- LLVM v10.0.0 or higher

A.3.4 Data sets

The artifact does not require specific data sets.

A.4 Installation

We provide two ways to install the package for this artifact.

Method 1: The first way is to use the docker image we provide where everything is built and installed:

Step 1: Start docker.

```
#!/bin/bash
docker run -it --gpus all \
  -w /tvm hguyue1/alcop:latest bash
```

Step 2: Inside the docker, set environment variables.

```
export TVM_HOME=/tvm
export PYTHONPATH=$TVM_HOME/python:${PYTHONPATH}
```

Method 2: The second way is to install the package from scratch. For the second way, you can follow the steps below:

Step 1: Start from an NVIDIA docker

```
docker run -it --gpus all -v /path/to/this/repo:/tvm \
  -w /tvm nvidia/cuda:11.4.0-cudnn8-devel-ubuntu20.04 \
  bash
```

Step 2: Inside the docker, install dependencies:

```
apt-get update
apt-get install -y python3 python3-pip \
    python3-dev python3-setuptools gcc \
    libtinfo-dev zlib1g-dev build-essential \
    cmake libedit-dev libxml2-dev \
    git llvm
pip install numpy decorator attrs tornado \
    psutil 'xgboost>=1.1.0' cloudpickle \
    matplotlib torch pytest
```

Step 3: Build the TVM shared library

```
# create build directory
mkdir build
cp cmake/config.cmake.template build/config.cmake

# build the shared library
cd build
cmake ..
make -j
```

Step 4: Set environment variables.

```
export TVM_HOME=/tvm
export PYTHONPATH=$TVM_HOME/python:${PYTHONPATH}
```

A.5 Experiment workflow

Experiment 1: run a simple GEMM example.

```
cd /tvm/auto_pipeline_exp/single_op
# baseline
python3 dense_tensorcore_in_topi.py
# optimized
python3 dense_tensorcore_autopipeline_example.py
```

Experiment 2: run the whole test suite. This means to search the entire configuration space with autotvm infrastructure for our optimizations, baseline vanilla TVM, and baseline TVM with double-buffering.

```
cd /tvm/auto_pipeline_exp/single_op
sh run.sh
```

A.6 Evaluation and expected result

Experiment 1: Expected output of the baseline script

```
# expected output of dense_tensorcore_in_topi.py
Test with pipelining
Result correct.
Time cost of 0.001948 throughput 70.571906 Tflops
```

Expected output of the baseline script

```
# expected output of
# dense_tensorcore_autopipeline_example.py
Test with pipelining
Running on target: cuda
[Info] calling pipeline buffer transformation
[Info] calling swizzle buffer transformation
Result correct.
Time cost of 0.000738 throughput 186.303047 Tflops
```

Experiment 2: The output log files are under the folder `/tvm/auto_pipeline_exp/single_op/result`. We provide the expected output files for several operators through the following shared drive. `https://drive.google.com/drive/folders/1pZhwS4zRFIIQfLlL8DfbJRtEjkxFouqA?usp=sharing`

A.7 Methodology

Submission, reviewing and badging methodology:

- `http://cTuning.org/ae/submission-20190109.html`
- `http://cTuning.org/ae/reviewing-20190109.html`
- `https://www.acm.org/publications/policies/artifact-review-badging`