Enabling Data Movement and Computation Pipelining in Deep Learning Compiler

Guyue Huang\textsuperscript{1*}, Yang Bai\textsuperscript{2}, Liu Liu\textsuperscript{1}, Yuke Wang\textsuperscript{1}, Bei Yu\textsuperscript{2}, Yufei Ding\textsuperscript{1}, Yuan Xie\textsuperscript{1}
\textsuperscript{1}University of California, Santa Barbara, \textsuperscript{2}Chinese University of Hong Kong
\textsuperscript{*}guyue@ucsb.edu

Abstract—Pipelining between data loading and computation is a critical tensor program optimization for GPUs. Multi-stage pipelining across the multi-level buffer hierarchy of GPU is particularly indispensable on the latest NVIDIA Ampere GPUs to reduce resource idleness and guarantee kernel performance. Currently, people rely on libraries written by experts such as cuBLAS to access the pipelining optimization instead of through a tensor program transformation, which is inextensible to new operators and un-composable with prior tensor compiler optimizations. We present ALCOP, an automatic pipelining framework based on TVM infrastructure that 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× speedup on average (up to 1.73×) over vanilla TVM. On end-to-end models, ALCOP can improve upon TVM by up to 1.18×, and XLA by up to 1.64×. 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× fewer trials.

I. INTRODUCTION

Deep learning (DL) has achieved great success in a variety of application fields, spanning computer vision, natural language processing, and recommendation systems [1]–[3]. The widespread use of GPUs [4], [5] 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 [6]–[11]. Because Tensor Core throughput continued to increase but memory bandwidth lagged, research on tiling and fusion to improve data re-use surged [12]–[13]. 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 [11] has studied double-buffering, a common but far less complicated 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. V-A).

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).

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\textsuperscript{\textregistered} (Automatic Load-Compute Pipelining), the first DL compiler solution (auto-scheduler, program transformation, auto-tuner) for automated

\textsuperscript{1}ALCOP is a copper-based alloy with significantly-enhanced endurance power of DL compilers for modern AI-GPUs.
Fig. 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.

Fig. 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.

multi-stage, multi-level pipelining; its architecture is shown in Figure 2. 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. II)

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

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. IV)

Experiments show that ALCOP pass can bring on average $1.23 \times$, and up to $1.73 \times$ speed-up on DL operators over TVM [10]. ALCOP brings $1.02-1.18 \times$ end-to-end inference speed-up for six DL models over TVM [10], and $1.01-1.64 \times$ over XLA [18]. Through combining the analytical model with ML-based tuning, we can identify schedules with 99% performance compared to the best schedule in the entire design space while reducing the number of trials by $40 \times$.

II. 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. Consequently, 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 III will turn the load-and-use structure into its pipelined version.

Two important questions must be addressed: First, we must determine 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.

A. 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 do not pipeline a buffer produced outside of a sequential loop. The purpose of pipelining is to overlap the data-loading operation of future iterations with the computation of the current iteration. This load-and-use loop must be sequential and cannot be parallelized (bound to parallel threads) or unrolled. This condition is typically violated by stencil algorithms that use tiling to increase the reuse of the input tensor, but the buffer is only filled and used once, as opposed to being generated in a sequential loop. Consequently, the pipelining approach cannot be applied to these buffers.

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 await certain loading instructions (e.g., instructions issued in the fourth-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.

B. 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 \( S_2 \) from tensor \( S_1 \), applying cache-reading means inserting a new tensor \( S_1_{buf} \) which is an identical copy of \( S_1 \) 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
Given a set of buffers we want to apply the Second Step.
The collection of pipelining hints inserted by the schedule of the pipelining pass. Figure 6 also depicts the transformation steps. This step increases the size of the memory buffer by the number of pipeline stages. Relevant transformed code is highlighted in light yellow.

**The First Step.** This step shifts the indices used in memory access. The relevant 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 Second Step.** This step handles indices for buffer rolling (circular access) and out-of-bound wrapping. Relevant code is

![Diagram](image_url)

**Fig. 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.

III. 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 II, 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.

A. 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 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 9, 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 5.

B. 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 relevant 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
Algorithm:
```c
/* MatMul */
C[i, j] = sum(A[i, k] * B[j, k],
  reduce_axis=(k,))
```

Schedule:
```c
/* cache read B is omitted for brevity */
A_shared = cache_read(A)
A_reg = cache_read(A_read)
```

```
/* tiling */
C.tile(TB_tile_i, TB_tile_j, TB_tile_k),
(Warp_tile_i, Warp_tile_j, Warp_tile_k)
```

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

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

```
A_shared.consumer_wait()
A_reg.pipeline(stage=2)
```

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

```
A_shared.consumer_release()
A_reg.pipeline(stage=2)
```

```
A_shared.producer_acquire()  
A_shared.consumer_wait()  
A_shared.producer_commit()
```

```
A_reg.pipeline(stage=2)
```

```
A_shared.consumer_release()
```

```
A_shared.producer_acquire()
```

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

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

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`. 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.

IV. STATIC ANALYSIS GUIDED TUNING

This section introduces how we combine a static analytical performance model with existing machine-learning (ML) based auto-tuning [19] 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 5.

A. Top-Level Model

Our analytical model is shown in Table I. At the top level, the threadblocks are grouped into threadblock-batches (threadblk_batch), and one threadblock-batch occupies all

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

with-memcpy_async-pipeline-pattern-multi
TABLE I: Analytical Performance Model

| Category                  | Model                                                                                           |
|---------------------------|-------------------------------------------------------------------------------------------------|
| Kernel Latency Model      | $T_{kernel} = T_{threadblk} \times N_{threadblk\_batch}$                                        |
| Pipeline Latency Model    | $T_{load\_use\_loop} = \frac{\text{Input: } T_{load\_use\_loop}}{\text{Output: } T_{load\_use\_loop}}$, $\begin{cases} \text{If } T_{load} \leq (N_{pipe} \times N_{threadblk\_batch} - 1) \times T_{use} : & T_{load\_use\_loop} = T_{use} \times N_{loop} \\ \text{Else: } : & T_{load\_use\_loop} = (T_{load} + T_{use}) \times N_{loop} \div N_{pipe} \end{cases}$ |
| Threadblock Latency Model | $T_{threadblk} = T_{init} + T_{main\_loop} + T_{epilogue}$                                       |
| Computation Latency Model | $T_{compute} = \frac{\text{FLOPs}_{\text{main}\_loop}}{\text{FLOPs}_{\text{load}\_use\_loop}}$         |
| Memory Latency Model      | $T_{LLC\_load} = \text{LAT}_{\text{LLC}} \times \frac{\text{Bytes}_{\text{threadblk\_batch}}}{\text{BW}_{\text{LLC}}}$, $T_{DRAM\_load} = \text{LAT}_{\text{DRAM}} + \frac{\text{Bytes}_{\text{threadblk\_batch}}}{\text{BW}_{\text{DRAM}}}$ |
| Epilogue Model            | $T_{epilogue} = \text{LAT}_{\text{DRAM\_write}} + \frac{\text{Bytes}_{\text{output\_size}} \times N_{threadblk\_batch}}{\text{BW}_{\text{DRAM\_write}}}$ |

Fig. 8: A high-level view of the performance model. Compared to prior work [20], our model takes into account the constraints and trade-offs among pipelining, tiling and spatial parallelism.

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 [20].

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, $N_{pipe}, N_{threadblk\_batch}$, which means the number of 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.

B. 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 I 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 ($N_{threadblk\_batch}$). 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., $(T_{load} + T_{use}) / N_{pipe}$.
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 tensorcore throughput in an SM. When determining the latency of memory copies, four parameters must be considered: the number of parallel workers (threadblocks or warps) to share with the bandwidth, and a constant round-trip latency \( \text{LAT} \). Note that GPU LLC is shared by all SMs. Hence the DRAM traffic cannot be computed by the sum of data loaded by all threadblocks because the data may hit in LLC. We model DRAM traffic by deciding the working set of a threadblock-batch.

C. 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 I compares our method (Model-Assisted XGB) with other available auto-tuning approaches.

V. EVALUATION

A. 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 [10] 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 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 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 II: Comparison of compiler search methods. |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|
| Cost Model | Grid Search | XGB | Anal. Only | Anal. + XGB (ours) |
| Prior Knowledge? | N.A. | ML | Yes | Yes |
| Sampling Method? | No | Yes | Yes | Yes |
| Cost-Model | Enumerate | Simulated Annealing | Cost-Model Ranking | Simulated Annealing |

Fig. 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.
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× speedup. Without multi-level and multi-stage pipelining, ALCOP w/o MS, ML can only give 1.01× speedup over TVM.

B. End-to-End Performance

To evaluate end-to-end model acceleration, we compare against two baselines: TVM [10] and XLA [18] (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 [2] and GPT-2 [21] are popular models in Natural Language Processing (NLP), ResNet-18, ResNet-50 [1] and VGG-16 [22] 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 III shows the end-to-end speedup in real models. We achieve 1.02-1.18× end-to-end speedup over TVM and 1.01-1.64× speedup over XLA.

C. Comparison with Libraries

We compare with kernels in vendor libraries (cuBLAS [16]/cuDNN [23]), 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 compilers 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.

D. 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.

### E. 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 II: (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 [24], and uses XGBoost [25] 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.

### VI. Related Work

**Pipelining.** Pipelining, as a GPU kernel optimization, is frequently used in GPU libraries like CUTLASS [6]. 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 [26–28] refers to operator-wise parallelism. Pipelining is also a hardware design technique widely used in accelerator designs [29–32], or hardware generation languages [33–36]. Software-pipelining has been studied to exploit instruction-level parallelism [37], [38] and multithread parallelism [39]. Compared to all these pipelining scenarios, 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 [20], [40]–[45]. The most relevant is DELTA [20], 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 IV-C.

Recently, static analysis has arisen to supplement the standard ML-based schedule tuning, whose cost model lacks hardware knowledge. Tuna [46] builds a performance model for CPU and GPU to replace the ML-based schedule tuning in AutoTVM [19]. We show that a combination of an analytical model and machine learning can achieve greater search efficiency than the Tuna technique.

### VII. 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 \( \times \) and maximally 1.73 \( \times \) speedup over vanilla TVM [10]. Additionally, we develop an analytical performance model which significantly improves the search efficiency of the schedule tuning process.
of the 36th annual international symposium on Computer architecture, 2009, pp. 152–163.

[41] V. Volkov, Understanding latency hiding on GPUs. University of California, Berkeley, 2016.

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

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

[44] Y. Zhang and J. D. Owens, “A quantitative performance analysis model for gpu architectures,” in 2011 IEEE 17th international symposium on high performance computer architecture. IEEE, 2011, pp. 382–393.

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

[46] Y. Wang, X. Zhou, Y. Wang, R. Li, Y. Wu, and V. Sharma, “Tuna: A static analysis approach to optimizing deep neural networks,” arXiv preprint arXiv:2104.14641, 2021.