Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning

Chen Zhang, *Tsinghua University;* Lingxiao Ma and Jilong Xue, *Microsoft Research;* Yining Shi, *Peking University & Microsoft Research;* Ziming Miao and Fan Yang, *Microsoft Research;* Jidong Zhai, *Tsinghua University;* Zhi Yang, *Peking University;* Mao Yang, *Microsoft Research*

https://www.usenix.org/conference/osdi23/presentation/zhang-chen

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

COCKTAILER: Analyzing and Optimizing Dynamic Control Flow in Deep Learning

Chen Zhang[†£◇*] Lingxiao Ma[◇] Jilong Xue[◇] Yining Shi[‡◇*] Ziming Miao[◇]
Fan Yang[◇] Jidong Zhai[†] Zhi Yang[‡] Mao Yang[◇]
[†]*Tsinghua University* [‡]*Peking University* [◇]*Microsoft Research*

Abstract

With the growing complexity of deep neural networks (DNNs), developing DNN programs with intricate control flow logic (e.g., loops, branches, and recursion) has become increasingly essential. However, executing such DNN programs efficiently on accelerators is challenging. Current DNN frameworks typically process control flow on the CPU, while offloading the remaining computations to accelerators like GPUs. This often introduces significant synchronization overhead between CPU and the accelerator, and prevents global optimization across control flow scopes.

To address this challenge, we propose COCKTAILER, a new DNN compiler that co-optimizes the execution of control flow and data flow on hardware accelerators. COCKTAILER provides the *u*Task abstraction to unify the representation of DNN models, including both control flow and data flow. This allows COCKTAILER to expose a holistic scheduling space for rescheduling control flow to the lower-level hardware parallelism of accelerators. COCKTAILER uses a heuristic policy to find efficient schedules and is able to automatically move control flow into kernels of accelerators, enabling optimization across control flow boundaries. Evaluations demonstrate that COCKTAILER can accelerate DNN models with control flow by up to 8.2× over the fastest one of the state-of-the-art DNN frameworks and compilers.

1 Introduction

In deep neural networks (DNNs), control flow plays a crucial role in accomplishing sophisticated tasks, akin to its usage in general programming languages. Examples of this include iterating over sequential data like text and time steps, activating different components of the model based on input-data-driven conditions, dynamically skipping some computation based on runtime decisions for efficient computation, and recursively traversing tree-based data structures. A DNN program is typically divided into two parts: control flow and data flow. The data flow is typically represented as a graph of DNN operators, which can be efficiently executed on specialized accelerators such as GPUs. The control flow, on the other hand, is either implemented as a special operator [4] or by directly reusing the built-in statements of programming languages [36], and is typically executed on a CPU. Therefore, the control flow and data flow are executed alternatively in an entire DNN computation: the control flow determines which part of the data flow should be executed, and then the corresponding data flow is sent to accelerators for processing and the result is obtained, which is used to decide the next step of control flow.

However, the interleaved execution paradigm on both sides in existing DNN frameworks often introduces significant efficiency issues. First, the control flow and data flow require frequent synchronization between the CPU and accelerator (e.g., for checking conditions based on results), resulting in significant communication overhead (e.g., across PCIE) in the critical path. Second, the control flow in a DNN program often establishes explicit boundaries between data flow operators, which prevents their holistic optimization for maximum efficiency, such as fusing two operators across a loop scope. Lastly, the control flow implicitly serializes the execution of data flow operators that could potentially be executed in parallel. We have observed that these overheads are prevalent in existing DNN models and can often occupy as much as 72% of the total time in PyTorch. These efficiency issues not only introduce obstacles to dynamic model developments but also make many optimizations, e.g., dynamically skipping some computation, hard to achieve theoretical speedup.

Based on our observation of DNN workload patterns, the fundamental reason for the inefficiency is the *parallelism mismatch* between the control flow and data flow. In particular, control flow operations, such as loops, branches, and recursion, are single-thread semantic and execute in a strictly sequential order. However, the data flow operators are parallelizable, running on multiple parallel threads (e.g., GPU cores) and synchronizing periodically across different scopes

[£]Tsinghua University, BNRist
[*]Work is done during the internship at Microsoft Research.

of threads (between operators or thread blocks). To control the execution flow of a parallel program, the mismatch between control flow and data flow forces existing practices to place the control flow either outside the DNN operators (e.g., in existing DNN frameworks) or inside an individual DNN operator, through implementing custom kernels in an ad-hoc way (e.g., Relu operator). This can be either inefficient or unscalable to support the increasing demands for control flow in DNN workloads.

In this paper, we present a new DNN compiler, COCKTAILER, that addresses the challenges of co-optimizing control flow and data flow in a single space. COCKTAILER is based on three key insights observed from studying DNN models and modern accelerators. First, the data flow in DNNs is inherently a multi-level parallel program, where individual operators are executed in different hardware parallelism, such as threads, thread blocks, or kernels in GPUs. Second, the control flow operations in DNNs are mostly applied at the operator level, where all lower-level parallelisms share the same control result. This implies that the control flow can be rescheduled to the low-level parallelism by replicating the control logic to all parallel tasks at different levels. Most importantly, modern hardware accelerators, such as GPUs, are designed to support the control logic in their low-level programming languages in each thread, which makes this rescheduling approach feasible in practice.

Based on these insights, COCKTAILER introduces the *u*Task abstraction as the primitive execution unit of a DNN program for both control flow and data flow. An operator in data flow can be naturally decomposed into different levels of granularity of *u*Tasks aligned with its computation parallelism. For control flow, COCKTAILER introduces three types of special *u*Tasks: loop *u*Task, branch *u*Task, and *u*Task reference, to represent the program with control flow as a special *u*Task. By unifying the DNN program into the *u*Task granularity, COCKTAILER creates a holistic space for co-scheduling control flow *u*Tasks with compute *u*Tasks, i.e., by assigning the control flow *u*Task to the most efficient parallel level with data dependencies resolved correctly. To facilitate this scheduling, COCKTAILER proposes a scheduling mechanism and a traverse-based bottom-up scheduling policy that incorporates all control flow optimizations such as function inline, loop unroll, and recursion unroll.

As a result, COCKTAILER is able to automatically move control flow operations, such as loops or branches, into accelerator side when applicable, enabling more optimization opportunities across the control flow boundary. COCKTAILER is built on top of general DNN tensor compilers by leveraging their kernel generation capabilities for *u*Task, allowing it to adapt to different accelerators such as CUDA GPUs and ROCm GPUs easily. COCKTAILER's approach can be applied to both DNN frameworks that implement control flow as special operators or language-built-in statements, by only compiling the sub-programs that can be optimized by COCKTAILER. Evaluation with 7 typical DNN models on CUDA GPUs and ROCm GPUs shows that COCKTAILER accelerates these models by up to 8.2× over the fastest one of state-of-the-art DNN frameworks and compilers. Furthermore, the evaluation shows that COCKTAILER not only reduces the overheads introduced by control flow but also enables scenarios like dynamically skipping some computation by achieving real speedup. The code has been open-sourced[1].

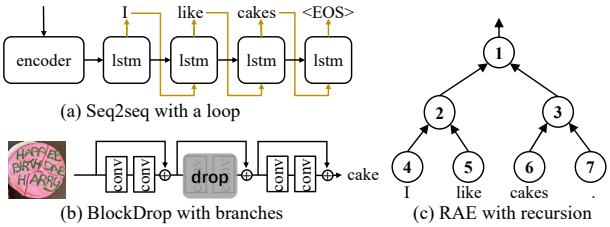


Figure 1: Models with control flow

2 Background and Motivation

DNNs have been successfully applied in many areas, such as computer vision, speech, and natural language. Meanwhile, the concept of control flow in programming languages is introduced to deep learning. The architecture of DNN models rapidly evolves from sequential feed-forward layers [15, 23, 38] to structures with complex control logic [16, 39–41, 46, 47], enabling dynamic computation and adaptability within the network architecture:

- **Dynamic computation**. Control flow enables constructing dynamic computation architectures, which can adapt their structure during runtime. For example, the loops are widely used to handle variable-length sequences (e.g., text, speech, time-series data) in RNNs [16, 40, 58] and Transformers [43].
- **Conditional computation**. Control flow enables the execution of specific parts of the model based on certain conditions [7, 24] like executing different parts of the model for images with different resolution.
- **Efficient computation**. Control flow can help reduce the computational resources required by DNN models by selectively executing parts of the model based on input data or intermediate results, e.g., the early-exiting mechanism [46, 47] that can skip some layers on easy input samples. Besides, control flow can be leveraged to adapt DNN models to different environments (e.g., different hardware accelerators) by trading off computation cost and model performance via control flow [26].

Dynamic computation for structural data is a common requirement in modern deep learning models. For instance, nearly 27% of the 52 models in PyTorch Hub (as of commit ID 1c747e2) contain structural data (e.g., sequence, tree). Furthermore, a survey on dynamic DNN models [13] indicates

[1] https://github.com/microsoft/nnfusion/tree/cocktailer_artifact/artifacts

that conditional computation and efficient computation are promising research directions.

In programming languages, control flow constructs are typically categorized into the following types: sequence, branch, loop, and subroutine (function). Similarly, a majority of DNN models with complex control flow can be categorized into models with loops for temporal-wise dynamism (e.g., LSTM [16], NASRNN [58], Seq2seq [40]), models with branches to skip computation (e.g., BlockDrop [47], SkipNet [46]), and models with tree-based architecture via recursion (e.g., RAE [39], Tree-LSTM [41]). We discuss the three representative categories with their representative models, as shown in Figure 1.

- **Loop**. Seq2seq [40] can generate arbitrary-length sequences. It contains a while loop that continues to emit new tokens until an end-of-sequence (EOS) token.
- **Branch**. BlockDrop [47] is a convolution neural network that can drop some convolution layers. Each layer is implemented by an if statement with two branches for whether executing the branch or not.
- **Recursion**. RAE [39] computes the embedding of a parse tree by traversing the tree. It can be implemented by a depth-first search using recursion.

To support these emerging DNN models, there are two mainstream approaches. The first one, represented by TensorFlow of version 1.x [4], supports these complex model architectures by introducing a set of control flow operators [52] like `Enter` and `NextIteration`. Then, these control flow operators are executed in the framework runtime with the CPU threads. The second one, represented by PyTorch [36] and JAX [11], leverages the programming language to represent and execute the control flow. For example, in PyTorch, algorithm designers program control logic in Python, and these control flow statements are running in the Python runtime.

Both approaches schedule data flow operators onto the accelerators while maintaining control flow in the CPU side for execution. The reason is the *parallelism mismatch* between the control flow and the data flow. Specifically, different from data flow operators (e.g., matrix multiplication) that have internal data parallelism, control flow operations are represented as single-thread computation. Modern hardware accelerators have massive hierarchical parallel processing units. For example, GPUs contain many parallel streaming multiprocessors (SMs) and each SM has many parallel cores. This architecture aligns with data flow operators' parallelism, but it is hard to schedule the single-thread control flow to the massive hierarchical parallel processing units for execution. Therefore, current practices [4, 11, 36, 52] schedule control flow operations to the CPU side for execution. Such approaches introduce boundaries between DNN operators of different basic blocks, resulting in performance issues.

Figure 2 compares JAX's performance of executing the three models via dynamic control flow to executing the corresponding traced static graph that has removed all the control

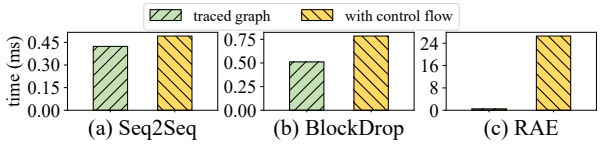


Figure 2: Control flow overhead in JAX

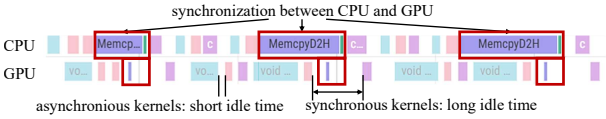


Figure 3: Timeline of BlockDrop in JAX. The data copy from GPU to CPU (`MemcpyD2H`) causes synchronization.

flow computation and only remains data flow computation. Compared with the traced graph baseline, the control flow computation of JAX causes 1.16×, 1.54×, and 56.22× slowdown in Seq2Seq, BlockDrop, and RAE, respectively. More results can be found in §5. The loop in Seq2seq and branches in BlockDrop use control flow operators. The recursion in RAE is executed in Python. Both approaches cannot match the performance of static traced graphs.

The performance issue comes from the following parts.

(1) Boundary overheads. Executing data flow operators on the accelerator and control flow operations on the CPU can incur synchronizations between the CPU and the accelerator. Take the BlockDrop model on a CPU-GPU system as an example, the DNN operators in the branch body are executed in the GPU side, while the branch operation is executed in the CPU side. The CPU stalls when waiting for the GPU to provide the data required for deciding the branch target, and then the GPU stalls to wait for the CPU to check the branch condition and send the following operations to the GPU. The boundary overheads mainly contain the communication between the CPU and the accelerator and the kernel launching. This boundary may also break the asynchronous execution in the accelerator side.

Figure 3 shows part of the timeline of JAX executing the BlockDrop model that the CPU-GPU synchronization not only has high synchronization overheads but also breaks the asynchronous execution and causes a long idle time without computation in the GPU side.

(2) Boundary limits the optimization scope. Executing control flow and data flow on separate sides divides the DNN program into sub-programs, each representing a static data flow that can be executed on the accelerator side. Many DNN optimizations (e.g., Rammer [28], kernel fusion [35, 56], etc.) are limited to only optimizing these sub-programs, resulting in sub-optimal performance. Consider a multi-layer LSTM model as an example: the DNN operators in LSTM cells across different layers can be scheduled for parallel execution. However, the loop control flow constrains the DNN optimizations within a cell, resulting in overlooking this parallelism.

(3) Boundary prevents parallelism in DNN programs. This boundary makes the DNN programs in different con-

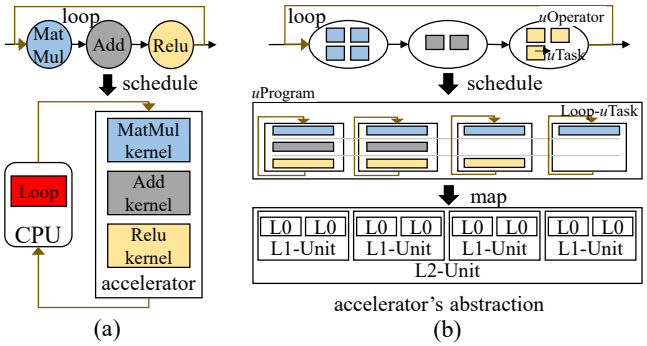


Figure 4: System overview of DNN computation in (a) existing DNN frameworks (e.g., JAX), and (b) COCKTAILER

trol flow statements executed sequentially due to the synchronizations between the CPU and the accelerator, which may prevent possible inter-operator parallelism. Take the RAE model as an example, the recursion builds up a tree-based architecture where operators without dependencies can be executed concurrently. However, because the control flow can only be executed sequentially, operators of nodes without dependencies are executed sequentially.

Observations and opportunities. Given the fundamental limitations of current approaches described above, it is desirable to schedule the DNN programs including control flow and data flow in a single space (i.e., the accelerator side). However, it is challenging to achieve this because of the parallelism mismatch between control flow and data flow. Fortunately, control flow in DNN programs is applied across DNN operators, that is to say, the DNN operators' computation under control flow shares the same control logic. On the other hand, most hardware accelerators (e.g., GPUs) have the ability to execute control flow instructions. If we represent control flow in a finer granularity that can be properly mapped to the parallel processing units for execution, we can schedule both data flow and control flow to the accelerator side.

3 COCKTAILER Design

The observation in §2 motivates COCKTAILER, a DNN compiler for co-optimizing control flow and data flow in a single space. Figure 4 shows the overview of COCKTAILER. First, COCKTAILER takes a DNN program with control flow and data flow as input, where each operator in the data flow is a *uOperator* that consists of independent and homogeneous *uTasks*. Each *u*Task can be scheduled to one compute unit of the accelerator. Second, instead of scheduling control flow on the CPU side and data flow on the accelerator side separately, COCKTAILER schedules control flow and data flow inside the program in a single space. COCKTAILER will generate the *uProgram* representation for the program, which contains multiple independent *u*Tasks (e.g., the Loop-*u*Tasks in Figure 4(b)) that can be scheduled to the parallel compute units in hardware accelerators for execution. Each *u*Task represents both the control flow and data flow logic of one compute unit.

COCKTAILER abstracts an accelerator of massive parallelism as multiple levels of parallel processing units. In each level, there are parallel and homogeneous processing units, which construct a higher level of processing unit. This hardware abstraction naturally aligns with common hardware accelerators. Take the NVIDIA GPU as an example, there are multiple homogeneous streaming multiprocessors (SMs) in a GPU, where each SM consists of multiple homogeneous cores. Therefore, NVIDIA GPUs can be mapped as an architecture with 3 levels of parallel processing units in COCKTAILER's hardware abstraction shown as Figure 4: L0 is the core (thread); L1 is the SM (thread block); L2 is the GPU device (kernel).

The example loop structure in Figure 4 is scheduled as a *u*Program mapped on the 3-level accelerator. The *u*Program consists of 4 loop-*u*Tasks for 4 L1-Units resepectively and each loop-*u*Task is mapped to a L1-Unit for execution. Both the data flow operators and the loop are scheduled into the loop-*u*Tasks. Take the first loop-*u*Task as an example, it has a loop control flow and a list of *u*Tasks for data flow operations containing 1 MatMul *u*Task, 1 Add *u*Task, and 1 Relu *u*Task.

The concepts of *u*Task, *u*Operator, and *u*Program are described in detail in §3.1. And the *u*Program scheduling is illustrated in §3.2.

3.1 *u*Task-based DNN Program

To co-schedule the control flow and the data flow of a DNN program to accelerators with massive parallel units, COCKTAILER defines the DNN program in fine grained with the concept of *unit-task* (*u*Task). Specifically, *u*Task is defined as the computation logic that can be scheduled to one of the multi-level processing units in hardware accelerators for execution. Note that the computation in a *u*Task can be a list of other *u*Tasks, i.e., a nested *u*Task. *u*Program represents the execution plan of the *u*Task-represented DNN program mapped to a level of parallel processing units on the hardware.

***u*Task and *u*Operator for data flow operators.** As Figure 5(a)(c) show, a data flow operator is represented as a group of independent and homogeneous *u*Tasks where each *u*Task is the computation to be scheduled to one processing unit. Specifically, each *u*Task takes a slice of the input tensor via `get_input_data()` and executes the corresponding computation defined in `compute()`. Then, a *u*Operator is defined as the collection of all *u*Tasks of the corresponding data flow operator. The *u*Tasks of a *u*Operator are indexed by the logical `uTask_id` and called by `compute(uTask_id)`. The total *u*Task count in an *u*Operator is reported by `get_uTask_num()`. When all *u*Tasks in an operator are executed, the execution of this operator is finished.

Data flow operators (e.g., matrix multiplication) are usually implemented as multiple independent and homogeneous tasks that are scheduled to the massive parallel units of accelerators

```
interface uTask {
 void compute();
 void get_input_data();
};
```
(a)

```
interface NestedUTask: uTask {
 void compute();
 void get_input_data();

 vector<uTask> body_uTasks;
};
```
(b)

```
interface uOperator {
 void compute(uTask_id);
 size_t get_uTask_num();

 set<uTask> uTasks;
};
```
(c)

```
interface uProgram {
  void compute(uTask_id);
  size_t get_uTask_num();
  set<uTask> uTasks;
  size_t unit_level;
};
```
(d)

Figure 5: The definition of *u*Task, *u*Operator and *u*Program

for execution. Each task consumes a slice of the input tensor, processes the corresponding computation over the input slice, and produces a slice of the output tensor. Take the NVIDIA GPU as an example, the kernel of an operator (e.g., matrix multiplication) is scheduled as multiple thread blocks and each of them is mapped to an SM for processing a tensor slice. Furthermore, a thread block is scheduled as multiple threads and mapped to cores for processing a tensor slice. Therefore, the concept of *u*Task is not only natural to represent the fine-grained computation of data flow operators, but also aligns with the hardware architecture of multi-level parallel processing units in accelerators.

***u*Task for control flow.** It is natural to represent data flow operators with *u*Tasks due to the internal data parallelism that can be divided into parallel tasks. However, different from DNN operators, the control flow cannot be divided into such parallel tasks. To enable the scheduling of control flow on the parallel processing units, we need to bridge this gap of the mismatching between the control flow computation and the massive parallelism in the accelerator.

Control flow operation applies to a scope of DNN operators in DNN programs. When the DNN operators can be divided into independent and homogeneous *u*Tasks, controlling the DNN operators is equal to applying control flow computation on each *u*Task. For example, assuming there is a loop structure that has a matrix multiplication operator in the loop body, compared with executing the loop over the operator, it is equally that let each unit of the hardware accelerator process the loop control flow over the *u*Task of the operator. If we apply such control flow on the scope of the fine-grained representation of these DNN operators, we can schedule such computation including the control flow to the parallel processing units of the hardware accelerators. That is to say, we can represent control flow in the *u*Task granularity by replicating the control flow computation to the multi-level parallel units that each unit executes the control flow independently and controls the *u*Tasks scheduled on the unit.

According to the observation, COCKTAILER represents control flow operations as `NestedUtasks` defined in Figure 5(b), where the computation in the body is represented in the `body_uTasks`. These *u*Tasks have data dependencies and should be executed sequentially on one processing unit. Different from data flow operators that the `get_input_data()` extracts a slice of the input tensor, the input data of the *u*Tasks in the `body_uTasks` of control flow is related to the results of the control flow. For example, in the LSTM model, the

```
1  interface LoopUTask: NestedUTask {
2      void compute();
3      void get_input_data();
4      void control_flow();
5      vector<uTask> body_uTasks;
6  };
```
(a) Loop-*u*Task

```
1  interface BranchUTask: NestedUTask {
2      void compute();
3      void get_input_data();
4      void control_flow();
5      vector<uTask> then_uTasks;
6      vector<uTask> else_uTasks;
7  };
```
(b) Branch-*u*Task

Figure 6: Control flow *u*Tasks

*u*Tasks in the body of the loop control flow require different values of the loop counter in different loop steps. Therefore, the `get_input_data()` for control flow should prepare the input data with consideration of the results of control flow. Note that different control flow operations have different data access patterns in the `body_uTasks`. We will discuss it in detail in the following.

According to Section 2, there are three types of control flow in DNN programs: loop, branch, and recursion. Therefore, COCKTAILER defines the concepts of loop-*u*Task, branch-*u*Task, and *u*Task reference correspondingly to represent the fine-grained *u*Task for control flow in DNN programs.

(1) Loop-uTask. Figure 6(a) shows the *u*Task definition for the loop control flow. COCKTAILER currently supports two types of loop control flow, i.e., for loop and while loop. The `control_flow()` interface implements the for loop or the while loop condition. The body computation of a loop represented in *u*Tasks is implemented in `body_uTasks`. Note that the `body_uTasks` is executed multiple times in a loop with different input data in each loop step. For example, in the LSTM model, the computation of a LSTM cell in each loop step requires the same model parameter tensors but different loop counter tensors and state tensors. The `get_input_data()` interface needs to prepare the corresponding tensors in each loop step.

(2) Branch-uTask. Figure 6(b) shows the *u*Task definition for the branch control flow. The `control_flow()` interface implements the condition computation in the branch. The branch-*u*Task has `then_uTasks` and `else_uTasks` to indicate the computation of two branches represented in *u*Tasks, respectively. The `get_input_data()` interface returns the required data for a branch indexed by the condition result.

(3) Function. A function can be natively represented

```
1  ScheduleOperator(op, D, unit_level, config);
2  ScheduleControlFlow(g, D, unit_level, config);
3  Config SetResource(D, unit_level, resource);
```

Figure 7: Scheduling interfaces

with NestedUTasks that each *u*Task represents the computation in the function body in the fine-grained *u*Tasks in the body_uTasks. The get_input_data() interface prepares input data tensors and The compute() interface executes the *u*Tasks in body_uTasks sequentially.

(4) Recursion and uTask reference. Functions can be represented with *u*Tasks. However, recursion is a special case in functions that a function may call itself in the function body. That is to say, a *u*Task may have itself in its body_uTasks. To support recursion, COCKTAILER introduces *u*Task reference to reference a *u*Task definition. The *u*Task reference can be considered as a function call to a *u*Task. The difference between a reference and a *u*Task is that the reference is a declaration for a *u*Task while a *u*Task defines the computation in a function. When executing a reference, COCKTAILER will find its *u*Task definition and execute this *u*Task.

The *u*Task abstraction in COCKTAILER is a general abstraction to represent control flow. We show how to represent loop, branch, function and recursion with *u*Task as most of current DNN models only contain these structures. More types of control flow can be represented by inheriting the NestedUTask.

***u*Program** The generated execution plan of the whole input DNN program is represented by a *u*Program. The *u*Program contains independent *u*Tasks, each of which is the compute logic scheduled to one processing unit of the unit_level of the accelerator. The *u*Tasks can be executed by compute, and the total *u*Task count of the *u*Program is reported by get_utask_num.

The *u*Task abstraction enables COCKTAILER to represent DNN programs with data flow operators and control flow in a fine granularity for accelerators with massive parallelism. This representation opens a new space for co-scheduling control flow and data flow.

3.2 *u*Program Scheduling

The *u*Task representation for DNN programs opens a large scheduling space for co-optimizing control flow and data flow in a single space. Instead of the pre-defined schedule in existing frameworks that executes data flow on the accelerator side while executes control flow on the CPU side, COCKTAILER chooses to explore this scheduling space at compile-time. To achieve this, COCKTAILER separates the scheduling policy from its mechanism. On the mechanism side, COCKTAILER provides *scheduling interfaces* with *scheduling constraints*. On the policy side, COCKTAILER provides a traverse-based *scheduling policy*. Note that the scheduling is generally designed for operators of *u*Task representation and can be executed automatically.

Scheduling interfaces. COCKTAILER provides three interfaces ScheduleOperator, ScheduleControlFlow and SetResource, to facilitate the scheduling process, as shown in Figure 7. Specifically, ScheduleOperator schedules an operator op, which can be either a data flow *u*Operator or a solely-scheduled control flow operation, into the target *u*Program with unit_level of the accelerator D. The config describes the current scheduling status including the target *u*Program and is initialized by the SetResource. ScheduleOperator will set the target *u*Program to NULL if it fails to schedule the *u*Operator. Similarly, ScheduleControlFlow schedules a control flow operation whose body has been scheduled to the required unit_level under the scheduling config, and returns NULL when failing to schedule this control flow. To ensure correctness, both schedule functions will add necessary barriers to enforce the desired *u*Task dependency. Moreover, as control flow should control the *u*Tasks in the body, a scheduling constraint is shown below,.

Constraint 1 *The unit_level of control flow should not be lower than the unit_level of data flow in the body.*

COCKTAILER also has a profiler that measures the execution time for a *u*Program. The profiled information could guide a policy on deciding whether to schedule a *u*Program to the unit_level of the accelerator.

Traverse-based bottom-up scheduling policy. Algorithm 1 describes a traverse-based scheduling policy to show how to use the interfaces and the profiler to schedule control flow and data flow in a single space to the accelerator side. This policy takes a DNN program g represented as control flow operations and *u*Operators in data flow and the accelerator D as input and returns a list of scheduled *u*Programs on this accelerator. The policy also accepts a unit_level parameter indicating the highest scheduled unit_level of the operators inside the graph g or NULL if the operators inside the graph are not scheduled yet, which is the initial case. If the input program has multiple operators, COCKTAILER will put these operators into a function operator before scheduling.

Initially, this policy schedules all the data flow *u*Operators to *u*Program (line 4 and line 21-27). The policy continues by progressively trying to schedule more parts of the program to the same *u*Program if the profiler suggests this schedule could reduce the overall execution time (line 5-27). Specifically, the policy will recursively traverse the program (line 7) until it only contains a *u*Operator (line 3-4) and schedule it to the *u*Program via ScheduleProgram which achieves this via ScheduleOperator. During the traverse, if all the operations in the input program are scheduled to accelerator's units (line 21), the policy will try to schedule this program (i.e., the control flow) to the *u*Program (line 21-27) via ScheduleProgram. ScheduleProgram implements scheduling an input program g to a unit_level of the accelerator D. Note that the input g is either a graph of operators

Algorithm 1: Traverse-based Scheduling Policy

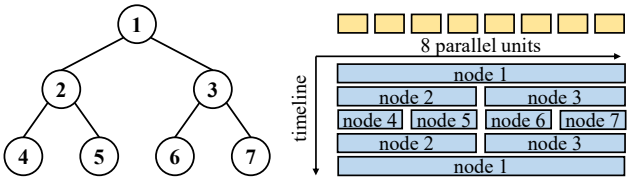
```
   Data: g: DNN program represented with uOperator; D: accelerator
   Result: uProgram
1  Function Schedule(g, D, unit_level = NULL):
2      ulevel=unit_level, ulevel_max=D.unit_levels.size()-1, uProgs=[];
3      if g ∈ D.Operators and ulevel is NULL then
4          ulevel = 0;
5      if ulevel is NULL then
6          for op ∈ g.TopoSort() do
7              g_op, ulevel_op = Schedule(op, D, NULL);
8              ulevel = max(ulevel, ulevel_op);
9              uProgram_p = uProgs[-1];
10             ulevel_m = max(ulevel_op, uProgram_p.ulevel);
11             if ulevel_m < ulevel_max then
12                 g_merge = uProgram_p.g + g_op;
13                 g_merge, ulevel_merge = Schedule(g_merge, D, ulevel_m);
14                 if ulevel_merge < ulevel_max then
15                     uProgs[-1] = g_merge.uProgs[0];
16                     ulevel = max(ulevel, ulevel_merge);
17                     continue;
18             uProgs.append(g_op.uProgs);
19     else
20         uProgs = g.uProgs
21     if ulevel < ulevel_max then
22         for ulevel_cur ∈ range(ulevel, ulevel_max) do
23             uProgram_cur = ScheduleProgram(g, D, ulevel_cur);
24             if uProgram_cur is not NULL then
25                 if Latency(uProgram_cur) < Latency(uProgs) then
26                     uProgs = [uProgram_cur];
27                     ulevel = ulevel_cur;
28     g.uProgs = uProgs;
29     return g, ulevel;
30 Function ScheduleProgram(g, D, unit_level):
31     // g is a graph of operators in uTask representation or a control
         flow operation that the body has been scheduled
32     resource = GetResource(D, unit_level); // calculate resource
33     cfg = SetResource(D, unit_level, resource);
34     if g ∈ D.ControlFlow then
35         return ScheduleControlFlow(g, D, unit_level, cfg);
36     else
37         for op ∈ g.TopoSort() do
38             ScheduleOperator(op, D, unit_level, cfg);
39         return cfg.uProg;
```

in *u*Task representation (including *u*Operators and scheduled control flow) or a control flow whose body has been scheduled as *u*Program. Therefore, `ScheduleProgram` calls `SetResource` to configure the scheduling and leverages the `config` to schedule the program with `ScheduleOperator` and `ScheduleControlFlow`. The `unit_level` is maintained as Constraint 1 during scheduling.

Several optimizations can be employed to reduce the scheduling time. For conciseness, these optimizations are not explicitly shown in the pseudo code. For example, trials on different `unit_levels` (line 22) can be performed in parallel.

Scheduling optimizations There are three optimization opportunities during the scheduling, depending on the inputs and the DNN programs.

Function inline. To remove function call overhead, COCKTAILER converts a function control flow without recursion to a sequence of computation. It removes the function control flow boundary and applies DNN optimizations to a larger program scope.



(a) A tree structure recursion (b) Schedule

Figure 8: Parallel execution of recursive calls

Loop unroll. COCKTAILER unrolls the loop control flow with some steps to explore more optimization opportunities. For example, unrolling the loops in a multi-layer RNN model can expose parallelism between RNN cells. Loop unrolling is applied during scheduling and is evaluated to decide whether to enable this unroll.

Recursion unroll. It is similar to loop unroll in that the recursion is also able to be unrolled to explicitly expose the recursion tree structure. COCKTAILER applies this optimization to DNN programs to unroll the recursion structure several times to expose more optimization opportunities. For example, the unrolled recursion tree can naturally expose parallelism between recursive calls, which can be leveraged for concurrent execution. Figure 8 shows an example of recursion unroll. By unrolling the recursive calls, computation without dependencies (e.g., nodes 2 and 3 in Figure 8) can be executed concurrently. Recursion unroll is applied during scheduling. The scheduler will evaluate the unrolled results to decide whether to enable this unroll and schedule the unrolled body to different computation units.

These optimizations are in `ScheduleControlFlow`. The scheduler will try to enable these optimizations and evaluate the performance with some sample data to decide whether to enable optimizations or not.

4 Implementation

COCKTAILER is implemented by about 10000 lines of code including Python and C++ on top of PyTorch [36] and Rammer [28]. COCKTAILER does not require any effort from model developers, who can still work on a native PyTorch program. COCKTAILER first exports the PyTorch program to an ONNX graph with built-in loop and branch operators and an extended invoke operator for representing recursion. With the converted ONNX graph, COCKTAILER automatically performs the scheduling of data flow and control flow, and applies control-flow-related optimizations described in §3. Then, COCKTAILER wraps the generated code as a customized PyTorch operator and replaces the PyTorch program with a call to this operator.

We implemented COCKTAILER for NVIDIA GPUs and AMD GPUs because they are the most popular accelerators

```
1  // y = matmul(x, w); out = tanh(y);
2  __device__ void UProg<NumUTask=2>(float* x,
     float* w => float* out | char* tmp,int id) {
3   if (id == 0) {
4    float *y = (float*)tmp;
5    MatmulUOp.compute(x, w => y, id=0);
6    MatmulUOp.compute(x, w => y, id=2);
7    Barrier(blocks={0,1});
8    TanhUOp.compute(y => out, id=0);
9   } else if (id == 1) {
10   float *y = (float*)tmp;
11   MatmulUOp.compute(x, w => y, id=1);
12   MatmulUOp.compute(x, w => y, id=3);
13   Barrier(blocks={0,1});
14   TanhUOp.compute(y => out, id=1);
15  }
16 }
```

Figure 9: Example of *u*Task

```
1  for i in range(10):
2      inpi = inp[i]
3      xi = matmul(inpi, wx)
4      h = tanh(xi + h)
```

(a) A simplified RNN model

```
1  __device__ void LoopUProg(float* inp, float* wx
     , float* h_in => float* h_out | float* tmp) {
2   float *inpi = tmp, *xi = tmp + 1024;
3   CopyUOp(h_in => h_out); Barrier();
4   for (int i = 0; i < 10; i++) {
5    GatherUOp(inp, &i => inpi); Barrier();
6    MatmulUOp(inpi, wx => xi); Barrier();
7    AddTanhUOp(xi, h_out => h_out); Barrier();
8   }
9  }
```

(b) Loop-*u*Task for the RNN model

Figure 10: Example of Loop-*u*Task

for DNNs. In the rest of this section, we describe the details about implementing COCKTAILER for NVIDIA CUDA GPUs, and briefly describe our implementation on AMD GPUs. COCKTAILER can be ported to other accelerators if they align with the hardware abstraction described in §3 and expose APIs to control the units (e.g., Graphcore IPUs).

4.1 COCKTAILER on NVIDIA CUDA GPUs

As described in §3, an NVIDIA GPU can be abstracted as a 3-level hardware. COCKTAILER implements the `ScheduleOperator` interface on top of Rammer [28], AutoTVM [6], Ansor [53], Roller [57], and manually-implemented kernels. Specifically, COCKTAILER first obtains the source code of each dataflow operator on the given `unit_level` by choosing from existing manual implementations of simple operators like element-wise ones or by tuning the operator with AutoTVM, Ansor, or Roller. COCKTAILER then leverages Rammer to covert the data flow operators' kernel source code to a *u*Operator with multiple *u*Tasks. After that, COCKTAILER schedules the program and generates the kernel code for the control flow body.

4.1.1 Code Generation for Nested-*u*Task

Overall structure A list of *u*Operators inside a function will be scheduled to a *u*Program with multiple Nested-*u*Tasks. It will be converted to a function with pointers to the related tensors. Specifically, we use `(A => B | C)` to represent a function with tensor A as input, tensor B as output, and tensor C as a buffer saving intermediate results. The function also accepts a `uTask_id` parameter for indexing the *u*Tasks in the *u*Program. Figure 9 provides an example Function-*u*Program with a `matmul` *u*Operator implemented by 4 *u*Tasks and a `tanh` *u*Operator implemented by 2 *u*Tasks. This Function-*u*Program contains 2 *u*Tasks, each of which contains 2 `matmul` *u*Tasks and 1 `tanh` *u*Tasks in the `body_uTasks` with proper barrier inserted (line 5-8, 11-14). The barrier can be implemented by using CUDA Cooperative Groups [1] or extending a lock-free GPU synchronization technique [48].

The Function-*u*Program allocates the storage for Tensor `y` (line 4,10) and wraps the code with function name and signature (line 2). The `__device__` function qualifier is used so that this function can be called by other *u*Tasks. We will omit the `uTask_id` in the following sections and only show the generated code of one *u*Task inside the *u*Operator for brevity.

Block alignment One challenge of scheduling multiple DNN operators into a single GPU kernel comes from the variance of thread count inside each GPU block (`blockDim`). The `blockDim` of the kernel for a *u*Program have to be set to the maximum `blockDim` of its *u*Operators, so that kernels with a large number of GPU blocks (`gridDim`) and small `blockDim` will execute inefficiently when they are scheduled into the same kernel of an operator with large `blockDim`. To address this problem, we re-implement the *u*Operators with configurable `blockDim` if possible (e.g., element-wise ones, reduction, and transpose). During schedule, COCKTAILER collects the fastest kernel of *u*Operators with predefined `blockDim` (e.g., matmul and convolution), and configure the `blockDim` of configurable *u*Operators to the maximum `blockDim` of the collected *u*Operators. If the `blockDim` of the collected *u*Operators varies greatly, COCKTAILER will leverage an extended Roller [57] to re-generate kernels with a fixed `blockDim`.

Register pressure The generated long-running GPU kernel may face register pressure. To alleviate this problem, COCKTAILER uses the profiling in §3.2 to detect performance drop due to register overuse and stop enlarging the current kernel. For control flow graph with no back edges, COCKTAILER can also utilize the branch reclustering technique in §4.1.3 to both schedule the control flow to the accelerator side and reduce the kernel size.

4.1.2 Code Generation for Loop-*u*Task

Overall structure Figure 10(a) shows a simplified RNN model. It is scheduled to a Loop-*u*Program with several Loop-*u*Tasks. Each Loop-*u*Task in the *u*Program contains `body_uTasks` from three types of *u*Operators, i.e., gather,

```
1  if (cond):
2      tmp1 = matmul(x, w1)
3      y = sigmoid(tmp1)
4      z = conv(y, w2)
5  else:
6      z = x + b
```

(a) A DNN model with branch

```
1  __device__ void BranchUProg(bool* cond, float*
     x, float* w1, float* w2, float* b, float*
     y_in => float* y_out, float* z_out | float*
     tmp) {
2   if (*cond) {
3    float *tmp1 = tmp;
4    MatmulUOp(x, w1 => tmp1); Barrier();
5    SigmoidUOp(tmp1 => y_out); Barrier();
6    ConvUOp(y_out, w2 => z_out);
7   } else {
8    AddUOp(x, b => z_out); // no Barrier
9    CopyUOp(y_in => y_out);
10  }
11 }
```

(b) Branch-*u*Task for the DNN model

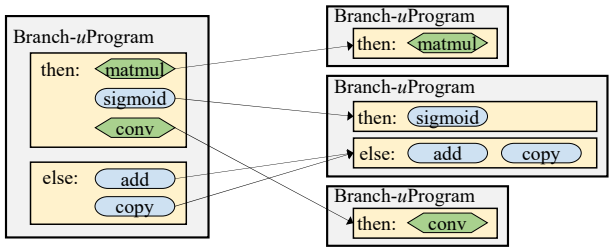
Figure 11: Example of Branch-*u*Task

matmul, and fused add-tanh operations. This Loop-*u*Program takes three input tensors named `inp`, `wx`, `h` (`h_in` in Figure 10(b)), and produces an updated tensor `h` (`h_out` in Figure 10(b)). The generated code of each Loop-*u*Task contains a loop (line 4) and the *u*Tasks (line 5-7) separated by barriers for synchronization across GPU blocks.

Memory management Different from existing DNN frameworks that allocate tensors at runtime, COCKTAILER needs to statically allocate tensor memory to execute the control flow operations on GPUs. The variables in the loop body can be divided into four categories: (1) constants (`wx`, `inp`); (2) intermediate results (`inpi`, `xi`); (3) iteration count (`i`); (4) loop-carried dependencies (`h`). All these variables are represented by pointers to the corresponding pre-allocated tensors and can be obtained from `get_input_data`. Specifically, the pointer to the constants are the corresponding function inputs, the intermediate results are allocated from a `tmp` buffer (line 2 in Figure 10(b)), and the pointer to the iteration count is `&i`. The pointers to the loop-carried dependencies are a little complex because the variable exists in both input tensors and output tensors of the Loop-*u*Operator. First, some `CopyUOperators` are inserted to copy the input tensors (`h_in`) to the corresponding output tensors (`h_out`). Then, the `body_uTasks` is generated via only visiting the output tensors. Additional `CopyUOperators` and dependencies between *u*Operators in the loop body are added to ensure the correctness of the overlapped input and output tensors.

4.1.3 Code Generation for Branch-*u*Task

Overall structure Figure 11(a) contains a DNN model with two branches, The then branch takes tensors `x`, `w1`, and `w2` as inputs and produces tensors `y` and `z`; The else branch takes tensors `x` and `b` as inputs and produces tensor `z`. The input of the generated Branch-*u*Program is the union of inputs of



(a) Single kernel (b) Branch reclustering

Figure 12: Optimize Branch-*u*Program by branch reclustering

the two branches as well as the `cond` tensor. The output is the union of the outputs of the two branches. If an output only exists in one branch, `CopyUOperators` will be added to the other branch to move the corresponding old value to the output tensor (line 9 of Figure 11(b)). The intermediate results are saved in tensors allocated from the `tmp` buffer. As only one branch may be executed in each run, the intermediate results of the two branches can use the same memory space.

Branch reclustering Scheduling a whole ControlFlow-*u*Program to a single GPU kernel is not always the best choice because different operations prefer different GPU occupancy (number of threads concurrently executed on an SM). For example, `matmul` uses a large amount of shared memory and registers for saving the tiles, resulting in limited occupancy, while element-wise operations prefer large occupancy to improve memory bandwidth. COCKTAILER also tries to schedule a Branch-*u*Program to multiple Branch-*u*Programs with each Branch-*u*Program containing *u*Operators with similar preferred occupancy and keeps the execution of branch condition on the GPU. The example model in Figure 12 contains limited-occupancy-*u*Operators `matmul` and `conv` (in green) and large-occupancy-*u*Operators `sigmoid`, `add`, and `copy` (in blue). These *u*Operators are scheduled into three Branch-*u*Programs for limited occupancy, large occupancy, and limited occupancy, respectively. The two branches are co-scheduled so each GPU kernel can contain *u*Operators from both branches. This branch reclustering technique reduces the kernel size, thus can also alleviate register pressure of large GPU kernels.

4.1.4 Code Generation for *u*Task Reference

Overall structure *u*Task reference is a special case that calls a *u*Task defined in another *u*Program. It is designed for recursions where a function may call its callers like Figure 13(a). To support recursion, the function declarations of all *u*Programs whose *u*Tasks are referenced by *u*Task references are generated at the start of the code (line 1 of Figure 13(b)). Then, all *u*Programs generate their function definitions. The maximum stack depth of our recursion implementation cannot be increased at runtime, so users need to manually set a limit to the stack depth, or COCKTAILER will use all free memory to save the intermediate results in the call stack. The base

```
def Recursion(l, r, is_leaf, inp, w, root):
 cond = is_leaf[root]
 if cond:
  output = inp[root]
 else:
  a = Recursion(l, r, is_leaf, inp, w, l[root])
  b = Recursion(l, r, is_leaf, inp, w, r[root])
  c = a + b
  output = matmul(c, w)
 return output
```

(a) A recursive model

```
__device__ void RecursionUProg(float* l, float*
    r, bool* is_leaf, float* inp, float* w, int*
    root => float* output | char* tmp);
__device__ void BranchUProg(float* cond, float*
    l, float* r, bool* is_leaf, float* inp,
   float* w, int* root => float* output | char*
   tmp) {
 if (*cond) {
  GatherUOp(inp, root => output);
 } else {
  float *a = tmp, *b = tmp+256, *c = tmp+512;
  RecursionUProg(l, r, is_leaf, inp, w, l + (*
    root) => a | tmp + 768); Barrier();
  RecursionUProg(l, r, is_leaf, inp, w, r + (*
    root) => b | tmp + 768); Barrier();
  AddUOp(a, b => c); Barrier();
  MatmulUOp(c, w => output);
 }
}
__device__ void RecursionUProg(float* l, float*
    r, bool* is_leaf, float* inp, float* w, int*
    root => float* output | char* tmp) {
 float *cond = tmp;
 GatherUOp(is_leaf, root => cond); Barrier();
 BranchUProg(cond, l, r, is_leaf, inp, w, root
    => output | tmp + 256);
}
```

(b) The generated code

Figure 13: Example of recursion with *u*Task reference

case check is kept in the function body as a branch operation.

Simulation of GPU stack Though NVIDIA GPUs have the built-in support of recursion, the stack is slow and with very limited supported depth. The reason is that GPU needs to save the registers of all threads during function calls. However, in a DNN program, we only need to save the pointers to tensors and the program counter of the current stack frame before performing a function call. Moreover, the same set of tensor pointers are shared by multiple *u*Tasks, and only a single copy needs to be saved. Therefore, we have the opportunity to reduce the size of saved information to both increase the stack depth and reduce the time for saving the stack frame.

To achieve this, COCKTAILER implements a stack in global memory to simulate the function call behavior. As it is dangerous to directly update the program counter, COCKTAILER choose to inline all *u*Programs to a single function and use "`goto`" together with "labels" inserted into the inlined function to simulate the update of the program counter. The labels are placed at the start of the function and at the end of each function call inside the function. Instead of maintaining program counters, the stack saves the label of each stack frame. Each stack frame only consumes tens of bytes of memory, so COCKTAILER can also save the stack in GPU shared memory to avoid the memory fence and inter-block barrier for maintaining a synchronized stack across different *u*Tasks when possible.

Model	Input shape	Description
LSTM	64, BS, 256	hidden 256, length 64, layer 10
NASRNN	1000, BS, 256	hidden 256, length 1000, layer 1
Attention	BS, 12, 64, 64	head 12, hidden 768, length 64
Seq2seq	BS, 256	hidden 256, embed 3797 × 256, max length: 50 dataset: tatoeba-eng-fra
BlockDrop	BS, 3, 32, 32	drop layers from ResNet-32, dataset: CIFAR-10
SkipNet	BS, 3, 224, 224	drop layers from ResNet-101, dataset: ImageNet
RAE	127, 512	hidden 512, dataset: Stanford Sentiment Treebank

Table 1: Model configurations. BS refers to "batch size".

4.2 COCKTAILER on AMD ROCm GPUs

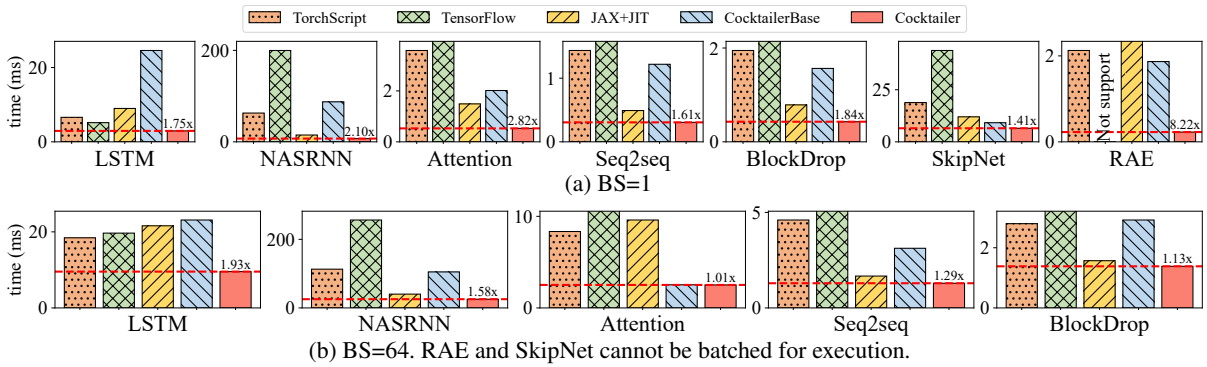
AMD ROCm GPUs provide a HIP programming model [2], which is similar to CUDA and is compatible with most CUDA statements. Besides, AMD provides a hipify tool to convert a CUDA kernel to a HIP kernel. COCKTAILER first generates a CUDA kernel and then leverages the hipify tool to convert it to the HIP version. Some *u*Operators are re-implemented due to the difference between CUDA and ROCm architectures.

5 Evaluation

Platform Our evaluation is on two accelerators: (1) NVIDIA Tesla V100-PCIE-32GB GPU with 2 Intel Xeon 5218 CPUs. The compiler is CUDA 11.5. (2) AMD Instinct MI100 GPU with 2 Intel Xeon 6338 CPUs. The compiler is ROCM 4.3.

Baselines We compare COCKTAILER with representative state-of-the-art deep learning frameworks including the most popular imperative framework PyTorch [36] v1.11 for CUDA and v1.10 for ROCM with TorchScript [3] enabled, the representative DAG-based framework TensorFlow v1.15 [4], and JAX v0.3.20 [11] with just-in-time compilation (JIT) enabled. ROCM 5.3 is used in JAX due to compatibility problems. Note that the latest TensorFlow 2 is redesigned as an imperative framework like PyTorch and JAX, therefore we choose TensorFlow v1.15 to evaluate the DAG-based framework. We also create a baseline that accelerates each basic block of the DNN program with Rammer [28] and relies on PyTorch for executing the control flow operations (COCKTAILERBASE). COCKTAILERBASE uses the same kernel implementation of each operator and the same compilation passes excluding the control-flow-related ones as COCKTAILER.

Benchmarks Our evaluation includes a set of representative DNN models that covers typical architectures like CNN, RNN, and transformers, different application domains including CV, NLP, and speech, and different types of control flow operations including loops, branches, and recursions. LSTM [16] is a representative RNN model for NLP and speech, and has been manually optimized by both deep learning frameworks and libraries. We use the built-in LSTM operators when possible, which are linked to the manually optimized LSTM im-



(a) BS=1

(b) BS=64. RAE and SkipNet cannot be batched for execution.

Figure 14: End-to-end DNN inference on NVIDIA V100 GPU

plementation in vendor libraries like cuDNN. NASRNN [58] is another RNN-based model created by network architecture search (NAS) that has not been manually optimized yet. Attention [43] is a widely used architecture in NLP and CV. We use an auto-regressive attention mechanism to continue sentences. The above three models contain loops with fixed iteration counts. Seq2seq [40] is a sequence-generation model that contains a while loop for continuously generating new tokens until an end-of-sequence (EOS) token is emitted or the maximum sequence length limit is reached. BlockDrop [47] and SkipNet [46] are two CNN-based CV models with branches for skipping some layers. Recursive Autoencoder (RAE) [39] is a well-known recursive model for NLP. The configuration of these models is listed in Table 1.

We set the batch size (BS) of the experiments to 1 and 64 to match the requirements for online inference and offline inference. The time is measured by averaging 100 tests after 100 warm-up runs. For models using real datasets, we randomly sample 100 $\times$ BS cases from the datasets.

5.1 End-to-end Evaluation on NVIDIA GPU

Figure 14 shows the inference performance of COCKTAILER by comparing with TorchScript, TensorFlow, and JAX with JIT enabled. All three frameworks support control flow operations by executing them on CPU. Overall, COCKTAILER outperforms the best baseline in each model by 1.85$\times$ in geometric mean (up to 8.22$\times$). Specifically, COCKTAILER outperforms TorchScript by 3.98$\times$ on average (up to 9.35$\times$), TensorFlow by 18.45$\times$ on average (up to 196.85$\times$), and JAX by 3.05$\times$ on average (up to 327.62$\times$). The time for compiling each model (except kernel tuning by AutoTVM and Ansor) is several seconds to minutes.

Models with loops LSTM has been manually optimized by many frameworks and vendor libraries, and we use the fastest built-in implementation in the baselines. The core control flow operations of LSTM are two loops iterating over the input sequence and the layers respectively. TensorFlow and TorchScript use the manually-optimized LSTM in cuDNN library, while JAX loops over manually-optimized LSTM cell implementation. According to profiling, TensorFlow uses the persistent-RNN [8] to optimize the loop over the input sequence, but it does not accelerate the loop iterating over the layers. TensorFlow with BS=64, TorchScript, and JAX only optimizes the operators in one LSTM cell, and does not perform joint optimizations on LSTM cells in different iterations. Different from these systems, COCKTAILER fully unrolls the static loop over layers and unrolls some steps of the loop over inputs, so that it can expose a large set of operators to the data flow optimization passes and benefit from the inter-operator schedule of Rammer. COCKTAILER outperforms all framework with handly-optimized implementations by 1.75$\times$ when BS=1 and 1.93$\times$ when BS=64.

The computation of NASRNN and Attention has not been manually optimized. These frameworks optimize the basic block using only passes for compiling static data flow, and execute the loop on CPU. COCKTAILER performs some loop optimizations and schedules the loop to thread block level. With such optimizations, COCKTAILER achieves 2.10$\times$ on NASRNN model and 2.82$\times$ speedup on Attention model over the fastest baseline when BS=1. However, COCKTAILER only achieves 1.01$\times$ speedup over COCKTAILERBASE on Attention BS=64 because control flow only take a small portion of execution time when the body computation is large enough.

Seq2seq is implemented with a while loop, and existing frameworks need to copy the decision from the accelerator to the CPU to decide whether to continue the loop. By executing the loop on GPU, COCKTAILER can both use fewer kernels and avoid such synchronization. The speedup over the fastest baseline is 1.61$\times$ and 1.29$\times$ when BS=1 and BS=64, respectively.

Models with branches BlockDrop and SkipNet drop some layers from ResNet with decisions generated at runtime. The baselines need to copy the decision from GPU to CPU to decide whether to launch the next layer. COCKTAILER avoids such synchronized copy by scheduling the branch to block level for BlockDrop BS=1, and using branch reclustering for BlockDrop BS=64 and SkipNet BS=1. COCKTAILER accelerates BlockDrop by 1.84$\times$ and 1.13$\times$ over the best baseline when BS=1 and BS=64, respectively, and accelerates SkipNet by 1.41$\times$.

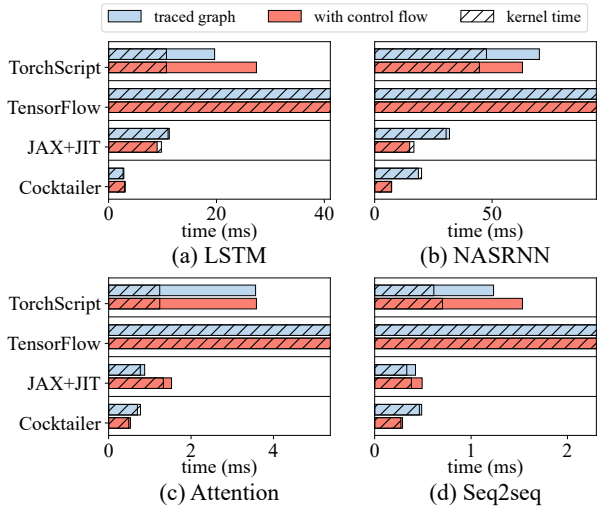


Figure 15: Control flow overhead of models with loops.

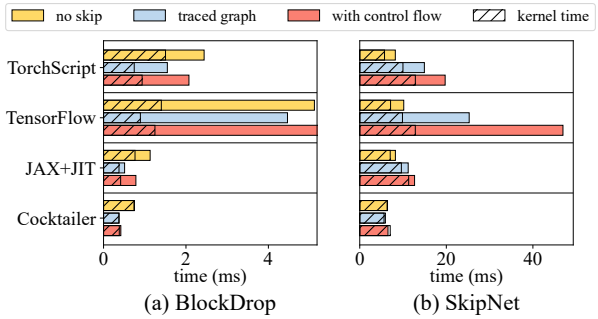


Figure 16: Control flow overhead of models with branches. "No skip" refers to running all layers of the ResNet model.

Model with recursion RAE is a recursive model. TensorFlow does not support recursion. PyTorch and JAX can only run this model in Python, resulting in poor performance. COCKTAILER schedules the recursion to block level with parallel execution and executes the recursive calls efficiently with the simulated stack, resulting in 9.35×, 327.62×, and 8.22× speedup over PyTorch, JAX, and COCKTAILERBASE respectively.

Discussion Whether a model is control flow bound or data flow bound depends on the ratio of control flow computation and data flow computation. According to the evaluation among different models in Figure 14, it is clear that COCKTAILER can achieve higher speedup when model execution has more control flow computation, e.g., NASRNN, RAE. When the data flow occupies the most computation (e.g., Attention in BS=64), COCKTAILER can achieve similar performance with the fastest baseline.

5.2 Control Flow Overhead Analysis

In this section, we evaluate the performance degradation caused by control flow boundary in different systems when BS=1. The results are shown in Figure 15, 16, and 18. For each model, we choose an input with a typical execution trace of the dataset. We compare the real scenario that executes control flow at runtime to executing the traced computation graph with no control flow to evaluate the overhead. The traced graph baseline of COCKTAILER is compiled by Rammer with the same kernel implementations and compilation passes for data flow as COCKTAILER.

Models with loops Figure 15 shows the control flow overhead of models with loops. The input data of LSTM, NASRNN, and Attention is a sequence with length provided in Table 1, and the input to Seq2seq generates a 10-token-sequence which is near to the average sequence length of the dataset.

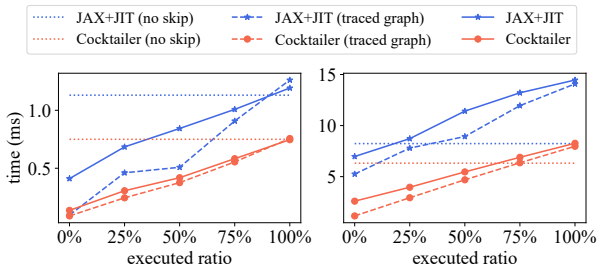
For LSTM model, Rammer can explore the parallelism of cells in different steps if all steps are unrolled. The dynamic unrolling of COCKTAILER provides similar performance with Rammer, but can support dynamic step count. Other systems do not explore such parallelism and are slower than COCKTAILER. To expose the loop of TorchScript and TensorFlow, we do not use their cuDNN LSTM here.

For NASRNN, Attention, and Seq2seq models, COCKTAILER schedules the loop to thread block level with only one GPU kernel, and is faster than Rammer which uses a larger number of kernels. A similar phenomenon also appeared in the NASRNN model with JAX. JAX generates thousands of different kernels for execution the unrolled loop and is slower than looping over the NASRNN cell with 3 kernels for 1000 times. This indicates that an efficient implementation of control flow can sometimes be faster than running the unrolled data flow.

For Seq2seq model, TorchScript, TensorFlow, and JAX need to copy the decision back to CPU to decide whether to execute the next iteration of the while loop, causing a synchronization between CPU and GPU. Therefore, when control flow is used, the increase of execution time is larger than that of kernel time. COCKTAILER does not have such a problem because all control flow operations are executed on GPU.

Models with branches Figure 16 shows the control flow overhead of models with branches. The two models skip some layers from a ResNet model, and we add a "no skip" which is a normal ResNet without skipping layers. The ratios of executed layers are 7/15 for BlockDrop and 23/33 for SkipNet, which are similar to the average ratio of the models respectively.

Due to the synchronization between CPU and GPU, the control flow operations of the baselines increase the execution time by at least 34% over the traced version for BlockDrop, while COCKTAILER only increases the execution time by 11%. Therefore, though more than half of the layers are skipped, the performance improvement of layer skipping compared with the original ResNet model is only at most 1.44× in the baselines, while COCKTAILER achieves 1.79× speedup. In SkipNet, the network for making the skip decision is heavier and the ratio of executed layers is larger, so the traced graph may take longer execution time than the original ResNet model. The slow execution of control flow makes the performance of



(a) BlockDrop (b) SkipNet

Figure 17: Different ratio of executed layers

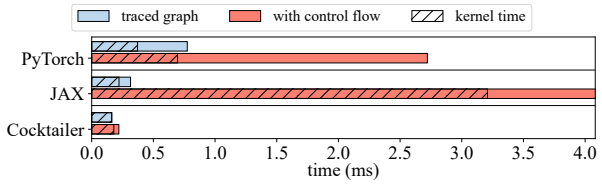


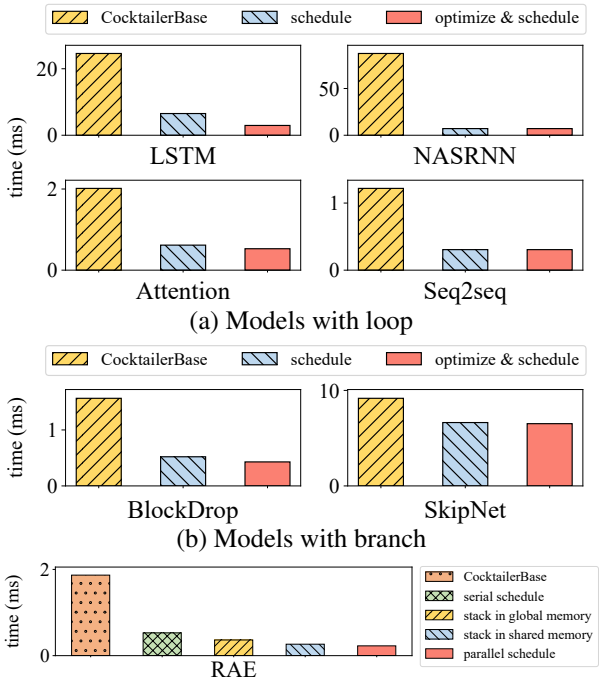
Figure 18: Control flow overhead of RAE with recursion.

this model even worse in baselines, while COCKTAILER can still provide reasonable performance.

Figure 17 shows the performance of BlockDrop and SkipNet at different ratios of executed layers. The results of JAX, the fastest baseline of the two models are also included. When the executed ratio is 0, the model executes all control flow operations but runs no layers, and COCKTAILER achieves 3.00× and 2.68× speedup over JAX on BlockDrop and SkipNet, respectively. This proves the low control flow overhead of COCKTAILER. In SkipNet, if the model is executed with JAX, the layer-skipping can improve the performance only when the ratio of executed layers is lower than 20%, while if executed with COCKTAILER, this ratio becomes about 65%.

Model with recursion Figure 18 shows the control flow overhead of the recursive RAE model. The input is a 65-node tree from the Stanford Sentiment Treebank dataset. PyTorch and JAX can only execute the recursion in Python and the time is much longer than executing the traced graph. Rammer processes nodes without dependencies in parallel with a static schedule that only works for this tree, while COCKTAILER executes the model by control flow operations on the GPU side and only increases the time by 11%.

Discussion Compared with the traced graph baseline which removes all the control flow operations in the models and can be considered as the optimal status, COCKTAILER achieves similar performance. Besides, the overall latency of COCKTAILER is similar to the kernel time, which indicates that COCKTAILER can minimize the overheads introduced by control flow. Furthermore, the evaluations on BlockDrop and SkipNet show that COCKTAILER also enables scenarios like efficient computation by achieving real speedup. We hope COCKTAILER can provide more flexibility for algorithm researchers to design DNN architectures with control flow.



(a) Models with loop

(b) Models with branch

(c) Model with recursion

Figure 19: Breakdown of models with BS=1

5.3 Breakdown of Optimizations

Figure 19(a) provides the breakdown of optimizations applied on models with loops. On average, scheduling the loop to block level provides 4.95× speedup over COCKTAILERBASE that executes the loop in PyTorch runtime. And applying the optimizations in §3.2, especially the dynamic loop unrolling further improves the performance of LSTM by 2.22× and Attention by 1.17×. In LSTM, the loop is re-scheduled to kernel level after loop unrolling.

Figure 19(b) provides the breakdown for models with branches. The branches of the two models are executed on GPU, with branch reclustering used in SkipNet. The scheduling provides 3.01× and 1.38× speedup over COCKTAILERBASE on BlockDrop and SkipNet, and the optimizations further accelerate the two models by 1.21× and 1.02×.

Figure 19(c) shows the performance of the RAE model. Executing the recursion on GPU provides 3.54× speedup over COCKTAILERBASE. The simulation of stack using global memory and shared memory are 1.45× and 1.99× faster than using the built-in GPU stack. And the parallel scheduling of *u*Programs further improves the performance by 1.17×.

5.4 End-to-end Evaluation on AMD GPU

Figure 20 compares TorchScript, TensorFlow, JAX with JIT enabled and COCKTAILER on AMD MI100 GPU with BS=1. COCKTAILER outperforms the three frameworks on all benchmarks by 2.97× over TorchScript on average (up to 5.86×), 21.28× over TensorFlow on average (up to 112.34×), and 3.22× over JAX on average (up to 272.63×).

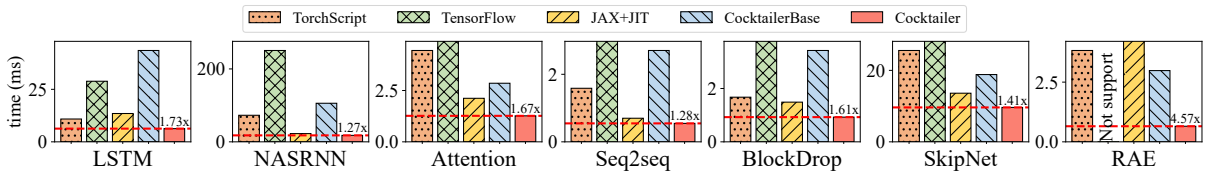


Figure 20: End-to-end DNN inference on AMD MI100 GPU with BS=1

6 Related Work

Supporting control flow in deep learning can be divided into two categories. The first one, represented by TensorFlow 1.x [4] and TorchScript [36], executes control flow operations in the framework runtime on CPU. Control flow is implemented as special operators (`NextIteration` for loops [52], `Switch` for branches [52], and `InvokeOp` for recursions [19]) or instructions in the runtime. The second one, represented by Chainer [42], PyTorch [36], and JAX [11], leverages the runtime of general-purpose language like Python to support the control flow operations. The control flow operations are expressed with Python statements and executed by the Python interpreter. AutoGraph [31], Janus [18], and Terra [22] show that the control flow operations expressed by general-purpose languages can sometimes be converted to the control flow operators in the framework runtime. Despite different ways of supporting control flow, the control flow operations in these works can only be executed by CPU.

Some special forms of control flow have been deeply optimized. VersaPipe [55] optimized pipelines for general GPU programs. Cortex [10] provides interfaces to describe recursion with data patterns (i.e., the recursion tree structure). It assumes that the jump direction of all control flow only depends on the input recursion tree structure, so it does not apply to control flow depending on dynamically computed data, e.g., the while loop with unknown iteration count in Seq2seq [40], and the branches whose direction is decided at runtime in BlockDrop [47] and SkipNet [46]. COCKTAILER does not assume the availability of such tree structures and works on these models.

Past works on batching (e.g., DyNet [33], Cavs [49], Tensorflow Fold [27], BatchMaker [12], Program-counter-autobatching [37], and ORCA [51]) enable the parallelization in different control flow operations by introducing a scheduler to batch the ready-to-execute operators, which is another applicable approach and is complementary to COCKTAILER. Specifically, COCKTAILER can compile subgraphs of a model, and then batching can be applied to these subgraphs. Applying batching on the more coarse-grained subgraph granularity can also reduce the scheduling cost in the batching scheduling.

There are many deep learning compilers for optimizing a computation graph without control flow, including TVM [6], TASO [20], Rammer [28], DNNFusion [35], PET [44], and AStitch [56]. These optimizations are compatible with COCKTAILER. COCKTAILER even enlarges their optimization scope because the boundary of control flow has been reduced. Compilation optimizations like function inline [5], loop unroll [9] have been introduced in general-purpose language compilers on CPU programs and have been implemented in COCKTAILER. COCKTAILER further introduces the new *u*Task abstraction to represent both data flow and control flow operations, which aligns with the parallelism of hardware accelerators, enabling analyzing and optimizing both data flow and control flow computation over heterogeneous accelerators (i.e., GPU). To scale DNN models on distributed architectures, frameworks and compilers like Tofu [45], FlexFlow [21], GSPMD [50], PipeDream [32], Tutel [17], FasterMOE [14], FlexMoE [34], BaGuaLu [29], Alpa [54] and SuperScaler [25] parallelize the execution of deep learning models across multiple hardware devices, but only focus on models with static architectures or specific types of dynamic models (e.g., Mixture-of-Experts [30]). COCKTAILER exposes the parallelism of control flow operations, which can be leveraged to support dynamic models over distributed devices.

7 Conclusion

DNN frameworks and compilers suffer from performance issues when supporting sophisticated dynamic DNN models. The parallelism mismatch between control flow and data flow results in separate execution of DNNs on the CPU and accelerator, causing not only overheads but also missed optimization opportunities. COCKTAILER supports sophisticated DNN models by co-scheduling the execution of control flow and data flow that (1) provides the fine-grained *u*Task abstraction for control flow and data flow in DNN programs to open a holistic scheduling space on hardware accelerators; (2) designs the scheduling mechanism and a heuristic policy to exploit this scheduling space; (3) provides control flow optimizations in both scheduling and code generation. Evaluations demonstrate that COCKTAILER significantly outperforms state-of-the-arts on sophisticated DNN models. By enabling the co-optimizing of control flow and data flow in a single space, COCKTAILER positions itself as a new enhancement to the deep learning infrastructure.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Prof. Wenjun Hu, for their extensive suggestions. This work is partially supported by National Key R&D Program of China under Grant 2021ZD0110104, National Natural Science Foundation of China (62225206).

References

[1] Cooperative Groups. https://devblogs.nvidia.com/cooperative-groups/.

[2] HIP Programming Guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html.

[3] TorchScript. https://pytorch.org/docs/stable/jit.html.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. USENIX Association.

[5] Pohua P Chang and W-W Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246–257, 1989.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[7] An-Chieh Cheng, Chieh Hubert Lin, Da-Cheng Juan, Wei Wei, and Min Sun. Instanas: Instance-aware neural architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3577–3584, 2020.

[8] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Adam Coates Mike Chrzanowski, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *In Proceedings of the 33nd International Conference on Machine Learning, (ICML 16)*, pages 2024–2033, 2016.

[9] Jack J Dongarra and A_R Hinds. Unrolling loops in fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

[10] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.

[11] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[12] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[13] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021.

[14] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 120–134, 2022.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[17] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *arXiv preprint arXiv:2206.03382*, 2022.

[18] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.

[19] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the*

27th ACM Symposium on Operating Systems Principles, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[22] Taebum Kim, Eunji Jeong, Geon-Woo Kim, Yunmo Koo, Sehoon Kim, Gyeong-In Yu, and Byung-Gon Chun. Terra: Imperative-symbolic co-execution of imperative deep learning programs. *Advances in Neural Information Processing Systems*, 34, 2021.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[24] Yanwei Li, Lin Song, Yukang Chen, Zeming Li, Xiangyu Zhang, Xingang Wang, and Jian Sun. Learning dynamic routing for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8553–8562, 2020.

[25] Zhiqi Lin, Youshan Miao, Guodong Liu, Xiaoxiang Shi, Quanlu Zhang, Fan Yang, Saeed Maleki, Yi Zhu, Xu Cao, Cheng Li, et al. SuperScaler: Supporting flexible dnn parallelization via a unified abstraction. *arXiv preprint arXiv:2301.08984*, 2023.

[26] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[27] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[28] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[29] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, et al. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 192–204, 2022.

[30] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *The Artificial Intelligence Review*, 42(2):275, 2014.

[31] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Autograph: Imperative-style coding with graph-based performance. volume 1, pages 389–405, 2019.

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[33] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[34] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. FlexMoE: Scaling large-scale sparse pre-trained model training via dynamic device placement. *arXiv preprint arXiv:2304.03946*, 2023.

[35] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 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*, pages 883–898, 2021.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[37] Alexey Radul, Brian Patton, Dougal Maclaurin, Matthew Hoffman, and Rif A Saurous. Automatically batching control-intensive programs for modern accelerators. *Proceedings of Machine Learning and Systems*, 2:390–399, 2020.

[38] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[39] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, pages 151–161, 2011.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[41] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[42] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[44] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[45] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[46] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.

[47] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8817–8826, 2018.

[48] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[49] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 937–950, 2018.

[50] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[51] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[52] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[54] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, 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)*, pages 559–578, 2022.

[55] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599. IEEE, 2017.

[56] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 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 Pro-*

gramming Languages and Operating Systems, pages 359–373, 2022.

[57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

[58] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

A Artifact Appendix

Abstract

This artifact helps to reproduce the results of OSDI'23 paper: COCKTAILER: Analyzing and Optimizing Dynamic Control Flow in Deep Learning.

Usage

The input of COCKTAILER is a PyTorch program. COCKTAILER exports the PyTorch program to ONNX format with ONNX loop and branch operators as well as an extended invoke operator for recursion. Then COCKTAILER generates the code with optimizations described in the paper and wraps the code to a PyTorch custom operator for execution.

Scope

The artifact can be used to reproduce the experiments of the paper, including the end-to-end comparison (Figure 14 and 20), control flow overhead analysis (Figure 2, 15, 16, and 18), performance of different ratio of executed layers (Figure 17), and breakdown of optimizations (Figure 19).

Contents

This artifact includes the code of COCKTAILER, input data of experiments, a guide for setting up the environment of the experiments, and scripts for running the experiments. It helps to reproduce the following Figures:

- Figure 2: Control flow overhead in JAX
- Figure 14: End-to-end DNN inference on NVIDIA V100 GPU
- Figure 15: Control flow overhead of models with loops
- Figure 16: Control flow overhead of models with branches
- Figure 17: Different ratio of executed layers
- Figure 18: Control flow overhead of RAE with recursion
- Figure 19: Breakdown of models with BS=1
- Figure 20: End-to-end DNN inference on AMD MI100 GPU with BS=1

Hosting

The main contents of COCKTAILER are hosted at https://github.com/microsoft/nnfusion/tree/cocktailer_artifact/artifacts, branch `cocktailer_artifact`.

Requirements

This artifact needs two machines:

- a machine with 8 NVIDIA V100 GPUs, with NVIDIA driver properly installed. Users can either follow the installation guide to setup the software environment or install the NVIDIA Container Toolkit to reproduce the results within the docker provided by the artifact.
- a machine with 1 AMD MI100 GPU, with ROCm driver and docker properly installed. Users can then reproduce the results within the dockers provided by the artifact.