先进编译实验室
Advanced Compiler

动态控制流编译优化论文分享：Cocktailer

COCKTAILER: Analyzing and Optimizing Dynamic Control Flow in Deep Learning

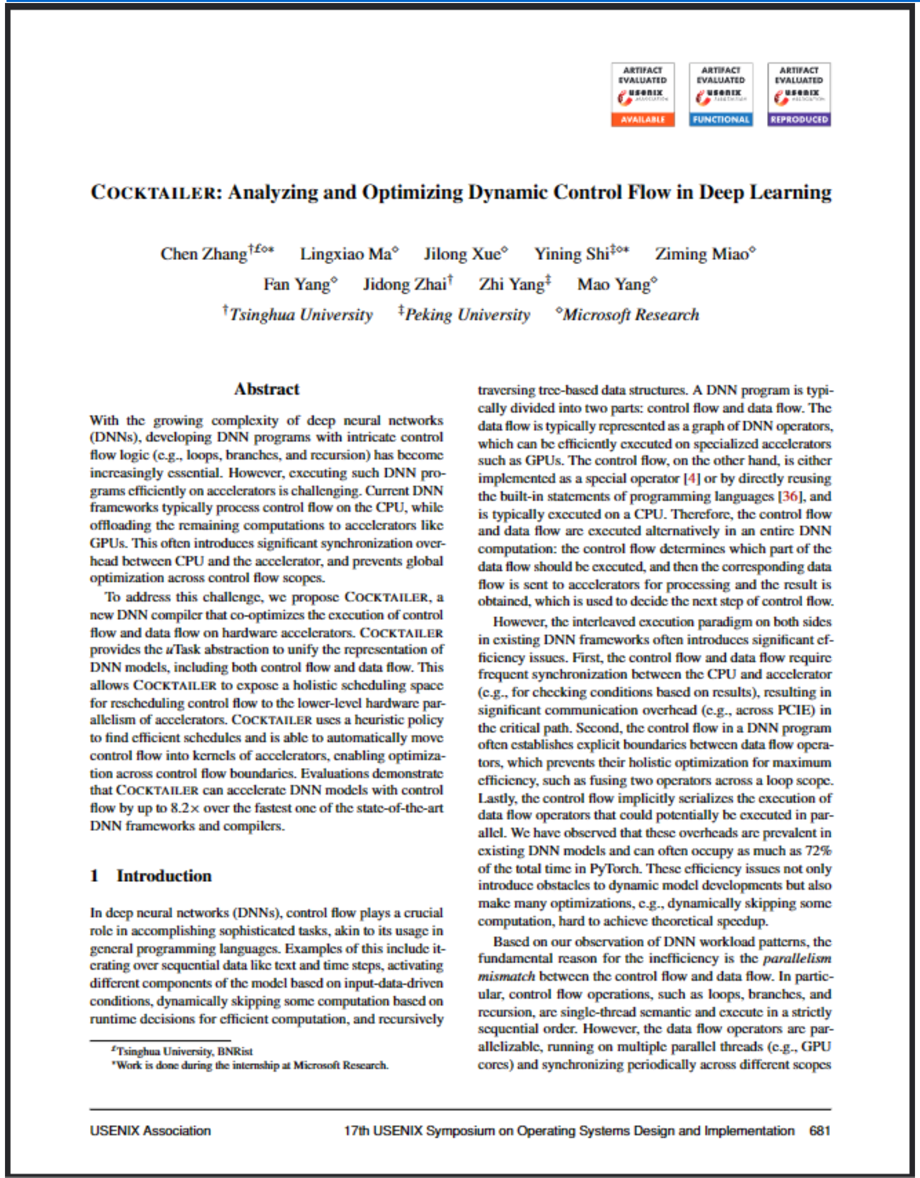
Chen Zhang[†£◇*] Lingxiao Ma[◇] Jilong Xue[◇] Yining Shi[‡◇*] Ziming Miao[◇]
Fan Yang[◇] Jidong Zhai[†] Zhi Yang[‡] Mao Yang[◇]
[†]*Tsinghua University* [‡]*Peking University* [◇]*Microsoft Research*

嘉宾：王磊

先进编译实验室
Advanced Compiler

Cocktailer : Analyzing and Optimizing Dynamic Control Flow in Deep Learning

论文提要



ARTIFACT EVALUATED usenix AVAILABLE | ARTIFACT EVALUATED usenix FUNCTIONAL | ARTIFACT EVALUATED usenix REPRODUCED

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.

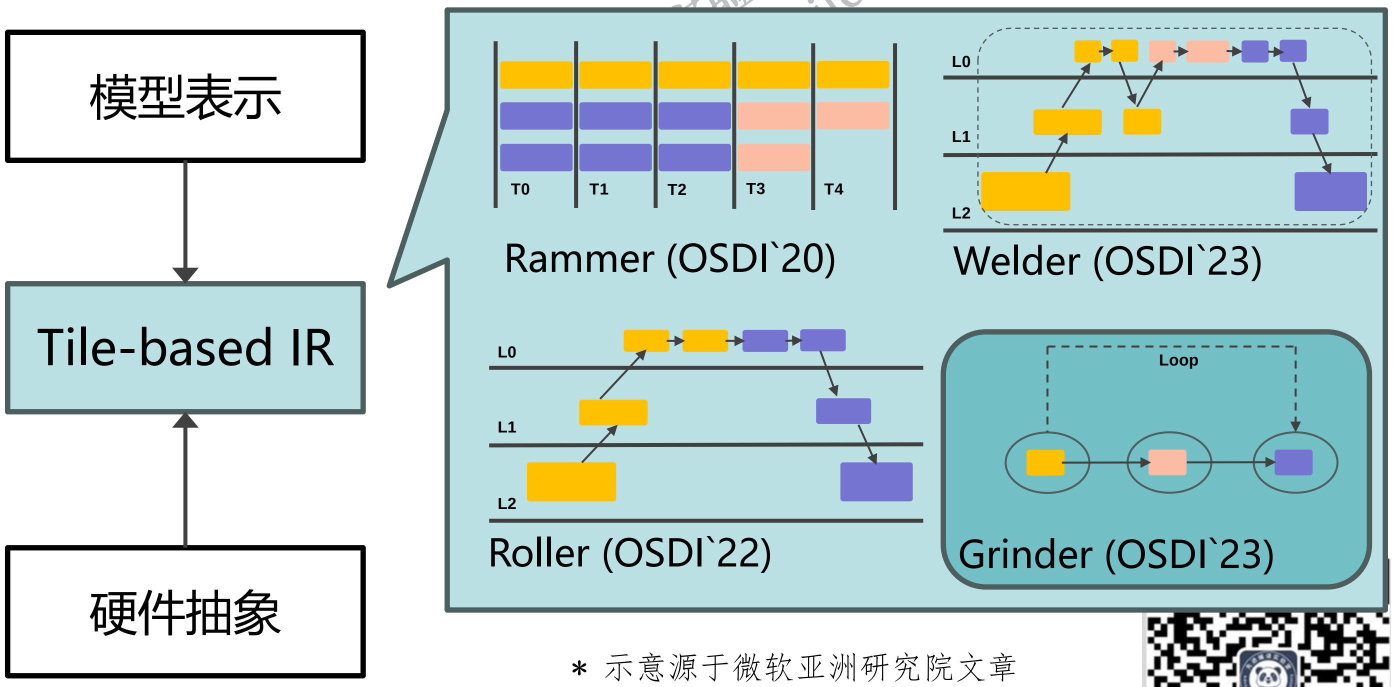
However, the interleaved execution paradigm on both sides in existing DNN frameworks often introduces significant efficiency issues. First, the control flow and data flow require frequent synchronization between the CPU and accelerator (e.g., for checking conditions based on results), resulting in significant 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.

USENIX Association 17th USENIX Symposium on Operating Systems Design and Implementation 681

Cocktailer : Analyzing and Optimizing Dynamic Control Flow in Deep Learning (OSDI '23)

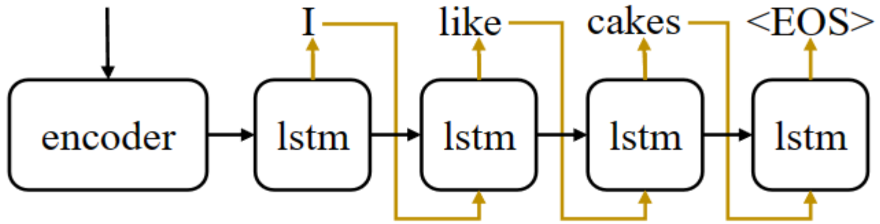


* 示意源于微软亚洲研究院文章
《微软亚洲研究院推出AI编译器界"工业重金属四部曲"》
https://www.msra.cn/zh-cn/news/features/ai-compilor

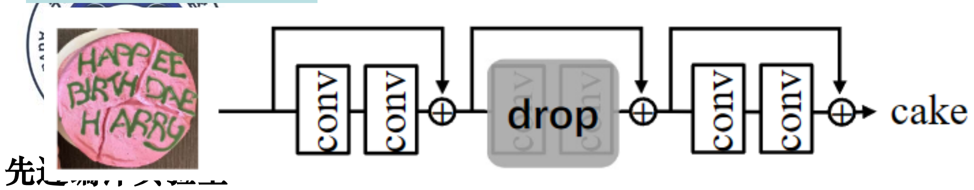
背景 > 控制流

- 深度学习模型：全连接 → 复杂控制逻辑结构：数据流 + 控制流
  - ◆ 动态计算 ◆ 条件计算 ◆ 高效计算
- 控制流分类
  - ◆ 循环 Loop ◆ 分支 Branch ◆ 递归 Recursion

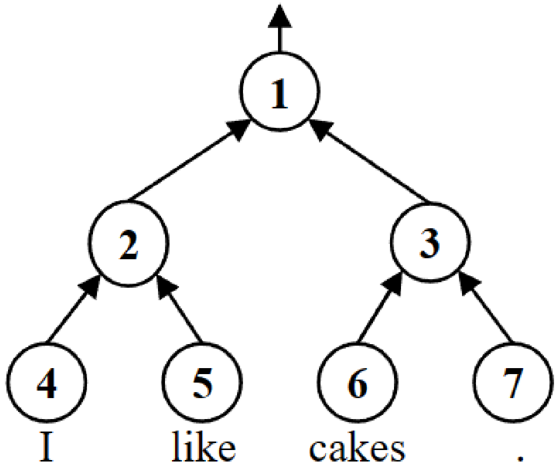
Seq2Seq



BlockDrop

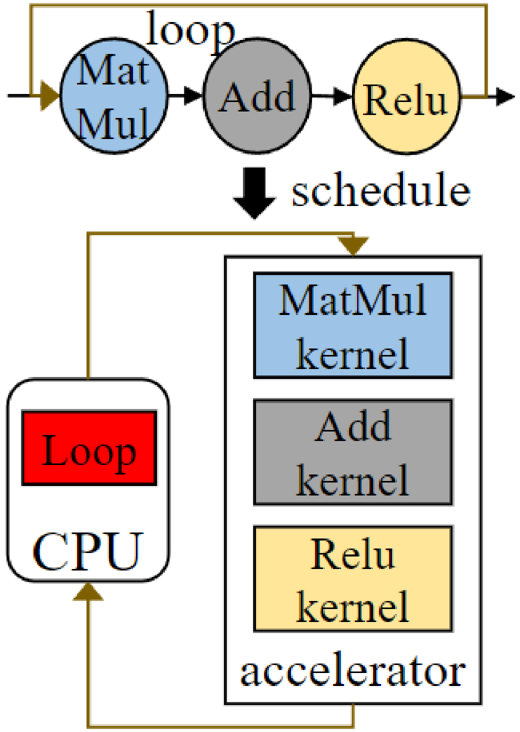


RAE



背景 > 控制流支持

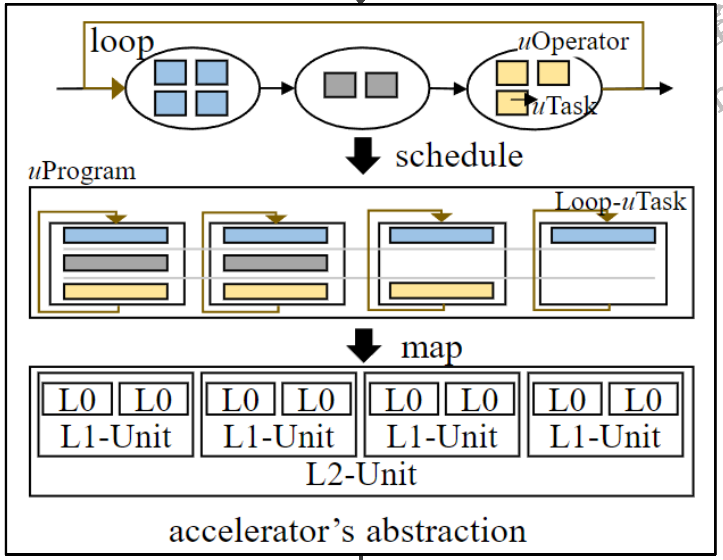
- 当前框架对控制流支持的实现
  - 控制流的表示：表示为特殊算子 or 使用编程语句
  - 计算的调度：数据流 → GPU等加速设备　控制流 → CPU
- 存在的问题
  - 带来频繁的同步和通信开销
  - 难以进行跨控制流边界优化
  - 忽略数据流的并行执行机会



在加速器侧协同调度数据流和控制流？ → 数据流并行性 ≠ 控制流并行性

设计

控制流并行性 ≠ 数据流并行性



在加速器侧协同调度数据流和控制流

- 数据流并行中每个算子多层次并行，且共享相同的控制逻辑
- GPU等加速设备支持控制流指令

- 程序表示：以更细的粒度表示控制流，统一数据流和控制流表示
- 协同调度：正确映射到并行处理单元执行

设计 > 程序表示

uTask > uOperator > uProgram

- uTask

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

- Nested uTask

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

 vector<uTask> body_uTasks;
};
```

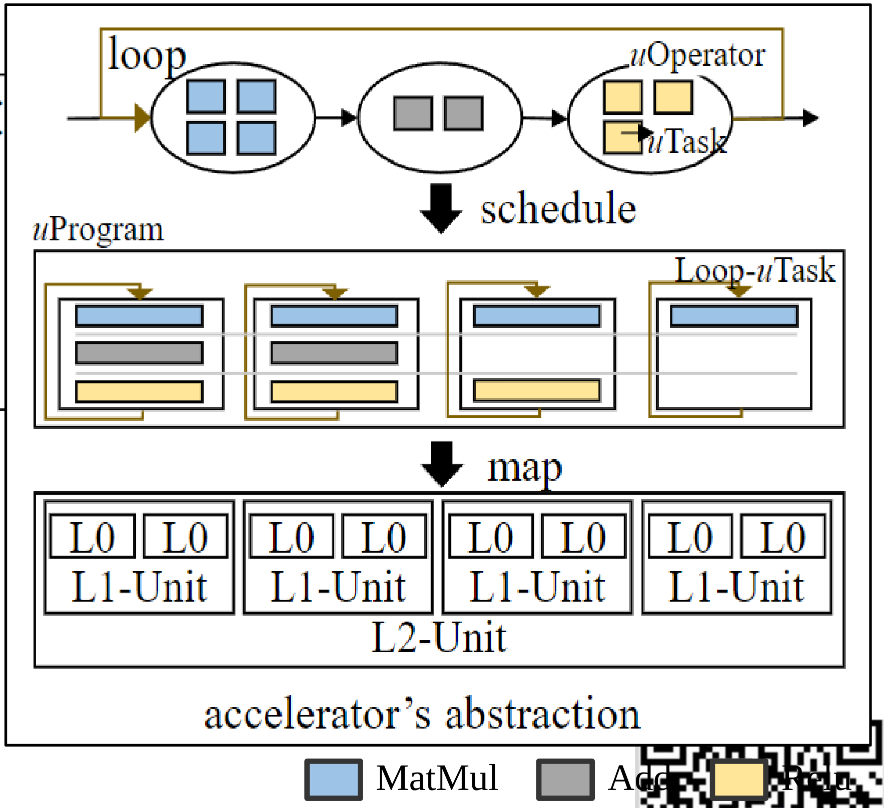
- uOperator

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

 set<uTask> uTasks;
};
```

- uProgram

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



设计 > 协同调度

- 调度接口

Function *ScheduleProgram(g, D, unit_level)*:
// *g* is a graph of operators in *u*Task representation or a control flow operation that the body has been scheduled
resource = GetResource(*D*, unit_level); // calculate resource
cfg = SetResource(*D*, unit_level, resource);
if $g \in D.ControlFlow$ **then**
  return ScheduleControlFlow(*g*, *D*, unit_level, cfg);
else
  for $op \in g.TopoSort()$ **do**
    ScheduleOperator(*op*, *D*, unit_level, cfg);
  return cfg.uProg;

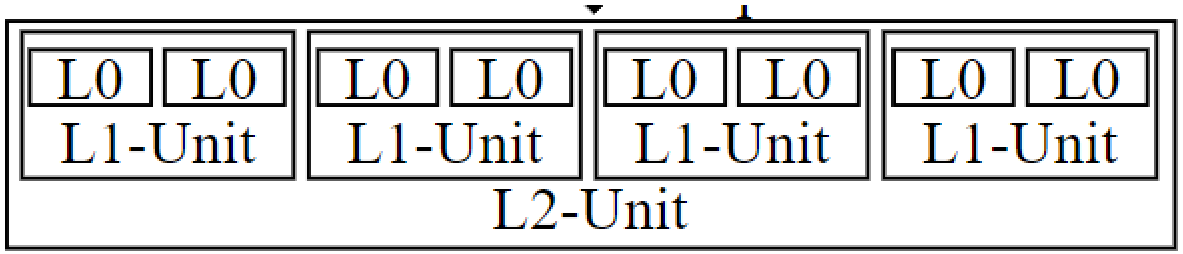
- 调度约束
  - 保证同步点uTask 间依赖关系
  - 控制流 unit_level ≤ 数据流 unit_level
- 调度优化
  - 函数内联 ◆ 循环展开 ◆ 递归展开

- 调度策略：自底向上遍历

Function *Schedule(g, D, unit_level = NULL)*:
ulevel=*unit_level*, ulevel_{max}=*D*.unit_levels.size()-1, uProgs=[];
if *g* ∈ *D.Operators and ulevel is NULL* **then**
  ulevel = 0;
if *ulevel is NULL* **then**
  for $op \in g.TopoSort()$ **do**
    g_{op}, ulevel_{op} = Schedule(op, *D*, NULL);
    ulevel = max(ulevel, ulevel_{op});
    $u\text{Program}_p$ = uProgs[-1];
    ulevel_m = max(ulevel_{op}, $u\text{Program}_p$.ulevel);
    if $ulevel_m < ulevel_{max}$ **then**
      $g_{merge} = u\text{Program}_p.g + g_{op}$;
      g_{merge}, ulevel_{merge} = Schedule(g_{merge}, *D*, ulevel_m);
      if $ulevel_{merge} < ulevel_{max}$ **then**
        uProgs[-1] = g_{merge}.uProgs[0];
        ulevel = max(ulevel, ulevel_{merge});
        continue;
    uProgs.append(g_{op}.uProgs);
else
  uProgs = *g*.uProgs
if $ulevel < ulevel_{max}$ **then**
  for $ulevel_{cur} \in range(ulevel, ulevel_{max})$ **do**
    $u\text{Program}_{cur}$ = ScheduleProgram(*g*, *D*, ulevel_{cur});
    if $uProgram_{cur}$ *is not NULL* **then**
      if $Latency(uProgram_{cur}) < Latency(uProgs)$ **then**
        uProgs = [$u\text{Program}_{cur}$];
        ulevel = ulevel_{cur};
g.uProgs = uProgs;
return *g*, ulevel;

实现

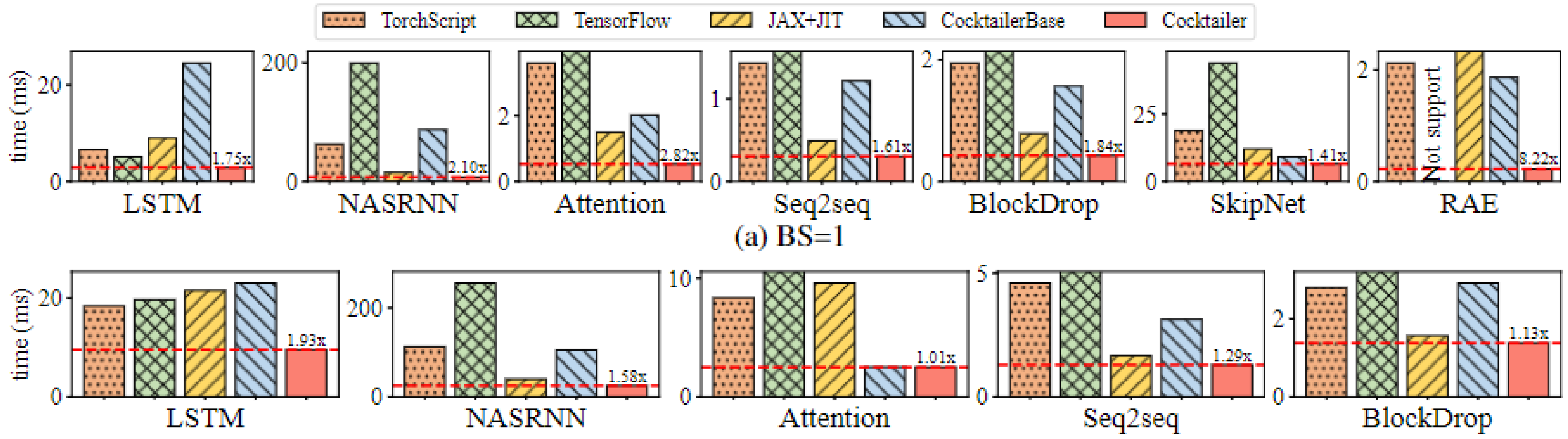
- 对接 框架/编译器
  - 框架：PyTorch程序 → ONNX计算图 → Cocktailer → 自定义PyTorch算子
  - 编译器：数据流算子Kernel → Rammer生成uTask组成的uOperator → Cocktailer
- 对接 加速器
  - NIVDIA GPU
  - AMD GPU
  - Graphcore IPU



- 其他实现细节
  - Block alignment
  - Memory management
  - Simulation of GPU stack
  - Register pressure
  - Branch reclustering

性能

- NIVDIA GPU



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

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

- AMD GPU

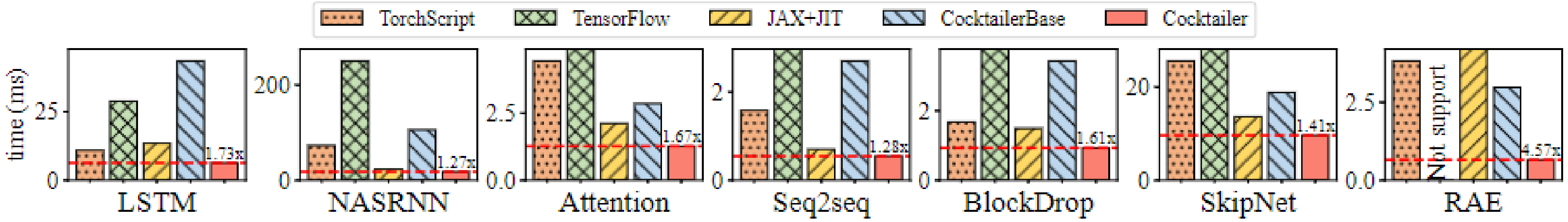


Figure 20: End-to-end DNN inference on AMD MI100 GPU with BS=1

参考资料

[1] 微软亚洲研究院推出AI编译器界“工业重金属四部曲”：https://www.msra.cn/zh-cn/news/features/ai-compilor

[2] 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. OSDI 2023.