RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with *r*Tasks

Lingxiao Ma[*][†][◇] Zhiqiang Xie[*][‡][◇] Zhi Yang[†] Jilong Xue[◇] Youshan Miao[◇]
Wei Cui[◇] Wenxiang Hu[◇] Fan Yang[◇] Lintao Zhang[◇] Lidong Zhou[◇]

[†]*Peking University* [‡]*ShanghaiTech University* [◇]*Microsoft Research*

Abstract

Performing Deep Neural Network (DNN) computation on hardware accelerators efficiently is challenging. Existing DNN frameworks and compilers often treat the DNN operators in a data flow graph (DFG) as opaque library functions and schedule them onto accelerators to be executed individually. They rely on another layer of scheduler, often implemented in hardware, to exploit the parallelism available in the operators. Such a two-layered approach incurs significant scheduling overhead and often cannot fully utilize the available hardware resources. In this paper, we propose RAMMER, a DNN compiler design that optimizes the execution of DNN workloads on massively parallel accelerators. RAMMER generates an efficient static spatio-temporal schedule for a DNN at compile time to minimize scheduling overhead. It maximizes hardware utilization by holistically exploiting parallelism through inter- and intra- operator co-scheduling. RAMMER achieves this by proposing several novel, hardware neutral, and clean abstractions for the computation tasks and the hardware accelerators. These abstractions expose a much richer scheduling space to RAMMER, which employs several heuristics to explore this space and finds efficient schedules. We implement RAMMER for multiple hardware backends such as NVIDIA GPUs, AMD GPUs, and Graphcore IPU. Experiments show RAMMER significantly outperforms state-of-the-art compilers such as TensorFlow XLA and TVM by up to 20.1×. It also outperforms TensorRT, a vendor optimized proprietary DNN inference library from NVIDIA, by up to 3.1×.

1 Introduction

Deep neural network (DNN) is now a widely adopted approach for image classification, natural language processing, and many other AI tasks. Due to its importance, many computational devices, such as CPU, GPU, FPGA, and specially designed DNN accelerators have been leveraged to perform DNN computation. Efficient DNN computation on these devices is an important topic that has attracted much research attention in recent years [22, 27, 31, 38, 49]. One of the key factors that affect the efficiency of DNN computation is scheduling, i.e. deciding the order to perform various pieces of computation on the target hardware. The importance of scheduling in general is well known and has been thoroughly studied [19, 37]. However, there is little work discussing scheduling for DNN computation on hardware devices specifically.

The computational pattern of a deep neural network is usually modeled as a data flow graph (DFG), where each node corresponds to an operator, which represents a unit of computation such as matrix multiplication, while an edge depicts the dependency between operators. This representation naturally contains two levels of parallelism. The first level is the inter-operator parallelism, where operators that do not have dependencies in the DFG may run in parallel. The second level is the intra-operator parallelism, where an operator such as matrix multiplication has inherent internal data parallelism and can leverage hardware accelerators that can perform parallel computation, such as a GPU.

To exploit the two levels of parallelism, current practice adopts a two-layered scheduling approach. An inter-operator DFG layer scheduler takes the data flow graph and emits operators that are ready to be executed based on the dependencies. In addition, an intra-operator scheduler takes an operator and maps it to the parallel execution units in the accelerator. This layering design has a fundamental impact on the system architectures of the existing DNN tool sets. For example, the DFG layer scheduler is typically implemented in deep learning frameworks such as TensorFlow [17] or ONNX Runtime [13]. The operator layer scheduler, on the other hand, is often hidden behind the operator libraries such as cuDNN [11] and MKL-DNN [8], and sometimes implemented directly in hardware, as is the case for GPUs.

While widely adopted by existing frameworks and accelerators, such a two-layer scheduling approach incurs fundamental performance limitations. The approach works well only

[*] Both authors contributed equally.

when the overhead of emitting operators is largely negligible compared to the execution time of operators, and when there is sufficient intra-operator parallelism to saturate all processing units in an accelerator. This unfortunately is often not the case in practice. DNN accelerators keep on increasing performance at a much faster pace than CPUs, thus making the operator emitting overhead more and more pronounced. This is exacerbated for DNN inference workloads when the batch size is small, which limits the intra-operator parallelism. Moreover, the two-layer scheduling approach overlooks the subtle interplay between the upper and lower layers: to optimize the overall performance, a system could reduce the degree of intra-operator parallelism in order to increase the level of inter-operator parallelism (§ 2).

To mitigate these limitations, we present RAMMER, a deep learning compiler that takes a holistic approach to manage the parallelism available in the DNN computation for scheduling. It unifies the inter- and intra-operator scheduling through a novel abstraction called *r*Task. *r*Task enables the scheduler to break the operator boundary and allows fine-grained scheduling of computation onto devices. Instead of the existing design that breaks scheduling into two pieces managed by software and hardware separately, RAMMER is a unified software-only solution, which makes it less dependent on underlying hardware and thus can be adopted by diverse DNN accelerators. In RAMMER, we make the following design decisions.

First, to exploit the intra-operator parallelism through a software compiler, RAMMER redefines a DNN operator as an *rTask*-operator or *r*Operator. An *r*Operator consists of multiple independent, homogeneous *rTask*s, each is a minimum schedulable unit runs on a single execution unit of an accelerator (e.g., a streaming multiprocessor SM in a GPU). Thus, *r*Task as the fine-grained intra-operator information is exposed to the RAMMER scheduler. RAMMER treats a DNN as a data flow graph of *r*Operator nodes, hence it can still see the coarse-grained inter-operator (DFG) dependencies.

Unfortunately, certain modern accelerators such as GPU do not expose interfaces for intra-operator (i.e., *r*Task) scheduling. To address this challenge, as a second design decision RAMMER abstracts a hardware accelerator as a *virtualized parallel device* (vDevice), which contains multiple virtualized execution units (vEU). The vDevice allows several *r*Tasks, even from different operators, to run on a specified vEU in a desired order. Moreover, a vEU can run a *barrier* *r*Task that waits for the completion of a specified set of *r*Tasks, thus ensuring the correct execution of *r*Tasks from dependent operators. The vDevice maps a vEU to one of the physical execution units in an accelerator to perform the actual computation of *r*Tasks.

Finally, fine-grained scheduling could incur significant runtime overheads, even more so than the operator scheduling overhead discussed previously. To address this issue, RAMMER moves the scheduling decision from runtime to compile time. This is driven by the observation that most DNN's DFG is available at the compile time, and the operators usually exhibit deterministic performance characteristics. Therefore, the runtime performance can be obtained through compile time profiling [42]. This not only avoids unnecessary runtime overheads, but also allows a more costly scheduling policy to fully exploit the inter- and intra- operator parallelism together.

RAMMER is compatible with optimizations developed in existing DNN compilers. RAMMER can import a data-flow graph from other frameworks like TensorFlow. Such a DFG can be optimized with techniques employed by a traditional graph optimizer such as [17]. An *r*Operator can also be optimized by an existing kernel tuner [22]. Our experience shows that, on top of existing optimizations, RAMMER can provide significant additional performance improvement, especially for DNN inference workloads.

RAMMER is hardware neutral. The abstractions proposed, such as *r*Task, *r*Operator and vEU are applicable to any massively parallel computational devices with homogeneous execution units. This includes almost all the computational devices proposed for DNN workloads. In this paper, in addition to describe in detail how RAMMER is implemented on NVIDIA GPUs, we will also discuss our experience retargeting RAMMER for several alternative computing devices.

We have implemented RAMMER with 52k lines of C++ code and open source the code[1]. Our evaluation on 6 DNN models shows that RAMMER significantly outperforms state-of-the-art compilers like XLA and TVM on both NVIDIA and AMD GPUs, with up to 20.1× speedup. RAMMER even outperforms TensorRT [12], a vendor optimized DNN inference library from NVIDIA, with up to 3.1× gain.

Our experience on RAMMER strongly suggests that the current industry-prevalent practice of vendor supplying highly optimized DNN operator implementations in a library form (such as cuDNN and MKL-DNN) is sub-optimal. This practice will incur significant efficiency cost for DNN workloads. The situation will become even worse in the coming years as modern accelerators keep on increasing the available hardware parallelism while new DNN architectures strive to save computation by replacing larger operators with many smaller ones [46,51]. We recommend vendors to supply optimized implementations in other forms, such as our proposed *r*Operator and vEU abstractions, in order to enable holistic optimization that can fully utilize hardware resources.

2 Motivation

In this section we highlight some results to illustrate the limitation of the two-layer design of existing deep learning frameworks. Without the loss of generality, we experiment with TensorFlow [17], a state-of-the-art DNN framework, on an NVIDIA GPU, using the same settings as in §5.

[1] https://github.com/microsoft/nnfusion

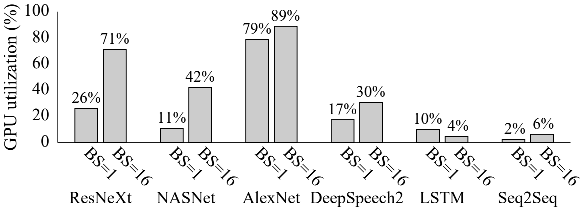

Figure 1: The average GPU utilization on different DNN model with different batch size (BS). The utilization only accounts for kernel execution, excluding other stages like operator emitting.

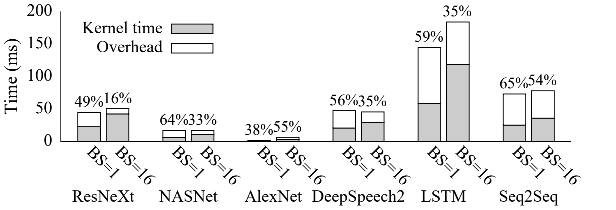

Figure 2: The average kernel time and end-to-end execution time on different DNN model with different batch size (BS).

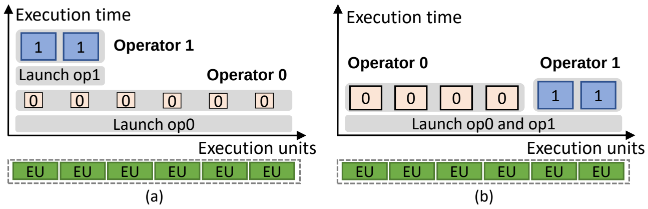

Figure 3: An illustration of (a) the inefficiency scheduling in existing approach; and (b) an optimized scheduling plan.

Hardware-managed intra-operator scheduling leads to low GPU utilization. The two-layer design delegates the intra-operator scheduling to the hardware scheduler in an accelerator like GPU. Figure 1 shows that such an approach could lead to low GPU utilization across different DNN models. When the batch size is 1, the GPU utilization could be as low as 2% for Seq2Seq model. Even when the batch size is increased to 16, the average GPU utilization across the 6 models is merely 40% [2]. To improve the scheduling efficiency, modern GPUs support the multi-streaming mechanism that allows independent operators to run concurrently. However, our measurement in §5 shows that multi-streaming often hurts rather than improves the overall performance.

High inter-operator scheduling overheads. The two-layer approach also incurs a higher inter-operator scheduling overheads. Here, we regard the time not spent doing actual computation in the GPU as the overhead for inter-operator scheduling. This overhead includes various operations to support operator emitting, including kernel launching, context initialization, communication between host and GPU, and so on. The percentage shown above each bar in Figure 2 depicts how much time the a DNN model is *not* spent in the actual GPU computation. From the figure it is clear that the overhead of inter-operator scheduling is quite significant. When batch size is 1, the average overhead is 55% across the 6 DNN models. Increasing the batch size to 16 slightly improves the situation, while the overhead is still not negligible (between 16% and 55%). Modern DNN compilers, including the one in TensorFlow, employs a technique called kernel fusion [16,22], which merges several DNN operators into a single one when allowed. However, our results in §5 show that this technique cannot reduce the overhead significantly.

Interplay between inter- and intra-operator scheduling. Separating scheduling into two layers ignores the subtle interplay between inter-operator and intra-operator scheduling, which may lead to suboptimal performance. For example, Figure 3(a) shows two independent operators being scheduled to a GPU. For operator 0, to maximize its performance, the system may choose the fastest implementation with a high degree of parallelism. Thus operator 0 could greedily span all the parallel execution units (EUs) of an accelerator (in this case the streaming multiprocessors of the GPU), while each EU may not be fully utilized. Since operator 0 occupies all the EUs, operator 1 has to wait for available resource. A better scheduler could reduce the degree of parallelism of operator 0 to increase the level of inter-operator parallelism, by mapping operator 1 alongside operator 0, as illustrated by Figure 3(b). We will discuss more details of this issue in §3.3 and §5.

Opportunities. Given the fundamental limitations of the two-layer design observed above, it is desirable to manage the scheduling of inter and intra-operator together. However, a naive implementation of this approach may incur even higher overheads than the already significant inter-operator scheduling overheads. Fortunately, most DNN's DFG is available at the compile time, and the operators often exhibit deterministic performance, therefore their execution times can be obtained through compile time profiling [42]. For example, Figure 4 shows the averaged GPU kernel time and the variance of all the operators in the ResNeXt [46] model. The kernel run-time weighted average of standard deviations among all operators is only 7%. This allows us to move the scheduling from runtime to compile-time, by generating an offline schedule plan to reduce runtime overhead.

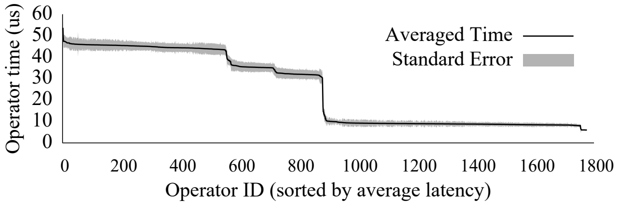

Figure 4: The profiled kernel time of all the operators in ResNeXt model. Each data point ran 1,000 times

[2] Note that the LSTM's GPU utilization is slightly higher when batch size is 1 compared to that when batch size is 16, because TensorFlow uses different kernel implementations of GEMM for different batch sizes.

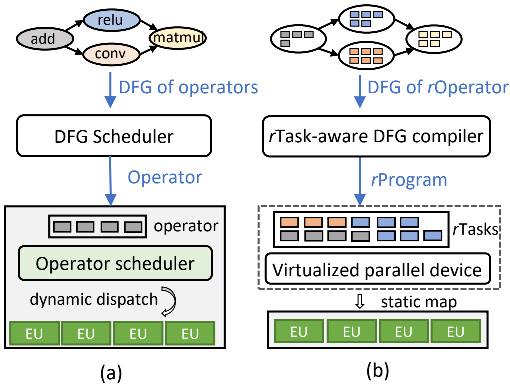

Figure 5: System overview of DNN computation in (a) existing DNN frameworks, and (b) RAMMER, where each node in a DFG is an *r*Operator that explicitly exposes the intra-operator parallelism through *r*Task; Rather than dynamically scheduling each *r*Operator, RAMMER compiles the DFG into a static execution plan called *r*Program (composed of *r*Tasks) and maps it to hardware by a software device abstraction called vDevice.

3 RAMMER's Design

The observations in §2 motivate RAMMER, a DNN compiler framework that manages both inter and intra-operator scheduling. Figure 5 shows the key differences between an existing deep learning framework and RAMMER. First, the input to RAMMER is a data-flow graph where a node is an *rOperator*, rather than a traditional operator. An *r*Operator explicitly exposes *r*Task, a fine-grained computation unit that could run on a parallel execution unit in an accelerator. We discuss details of *r*Task in §3.1. Second, instead of separating the two-layer scheduling between software and hardware, RAMMER introduces *rTask-aware DFG compiler* to manage the inter and intra-operator scheduling in one place. The *r*Task-aware DFG compiler will generate a static execution plan for runtime execution. Often, it is not efficient or not possible to pack the entire DNN computation in a single accelerator device invocation. Therefore, the execution plan is breaking into multiple *r*Programs, each contains a piece of computation to be carried out on the hardware. Instead of emitting one operator at a time for an accelerator, RAMMER emit an *rProgram* at a time. The details of the *r*Task-aware DFG compiler will be discussed in §3.3. To carry out the execution plan, RAMMER abstracts a hardware accelerator as a *virtualized parallel device* (vDevice), which includes multiple virtualized execution units (vEUs). The vDevice provides the scheduling and synchronization capabilities at the *r*Task level so that the *r*Program can be mapped to the corresponding vEUs at compile time. The vEUs, together with the vDevice will be mapped to the hardware at runtime. We introduce virtualized device in §3.2.

```
1 interface Operator { void compute(); };
2 interface rOperator {
3   void compute_rtask(size_t rtask_id);
4   size_t get_total_rtask_num();
5 };
```

Figure 6: The execution interfaces of (a) traditional operator and (b) *r*Operator. More details in §4.

3.1 *r*Operator

An *r*Operator is defined as a group of independent, homogeneous ***r*Tasks** (short for RAMMER task), where an *r*Task is the minimum computation unit in an operator to be executed on a processing element of the accelerator device. The concept of *r*Task naturally aligns with the parallel architecture of DNN accelerators, e.g., the SIMD architecture of GPU. To maximize efficiency, the computation on such an accelerator need to be divided into multiple parallel (homogeneous) tasks. Each of these parallel tasks can be represented by an *r*Task, thereby exposing the intra-operator parallelism not only to the underlying hardware, but to the RAMMER compiler. Given that an *r*Task is logically identical to a parallel task, RAMMER relies on external tools to partition an *r*Operator into *r*Tasks (e.g., TVM [22]). In another word, RAMMER uses external heuristics to decide a reasonable granularity of *r*Task.

As a concrete example, a matrix multiplication operator can be divided into multiple homogeneous *r*Tasks, each computes a tile of the output matrix, while the tiling strategy is assumed to be given. For a more complicated DNN operator that cannot be simply divided into homogeneous *r*Tasks, it could be further divided into multiple dependent *r*Operators, each can be further partitioned into *r*Tasks.

An *r*Task is indexed by a logical `rtask_id`. The *r*Tasks in an *r*Operator are numbered continuously. To execute an *r*Task, the parallel execution unit could call the `compute_rtask()` interface (line 3 Figure 6). To generate an *r*Program, RAMMER needs to know the total number of *r*Tasks in an operator. This is available through the interface `get_total_rtask_num()`. In contrast, a traditional operator has only one interface `compute()`(line 1 Figure 6). The implementation of an *r*Operator is called *r*Kernel, which realizes the concrete *r*Task computation logics and decides the total number of *r*Tasks. One *r*Operator might have multiple versions of *r*Kernels based on different tiling strategies, e.g., trading off between resource efficiency and overall execution time.

The *r*Operator abstraction allows RAMMER to expose both inter- and intra-operator parallelisms. This opens up a new space to optimize DNN computation holistically.

3.2 Virtualized Parallel Device

Modern accelerators do not provide interfaces to map an *r*Task to a desired execution unit directly. For example, a GPU only allows to execute one operator (in the form of a kernel) at a time. To address this challenge, RAMMER abstracts a hardware accelerator as a software-managed virtual device called *virtualized parallel device* (*vDevice*). A vDevice further presents multiple parallel *virtual execution units* (*vEU*s), each of them can execute *r*Tasks independently.

With vDevice, RAMMER organizes the computation of *r*Task-aware DFG as an *r*Program on a vDevice. An *r*Program is represented as two dimensional array of *r*Task `prog[vEU_id][order]`, where `vEU_id` denotes the vEU the *r*Task is assigned to, and `order` denotes the execution order of the *r*Task in this vEU. For example, `prog[0][0]` denotes the first *r*Task to be executed in vEU `0`. To ensure the correct execution of dependent *r*Tasks in a plan, RAMMER introduces ***barrier-rTask***. A barrier-*r*Task takes the argument of a list of pairs `<vEU_id, order>`. The barrier-*r*Task will wait until the completion of all *r*Tasks indexed by each pair. The barrier-*r*Task provides a fine-grained synchronization mechanism to enable *r*Task schedule plan execution.

For the execution of DNN computation, a vDevice need to be mapped to a physical accelerator at runtime. We will discuss how RAMMER implements the mapping of vDevice to different hardware accelerators in §4.

3.3 *r*Task-aware DFG Compiler

The *r*Task abstraction and the fine-grained *r*Task execution capability exposed by the vDevice open up a large optimization space. RAMMER aims to generate a high-quality schedule in this space, represented as a sequence of *r*Programs. To this end, the *r*Task-aware DFG compiler separates the scheduling mechanism from its policy. On the mechanism side, it provides two capabilities: (1) Two scheduling interfaces for a policy to generate an execution plan. (2) A profiler to supply profiling information requested by a scheduling policy.

Scheduling interfaces. RAMMER's *r*Task-aware DFG compiler introduces two scheduling interfaces, `Append` and `Wait`. `Append(task_uid, vEU_id)` assigns an *r*Task from an operator to the specified vEU in a sequential order. Here `task_uid` is a global identifier for an *r*Task, which is essentially the operator `id` combined with the `rtask_id` within the operator. The second API, namely `Wait(wtask_uid, list<task_uid>)`, allows an *r*Task specified by `wtask_uid` to wait for *r*Tasks in `list<task_uid>`. The `Wait` interface will implicitly `Append` a *barrier*-*r*Task (discussed in §3.2) right before the *r*Task `wtask_uid`. As an optimization, when waiting for multiple consecutive *r*Tasks $r_1, r_2, ..., r_n$ sequentially appended to the same vEU, the *r*Task only need to include the last one, i.e., r_n, in the waiting list.

Compile-time profiling. RAMMER profiler provides the following three types of information: 1) individual *r*Task execution time on a vEU; 2) resource usage of an *r*Task such as the local memory or registers used and 3) the overall execution time of an *r*Program. This profiling information can guide a policy to generate an efficient scheduling plan.

Algorithm 1: Wavefront Scheduling Policy

```
   Data: G: DFG of rOperator, D: vDevice
   Result: Plans: rPrograms
1  Function Schedule(G, D):
2      $P_{curr} = \{\}$;
3      for $W = Wavefront(G)$ do
4          $P_1 = ScheduleWave(W, P_{curr}, D)$;
5          $P_2 = ScheduleWave(W, \{\}, D)$;
6          if $time(P_1) \leq time(P_{curr}) + time(P_2)$ then
7              $P_{curr} = P_1$;
8          else
9              Plans.push_back($P_{curr}$);
10             $P_{curr} = P_2$;
11     return Plans;
12 Function ScheduleWave(W, P, D):
13     $SelectRKernels(W, P)$;
14     for $op \in W$ do
15         for $r \in op.rTasks$ do
16             $vEU = SelectvEU(op, P, D)$;
17             P.Wait($r$, Predecessor(op).rTasks);
18             P.Append($r$, $vEU$);
19     return $P$;
```

Scheduling policy. Algorithm 1 illustrates how to use the above scheduling interfaces and the profiler to implement a scheduling policy to exploit both inter- and intra-operator parallelisms. This policy takes an *r*Task-aware DFG and schedules operators in *waves* [35]. The operators in a wave are the fringe nodes of a breadth-first-search on the DFG. The policy will include a wave's operators in the current *r*Program if the profiling results (denoted by `time()`) suggest it will reduce the total execution time. Otherwise, the policy will create a separate *r*Program (line 2-10).

First of all, we assume that each *r*Operator has one or more implementations called *r*Kernels, each *r*Kernel is a way to break the operator into *r*Tasks with different resource and runtime trade-offs. Among the *r*Kernels of a particular *r*Operator, there is the *fastest* one with the smallest runtime, and there is the most *efficient* one with the smallest product of runtime and the total number of *r*Tasks.

For each wave, the policy selects the implementations of the operators through `SelectRKernels()` (line 13), with the following heuristics: If combine all the *r*Tasks in the wave with the *fastest* operator implementations still cannot occupy all the parallel execution units in the accelerator, the policy will just select them. Otherwise, the policy will find the most *efficient* *r*Kernels and then perform a profiling. The policy will choose these *r*Kernels if the profiling results show better execution time, otherwise it will stick with the fastest *r*Kernels. This heuristic considers the interplay between the inter- and intra-operator scheduling by evaluating the *r*Operators (and their *r*Tasks) in a wave, instead of individually. After the *r*Kernel selection, the policy calls `SelectvEU()` to decide which vEU an *r*Task should be scheduled to (line 16). Given the current *r*Program `P`, `SelectvEU()` chooses the vEU that can execute the *r*Task at the earliest, based on the profiled

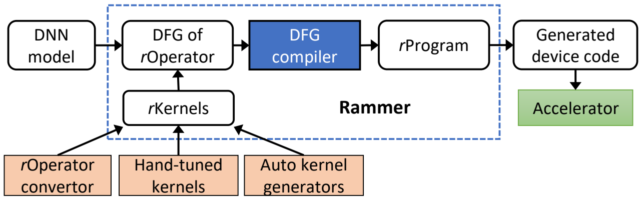

Figure 7: Overall workflow of Rammer.

execution time of each *r*Task in `P`. Finally, the policy calls `Wait()` to ensure *r*Task level dependency (derived from the DFG) and `Append()` to assign the *r*Task to the selected vEU (line 17-18). The policy in Algorithm 1 demonstrates how Rammer separates the scheduling mechanism from scheduling policy. As shown in §5, this simple policy can already outperform the state-of-the-art, sometimes significantly. We envision the proposed scheduling mechanism could enable future research on more advanced scheduling policies to further explore the optimization space.

4 Implementation

We implement Rammer with 52k lines of C++ code, including 3k line of codes for the core compiler and scheduling function. The input of Rammer is a DNN model in either TensorFlow [17] frozen graph, TorchScript [15] or ONNX [13] format. Rammer first converts the input model into a DFG of *r*Operators. Since the input model is often not optimized, like other compilers, we also implemented common graph optimizations such as constant folding, common sub-expression elimination, pattern-based kernel fusion, etc. For each *r*Operator from an optimized DFG, Rammer loads one or multiple versions of *r*Kernel implementations from different sources, e.g., auto-kernel generators [22], hand-tuned kernels, or converted from existing operators in other frameworks. Rammer compiler will then partition the DFG into sub-graphs (e.g., based on the policy in Algorithm 1) and compile each of them as an *r*Program. As an output, each *r*Program is further generated as a device code (e.g., GPU kernels) that runs on the accelerator. Figure 7 summarizes the overall workflow of Rammer.

In the rest of this section, we describe the details about Rammer's implementation for CUDA GPU. We focus on NVIDIA GPUs and the CUDA eco-system because they are the most widely used accelerators for DNN. To demonstrate that the vDevice abstraction enables Rammer compiler to support different accelerators with an uniform interface, we will also briefly describe our experience with other DNN accelerators, including AMD GPUs and Graphcore IPU, at the end of this section.

```
1  __device__ void matmul_rTask(float *A, float *B,
2      float *C, size_t rtask_id) {
3    size_t tile_x = rtask_id / (M/32);
4    size_t tile_y = rtask_id % (N/32);
5    size_t i = threadIdx.x/32 + tile_x*32;
6    size_t j = threadIdx.x%32 + tile_y*32;
7    C[i][j] = 0;
8    for (size_t k = 0; k < K; k++)
9      C[i][j] += A[i][k] * B[k][j];
10 }
11
12 class MatmulROperator {
13  __device__ void compute_rtask(size_t rtask_id){
14    matmul_rTask(input0,input1,output0,rtask_id);}
15  size_t get_total_rtask_num(){return M/32*N/32;}
16 };
```

Figure 8: A CUDA implementation of a naive matrix multiplication with the *r*Operator abstraction.

4.1 Rammer on NVIDIA CUDA GPUs

An NVIDIA GPU usually consists of tens to hundreds of *streaming multiprocessors (SM)*, each containing tens of cores. Computation on SM follows the Single Instruction Multiple Thread (SIMT) model. In this paper we assume the readers are familiar with the basic concepts of CUDA [4], the programming paradigm introduced by NVIDIA to program their GPUs. A single CUDA program (often referred to as a CUDA kernel) groups multiple threads into *block*s, each thread-block is assigned to run on an SM, where the scheduling is performed by GPU hardware. Rammer naturally maps each vEU to an SM and implements an *r*Task as a thread-block.

4.1.1 *r*Operator in CUDA

Figure 8 shows a naive CUDA implementation of an *r*Operator that multiplies a $M \times K$ matrix A by a $K \times N$ matrix B. For simplicity, we assume M and N are evenly divisible by 32. In the code, each *r*Task computes a 32×32 tile of the output matrix C. Line 1-10 in Figure 8 shows the computation of one thread in one *r*Task. The thread uses `rtask_id`, a Rammer assigned id to identify the tile to be computed by this *r*Task (line 3-4), and uses `threadIdx`, a CUDA built-in thread index to identify the data element to be computed (line 5-6) by this thread. The identified element is then computed in line 7-9. Line 13 shows the interface exposed by this *r*Operator, which will be called by a vEU's parallel thread. The total *r*Tasks needed in this operator is determined by the matrix dimension M and N, and can be obtained through the `get_total_rtask_num` interface (line 16). The key difference between code in Figure 8 and a traditional CUDA code is that an *r*Task uses `rtask_id`, a logical index controlled by Rammer, instead of `blockIdx`, a built-in thread-block index controlled by the GPU's hardware scheduler. This enables Rammer to map an *r*Task to a desired vEU by executing `compute_rtask()` with a proper `rtask_id`. Note that the code shown in Figure 8 is for illustrative purpose. The evalu-

```
// config: <<<(vEU_size,1,1), (vEU#,1,1)>>>
__global__ void vdevice_run() {
  if (Get_vEU_Id() == 0) { // vEU 0
    MatmulrTaskOp.compute_rtask(0);
    MatmulrTaskOp.compute_rtask(1);
    // wait the rTask on vEU 1 with order=3
    BarrierTask({<1, 3>}).compute_rtask();
    Conv2DrTaskOp.compute_rtask(0);
  }
  else if (Get_vEU_Id() == 1) { // vEU 1
    for (auto i : 4)
      RelurTaskOp.compute_task(i);
    // wait the rTask on vEU 0 with order=1
    BarrierTask({<0, 1>}).compute_rtask();
    Conv2DrTaskOp.compute_rtask(1);
  }
}
```

Figure 9: The CUDA code for a vDevice with two vEUs.

ation shown in §5 uses a more complicated tiled version of matrix multiplication *r*Operator for better performance.

4.1.2 vDevice and vEU on CUDA GPU

On a CUDA GPU, the intra-operator scheduling is usually managed by the GPU's built-in scheduler. To bypass the built-in scheduler, Rammer leverages a persistent thread-block (PTB) [28] to implement a vEU in a vDevice. PTB is a thread-block containing a group of continuously running threads, where Rammer is able to "pin" the PTB to the desired SM. Given an *r*Program, each thread in the PTB (and hence the vEU) executes the `compute_rtask()` according to the sequence specified by the *r*Program. To execute the `compute_rtask()` from multiple *r*Tasks continuously in a PTB, a function qualifier `__device__` is required by CUDA for `comptue_rtask()` and any sub functions executed therein (e.g., line 1 and 13 in Figure 8).

Figure 9 illustrates the CUDA code for a vDevice with two vEUs, i.e., a CUDA kernel function with two PTBs. This vDevice executes an *r*Program compiled from a DFG with three *r*Operators: a `Matmul`, a `Relu`, and a `Conv`. Specified by the execution plan, the vDevice executes two *r*Tasks of the `Matmul` operator on vEU 0, and in parallel it also runs four *r*Tasks of the `Relu` operator on vEU 1. Then a global barrier is inserted to the two vEUs, each runs a barrier-*r*Task: vEU 0 waits for the 4*th* *r*Task on vEU 1, and vEU 1 waits for the 2*nd* *r*Task on vEU 0. Finally, the vDevice executes two *r*Tasks of the `Conv` operator on the two vEUs respectively. On each vEU, Rammer runs the *r*Tasks sequentially in a code branch, executed only if the current vEU Id matches the one generated by the *r*Program.

Before a lengthy DNN computation, Rammer dispatches each vEU (implemented by a PTB) to a desired SM through the GPU scheduler [45]. To improve hardware utilization, an SM can run multiple vEUs (PTBs) concurrently. Since CUDA uses a SIMT model, all vEU are homogeneous, the number of vEUs an SM can support depends on the most demanding *r*Task across all the vEUs, i.e., the *r*Task that requires the most thread number, register number, shared memory size, etc. In practice, we set the number of vEUs on each SM according to the maximum active PTB number provided by the CUDA compiler `nvcc` [5]. With the vDevice abstraction, the optimizations in Rammer become hardware agnostic.

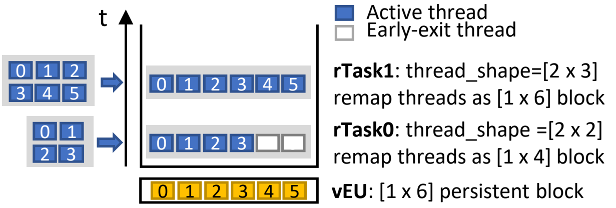
t
■ Active thread
□ Early-exit thread
0 1 2
3 4 5
0 1 2 3 4 5
rTask1: thread_shape=[2 x 3]
remap threads as [1 x 6] block
0 1
2 3
0 1 2 3
rTask0: thread_shape =[2 x 2]
remap threads as [1 x 4] block
0 1 2 3 4 5
vEU: [1 x 6] persistent block

Figure 10: Executing two heterogeneous *r*Tasks on a vEU.

4.1.3 Executing *r*Task on vEU in CUDA

Executing heterogeneous *r*Tasks. In a CUDA kernel, the number of threads in a thread block is fixed in the entire execution lifecycle. This force Rammer to require that all the *r*Tasks on a vEU to run on with the same number of persistent threads. In practice, different *r*Operators may use different number of threads to balance parallelism and per-thread resource usage. To address this problem, Rammer sets the number of threads of a vEU to be the maximum number of threads used by an *r*Task in the vEU. For an *r*Task with less threads, Rammer inserts early-exit logic in the extra threads to skip the unnecessary (and invalid) execution. However, early-exit may lead to dead-lock: a global barrier might never return because early-exit logic may skip the barrier. To avoid this issue, Rammer leverages the CUDA cooperative group primitives [3], which can explicitly control the scope of threads during a synchronization.

Implementing barrier-*r*Task. To implement an efficient barrier-*r*Task, Rammer introduces a step array, where each element is an integer tracking the number of finished *r*Tasks in each vEU. When finished, an *r*Task will use its first thread to increase the corresponding element in the step array by 1. When waiting for a list of *r*Tasks on N vEUs, a barrier-*r*Task uses its first N threads to poll on the corresponding elements in the step array until the steps are larger than the `orders` of those *r*Tasks. After that, the barrier-*r*Task calls `__syncthreads` to ensure all threads in this vEU are ready to run the next *r*Task.

4.1.4 Transforming Legacy CUDA Operators

Many operators for DNN are already available as CUDA kernel code. To reduce development efforts, Rammer introduces a source-to-source converter to transform a legacy CUDA operator into an *r*Operator. The key insight of the converter lies on the facts that to exploit the intra-operator parallelism, legacy CUDA operators are also implemented as thread-blocks, although they use blockIdx and let CUDA GPU hardware control the intra-operator scheduling directly. *r*Operator can just compute the desired blockIdx from rtask_id without changing computation logic in the legacy kernel.

One challenge in this transformation is that the thread-blocks in existing operator could be laid out in 1, 2, or 3-dimensional shape, while in a vEU threads are laid out in a 1-dimensional shape. This means our vEU needs to support *r*Task with different threads shapes. For example, Figure 10 illustrates a vEU executing two *r*Tasks with the thread shapes of $[2 \times 2]$ and $[2 \times 3]$ respectively. Our solution is to stick to a 1-D persistent thread shape for a vEU, and apply a thread index remapping to compute the desired threadIdx in the legacy kernel with the vEU's 1-D threadIdx. Notice that, as discussed before, the number of threads of a vEU is the maximum number of threads of all *r*Task in the vEU, so that such a remapping is always possible. For example, in Figure 10 we configure the vEU with $[1 \times 6]$ persistent threads. When executing *r*Task 0 with a legacy $[2 \times 2]$ thread shape, RAMMER remaps the $[2 \times 2]$ shape to the vEU's $[1 \times 6]$ thread.

In summary, to convert a legacy DNN operator to an *r*Operator, one needs remap thread and block index, implement the early-exit logic, and use CUDA cooperative group primitive to support local barrier on the active (i.e. not early-exited) threads. RAMMER implements these changes by inserting a compiler-generated code segment at the entry point of the legacy operator kernel code. With these modifications, RAMMER can preserve the legacy operator implementation, and reuse it as an *r*Task operator. In RAMMER, we have transformed and implemented total 150 *r*Kernels for 70 *r*Operators.

4.2 RAMMER on Other Accelerators

The design of RAMMER is not limited to CUDA and NVIDIA GPUs. In fact, our *r*Task, *r*Operator and vEU abstractions are applicable to any massively parallel computational devices with homogeneous execution units, including most of the devices that used for DNN computation. In this section we discuss how to port RAMMER to support other devices.

4.2.1 RAMMER on AMD GPUs

AMD GPUs are similar to NVIDIA GPUs, which also consist of many parallel execution units called *compute units (CU)*. AMD GPU has a HIP programming model [7], which is similar to CUDA. AMD provides a hipify tool that can convert a CUDA kernel to a HIP kernel. hipify can help convert most CUDA *r*Operators to the HIP version. Some CUDA kernel configurations, such as the number of threads per thread-block and size of local memory, are not optimized for AMD GPUs due to the minor architecture differences. We re-implemented 41 *r*Kernels for AMD GPUs for better performance. hipify can also convert the CUDA implementation of vDevice (i.e., PTBs) to the HIP version. The only exception is that AMD GPUs do not support cooperative group primitives. To address this issue, we introduce a new API in *r*Operator to provide the number of (block-wise) synchronizations S (i.e. calls to __syncthreads). For early-exit threads, instead of exit immediately, RAMMER will insert code to call the __syncthreads primitive S times.

4.2.2 RAMMER on Graphcore IPU

The Graphcore IPU (Intelligence Processing Unit) [9] is a state-of-the-art DNN accelerator with an architecture quite different from GPUs. IPU is a massively parallel MIMD processor with a bulk-synchronous-parallel (BSP) communication model. Each IPU contains 1,216 parallel processing units called *tiles*; a tile consists of a hyper-threaded computing core plus 256 KB of local memory. DNN computation on an IPU is explicitly programmed as a data-flow graph, where each vertex implements the code executed on a tile and each edge depicts the data transfers between vertices. The IPU compiler is responsible for mapping each vertex to a tile.

RAMMER's *r*Task abstraction can also map to IPU's MIMD model: a vEU can map to a tile and a vertex can be treated as an *r*Task. Thus, an *r*Operator on IPU can be implemented as a set of vertices. More importantly, IPU compiler allows to control the vertex-tile mapping at compile-time. This provides the core functionality required in vDevice abstraction. Restricted by the hardware BSP model, IPU does not provide a fine-grained synchronization mechanism. We therefore implement barrier-*r*Task with a global barrier, which may reduce scheduling space for RAMMER. Even with this limitation, RAMMER still can schedule *r*Tasks of different operators at the same computing step to increase utilization. To evaluate RAMMER, we implemented total 15 *r*Operators and 18 *r*Kernels.

4.2.3 RAMMER on x86 CPUs

We also implemented RAMMER on multi-core x86 CPUs. However, we see little performance benefit of adopting the RAMMER abstractions on x86-based platforms. On x86, the operator runtime is high due to the relatively low performance of x86 cores for numerical computations, and the small number of cores can be fully occupied by almost any DNN operators. Moreover, scheduling overhead is not significant because kernel launch is just a regular function call. Therefore, RAMMER cannot provide additional benefit compared with the traditional two-layered scheduling approach.

5 Evaluation

In this section, we present the detailed evaluation results to demonstrate the effectiveness of RAMMER with comparison with other state-of-the-art frameworks.

5.1 Experimental Setup

Machine environment. We evaluated RAMMER on three servers with different accelerators equipped. The CUDA GPU evaluations use an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB) GPUs, with Ubuntu 16.04, CUDA 10.0 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 2 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 3.1.1 [1]. The IPU evaluations use an Azure ND40s_v3 preview VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare RAMMER with other DNN frameworks and compilers, including TensorFlow (v1.15.2) representing the state-of-the-art DNN framework, TVM (v0.7) [22] and TensorFlow-XLA representing the state-of-the-art DNN compilers, and TensorRT (v7.0) (with TensorFlow integration version), a vendor-specific inference library for NVIDIA GPUs.

Benchmarks and datasets. Our evaluation is performed using a set of representative DNN models that covers typical deep neural architectures such as CNN and RNN; and different application domains including image, NLP and speech. Among them, ResNeXt [46] is an improved version of ResNet [29]; NASNet [51] is a state-of-the-art CNN model obtained by the neural architecture search; AlexNet [34] represents a classic CNN model with a simple architecture. LSTM-TC [30] is an RNN model for text classification; DeepSpeech2 [18] is a representative speech recognition model; and Seq2Seq [43] is for neural machine translation. All the implementations of these benchmarks, including the *r*Kernels used in each model, are available in our artifact evaluation repository[3].

We focus our evaluation on model inference. There is no fundamental reason limiting RAMMER from model training, except that supporting training requires us to develop more operators. We evaluate these models on a set of datasets including CIFAR-10 [2], ImageNet [25], LibriSpeech [10] and synthetic datasets. Table 1 lists the models, hyper-parameters, and the corresponding datasets used. All performance numbers in our experiments are averages over 1,000 runs; in all cases we observed very little variations.

5.2 Evaluation on CUDA GPUs

This section answers the following questions: 1) How does RAMMER perform comparing with the state-of-the-art DNN frameworks or compilers? 2) How well does RAMMER utilize the GPU's parallel resource? 3) How much does RAMMER reduce the runtime scheduling overhead? 4) How much performance gain comes from RAMMER's scheduling leveraging both the intra and inter operator parallelism? 5) How effective is the fine-grained synchronization in improving the overall performance?

5.2.1 End-to-end Performance

We first demonstrate the end-to-end efficiency of RAMMER by comparing with TensorFlow (TF), TensorFlow-XLA (TF-XLA), TVM and TensorRT (TF-TRT). To show the benefit of the abstractions introduced in RAMMER, we create a baseline version of RAMMER (called RAMMERBASE), which only implements the optimizations similar to those in existing compilers and still use a two-layered scheduling approach. Thus, RAMMERBASE can be treated as just another regular DNN compiler implemented in the same codebase of RAMMER. Figure 11 shows the execution time of the benchmarks with batch size of 1.

First, RAMMER significantly outperforms TF by 14.29× on average, and up to 33.94× for the LSTM-TC model. The performance improvement of RAMMER against TF is mainly because TF suffers from heavy runtime scheduling overhead at DFG level, especially when the individual operator's execution time is relatively short, as is the case in small batch inference. TF-XLA, as a DNN compiler, can improve TF's performance through DFG level optimizations (e.g., operator fusion) and operator-level code specializations (e.g. customized kernel generation). However, it still cannot fully avoid scheduling overhead, which leads to an average of 11.25× (up to 20.12×) performance gap compared to RAMMER. We observed that TF-XLA incurs even higher overhead for some CNN models such as ResNeXt and NasNet compared with TF. TVM, as another state-of-the-art DNN compiler, mainly leverages a kernel tuning technique to generate a specialized kernel for each operator. In our evaluation, TVM tunes 1,000 steps and chooses the fastest kernel for each operator. With such specialized optimization, TVM can improve the performance significantly compared with TF and TF-XLA. Still, RAMMER can outperform TVM by 3.48× on average and up to 6.46×. Even though TVM can make individual operator run faster through tuning, it still lacks the capability to leverage the fine-grained parallelism as RAMMER. An exception is that, for AlexNet, RAMMER can only achieve comparable performance with TVM. This is mainly because AlexNet, being one of the earliest modern DNN models, can be easily optimized due to its simple sequential model architecture and relatively fewer, but larger operators. Finally, TensorRT is a specialized DNN inference library with highly optimized operators provided by NVIDIA. We use its official TensorFlow-integration version (TF-TRT) to compile and run our models, as its stand-alone version fails to directly compile these benchmarks. However, for RNN models like DeepSpeech2, LSTM-TC and Seq2Seq-NMT, TF-TRT failed to produce results after compiling for over 50 hours. Thus, we reimplemented these three models with the TensorRT native APIs. Our evaluation shows that RAMMER can outperform

[3] https://github.com/microsoft/nnfusion/tree/osdi20_artifact

Model	Dataset	Model Type	Note
ResNeXt	CIFAR-10	Computer Vision	layers: 29, cardinality: 16 (paper parameter)
NASNet	CIFAR-10	Computer Vision	(paper parameter)
AlexNet	ImageNet	Computer Vision	(paper parameter)
DeepSpeech2	LibriSpeech	Speech	input length: 300; CNN layer: 2; RNN: type: uni-LSTM, layer: 7, hidden size: 256;
LSTM (-TC)	synthetic	Language Model	input length: 100, hidden size: 256, layer: 10

Seq2Seq (-NMT)	synthetic	Language Model	Encoder: input length: 100, type: uni-LSTM, hidden size: 128, layer: 8;
			Decoder: output length: 30, type: uni-LSTM, hidden size: 128, layer: 4;

Table 1: Deep learning models and datasets.

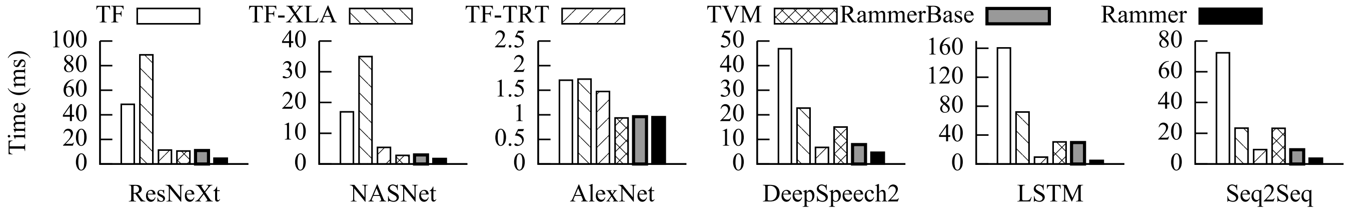

Figure 11: End-to-end execution time on NVIDIA V100 GPU.

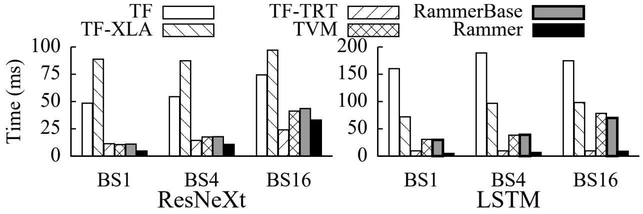

Figure 12: Performance with different batch sizes (BS).

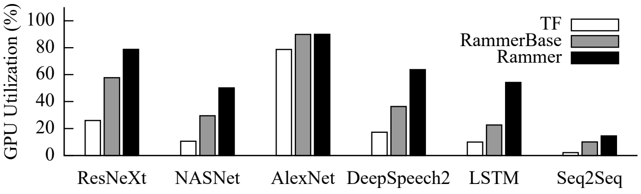

Figure 13: Comparison of GPU utilization.

the vendor optimized TensorRT on all the benchmarks, with an averaged 2.18× and up to 3.09× lower latency. Finally, compared to RammerBase, Rammer can further improve the end-to-end performance by 2.59× and up to 6.29×.

We also evaluate Rammer's performance with larger batch sizes. Figure 12 shows the performance comparison on two representative CNN and RNN models, i.e., ResNeXt and LSTM-TC, with batch sizes of 4 and 16. We limit our benchmarks in this test due to the cost of developing optimized *r*Operator kernels for Rammer: we have to hunt for efficient open-sourced operator kernel implementations or perform tuning by hand or through automatic tuning tools, which is time consuming. As it shows, using larger batch sizes can reduce scheduling overhead in existing frameworks due to the increased per-operator execution time. Even so, Rammer can still outperform all the systems except for TensorRT on the ResNeXt model with batch size of 16. For this case, TensorRT uses some operators whose source codes are not publicly available, and our implementations do not yet match their performances. In fact, implementing operators to match the performance of close-sourced kernels is one of the major challenges for Rammer. Compared to the other open source frameworks and compilers, Rammer has a significant gain. For example, when using batch size of 16, Rammer can outperform TF by 2.25×, and TVM by 1.25× on ResNeXt. For the LSTM-TC model, Rammer can get 20.08× and 9.0× performance gains compared with TF and TVM respectively.

Note that in the above evaluations, RammerBase can already get a comparable or even better performance than compilers like TF-XLA and TVM. Thus, we will use RammerBase as a state-of-the-art compiler baseline to evaluate the benefits of Rammer in the rest of the evaluations. This can help remove the side effects caused by different implementations.

5.2.2 GPU Utilization

Rammer's scheduling enables *r*Tasks from different operators to execute alongside each other to achieve better GPU utilization. We evaluate the utilization improvement by Rammer through comparing it with both TF and RammerBase. Figure 13 shows the average utilization for the 6 DNN models (with batch size of 1) through their execution time. The average GPU utilization only accounts for kernel execution, excluding other stages like operator emitting. Specifically, we use the metric *SM-efficiency* provided by NVIDIA profiler `nvprof` [6] to measure the utilization, which calculates the percentage of time when at least one warp is active on a multiprocessor. Compared to TF, Rammer can significantly improve GPU utilization by 4.18× on average across different models. This improvement comes from both the lower runtime scheduling overhead and the capability to co-schedule operators in Rammer. Through comparing with highly optimized RammerBase, our evaluation shows that Rammer's

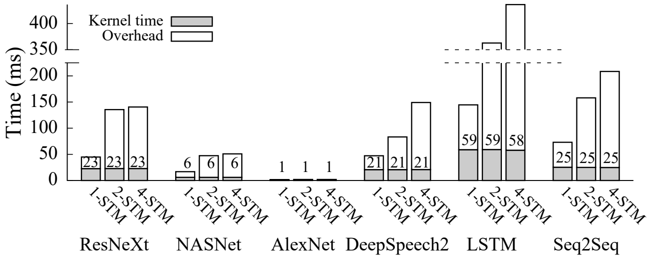

Figure 14: TF performance with different stream(STM) number. (Note: The number atop a bar indicates kernel time.)

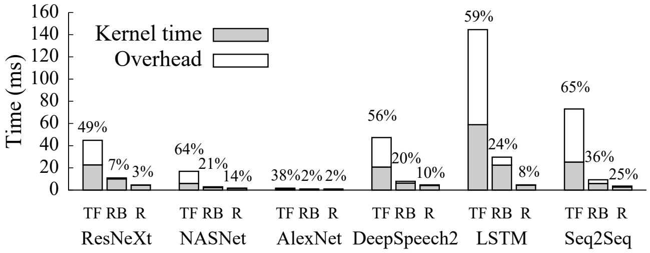

Figure 15: GPU scheduling overhead on different models. (The number atop a bar indicates the overhead in percentage.) TF: TensorFlow, RB: RammerBase, R: Rammer

scheduling by itself can improve the utilization by 1.61× on average, and up to 2.39× for the LSTM-TC model.

As mentioned in §2, modern GPUs support the multi-streaming mechanism to increase utilization through concurrently scheduling independent kernels. We evaluate the efficiency of multi-streaming by increasing the stream numbers in TF. Figure 14 shows both the end-to-end execution time and the kernel time when using stream number of 1, 2, and 4 for each model. We observe that using more streams can harm the end-to-end performance, a phenomenon observed by others [42]. For example, using 4 streams increases the end-to-end time by 2.72× on average compared with using a single stream. Moreover, the kernel time in each model only see very small reduction after enabling multi-streaming, which implies most kernels are still sequentially executed, thus providing little improvement on the GPU utilization. The major reason is because multi-streaming introduces even higher operator scheduling overhead, as shown in Figure 14.

5.2.3 Scheduling Overhead

The techniques proposed by Rammer can effectively reduce scheduling overhead. To verify this, we evaluate the run-time scheduling overhead by comparing Rammer with both TF and RammerBase. Figure 15 shows the total kernel time and the scheduling overhead (i.e., the time not spent on actual computation) for each model. Specifically, compared with TF, RammerBase can reduce the scheduling time from an average of 32.29 milliseconds to only 2.27 milliseconds over all models. RammerBase achieves this reduction by optimizing the scheduling execution code path and leveraging operator fusion to reduce kernel launches. The significant reduction demonstrates the heavy overhead of operator scheduling in existing DNN frameworks. Compared with RammerBase, Rammer can further reduce the average overhead from 2.27 milliseconds to 0.37 milliseconds, a 6.14× reduction. This significant reduction is due to static compile-time operator scheduling, i.e. packing operators into *r*Program so that several operators can be executed by a single GPU kernel launch.

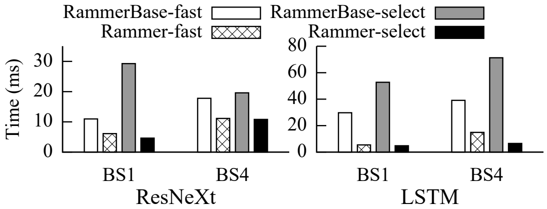

Figure 16: Performance with different kernel sets and batch sizes(BS).

5.2.4 Interplay of Intra and Inter Operator Scheduling

Rammer enables scheduling policies to optimize the interplay of intra and inter operator scheduling, instead of just focusing on making individual operators fast. This is implemented through selecting appropriate *r*Kernel for each *r*Operator, as introduced in §3.3. We evaluate the effect of such scheduling by using two sets of kernels: the fastest kernels only for each individual operator, and the kernels selected by Rammer's scheduling policy. Figure 16 shows the performance of Rammer and RammerBase with these two kernel sets on two representative CNN and RNN models, i.e., ResNeXt and LSTM-TC. First, no matter which set of kernels is used, Rammer can always improve the performance significantly. For example, if Rammer uses the same fastest kernels (i.e., the Rammer-fast) as used in RammerBase (i.e., RammerBase-fast), it can improve the performance by 2.89× on average. If more *r*Kernels are available for a given *r*Operator and Rammer can select kernels based on its policy (i.e., the Rammer-select), it can further improve the end-to-end performance by 1.44× on average, and up to 2.28× compared with Rammer-fast, even though the selected kernels may be not the fastest in isolation. In fact, if we use these kernels in RammerBase (i.e., the RammerBase-select), its performance will drop by 1.84× on average.

We further perform detailed analysis of the kernels used in LSTM-TC model with batch size of 4. For example, for the `Matmul` operator, the fastest kernel uses 1,024 *r*Tasks to get the optimal execution time of 4.28 microseconds; while the selected kernel by Rammer only consists of 16 *r*Tasks and gets a slower execution time of 7.46 microseconds when launched alone. However, Rammer chooses this kernel to trade a slower individual kernel (by reducing intra-operator parallelism) for a better overall performance (through increasing the inter-operator parallelism), thanks to the holistic schedul-

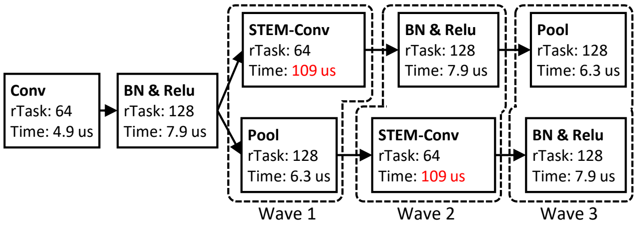

Figure 17: An irregular DFG generated by NASBench

ing capability of RAMMER.

5.2.5 Fine-grained Synchronization

As a synchronization mechanism, barrier-*r*Task provides some extra optimization spaces for the DFGs with irregular structure, which is common in the models generated by neural architecture search (NAS) [51]. To highlight such extra benefit, we leverage NASBench [47], a state-of-the-art NAS benchmark, to randomly generate 5,000 modules, where each module is a small DFG that consists of up to 9 operators and 7 edges. We first compare the end-to-end performance of RAMMER and RAMMERBASE on all these modules, which shows RAMMER can improve the performance by 1.28× on average, and up to 3.40× than RAMMERBASE. Among all these modules, our measurement shows that 28.3% of them has obvious irregular structures, e.g., heterogeneous operators in a wave decided by our policy (Algorithm 1). For these modules, we compare the end-to-end performance of using our barrier-*r*Task implementation and a global barrier. The results show that using the barrier-*r*Task can provide extra performance speedup of 1.11× on average and up to 1.89×. Figure 17 illustrates one of such modules, where the execution time and *r*Task number in each operator are also listed. For such a DFG, our barrier-*r*Task provides a possibility to overlap the execution of operators from different waves (e.g., the two `STEM-Conv` operators from wave 1 and 2) through removing the global barriers between waves and inserting fine-grained *r*Task-level synchronizations.

5.3 Evaluation on Other Accelerators

5.3.1 End-to-end Performance on ROCm GPUs

We evaluate the efficiency of RAMMER on AMD ROCm GPUs by comparing it with TF, TVM, and RAMMERBASE. TF-XLA is not included because it cannot be successfully enabled on AMD GPUs in our experiments, and TensorRT is not included because it is proprietary and is exclusive for NVIDIA. Figure 18 shows the end-to-end performance of the 6 benchmarks with batch size of 1. Compared with TF, RAMMER can outperform it by 13.95× on average, and up to 41.14× for the LSTM-TC model. Compared to TVM, RAMMER can improve the performance by 5.36× on average, and up to 7.57×. Note that we fail to make the TVM auto tuning feature works on ROCm GPUs, so TVM just uses its default kernels in this experiment. Compared with RAMMERBASE, we can see that the proposed scheduling of RAMMER's can bring average of 2.19× and up to 4.12× speedup. Finally, RAMMERBASEK in the figures is exactly the same as RAMMERBASE, except that it uses kernels from RAMMER. Notice that RAMMER might not always choose the fastest kernel implementations for the *r*Operators. Though there are little performance change for most models, for the ResNeXt model there is a 3.02× performance drop. This demonstrates the importance of the interplay of scheduling and kernel selection.

5.3.2 End-to-end Performance on Graphcore IPU

We also conduct a preliminary evaluation of RAMMER on a Graphcore IPU. In this experiment, we choose only the three RNN benchmarks, again, because it takes effort implementing efficient *r*Operators to support other models. Currently, RAMMER only support a single IPU device. We leave the multi-IPU support of RAMMER to future work. For the three RNN models, due to the limited memory available on IPU (256 KB on each tile), we configure the layers of these models to 4 in order to fit in a single IPU. Figure 19 shows the end-to-end performance of RAMMER on these models with batch size of 1. It shows that RAMMER's preliminary implementation can bring up to 5.37× performance improvement compared with RAMMERBASE, which demonstrates the applicability and effectiveness of the abstractions of RAMMER on new accelerator architectures.

6 Discussion

Having shown the advantages, we discuss some RAMMER's limitations and future work in this section.

Performance gain on large batch sizes. RAMMER's benefits are more significant when the intra-operator parallelism is insufficient to saturate hardware. This is the case when the input batch size is small, often found in online DNN inference. Moreover, our preliminary experiment shows that this is also the case for some model training workloads with large batch sizes, such as the LSTM-TC model. Figure 20 shows the training performance of LSTM-TC with batch size of 256. As it shows, with holistic optimizations on both intra- and inter-operator parallelism, RAMMER improves the performance by 2.28× than our baseline implementation RAMMERBASE and 2.36× than TF-XLA. We will leave a more detailed analysis and further optimizations on model training with large batch sizes as our future work.

Dynamic graph. Currently, RAMMER only supports static graph. For DFGs with dynamic control flow [48], RAMMER can compile each static sub-graphs, e.g., a branch of conditionals or a body of loops, into individual rPrograms. We leave this implementation to our future work.

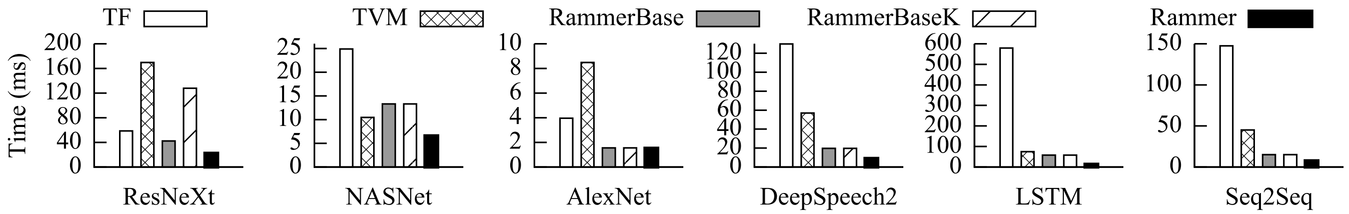

Figure 18: End-to-end execution time on AMD MI50 GPU.

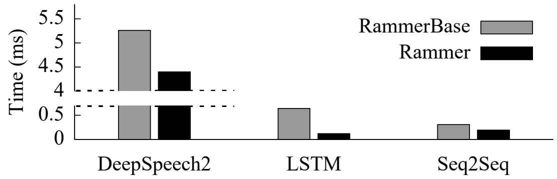

Figure 19: End-to-end execution time on Graphcore IPU.

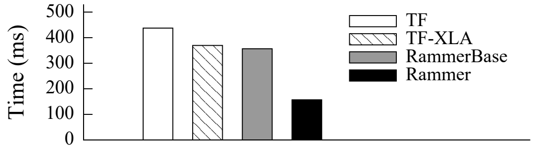

Figure 20: End-to-end model training time of LSTM-TC with batch size of 256 on NVIDIA V100 GPU. Note that TVM and TF-TRT do not support training, hence the data is missing.

Inter-job scheduling. RAMMER focuses on optimizing a single deep learning job and is orthogonal to inter-job scheduling, e.g., through scheduling multiple models in a batch or precisely controlling each job's hardware resource with vDevice. Nevertheless, it is an interesting topic to explore the possibility to co-schedule *r*Tasks not only from different operators, but from different jobs within an accelerator.

7 Related Work

DNN compiler optimization can be generally divided into two classes based on its two-layered representations. DFG-level optimizations, such as operator fusion, are exploited in many DNN frameworks and compilers, e.g., TensorFlow [17], PyTorch [14], TVM [22], XLA [16], etc. TASO [33] proposes an automatic graph substitutions approach to optimize the DFG. On the operator-level, recent work has leveraged different approaches to tune and generate efficient hardware-specific operator code, e.g., AutoTVM [23], Tensor comprehension [44], FlexTensor [50], Tiramisu [20], Halide [40], etc. RAMMER is compatible with all these optimizations through taking an optimized DFG as input and generating efficient *r*Kernels with those kernel generators.

DNN inference and its optimization have attracted a lot of recent attention. DeepCPU [49], BatchMaker [26], GRNN [31], and NeoCPU [38] optimize the inference for RNN or CNN specific models on either CPU or GPUs. Jain et al. [32] proposes to leverage both temporal and spatial multiplexing for multiple inference jobs to improve the GPU utilization. RAMMER differentiates with these works in two aspects: 1) RAMMER can apply to general DNN models and accelerators; and 2) more than just compiler optimizations, RAMMER provides a new abstraction and a larger optimization space for DNN computation. Astra [42] exploits the predictability of DNN to perform online optimization for DNN training, while RAMMER leverages the same property to reduce the individual *r*Task scheduling overhead. There are also many inference systems proposed to optimize the overall throughout under the guaranteed query latency, e.g., Nexus [41], PRETZEL [36], Clipper [24], TF-serving [39], etc. RAMMER instead focuses on optimizing a single model and is orthogonal to these works.

Some other work from the GPU community has proposed software-based schedulers within a GPU to schedule general workload. For example, Juggler [21] proposes a framework to dynamically execute a job represented as a DAG of tasks. Wu et al. [45] proposes a software approach to control the job locality on SMs. However, driven by the property of DNN workload, RAMMER proposes a new computation representation with *r*Task and *r*Operator; and adopts a compile-time scheduling approach to avoid runtime overhead systemically.

8 Conclusion

DNN computation suffers from unnecessary overheads due to the fundamental limitations of existing deep learning frameworks, which adopt a two-layer scheduling design that manages the inter-operator scheduling in the framework and delegates intra-operator scheduling to the hardware accelerator. RAMMER addresses this issue with a holistic compiler solution that (1) provides an *r*Task-operator abstraction that exposes the fine-grained intra-operator parallelism. (2) virtualizes the modern accelerator with parallel execution units to expose the hardware's fine-grained scheduling capability. (3) leverages the predictability of DNN computation to transform run-time scheduling into a problem of generating compile-time *r*Task execution plans. Our evaluations show that RAMMER can achieve significant improvements compared to native deep learning frameworks, compilation frameworks and even vendor-specific inference engine on GPUs. This positions RAMMER as a new enhancement to the existing ecosystem of DNN compiler infrastructure.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Jinyang Li, for their extensive suggestions. We thank Jim Jernigan and Kendall Martin from the Microsoft Grand Central Resources team for the support of GPUs. Fan Yang thanks the late Pearl, his beloved cat, for her faithful companion when writing this paper. This work was partially supported by the National Natural Science Foundation of China under Grant No. 61972004.

References

[1] AMD ROCm Platform. https://github.com/RadeonOpenCompute/ROCm.

[2] CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

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

[4] CUDA Driver API. http://docs.nvidia.com/cuda/cuda-driver-api.

[5] CUDA NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[6] CUDA nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/.

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

[8] Intel MKL-DNN. https://github.com/oneapi-src/oneDNN.

[9] IPU PROGRAMMER'S GUIDE. https://www.graphcore.ai/docs/ipu-programmers-guide.

[10] LibriSpeech ASR corpus . http://www.openslr.org/12/.

[11] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[12] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[13] ONNX. https://onnx.ai/.

[14] PyTorch. https://pytorch.org/.

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

[16] XLA. https://www.tensorflow.org/xla.

[17] 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.

[18] Dario Amodei and Sundaram Ananthanarayanan et al. Deep speech 2 : End-to-end speech recognition in english and mandarin. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20–22 Jun 2016. PMLR.

[19] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[20] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[21] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. Juggler: A dependence-aware task-based execution framework for gpus. *SIGPLAN Not.*, 53(1):54–67, February 2018.

[22] 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.

[23] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in*

Neural Information Processing Systems 31, pages 3389–3400. Curran Associates, Inc., 2018.

[24] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, 2017. USENIX Association.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[26] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *In Proceedings of the Thirteenth EuroSys Conference (EuroSys 18)*, 2018.

[27] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), March 2019.

[28] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, pages 1–14, 2012.

[29] 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.

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

[31] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 41. ACM, 2019.

[32] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *CoRR*, abs/1901.00041, Dec 2018.

[33] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. pages 47–62, 10 2019.

[34] 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.

[35] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, USA, 2006.

[36] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[37] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., USA, 2004.

[38] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1025–1040, USA, 2019. USENIX Association.

[39] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[41] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 909–923, New York, NY, USA, 2019. Association for Computing Machinery.

[43] 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.

[44] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[45] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. ACM.

[46] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *arXiv preprint arXiv:1611.05431*, 2016.

[47] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[48] 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.

[49] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *In 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951––965. USENIX Association, 2018.

[50] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. pages 859–873, 03 2020.

[51] 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.