ACE: Efficient GPU Kernel Concurrency for Input-Dependent Irregular Computational Graphs

Sankeerth Durvasula
University of Toronto
Toronto, Canada
sankeerth.durvasula@mail.utoronto.ca

Junan Zhao
University of Toronto
Toronto, Canada
adrian.zhao@mail.utoronto.ca

Raymond Kiguru
University of Toronto
Toronto, Canada
raymond.kiguru@mail.utoronto.ca

Yushi Guan
University of Toronto
Toronto, Canada
yushi.guan@mail.utoronto.ca

Zhonghan Chen
University of Toronto
Toronto, Canada
zhonghan.chen@mail.utoronto.ca

Nandita Vijaykumar
Vector Institute, University of Toronto
Toronto, Canada
nandita@cs.toronto.edu

Abstract

GPUs are widely used to accelerate many important classes of workloads today. However, in this work, we observe that several important emerging classes of workloads, including simulation engines for deep reinforcement learning and dynamic neural networks, are unable to fully utilize the massive parallelism that GPUs offer. These applications tend to have kernels that are small in size, i.e., have few threads and thread blocks that cannot saturate the GPU's compute resources. Executing independent kernels *concurrently* is a promising approach to improve parallelism and utilization. However, this inter-kernel concurrency is difficult to leverage in such workloads with existing approaches: First, the inter-kernel dependencies and computational graph are input-dependent and vary each time the application is executed. Second, the computational graphs tend to be irregular, requiring fine-grain scheduling and synchronization; thus incurring significant synchronization overheads if kernel execution is parallelized. In this work, we propose ACE, a new framework that enables lightweight detection of inter-kernel dependencies and low overhead kernel scheduling at runtime. The key idea behind ACE is to perform inter-kernel dependency checks for a small window of kernels at runtime, similar to out-of-order instruction scheduling. This enables concurrent execution of kernels in applications whose computational graphs are input-dependent and require fine-grained scheduling. We propose ACE-SW, a software-only open-source implementation of ACE and ACE-HW, a hardware-software cooperative implementation. ACE-HW further reduces synchronization overheads by reducing communication between the CPU and GPU. We evaluate ACE for deep RL simulation engines and dynamic and static DNNs on both real hardware and a GPU simulator. We demonstrate speedups of up to 2.19× (1.56× on average) by improving GPU utilization with concurrent kernel execution.

PACT '24, October 14–16, 2024, Southern California, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676897

CCS Concepts

• **Hardware**; • **Computer systems organization** → *Architectures*;

Keywords

Workload performance Analysis, GPU Architecture

ACM Reference Format:
Sankeerth Durvasula, Junan Zhao, Raymond Kiguru, Yushi Guan, Zhonghan Chen, and Nandita Vijaykumar. 2024. ACE: Efficient GPU Kernel Concurrency for Input-Dependent Irregular Computational Graphs . In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Southern California, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3656019.3676897

1 Introduction

Graphics Processing Units (GPUs) today are commonly used to accelerate a diverse set of applications, such as deep neural network (DNN) processing, scientific computing, graphics, and cryptography. The massive parallelism offered by GPUs enables efficient computations on large amounts of data concurrently. However, we observe that certain important classes of applications, such as simulation engines for deep reinforcement learning (RL) [29, 33, 56, 60, 72] and dynamic neural networks [18, 25, 38, 51, 53, 77, 80, 82, 83, 85–87, 93, 94], are unable to fully utilize the significant compute capability GPUs offer. This is because these applications comprise a large number of small kernels, i.e., kernels with few thread blocks that are unable to fully saturate GPU cores. To understand the challenges in alleviating this underutilization, we evaluate two important classes of applications and introduce their properties.

Simulation Engines for Deep RL. With reinforcement learning (RL), an agent (for example, a robot) learns to perform tasks such as robotic locomotion, manipulation, and navigation [23, 45] by trial and error from interactions with the environment. Deep RL training involves using a DNN to learn policies that optimize for rewards from data collected by simulation. By leveraging the benefits of DNNs, deep RL has recently gained widespread application for many challenging and important tasks [15, 21, 45, 48, 58, 69, 75, 89]. Despite leveraging GPUs, a significant fraction of the deep RL runtime is the data collection phase (up to 70%), where physics simulations are used to generate training data. We observe that these physics simulations contain kernels with few thread blocks and heavily underutilize the GPU, only achieving an occupancy of 34% on average. Programming larger kernels is impractical as each

instance simulates a different scenario, and large kernels would lead to thread divergence.

Dynamic DNNs. Several recent types of DNNs [18, 25, 65, 93] have emerged as a promising approach to reduce inference latencies in resource-constrained devices by re-configuring/specializing the architecture based on the input to the DNN. For example, InstaNAS [25] configures the network architecture at runtime based on the input image. Our evaluations demonstrate that, while these architectures require significantly fewer FLOPs and lower inference latencies, there is still significant underutilization of GPU resources (achieving an occupancy of only 39% on average). Similar to the simulation engines, we find that this underutilization is caused by small kernels that do not fully utilize the GPU cores.

GPU kernels from such applications are typically executed *serially*, and thus the utilization is determined by the size (i.e., the number of threads and thread blocks) of the kernel. However, we observe that many kernels are independent and thus can be executed concurrently. By concurrently executing independent kernels, we can effectively improve GPU utilization and thus performance. Existing GPU architectures allow for concurrent execution of kernels by using multiple command queues [4] which are abstracted in software (such as CUDA Stream [5]), allowing the programmer to identify and launch independent kernels in parallel. However, enabling concurrent kernel execution for these applications is still a challenging task for two major reasons.

Challenge 1: Input-dependent computational graphs. For these applications, the computational graph (i.e. the kernels to be executed and their dependencies) is only resolved at runtime based on the input, and each input or set of inputs leads to a different computational graph. This means that identifying independent kernels to launch in parallel requires performing inter-kernel dependency checks at runtime. These workloads have short running kernels that significantly exacerbate the scheduling and dependency checking overheads, making this a challenging problem to solve. Frameworks such as CUDA Graph [1] and AMD ATMI [2] allow programmers to define the inter-kernel dependency information and construct a directed acyclic graph (DAG) of kernels. These frameworks enable concurrent kernel execution. However, when inter-kernel dependencies vary by input, we must incur the significant latency of constructing the dependency graph and scheduling independent kernels every time the application is executed, significantly increasing run time (§ 2.3 and § 6).

Challenge 2: Irregular inter-kernel dependencies require fine-grain scheduling. The computational graph for a given input tends to be highly irregular. In other words, the kernels cannot be easily partitioned into independent streams and fine-grain scheduling is required to expose inter-kernel parallelism. Thus, parallel execution of kernels requires frequent synchronization to ensure correctness, leading to significant synchronization overheads from communicating with the CPU and from kernel launches (§ 2.3).

To address these challenges, **our goal** in this work is to enable kernel concurrency with *(i)* lightweight scheduling and dependency checking of kernels that can be performed at runtime and *(ii)* low overhead synchronization for scheduling and kernel launch. To this end, we propose ACE, a new framework for Automatic Concurrent Execution with two implementations: *(i)* ACE-SW, a software-only mechanism to enable lightweight kernel scheduling at runtime and *(ii)* ACE-HW: a hardware-software mechanism to further reduce synchronization overheads for efficient kernel concurrency.

The key idea of ACE is to perform dependency checks between sequentially launched kernels within a fixed window at runtime, similar to out-of-order instruction scheduling. We refer to this window as the *scheduling window*. When a kernel is inserted into the scheduling window, the kernels that it is dependent on are identified. As kernels complete execution, kernels in the scheduling window are marked ready based on the identified dependencies. Ready kernels can then be concurrently launched as they have no more dependencies. Since at any given time, only a small set of kernels are scheduled and tracked (instead of the entire computational graph), this approach enables efficient kernel parallelization and scheduling at runtime. To perform dependency checks between kernels, ACE leverages annotations from the application that specify the memory address ranges that are read/written by each kernel. This metadata is then used to identify inter-kernel dependencies at runtime when kernels are inserted into the scheduling window. Compared to prior approaches (§ 3.1), this method alleviates the significant kernel scheduling and dependency-check overheads for kernel parallelization.

ACE-SW implements the above out-of-order runtime kernel scheduling in software as an application runtime system using CUDA streams. ACE-SW however still incurs synchronization overheads from communication with the CPU and kernel launch. On the other hand, ACE-HW implements the out-of-order kernel scheduler in the GPU hardware and can alleviate the synchronization overheads. We propose an efficient implementation of ACE-HW that reduces synchronization and kernel overheads by reducing communication with the CPU. Prior works such as task superscalar [31], carbon [49], TDM [19] and ADM [70] propose similar out-of-order scheduling to leverage irregular parallelism between tasks in CPU multiprocessors. However, the major challenge in CPUs is the latency of runtime dependence checking. The primary bottleneck with GPUs is the latency for launch/signal completion of kernels rather than dependence checking (§ 4.4). ACE addresses this challenge and provides an efficient approach to enable out-of-order kernel scheduling in GPUs.

We demonstrate the effectiveness of ACE in improving GPU utilization and thus performance for physics simulation workloads, a range of dynamic neural networks, as well as static neural networks. We demonstrate an average speedup of up to 1.87× using our software-only approach and up to 2.19× from hardware-software implementation. The major contributions of this work are:

- We identify and characterize GPU underutilization as a result of small kernels in applications with input-dependent irregular computational graphs, e.g., deepRL, dynamic DNNs.
- We introduce ACE, a mechanism that improves GPU utilization by enabling concurrent execution of GPU kernels with a lightweight dependency tracking and scheduling framework.
- We provide an open source software-only implementation of ACE that can be used on real hardware to enable low overhead GPU kernel concurrency.
- We evaluate the effectiveness of ACE-SW and ACE-HW on a range of important GPU applications and demonstrate significant speedups and improved GPU utilization.

2 Motivation

2.1 Case Study 1: Simulation Engines for Deep Reinforcement Learning

Deep Reinforcement Learning (RL) has widely gained attention as a promising approach to learning control policies in robotics and dynamical systems for tasks such as locomotion on legged robots [33, 69, 89], dexterous hand manipulation [23], autonomous driving [21, 45], and drone control [15, 48, 58]. Deep RL involves training a DNN to learn policies that maximize the reward, based on the actions that the agent (e.g., four-legged robot) performs in a given environment. This training process requires data from the agent interacting with a physics simulator. Typically, each training step requires data from thousands of physics simulations. Recent works [29, 33, 35, 56, 60, 72] accelerate this data generation phase by leveraging GPUs. GPUs can accelerate data generation by performing multiple simulations simultaneously and also parallelizing within a single simulation. Hence this makes them an appropriate candidate workload for GPU execution. Despite GPU acceleration, the simulation/data generation phase is still the predominant computation in deep RL—taking about 30 − 70% of training time depending on the complexity of the simulated environment. Thus accelerating simulation engines is critical for deep RL performance.

To evaluate the efficiency of physics simulations, we analyzed a set of physics simulations with different environments on a GPU (parameters in § 5) with the widely used Brax [33] framework. We evaluate the utilization of the GPU by measuring achieved occupancy (average ratio of active warps), depicted in Fig. 1. We find that as much as 65% of the GPU cores are underutilized on average. To evaluate the cause of this underutilization, we analyze the number of kernel launches required to generate one batch of training data in Fig. 2. We also present the average number of CTAs per kernel in Fig. 3 and depict the distribution of kernel sizes observed for the `ant` environment in Fig. 4. We observe that physics simulations in our evaluations generate a large number of *small* kernels that have few threads and CTAs. Each CTA is mapped to a single SM at kernel launch time. Running kernels with fewer than 50 CTAs leads to many idle SMs, leading to underutilization.

This is a fundamental problem, because the simulation engine cannot be efficiently mapped into large kernels, since each thread simulates a different scenario, which leads to high thread divergence. Thus, the application is instead programmed as many short-running kernels. This phenomenon has also been observed by recent works [34, 35]. However, there is still parallelism between kernels that can be leveraged during the collisions detection between every pair of rigid bodies, which can be parallelized.

2.2 Case Study 2: DNNs with Dynamic Graphs

Recent research has extensively investigated specialized DNNs for edge devices with limited compute resources and power budgets as direct deployment of large neural network architectures on these devices leads to high-inference times. Automated DNN architecture design (neural architecture search) is a promising approach to generate faster neural network architectures while retaining or improving accuracy [52, 67, 88, 96]. These optimized architectures tend to have irregular elaborate connections between convolution operations. Fig. 5a depicts an example DNN with irregular structure. Additionally, an emerging trend in recent research [65] shows that *dynamic inference models* [14, 25, 51, 53, 73, 77, 80, 82, 83, 86, 87, 91–94] are very promising to significantly reduce inference latency and FLOPs. With these dynamic inference models, the path of execution through the network is determined by the *input*. Thus, the computational graph is not known ahead of time. For example, Fig. 5b shows an example CNN model with different paths of execution based on the input [25]. Dynamic DNNs have the structure as shown in Fig. 5b, containing operations (MBConv blocks implemented as kernels) that can be parallelized. A subset of these kernels can be scheduled for concurrent execution.

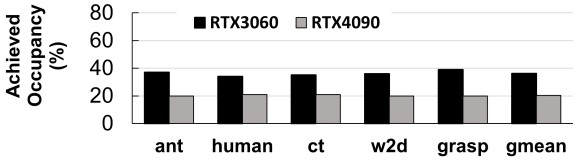


Figure 1: Simulation engines: Achieved occupancy.

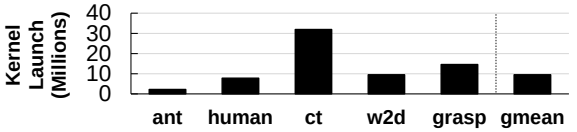


Figure 2: Simulation engines: Kernels for 1 batch of data

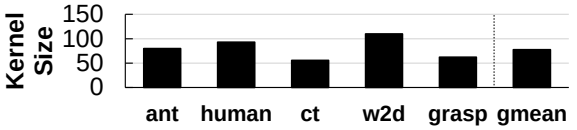


Figure 3: Simulation engines: Average kernel size (in CTAs)

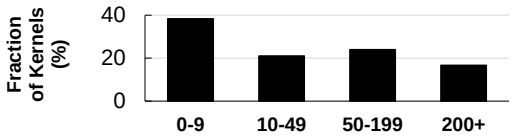
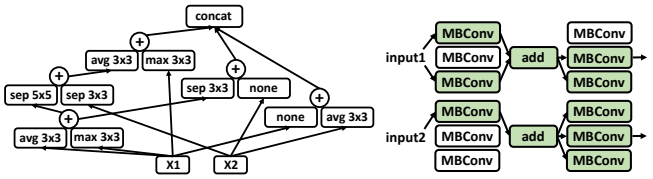


Figure 4: Kernel size distribution for the `ant` environment

Similar to § 2.1, we evaluate the efficiency of these workloads on a GPU (an NVIDIA RTX 3060 and an NVIDIA RTX 4090) and depict the resulting utilization in Fig. 6 (evaluation and workload settings are in § 5). We find that the total achieved occupancy is around 39% in the InstaNAS-A [25] workload on RTX 3060. We observe a higher degree of underutilization on a more recent GPU, with a higher SM count. Similar to the simulation engines, we root cause this underutilization to the existence of a large number of small kernels, as depicted in Fig. 7, where a large fraction of the kernels have fewer than 200 CTAs. Thus, these small kernels are unable to fully utilize the GPU. In these workloads, the small kernels are due to convolution layers that were optimized for fewer FLOPs with smaller filters.



(a) Amoebanet [67] (b) InstaNAS [25] with MB-Conv [71] units

Figure 5: DNNs with irregular or dynamic structures

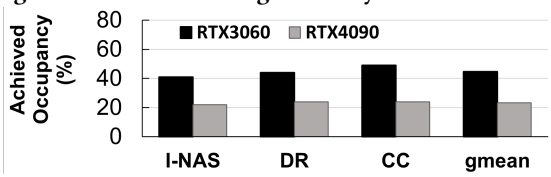


Figure 6: Achieved occupancy in dynamic DNNs

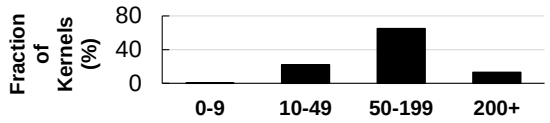


Figure 7: Kernel size distribution (CTAs) for InstaNAS-A [25]

2.3 Key Observations

While small-sized kernels lead to underutilization, we observe that there are typically many kernels that can be executed *concurrently*. Thus we can improve GPU utilization and reduce runtimes by identifying independent kernels and scheduling them for concurrent execution. However, this is a challenging task for these classes of applications for the following reasons.

(1) Input-dependent kernel dependencies. The computational graph, and hence, the dependencies between kernels are only determined at *runtime* for each input. For example, with the instance-aware dynamic DNNs [25, 53, 93, 94] described in § 2.2, for the classification inference task, the computational graph is different for each image. As a result, the determination of kernel dependencies and scheduling of kernels for the entire computational graph needs to be done for *each input*. This adds significant latencies to the runtime.

CUDA Graphs [1] and AMD ATMI [2] are software frameworks that allow developers to specify dependencies between different kernels as edges of a directed acyclic graph (DAG). The challenge with this approach is that the DAG needs to be constructed in full (with dependencies, kernel launches, and barriers determined) before the application is executed on the GPU, *for each input*. This process adds high latency in compiling the complete dependency information. We perform an experiment to measure the DAG construction and launch time on Brax [33] simulation engine (§ 5) compared to the program execution time, shown in Fig. 8. We observe that the time taken to construct the graph is exceedingly high (average of 47% of overall execution time).

(2) Irregular kernel dependencies. These classes of applications have *irregular* computational graphs that are challenging to easily partition into CUDA streams (§ 2.2). Popular deep learning frameworks [8, 59] use a single stream by default. The stream abstraction works best if the entire graph can be partitioned into independent streams of kernels. However, these graphs with irregular dependencies would require fine-grained scheduling and

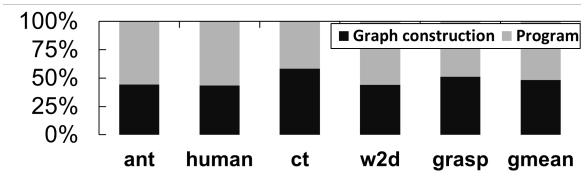


Figure 8: DAG construction time as % of execution time

heavy use of synchronization (e.g., cudaDeviceSynchronize and cudaStreamSynchronize) when parallelizing using CUDA streams. This synchronization may lead to large overheads as it requires communication between the GPU and CPU. Fig. 9 depicts the different overheads when CUDA streams are used for fine-grained scheduling with irregular graphs: kernel launch overheads ❶, CPU execution overheads ❷ and the synchronization overheads ❸. Based on our profiling, the synchronization and launch overheads vary between 5-20*us*.

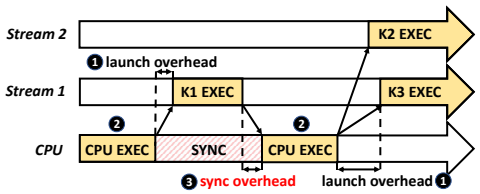


Figure 9: Kernel launch and synchronization overheads

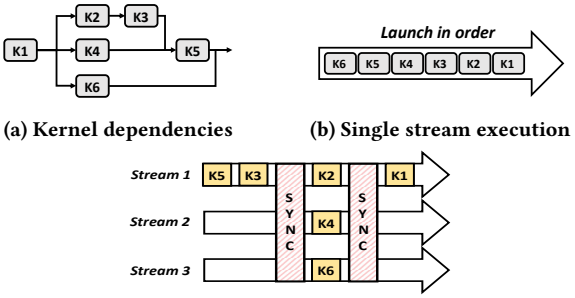
3 Approach

Our **goal** in this work is to design a framework that enables efficient concurrent execution of GPU kernels *(i)* whose computational graph may only be known at runtime, *(ii)* without incurring significant synchronization overheads. To this end, we introduce ACE, a new framework that concurrently schedules independent kernels with a lightweight runtime mechanism.

3.1 Prior Mechanisms

We consider the baseline hardware model in modern GPU architectures [64]. The host communicates with the command processor (CP) of the GPU via a virtual memory region mapped to the GPU that is accessible by the command processor. This enables communication between the CPU and GPU via the command queue. The CPU transmits kernel launch packets to the GPU by writing them to the user mode command queue. The CP is responsible for decoding and dispatching the kernels in these command queues for execution. The CP accesses the command queue and schedules the kernels at the head for execution. The GPU runtime may launch kernels into different streams. These streams are mapped to one of the command queues in the device-mapped memory. The command processor schedules kernels at the head of these queues concurrently, thus enabling concurrent kernel execution. However, neither the command processor nor the kernel launch packets in the command queues have information on inter-kernel data dependencies. Kernels in different queues are assumed to be independent of each other, and kernels in the same queue are executed in order. Hence, to leverage parallelism in kernel executions, the task of checking inter-kernel dependencies and determining the kernels eligible for concurrent

execution *must be done by the host application.* However, this adds significant dependency-checking latency at the run time. It also requires communication with the host (through a synchronization routine) to be performed each time a kernel completes execution, adding to overhead. Several prior works describe approaches to efficiently schedule kernels into multiple streams. Fig. 10 depicts approaches to scheduling a computational graph (Fig. 10a). Fig. 10b is the baseline approach used by many existing frameworks [8, 59], where a single stream is used to execute all kernels serially. This approach leads to underutilization (§ 2.2). Fig. 10c shows prior works [30, 50] that use the computational graph to identify independent kernels and the *entire graph* is scheduled ahead of time into multiple streams. However, this fine-grained scheduling and synchronization leads to large overheads.



(a) Kernel dependencies **(b) Single stream execution**

(c) Multiple streams with synchronization between streams

Figure 10: Scheduling kernels in a computational graph

One way to avoid using device-level synchronizations and enable asynchronous kernel execution without communication with the CPU is to use CUDA events. Events serve as signaling mechanisms to indicate the occurrence of specific operations in a stream. This allows synchronization between kernels across streams through the cudaStreamWaitEvent API, facilitating asynchronous kernel execution without blocking the host. By strategically placing events and using cudaStreamWaitEvent, it is possible to orchestrate the order in which kernels are executed on the GPU without communication with host. However, this approach still requires deriving dependencies between all kernels beforehand, incurring overhead.

Another set of approaches [22, 50, 57], define static dependencies between kernels as a DAG, which is then scheduled with DAG frameworks (CUDA Graph [1]/ATMI [2]). These approaches cannot be applied to input-dependent computation graphs, as constructing the entire computational graph is too time-consuming to be done at runtime. To convey the DAG information, ATMI sends barrier packets [63] along with kernel launch packets to the command queue. A barrier packet [40] is a 64-byte data packet that contains id information about a kernel and a set of kernels that depend on it. This packet can be inserted into the command queue by the device runtime. The barrier packet blocks the launching of dependent kernels until the independent kernel completes execution. The barrier packet however does not contain any information regarding the current status of the executing kernels in the GPU and thus cannot perform any additional runtime reordering of kernels. It simply follows the dependencies already specified by the DAG. While it is possible to devise a framework that dynamically launches barrier packets and launch commands onto the GPU command queue in memory, this would require hardware support and would still incur synchronization overheads with the CPU. Our approach is specifically designed to mitigate this scheduling cost by avoiding direct communication from the GPU to the CPU, thereby reducing potential overheads. Persistent threads (PT) eliminate the scheduling and launch overheads but are only effective when all kernels are homogeneous [24]. CUDA dynamic parallelism [3] (CDP) or AMD's device enqueue [7] (DE) enables parent kernels to launch child kernels, only allowing data dependencies between one parent and its children. These workloads however involve kernels that depend on multiple kernels, and it is an open problem how to use CDP for these types of dependencies.

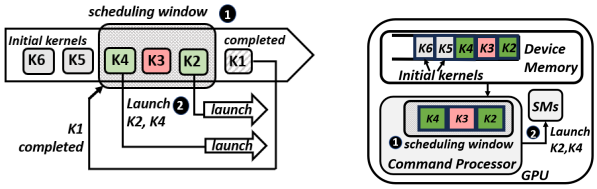
We summarize different approaches for parallel kernel scheduling in Table 1, in terms of applicability (whether input-dependent irregular workloads can be effectively mapped), synchronization/launch overheads and preparation overhead (resolving dependencies, constructing, and scheduling the computational graph).

Method	Applicability	Sync+Launch Overhead	Preparation Overhead
Multi-Stream [30, 50]	✓	x	✓
DAG Frameworks [1, 2]	✓	✓	x
PT [16, 24, 78]	x	✓	✓
CDP [3] / DE [7]	x	x	✓
ACE-SW (Our approach)	✓	x	✓
ACE-HW (Our approach)	✓	✓	✓

Table 1: ACE compared to other scheduling frameworks

3.2 Key Idea of ACE

With ACE, the key idea is to instead perform the dependence checking and scheduling within a small window of kernels at *runtime* similar to out-of-order instruction scheduling. We perform this scheduling over a single command queue (or a single initialized stream). Fig. 11a depicts out-of-order kernel dispatch with ACE. Fig. 11b shows the corresponding high-level hardware modifications for ACE. A fixed number of kernels in the original stream (scheduling window ❶) are evaluated for dependencies. When a kernel completes execution, we evaluate which kernels within the scheduling window are now ready for execution ❷. All such kernels are marked ready and can be scheduled concurrently.



(a) Out-of-order kernel dispatch from the scheduling window **(b) CP scheduling kernels in out of order manner**

Figure 11: ACE: Runtime out-of-order kernel scheduling

We propose two implementations of ACE: ACE-SW, a SW-only approach and ACE-HW, a hardware-software cooperative mechanism,

which we describe in the following sections. ACE-SW emulates the out-of-order kernel scheduling mechanism by scheduling independent kernels into multiple streams and can be implemented with purely software changes, however the hardware support in ACE-HW is more efficient as it alleviates synchronization overheads.

3.3 Design Overview

To design ACE to perform the runtime kernel scheduling as depicted in Fig. 11a, we need *(i)* a mechanism to determine inter-kernel dependencies in the scheduling window; *(ii)* to identify kernels that are ready for execution; and *(iii)* alleviate synchronization and kernel launch overheads.

Determining inter-kernel dependencies. In order to determine dependencies between kernels, the application adds additional metadata to each kernel invocation. This metadata defines the range of global memory addresses that are written to and read from by each kernel. This metadata is provided to ACE by using a kernel wrapper (described in § 4.2) and can be defined by the programmer, library-writer, or compilation tools. By checking for overlaps between read segments and write segments, we determine dependencies between kernels. The kernel wrapper defines the pointers to the read and write data segments (`start_addr`) along with the size of the segments (Fig. 12). The actual virtual addresses associated with the pointers are resolved just before kernel launch in order to perform the dependence checks (§ 4.1). We refer to these memory ranges as `read_segments` and `write_segments`. The runtime software performs dependency checks between a new kernel and a set of kernels launched earlier within a window. This information is utilized by the hardware to identify independent kernels and schedule them. Identifying dependencies and communicating them to the GPU is done by the software runtime while kernels are executing. Fast dependency checks and communication to GPU can be hidden by GPU kernel execution. As fast dependency checks can be performed in software, and scheduling independent kernels for execution can be done without needing CPU communication by the hardware, the task of dependency checking is delegated to the runtime and scheduling is delegated to hardware.

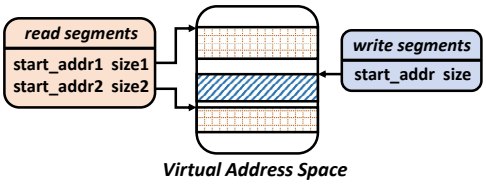


Figure 12: Memory regions written to/accessed by the kernel

Tracking kernel state at runtime. Fig. 13 depicts the scheduling window (❶), with the additional state required for scheduling. The kernels in the window can be `ready`, `pending`, or `executing` (❸). Kernels in the scheduling window become ready for launch (`ready`) when the kernels it is dependent on (referred to as *upstream* kernels ❷) complete execution. For each kernel in the scheduling window, we track a list of the corresponding upstream kernels. The upstream kernels are determined using the above dependency checks when inserting into the scheduling window. When the upstream list is empty, the kernel is marked `ready` for execution. After each kernel completes execution, the upstream list is updated for all kernels in the scheduling window. For ACE-SW, these checks are performed in the software runtime system (§ 4.2), and for ACE-HW, we implement them in hardware (§ 4.3).

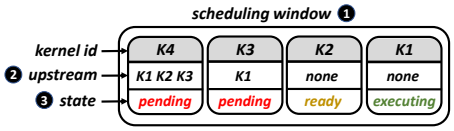


Figure 13: Kernels in the scheduling window with their state and corresponding upstream kernels (i.e., dependencies)

Eliminating CPU synchronization overheads. In order to eliminate synchronization and launch overheads resulting from communication between the CPU and GPU, we implement the scheduling window in the GPU hardware in ACE-HW. The management of the scheduling window is done entirely in hardware, including the determination of `ready` kernels. Similarly, once a kernel completes execution, the scheduling window is updated without requiring synchronization with the CPU.

3.4 Mechanism Walkthrough

Fig. 14 depicts a high level walkthrough of ACE. For each GPU kernel invoked by the application ❶, the read and write segments are resolved (detailed in § 4.1). All invoked kernels along with the corresponding read/write segments are entered into the input FIFO queue to await scheduling ❷. Kernels are then added to the fixed size scheduling window in a FIFO manner ❸. When the kernel enters the scheduling window ❹, the write segments of the current kernel are compared against read and write segments of all kernels in the scheduling window. The kernels with overlap are added to the corresponding upstream kernel list and are marked `pending`. When an `executing` kernel completes execution, all corresponding upstream kernel lists are updated. Any kernel that has an empty list is marked `ready` for the scheduler to launch.

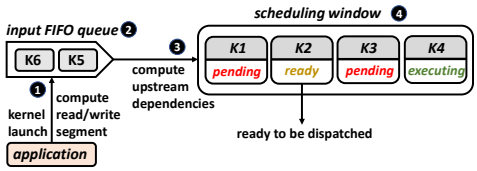


Figure 14: High level overview of ACE

4 Detailed Design

4.1 ACE Kernel Wrappers

To perform runtime dependency checks, the application defines the read/write segments for each kernel. These segments are defined using a kernel wrapper, `ACE_wrapper` (defined in Fig. 15). Since virtual addresses can only be resolved at runtime, the programmer instead defines a function `get_addresses` which populates `__read_segments__` and `__write_segments__` lists (lines 6 and 7 in Fig. 15). The `get_addresses` function takes kernel launch arguments as the input (lines 12 to 15). These arguments are then used to compute the read/write segments.

Just before kernel launch, the CUDA runtime calls `get_addresses`. At this point, `__read_segments__` and `__write_segments__` are populated with resolved virtual addresses. In our implementation of ACE-SW, since the CUDA drivers are closed-source, we implement an intermediate user-level kernel launch function that calls `get_addresses` instead. Fig. 16 depicts an example implementation of the `get_addresses` function. ACE assumes that the programmer or the kernel library provider has knowledge of the memory regions accessed by the kernel from the kernel function prototype. For a wide range of commonly used kernels, such as matrix multiplication, convolution, addition, etc., which operate on data stored as contiguous regions in memory, this task is straightforward. ACE does not require additional programming effort from the application programmer. This is because programmer annotations for these workloads would be written by the library writer and thus, not necessarily the programmer. Standard functions/kernels along with the annotations are provided as metadata in the library.

```
struct ACE_wrapper {
   //list of read,write segments defined as
   //[{start_adr1,size1},{start_adr2,size2}..]
   list __read_segments__;
   list __write_segments__;
   // function which gets called at kernel
   // launch to populate read,write segments
   void get_addresses(
        dim3 blocks, dim3 threads, ...
   );
   // function declaration of the kernel
   static __global__ void kernel(...);
};
```

Figure 15: The `ACE_wrapper` definition

```
// get address function for matrix multiply
// input matrices: input1 (mxn), input2(nxk)
// output matrix: output(mxk)
void ACE_wrapper::get_addresses(
    dim3 blocks, dim3 threads,
    int* input1, int* input2, int* output1,
    int m, int n, int k) {
   // input1 reads m*n elements
   // input2 reads n*k elements
   __read_segments__ = {
       {(void*)input1, m*n*sizeof(int)},
       {(void*)input2, n*k*sizeof(int)}
   };
   // output reads m*k elements
   __write_segments__ = {
       {(void*)output, m*k*sizeof(int)},
   };
}
```

Figure 16: Example: `get_addresses` function

4.2 ACE-SW Design

ACE-SW is implemented as a user-level runtime that is called by the application. The functionalities of ACE-SW are performed by multiple independent threads that are launched simultaneously. The ACE-SW runtime performs two major tasks: *(i)* implementing and maintaining the scheduling window (window module); and *(ii)* scheduling kernels ready for execution (scheduling module).

4.2.1 The window module. The window module is implemented as a separate thread that manages the input FIFO queue and the scheduling window. Scheduling window, dependency tracking, and state management are performed in software within this module. This module is called in two ways: First, when a kernel is invoked by the application thread, this module inserts kernels into the input queue. Second, the scheduler module (implemented as a separate thread(s)) calls the window module when a kernel completes execution. At this point, the state of upstream lists is updated and the kernel is removed from the scheduling window. The window module constantly polls the input queue and the scheduling window. When there is a vacancy in the scheduling window and a pending kernel in the input queue, the kernel is moved into the scheduling window. At this point, the window module performs the necessary dependency checks and bookkeeping. Algorithm 1 describes how the dependency check is performed.

Algorithm 1 Dependency check algorithm

Input: $rslist_1$, $wslist_1$, $wslist_2$ ▷ RW segments of scheduling window kernel, w-segment of kernel in inputFIFO
Output: $is\_dependent$ ▷

1: $is\_dependent = false$ ▷ initial state of is_dependent
2: $rwslist_1 \leftarrow wslist_1 \bigcup rslist_1$ ▷ Read+Write segments
3: **for each** $segment_1$ in $rwslist_1$ **do** ▷ Test for every pair of segments
4: **for each** ws_2 in $wslist_2$ **do**
 ▷ get start and end virtual memory addresses
5: $start_1 \leftarrow segment_1.start$
6: $end_1 \leftarrow segment_1.start + segment_1.size$
7: $start_2 \leftarrow ws_2.start$
8: $end_2 \leftarrow ws_2.start + ws_2.size$
9: **if** $start_1 < end_2$ and $end_1 > start_2$ **then**
10: $is\_dependent = true$ ▷ check start&end address overlaps
11: **end if**
12: **end for each**
13: **end for each**

Algorithm 2 The scheduler module in software

Input: SchedulingWindow SW, $stream\_id$

1: **while** $notstop()$ **do** ▷ poll for kernels until stop signal
2: ACQUIRE_LOCK(SW)
3: **if** $SW.ready.exists()$**then** ▷ check ready kernels
4: $kernel \leftarrow SW.ready.pop()$ ▷ get ready kernel
5: **end if**
6: RELEASE_LOCK(SW)
7: LAUNCH(kernel, stream_id) ▷ launch kernel
8: STREAM_SYNC(stream_id) ▷ wait for completion
9: **end while**

4.2.2 The scheduler module. This module schedules and launches `ready` kernels for execution. This module is implemented as a configurable fixed number of threads, each of which launches kernels into an independent CUDA stream for concurrent execution, as depicted in Fig. 17. Each stream contains only one kernel at any given time. Threads with empty streams poll the scheduling window for a `ready` kernel ❶, which is then launched in its CUDA stream ❷. The thread then waits for the kernel to complete execution using the StreamSync primitive ❸. Once the kernel completes execution, the thread calls the window module as described above. This algorithm is described in Algorithm 2.

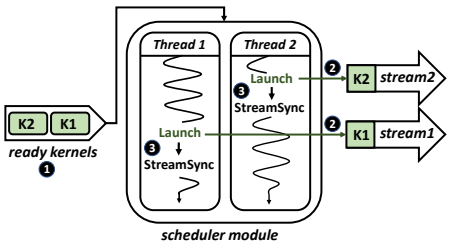


Figure 17: ACE-SW: The scheduler module

4.3 ACE-HW Design

While ACE-SW enables concurrent kernel execution and can be fully realized in software, it still incurs overheads from *(i)* synchronization with the CPU when a kernel completes execution, i.e., the StreamSync primitive that blocks the scheduler module thread; and *(ii)* the launch overhead when the scheduler module launches a kernel. ACE-HW is designed to alleviate these overheads with hardware support for kernel scheduling in the GPU.

Fig. 18 depicts an overview of ACE-HW. ACE-HW comprises a software runtime system similar to ACE-SW that maintains an input FIFO queue containing the kernels that were invoked by the application ❶. The scheduling window and its management are however implemented in hardware on the GPU side ❷. The input queue is essentially implemented as a CUDA stream that dispatches kernels to the GPU. In addition to the input FIFO queue, the software runtime also maintains a list of kernels in the GPU's scheduling window, which we call the `scheduled_list` ❸. To avoid frequent synchronization between the CPU and GPU, we allow this list to be stale. Before a kernel is inserted into the scheduling window, the software runtime performs dependency checks with the `scheduled_list` to determine the upstream kernels. Note that since the `scheduled_list` may be stale, this upstream list needs to be further updated before insertion into the scheduling window (discussed below).

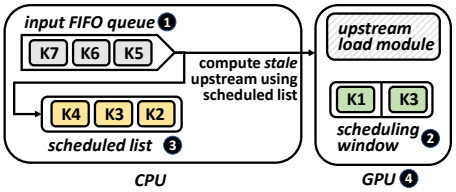


Figure 18: ACE-HW: Design overview

The hardware component ❹ consists of two modules: *(i)* the scheduling window and *(ii)* the upstream load module.

The hardware scheduling window structure is depicted in Fig. 19 and comprises a fixed number of slots (N) ❶. Each slot contains an 8-bit kernel identifier and (N-1) 8-bit upstream kernel identifiers that are implemented with SRAM ❷. Each slot of the SRAM module is implemented as a single bank of SRAM, containing N-1 fully associated units to store upstream kernel identifiers. These upstream identifiers are used to determine when a kernel is `ready`. An additional two bits are used to identify the state of each kernel (i.e., `ready`, `pending`, and `executing`). When a kernel completes execution, the upstream identifiers are updated and corresponding state of each kernel is updated. The completed kernel is removed from the scheduling window. Any kernels that are now `ready` are dispatched to the GPU's kernel dispatch unit for execution ❸.

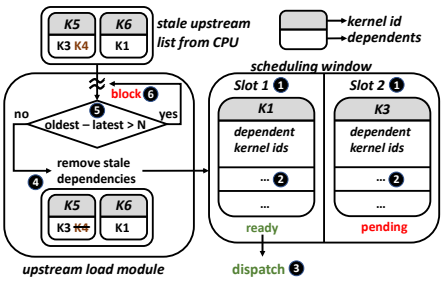


Figure 19: HW scheduling window and upstream load module

The upstream load module is responsible for refining the upstream list provided by the CPU, which may be stale in two ways. It may contain kernels that have (1) already completed execution and (2) may miss long-running kernels that are still executing. The first case is handled by the upstream module by checking against a list of kernels in the scheduling window ❹. The second case is avoided by ensuring that the `scheduled_list` (of size M) in the CPU never misses kernels that are still executing. The upstream load module tracks the oldest scheduled kernel ❺. If the number of newer kernels exceeds M (size of the `scheduled_list`), this module blocks the insertion of more kernels from the CPU ❻.

4.4 ACE Overheads

(1) Hardware area overhead. ACE-HW introduces the hardware scheduling window which contains N slots, where N is the size of the scheduling window. Each slot contains N kernel ids of upstream data of 8 bytes each and 2 bits for status. Assuming a scheduling window of length $N = 32$, we require 1KB of SRAM for the scheduling module (for the entire GPU). The upstream module keeps track of the oldest executing kernel with an 8-bit unit of memory in SRAM, and the number of kernels scheduled or completed execution since this oldest executing kernel (❺ in Fig. 19).

(2) Storage overheads. The read and write segments that are saved as metadata in the input FIFO and the `scheduled_list` by the software runtime in the CPU utilize memory. Each read, write segment requires 48 bits to hold the start addresses and the size.

(3) Mechanism latencies. ACE-HW requires updating all upstream kernels in each slot of the scheduling window every time a kernel completes execution. ACE-HW updates each slot in N-1 cycles (where N is the size of the scheduling window). Additionally, ACE-HW requires N cycles to insert a kernel ID with its upstream kernel IDs into the scheduling window. For a scheduling window of size 64, this operation adds 64 cycles (about 50-100ns) overhead to dispatch a `ready` kernel for launch. Thus, ACE-HW adds negligible runtime to the application compared to the baseline kernel launch overhead (in the order of a few microseconds).

(4) Dependency checking overheads To determine the list of upstream kernels, the CPU checks for overlaps between the write segments of the kernel in the input queue and the read-write segments of the kernels in the `scheduled_list`. As the `scheduled_list` can fit completely into the cache (4KB), dependency-checking is compute-bound and dependent on the

number of read and write segments. Table 2 presents the time required to do dependence checking. For a processor with P execution units, effective utilization requires dependency checks to be performed in no more than T/P, where T is the task execution time [19, 31]. We estimate T/P to be around $4us$, which is more than the dependency check latency.

Window size	Number of RW-segments	Dependency check time
16	6	410ns
	10	700ns
32	6	510ns
	10	1640ns

Table 2: Dependency checking overhead analysis

5 Methodology

We evaluate ACE-SW on real hardware with an Intel 11700K CPU and an NVIDIA RTX3060 GPU. We model ACE-HW using Accel-Sim [46], configured with parameters of RTX3070 (Table 3) and an RTX 4090 GPU (Table 4). We set the scheduling window size to 32.

Shader core 46 SMs, 1.4GHz; 4 schedulers per SM
SM Resources 32768 Registers, 32KB Shared memory, 128KB L1D
DRAM 2-channel; 16-bank; open-row policy, 16GB DDR4

Table 3: Simulated GPU configuration 1

Shader core 128 SMs, 1.4GHz; 4 schedulers per SM
SM Resources 32768 Registers, 128KB shared memory+ L1D
DRAM 2-channel; 16-bank; open-row policy, 24GB DDR4

Table 4: Simulated GPU configuration 2

Workloads. We evaluate ACE using:

(1) Deep RL physics simulations. Brax [33] is a GPU accelerated simulation engine for control tasks in reinforcement learning. We evaluate ACE with the Ant (`ant`), Grasp (`grasp`), Humanoid (`human`), Cheetah (`ct`), and Walker2d (`w2d`) simulation environments. These environments are MuJoCo [81] simulations for training RL agents to perform a specific task. For example, `ant` contains a robot (agent) with one body and 4 legs, each with a knee joint. The goal is to move in a particular direction by controlling its legs.

(2) Dynamic DNNs. We evaluate our approach for 3 dynamic DNN workloads: InstaNAS[25] (`I-NAS`) is a dynamic CNN for image classification. We evaluate our approach using the InstaNAS-A architecture on the CIFAR10 dataset. Dynamic routing [18] (`DR`) is a DNN for image semantic segmentation. We evaluate ACE on the Dynamic-A 16 layer architecture using Cityscapes dataset [27]. Conditional Convolution [91] (`CC`) is a mixture-of-experts CNN model for image classification where the weights are computed at runtime. We evaluate Conditional Convolution with 4 experts that use an efficientnet b4[79] network as backbone. All dynamic DNNs are designed for a batch size of 1 and the input image defines the DNN execution. We use Pytorch [59] implementations.

(3) Static DNNs. CNNs optimized for low inference latency using neural architecture search (NAS): NASNet [96] (`NASNet`), AmoebaNet [67] (`Amoeba`), SqueezeNet [41] (`Squeeze`), and RandomWire [88] (`RW`). These CNNs have highly irregular structures with many small kernels. We evaluate ACE with a batch size of 1.

6 Evaluation

We evaluate ACE using the following configurations: *(i)* Baseline: cuDNN implementation (for DNNs) and jax implementation [33] (for deep RL simulation). *(ii)* `ACE-SW`: Our software-only mechanism is evaluated on real hardware. *(iii)* `ACE-SW-Sim`: Our software-only mechanism evaluated on the GPU simulator with configurations close to RTX3060. We include this to compare against `ACE-HW`. *(iv)* `ACE-HW`: Our hardware-software cooperative mechanism evaluated on the GPU simulator with configuration 1 (Table 3). *(v)* `ACE-SW-SimC2`: Our software-only mechanism evaluated on the GPU simulator with configuration as in Table 4. (vi) `ACE-HWC2` Our hardware-software cooperative mechanism evaluated on the GPU simulator with configuration 2 (Table 4). *(vii)* `CUDAGraph`: Framework where the inter-kernel dependencies are generated on the CPU and sent to the GPU. We only present `ACE-SW` results for the deep RL workloads as the dynamic and static DNNs rely on cuDNN libraries, which cannot be modified to use different streams.

6.1 Deep RL Physics Simulations

Fig. 20 depicts the runtimes for the generation of a single batch of training data from different simulation environments using `ACE-SW`, normalized to the baseline approach.

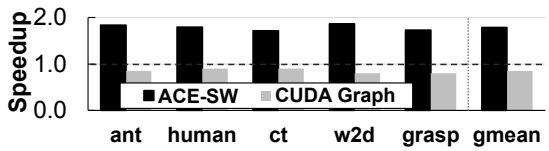


Figure 20: Deep RL physics simulations: Normalized Speedup

Fig. 21 depicts the runtimes for `ACE-SW-Sim`, `ACE-HW`, `ACE-SW-SimC2`, `ACE-HWC2` normalized to the baseline implementation. We make two observations. First, `ACE-SW-Sim`, provides similar speedups as in real hardware compared to the baseline implementation (up to 1.79× and 1.66× on average). Second, `ACE-HW` is able to further improve performance compared to the software-only approach by alleviating the synchronization and kernel launch overheads. We observe a slowdown with `CUDAGraph` due to the significant latency of constructing the kernel dependency graph and sending the information to the GPU.

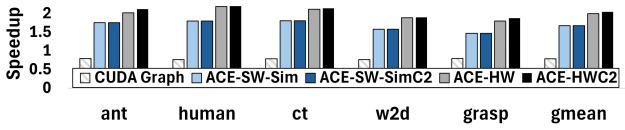


Figure 21: Deep RL physics simulations: Normalized speedup

In Fig. 22, we depict the achieved occupancy for the three configurations. Achieved occupancy is calculated as the number of active warps divided by the maximum number of active warps supported by the GPU averaged over all clock cycles. We observe that the ACE is able to significantly increase the achieved occupancy and thus the utilization. `ACE-HW` reports higher achieved occupancy because `ACE-SW` incurs additional costs before the launch of a new kernel (launch + synchronization time), during which there is a lower

number of active warps executing in GPU compared to ACE-HW. These gaps in launch times contribute to the overall decrease in achieved occupancy in ACE-SW.

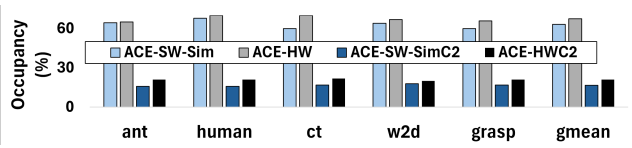


Figure 22: Deep RL physics simulations: Achieved occupancy

6.2 Inference on Dynamic DNNs

Fig. 23 depicts speedup over the baseline for the dynamic DNNs described in § 5. We observe that ACE is able to provide speedups of up to 1.39× on dynamic DNN workloads with ACE-HW and on average 1.05× with ACE-SW and 1.3× with ACE-HW (average 1.07× with ACE-SW-SimC2 and 1.32× with ACE-HWC2). I-NAS suffers a slowdown with ACE-SW because this workload has significant kernel launch overheads when parallelized but are hidden in the baseline case where the kernels are simply launched serially into a single stream without synchronization. We observe that CUDAGraph exhibits a significant slowdown due to the overhead incurred during the construction and communication of the DAG dependencies. Fig. 24 depicts the corresponding achieved occupancy. We find that the ACE configurations are able to improve utilization, leading to performance improvements. As is the case in § 6.1, ACE-HW has a higher achieved occupancy than ACE-SW because of lower kernel launch + sync time.

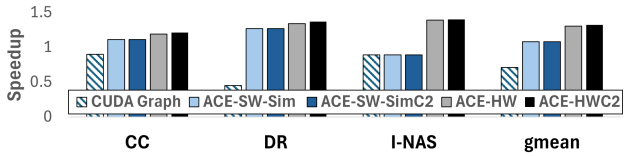


Figure 23: Dynamic DNNs: Normalized speedup

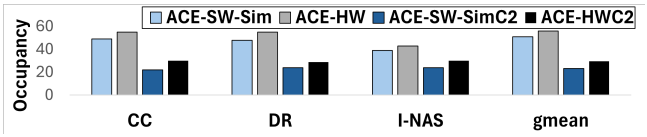


Figure 24: Dynamic DNNs: Achieved occupancy

6.3 Inference on Static DNNs

While our approach is designed for applications with dynamic computational graphs, we also evaluate its effectiveness in improving the concurrency of static DNNs. We depict the speedups obtained normalized to the baseline in Fig. 25. We observe an average speedup of 1.31× with ACE-HW, and a speedup of 1.16× with ACE-SW. Fig. 26 depicts the corresponding achieved occupancy. We find that ACE leads to higher GPU utilization, leading to performance improvements. ACE-HW has a higher achieved occupancy than ACE-SW because of lower kernel launch + synchronization time.

We observe that CUDAGraph exhibits similar execution times for all but one workload as ACE-HW for static graphs. The difference in performance in one workload occurs because ACE-HW can detect inter-kernel dependencies over a limited window of kernels. CUDAGraph is able to leverage additional parallelism in the application as it contains information on kernel dependencies across all kernels launched. However, this performance difference is marginal, and occurs on a single one of the tested static-DNN workloads.

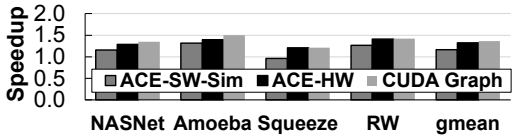


Figure 25: Static DNNs: Normalized speedup

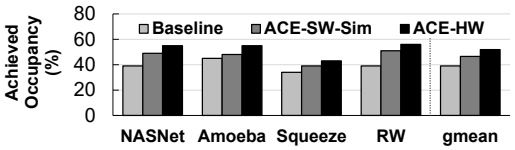


Figure 26: Static DNNs: Achieved occupancy

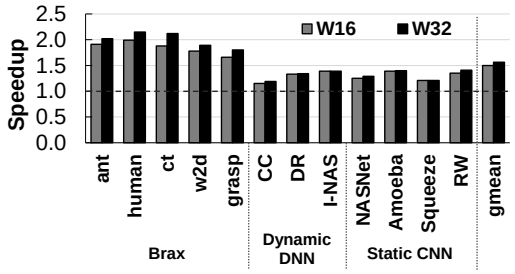


Figure 27: Speedups on varying scheduling window size

6.4 Sensitivity Analysis

Fig. 27 compares the speedups obtained on using scheduling window sizes of 16 and 32 for ACE-HW over baseline. We observe that the Brax simulations have higher performance (4.5% on average) with a window size of 32 compared to 16. However, the window size has less of an impact on the DNNs. This is because the simulation engines have more inter-kernel parallelism that is exposed with a larger scheduling window. This means that while there is still room to schedule newer CTAs, the amount of inter-kernel parallelism in the deepRL/dynamic DNNs is not sufficient enough to further increase utilization significantly.

6.5 Energy Consumption

Fig. 28 shows the normalized energy consumption for the evaluated workloads. We observe energy savings of 21.6% on average on all workloads with ACE-HW, and 6.1% with ACE-SW as a result of the reduction in execution time.

7 Related Work

In this work, we (i) observe that input-dependent inter-kernel dependencies and small kernels are a significant performance bottleneck in a range of important applications such as simulation engines in deep RL and dynamic neural networks; and (ii) propose both a software-only and hardware-software cooperative mechanism to enable concurrent execution of kernels with statically unknown

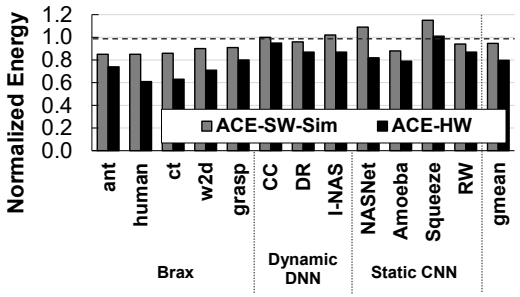


Figure 28: Normalized energy consumption

inter-kernel dependencies. In this section, we describe prior work that aim to improve GPU utilization and kernel concurrency.

Leveraging concurrent streams in DL workloads. Mainstream deep learning frameworks like Tensorflow [8] and Pytorch [59] launch GPU kernels into a single CUDA stream that executes them sequentially. Recent works [30, 55, 57, 95] propose software techniques to enable concurrent execution of GPU kernels using multiple streams with static scheduling and stream assignment before application execution. Inter-operator scheduling [30] partitions a computation graph into sections of kernels that can execute in parallel. Out-of-order backprop [57] observes that gradient computation can be parallelized using CUDA streams into weight gradients and the output gradient computation during backpropagation. However, these works are only applicable to DL workloads whose computation graph is static and known ahead of time, often requiring significant compilation times. Furthermore, these approaches incur high synchronization overheads.

Task-based programming frameworks in CPUs. Task-based frameworks [17, 28, 68] enable programmers to describe a program as multiple tasks scheduled for execution in multiprocessor architectures [66]. Works such as task superscalar [31], carbon [49], TDM [19] and ADM [70] propose out-of-order task scheduling to efficiently leverage irregular parallelism in multiprocessors. The major bottleneck in these applications is the long latency for dependency checks. Thus, prior work [19, 31, 49, 70] propose hardware accelerators to alleviate dependence checking bottleneck at runtime. However, the primary bottleneck on GPUs is the long latency to launch/signal completion of kernels instead, requiring a different approach to enable out-of-order scheduling.

Programmer annotations Prior works leverage programmer annotations as parallelization hints to the compiler. Sinclair et. al [76], DeNovo [26] use programmer annotations that encode the data read and written to by each function. This information is leveraged to determine independent tasks. Some frameworks [11, 28, 36, 61, 62] allow programmers to annotate array regions accessed by each task as a compile time directive. In ACE, we use a similar approach of using programmer annotations to help determine parallelism at runtime to enable out-of-order kernel scheduling.

Software techniques to improve GPU utilization with concurrent kernel execution. CUDA Graphs [1] and AMD ATMI [2, 12, 13] are frameworks that allow users to define dependencies between kernels as a directed-acyclic-graph (DAG) prior to execution. This approach eliminates synchronization and kernel launch overheads due to communication with the CPU. Nimble [50] identifies independent GPU kernels prior to execution and concurrently schedules independent kernels using CUDA streams. This approach uses CUDA Graphs [1] to reduce synchronization and kernel launch overheads. Irregular graphs are also seen in solving sparse linear equations for CFD simulations [39] and hyperplane sweep routines [44], where DAG frameworks have been shown to be effective.We quantitatively compared ACE against a CUDA graph implementation in § 6. None of these approaches is applicable to dynamic input-dependent computational graphs. While newer version of CUDA drivers improve the graph construction times, caching dependency information and constructing CUDA Graphs incur non-trivial latencies (§ 2.3).

Hardware support for concurrent kernels. Wireframe [10] proposes merging multiple kernels into a single large kernel and performs CTA scheduling with data dependency checks between CTAs. Blockmaestro [9] enables concurrently running kernels by identifying dependencies between their CTAs. These approaches however perform dependence checks by tracing and extracting the memory loads and stores performed by each thread block of every kernel. Similar to the software approaches, these approaches are designed for static computational graphs. The proposed scheduling and dependency check techniques would be too time-consuming for runtime scheduling. GPU dynamic parallelism [3, 20, 37, 84] enables launching kernels from the device itself and allows data dependencies between a single parent and multiple child kernels. However, Dynamic-NN and RL simulation workloads contain kernels that depend on multiple kernels, making it difficult to apply GPU dynamic parallelism.

GPU sharing framework. Multi-Instance GPU (MIG) [6] partitions GPU resources to allow concurrent execution of jobs launched by *multiple* users. Since fewer SMs are allocated to a single application, this would improve overall GPU utilization. However, applications such as deep RL and dynamic DNN inference with low utilization would still be slow as much of the inter-kernel parallelism is not being exploited. As ACE leverages this parallelism and reduces underutilization, ACE can improve performance even when sharing with another application.

Compilers, runtime systems for dynamic neural networks. Prior software [32, 42, 43, 54, 74, 85, 90] and hardware approaches [47] optimize CPU-GPU communication overheads, and blocking synchronization calls for dynamic computational graphs. These approaches introduce techniques such as dynamic batching and kernel fusion. These works are orthogonal to our approach.

8 Conclusion

We introduce ACE, the first framework that enables automatic concurrent kernel execution with low overhead runtime scheduling and dependency checks. The key idea behind ACE is to dynamically schedule a small window of kernels by identifying which kernel(s) within the window is ready for execution. ACE leverages kernel annotations to automatically identify kernel dependencies at runtime. We implement ACE as both a software framework and a hardware-software mechanism that is able to further reduce synchronization overheads from CPU-GPU communication. We demonstrate that ACE can improve the performance of important emerging classes of workloads, such as RL simulations and dynamic DNNs, whose kernel dependencies are irregular and vary with input.

References

[1] 0 [n. d.]. NVIDIA Inc, Getting started with CUDA Graphs. Retrieved 2020-09-30 from https://developer.nvidia.com/blog/cuda-graphs/

[2] 1 [n. d.]. Radeon Open Compute, ATMI (Asynchronous Task and Memory Interface). https://github.com/RadeonOpenCompute/atmi. Accessed: 2022-09-30.

[3] 2 [n. d.]. NVIDIA Inc, Cuda Dynamic Parallelism. https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/. Accessed: 2022-09-30.

[4] 3 [n. d.]. NVIDIA Inc, HyperQ. Retrieved 2023-07-21 from https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[5] 5 [n. d.]. NVIDIA Inc, CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams. Accessed: 2022-11-21.

[6] 6 [n. d.]. NVIDIA Inc, Multi-Instance GPU. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html. Accessed: 2023-10-10.

[7] 8 [n. d.]. AMD Inc, ROCm Device Enqueue. https://sep5.readthedocs.io/en/latest/Programming_Guides/Opencl-programming-guide.html#device-side-enqueue. Accessed: 2022-09-30.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Z. Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. *ArXiv* abs/1605.08695 (2016).

[9] AmirAli Abdolrashidi, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael B. Abu-Ghazaleh, and Daniel Wong. 2021. BlockMaestro: Enabling Programmer-Transparent Task-based Execution in GPU Systems. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 333–346.

[10] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi N. Bhuyan, and Daniel Wong. 2017. WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), 600–611.

[11] Matthew D Allen, Srinath Sridharan, and Gurindar S Sohi. 2009. Serialization sets: a dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 85–96.

[12] AMD Research. 2017. DAGEE. https://github.com/AMDResearch/DAGEE.git, Last accessed on 2023-02-14.

[13] AMD Research. 2017. HipGraph. https://github.com/HipGraph/, Last accessed on 2023-02-14.

[14] Haoyue Bai, Fengwei Zhou, Lanqing Hong, Nanyang Ye, Shueng-Han Gary Chan, and Zhenguo Li. 2021. NAS-OoD: Neural Architecture Search for Out-of-Distribution Generalization. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 8300–8309.

[15] Luca Bartolomei, Lucas Teixeira, and Margarita Chli. 2021. Semantic-aware Active Perception for UAVs using Deep Reinforcement Learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 3101–3108. https://doi.org/10.1109/IROS51168.2021.9635893

[16] Mehmet Esat Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: a dependence-aware task-based execution framework for GPUs. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2018).

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *PPOPP '95*.

[18] Shaofeng Cai, Yao Shu, and Wei Wang. 2021. Dynamic Routing Networks. *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2021), 3587–3596.

[19] Emilio Castillo, Lluc Alvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, José Luis Bosque, Ramón Beivide, and Mateo Valero. 2018. Architectural Support for Task Dependence Management with Flexible Software Scheduling. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), 283–295.

[20] Guoyang Chen and Xipeng Shen. 2015. Free launch: Optimizing GPU dynamic kernel launches through thread reuse. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015), 407–419.

[21] Jianyu Chen, Shengbo Eben Li, and Masayoshi Tomizuka. 2020. Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning. *arXiv preprint arXiv:2001.08726* (2020).

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv* abs/1512.01274 (2015).

[23] Tao Chen, Jie Xu, and Pulkit Agrawal. 2022. A system for general in-hand object re-orientation. In *Conference on Robot Learning*. PMLR, 297–307.

[24] Yuxin Chen, Benjamin Brock, Serban D. Porumbescu, Aydin Bulucc, Katherine A. Yelick, and John Douglas Owens. 2021. Atos: A Task-Parallel GPU Dynamic Scheduling Framework for Dynamic Irregular Computations. *ArXiv* abs/2112.00132 (2021).

[25] A. Cheng, Chieh Hubert Lin, Da-Cheng Juan, Wei Wei, and Min Sun. 2020. InstaNAS: Instance-aware Neural Architecture Search. In *AAAI*.

[26] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. *2011 International Conference on Parallel Architectures and Compilation Techniques* (2011), 155–166.

[27] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3213–3223.

[28] Leonardo Dagum and Ram Menon. 1998. OpenMP: an industry standard API for shared-memory programming. In *OpenMP: an industry standard API for shared-memory programming*.

[29] Steven Dalton and Iuri Frosio. 2020. Accelerating Reinforcement Learning through GPU Atari Emulation. *arXiv: Learning* (2020).

[30] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. IOS: Inter-Operator Scheduler for CNN Acceleration. *ArXiv* abs/2011.01302 (2021).

[31] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task superscalar: An out-of-order task pipeline. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 89–100.

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

[33] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. *ArXiv* abs/2106.13281 (2021).

[34] James Gleeson, Srivatsan Krishnan, Moshe Gabel, Vijay Janapa Reddi, Eyal de Lara, and Gennady Pekhimenko. 2021. RL-Scope: Cross-Stack Profiling for Deep Reinforcement Learning Workloads. *ArXiv* abs/2102.04285 (2021).

[35] James Gleeson, Daniel Snider, Yvonne Yang, Moshe Gabel, Eyal de Lara, and Gennady Pekhimenko. 2022. Optimizing Data Collection in Deep Reinforcement Learning. *ArXiv* abs/2207.07736 (2022).

[36] Gagan Gupta and Gurindar S Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th annual IEEE/ACM international symposium on Microarchitecture*. 59–70.

[37] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan S. Milojicic, and Wen mei W. Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), 1–12.

[38] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2022. Dynamic Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44 (2022), 7436–7456.

[39] Ahmed E. Helal, Ashwin M. Aji, Michael L. Chu, Bradford M. Beckmann, and Wu chun Feng. 2019. Adaptive Task Aggregation for High-Performance Sparse Solvers on GPUs. *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2019), 324–336.

[40] HSA Foundation. 2017. HSA Standard. http://hsafoundation.com/standards/, Last accessed on 2023-02-14.

[41] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *ArXiv* abs/1602.07360 (2016).

[42] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. 2019. {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)*. 453–468.

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

[44] Anirudh Mohan Kaushik, Ashwin M. Aji, Muhammad Amber Hassaan, Noel Chalmers, Noah Wolfe, Scott Moe, Sooraj Puthoor, and Bradford M. Beckmann. 2019. Optimizing Hyperplane Sweep Operations Using Asynchronous Multi-grain GPU Tasks. *2019 IEEE International Symposium on Workload Characterization (IISWC)* (2019), 59–69.

[45] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John M. Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. 2019. Learning to Drive in a Day. *2019 International Conference on Robotics and Automation (ICRA)* (2019), 8248–8254.

[46] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), 473–486.

[47] Farzad Khorasani, Hodjat Asghari Esfeden, Nael B. Abu-Ghazaleh, and Vivek Sarkar. 2018. In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), 377–389.

[48] Srivatsan Krishnan, Behzad Boroujerdian, William Fu, Aleksandra Faust, and Vijay Janapa Reddi. 2021. Air Learning: a deep reinforcement learning gym for autonomous aerial robot visual navigation. *Mach. Learn.* 110 (2021), 2501–2540.

[49] Sanjeev Kumar, Christopher J. Hughes, and Anthony D. Nguyen. 2007. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *International Symposium on Computer Architecture*.

[50] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. In *NeurIPS*.

[51] Yunsheng Li, Yinpeng Chen, Xiyang Dai, Dongdong Chen, Mengchen Liu, Lu Yuan, Zicheng Liu, Lei Zhang, and Nuno Vasconcelos. 2021. MicroNet: Improving Image Recognition with Extremely Low FLOPs. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 458–467.

[52] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[53] Lanlan Liu and Jia Deng. 2018. Dynamic Deep Neural Networks: Optimizing Accuracy-Efficiency Trade-offs by Selective Execution. In *AAAI*.

[54] Moshe Looks, Marcello Herreshoff, DeLesley S. Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. *ArXiv* abs/1702.02181 (2017).

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

[56] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, N. Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. *ArXiv* abs/2108.10470 (2021).

[57] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-of-order backprop: an effective scheduling technique for deep learning. *Proceedings of the Seventeenth European Conference on Computer Systems* (2022).

[58] Jacopo Panerati, Hehui Zheng, Siqi Zhou, James Xu, Amanda Prorok, Angela P. Schoellig University of Toronto Institute for A Studies, Vector Institute for Artificial Intelligence, and University of Cambridge. 2021. Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control. *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2021), 7512–7519.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *ArXiv* abs/1912.01703 (2019).

[60] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *ICML*.

[61] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2009. Hierarchical Task-Based Programming With StarSs. *The International Journal of High Performance Computing Applications* 23 (2009), 284 – 299.

[62] Antoniu Pop and Albert Cohen. 2012. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.* 9 (2012), 53:1–53:25.

[63] Sooraj Puthoor, Ashwin M. Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M. Beckmann, and Gregory P. Rodgers. 2016. Implementing directed acyclic graphs with the heterogeneous system architecture. *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit* (2016).

[64] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. *Proceedings of the 11th Workshop on General Purpose GPUs* (2018).

[65] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *ICML*.

[66] Alex Ramírez, Felipe Cabarcas, Ben H. H. Juurlink, Mauricio Alvarez-Mesa, Friman Sánchez, Arnaldo Azevedo, Cor Meenderinck, Cătălin Bogdan Ciobanu, Sebastián Isaza, and Georgi Gaydadjiev. 2010. The SARC Architecture. *IEEE Micro* 30 (2010), 16–29.

[67] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*.

[68] James Reinders, Michael J. Voss, Pablo Reble, and Rafael Asenjo-Plaza. 2020. ++ for Heterogeneous Programming: oneAPI (DPC++ and oneTBB). In *C++ for Heterogeneous Programming: oneAPI (DPC++ and oneTBB)*.

[69] N. Rudin, David Hoeller, Philipp Reist, and Marco Hutter. 2021. Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning. *ArXiv* abs/2109.11978 (2021).

[70] Daniel Sánchez, Richard M. Yoo, and Christoforos E. Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *ASPLOS XV*.

[71] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), 4510–4520.

[72] Brennan Shacklett, Erik Wijmans, Aleksei Petrenko, Manolis Savva, Dhruv Batra, Vladlen Koltun, and Kayvon Fatahalian. 2021. Large Batch Simulation for Deep Reinforcement Learning. *ArXiv* abs/2103.07013 (2021).

[73] Noam M. Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *ArXiv* abs/1701.06538 (2017).

[74] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. *ArXiv* abs/2006.03031 (2021).

[75] Zilin Si and Wenzhen Yuan. 2022. Taxim: An Example-Based Simulation Model for GelSight Tactile Sensors. *IEEE Robotics and Automation Letters* 7, 2 (2022), 2361–2368. https://doi.org/10.1109/LRA.2022.3142412

[76] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015), 647–659.

[77] Pravendra Singh and Vinay P Namboodiri. 2020. SkipConv: skip convolution for computationally efficient deep CNNs. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[78] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph.* 33 (2014), 228:1–228:11.

[79] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[80] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. *2016 23rd International Conference on Pattern Recognition (ICPR)* (2016), 2464–2469.

[81] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 5026–5033.

[82] Andreas Veit and Serge J. Belongie. 2019. Convolutional Networks with Adaptive Inference Graphs. *International Journal of Computer Vision* 128 (2019), 730–741.

[83] Huanyu Wang, Songyuan Li, Shih-Chieh Su, Zequn Qin, and Xi Li. 2021. RDI-Net: Relational Dynamic Inference Networks. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 4601–4610.

[84] Jin Wang, Norman Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2015. Dynamic Thread Block Launch: A lightweight execution mechanism to support irregular applications on GPUs. *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015), 528–540.

[85] Jinliang Wei, Garth Gibson, Vijay Vasudevan, and Eric Xing. 2018. Dynamic scheduling for dynamic control flow in deep learning systems. *URL http://www.cs. cmu. edu/jinlianw/papers/dynamic_scheduling_nips18_sysml. pdf* (2018).

[86] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. 2018. BlockDrop: Dynamic Inference Paths in Residual Networks. In *CVPR*.

[87] Wenhan Xia, Hongxu Yin, Xiaoliang Dai, and Niraj Kumar Jha. 2022. Fully Dynamic Inference With Deep Neural Networks. *IEEE Transactions on Emerging Topics in Computing* 10 (2022), 962–972.

[88] Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. 2019. Exploring Randomly Wired Neural Networks for Image Recognition. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019), 1284–1293.

[89] Zhaoming Xie, Xingye Da, Buck Babich, Animesh Garg, and Michiel van de Panne. 2021. Glide: Generalizable quadrupedal locomotion in diverse environments with a centroidal model. *arXiv preprint arXiv:2104.09771* (2021).

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

[91] Brandon Yang, Gabriel Bender, Quoc V. Le, and Jiquan Ngiam. 2019. CondConv: Conditionally Parameterized Convolutions for Efficient Inference. In *NeurIPS*.

[92] Zhao You, Shulin Feng, Dan Su, and Dong Yu. 2021. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts. *arXiv preprint arXiv:2105.03036* (2021).

[93] Kun Yuan, Quanquan Li, Shaopeng Guo, Dapeng Chen, Aojun Zhou, Fengwei Yu, and Ziwei Liu. 2021. Differentiable Dynamic Wirings for Neural Networks. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), 317–326.

[94] Zhihang Yuan, Bingzhe Wu, Zheng Liang, Shiwan Zhao, Weichen Bi, and Guangyu Sun. 2020. S2DNAS: Transforming Static CNN Model for Dynamic Inference via Neural Architecture Search. *ArXiv* abs/1911.07033 (2020).

[95] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020*

USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 337–352. https://www.usenix.org/conference/atc20/presentation/zhu-hongyu

[96] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), 8697–8710.