ARK: GPU-driven Code Execution for Distributed Deep Learning

Changho Hwang[1,2], KyoungSoo Park[1], Ran Shu[2], Xinyuan Qu[2,†], Peng Cheng[2], and Yongqiang Xiong[2]

[1]*KAIST* [2]*Microsoft Research*

Abstract

Modern state-of-the-art deep learning (DL) applications tend to scale out to a large number of parallel GPUs. Unfortunately, we observe that the collective communication overhead across GPUs is often the key limiting factor of performance for distributed DL. It under-utilizes the networking bandwidth by frequent transfers of small data chunks, which also incurs a substantial I/O overhead on GPU that interferes with computation on GPU. The root cause lies in the inefficiency of CPU-based communication event handling as well as the inability to control the GPU's internal DMA engine with GPU threads.

To address the problem, we propose a ***GPU-driven*** code execution system that leverages a GPU-controlled hardware DMA engine for I/O offloading. Our custom DMA engine pipelines multiple DMA requests to support efficient small data transfer while it eliminates the I/O overhead on GPU cores. Unlike existing GPU DMA engines initiated only by CPU, we let GPU threads directly control DMA operations, which leads to a highly efficient system where GPUs drive their own execution flow and handle communication events autonomously without CPU intervention. Our prototype DMA engine achieves a line-rate from a message size as small as 8KB (3.9x better throughput) with only 4.3µs of communication latency (9.1x faster) while it incurs little interference with computation on GPU, achieving 1.8x higher all-reduce throughput in a real training workload.

1 Introduction

Modern machine learning (ML) applications tend to harness an increasingly larger number of accelerators (especially GPUs in this work) [19, 26]. State-of-the-art deep learning (DL) algorithms often need to scale out to thousands of GPUs for higher throughput and accuracy [26]. Unfortunately, this poses a substantial communication overhead to the entire system, which harms GPU utilization by delaying or interfering with numeric computation.

† Now at Horizon Robotics.

The communication overhead mainly arises in two different aspects. First, collective communication (e.g., all-reduce, split-and-gather, all-to-all, etc.), which is widely adopted in most of popular DL algorithms, often splits the data for transfer into multiple small chunks for pipelining or for sending to multiple different destinations. The chunk size tends to get smaller as we scale out, which is detrimental to efficient utilization of networking bandwidth. Second, popular communication libraries for GPUs such as NCCL [32] and RCCL [5] often incur a severe I/O overhead on GPU. This is because they commonly leverage memory-mapped I/O (MMIO) for data copies between GPUs, which consumes a substantial amount of GPU resources (i.e., core cycles and L2 cache/DRAM bandwidth). We observe that concurrent execution of collective communication and numeric computation on GPU heavily interferes with each other – in our training experiment with BERT-Large [10], the throughput of parallel computation drops by 45% while it achieves only 53.6% of the peak communication throughput (see details in Section 2.3).

Unfortunately, it is challenging for existing systems to address both issues (i.e., large transfer delay for small chunks and I/O overhead on GPU) at the same time. One may avoid the I/O overhead by offloading the I/O to a hardware DMA engine instead of employing MMIO with GPU threads. However, the current DMA engine on commodity GPU is initiated only by CPU threads, which often enrolls CPU's control on the critical path of communication. This incurs the CPU-GPU synchronization overhead that bloats up the communication latency, especially detrimental to the throughput of small data chunk transfer. In fact, one can observe hundreds of µs of communication latency in a popular DL framework as it leverages the DMA engine. Similarly, if one does not employ the DMA engine for communication of data chunks, the communication would suffer from high I/O overhead on GPU.

This paper proposes the *GPU-driven* system named *ARK*, a communication-motivated DL system design. The key idea of the GPU-driven system lies in autonomous execution control of GPU code without any control by external devices. This regime tightly connects computational power of every GPU core across machines by allowing GPU threads to communicate directly with remote GPUs without any external control signals, which ends up achieving low-latency communication. At the same time, to avoid the I/O overhead on GPU, we design a GPU-controlled DMA engine. Specifically, our custom DMA engine is directly initiated by GPU threads, which avoids the heavy MMIO without CPU intervention.

Our evaluation shows that our DMA engine prototype is especially beneficial for small messages, achieving a high communication throughput (3.87x over `cudaMemcpy` with 8KB messages) at low latency (9.1x faster over CPU intervention). Furthermore, it does not interfere with computation on GPU, which delivers both computation and communication throughput gains over using MMIO-based libraries [5,32] (1.8x faster all-reduce in BERT-Large [10] training, see Section 5.3).

To realize the GPU-driven system, we also present an efficient scheduler of autonomous execution on GPU. Our key observation is that online dynamic scheduling is unnecessary as DL workloads are typically deterministic at runtime. Instead, we present the *virtual Cooperative Thread Array* (vCTA) framework that abstracts *offline* GPU scheduling. Offline scheduling allows eliminating the runtime scheduling overhead at the back-end, while reusing the existing front-end interface and GPU kernel implementations.

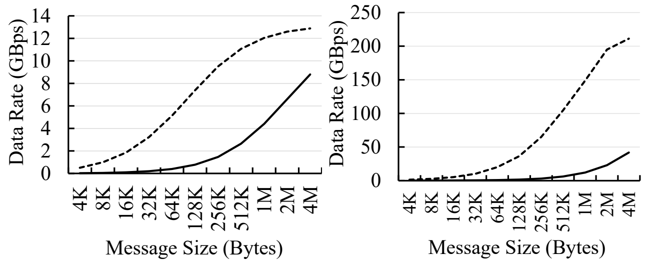
ARK supports efficient and flexible parallel execution models for data-, tensor-, and pipeline-parallelisms. Our evaluation demonstrates that ARK delivers substantial performance gains both in training and inference, achieving 2.5x and 3.6x throughput improvement, respectively.

2 Background & Motivation

This section explains existing inter-GPU communication technologies and their limitations.

2.1 Small Data Transfer in Distributed DL

Collective communication consists of several communication primitives that concurrently exchange the data across multiple GPUs, which is widely adopted to implement various parallelism methods in distributed DL. Popular use cases include *all-reduce* for data-parallelism, *split-and-gather* for tensor-parallelism [22,40], and *all-to-all* for expert-parallelism [11]. As the number of employed GPUs gets larger, the size of unit data transfer in collective communication becomes smaller as it splits the local data into multiple pieces to be delivered to different GPUs. This small transfer size makes the overall performance of collective communication highly dependent on the control plane overhead before and after each data transfer. Unfortunately, we observe that the control plane overhead either with *CPU-controlled* or even *GPU-controlled* communication is pretty substantial (See Section 2.2 and Section 2.3). Also, existing workarounds (e.g., *tensor fusion* [39]) that batch a large amount of data to avoid small transfers would not completely address the problem as they trade off computational throughput by intentionally delaying data transfer.



(a) PCIe v3. (b) NVLink v3.

Figure 1: Data dependency between GPUs decreases the inter-GPU data rate due to event handling delays. Solid lines refer to actual data rate (for sending one message at a time) in TensorFlow's CPU-controlled communication crossing a PCIe v3 or a NVLink v3 switch while dashed lines indicate the ideal data rate without event handling delays.

2.2 External Execution Control Overhead

Existing GPU program execution heavily relies on an external processor (i.e., CPU) to submit GPU commands for kernel execution or data transfer. Unfortunately, this model often incurs a large overhead due to the delay for command delivery from the host side to GPU hardware queue (i.e., *stream*). One can use the conventional GPU event interface (i.e., `cudaEvent`) to hide the delay, but it would also suffer from substantial delay for event handling. When adopted to inter-GPU communication, which we call *CPU-controlled* communication (in contrast to *GPU-controlled* communication by NCCL [32]), we observe that event handling becomes the primary cause for large communication delay beyond the data transfer itself.

We consider a common communication scenario where two GPUs have a data dependency – one GPU receives computation results of another GPU to feed them as input to its own computation. In every data transfer, event handling is needed to check the dependency between the copy and the GPU commands around the copy operation, which reduces the actual data rate between GPUs. Figure 1 compares the ideal inter-GPU data rate (`cudaMemcpy` throughput) with the actual data rate in TensorFlow's CPU-controlled communication, which is still used along with NCCL especially for model-parallelism implementations. We see that the event handling overhead with `cudaMemcpy` drastically lowers the data rate both in the PCIe and NVLink interfaces. We explain two implementations when GPU A sends data to GPU B.

2.2.1 Runtime Intervention for the Control

CPU can serve as an intermediary to deliver an event between two communicating GPUs. In fact, if GPUs are located in different NUMA nodes or on different machines, the runtime intervention by CPU is required for communication. Also, some frameworks like TensorFlow implement a generic interface that always uses CPU for GPU event handling regardless of the placement. Figure 2 illustrates the event handling overhead due to CPU intervention when GPU A sends its data to

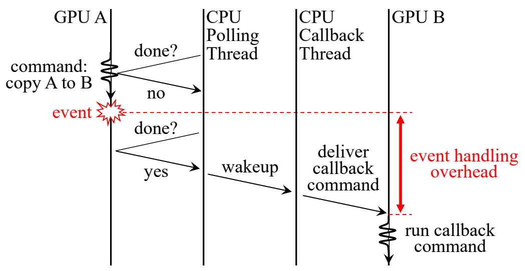


Figure 2: CPU intervention in inter-GPU event handling.

Overhead Detail	Delay (µs)
Initiation	
Trigger send ready event on the GPU	3.8
Sync comp. stream and comm. stream	11.6
Completion Check	
Event polling gap	58.3
Delay of pthread mutex lock	58.7
GPU kernel launch overhead	19.2
Total	**151.6**

Table 1: Breakdown of the constant overhead of inter-GPU data transfer using TensorFlow in Figure 1.

GPU B that plans to run the next command with the data.

We notice three places for the overhead. First, it is inefficient for a CPU thread to poll GPU events because the event interface disallows the CPU thread to monitor multiple events at the same time. While it takes only ∼3µs for a dedicated busy-waiting CPU thread to be notified of a triggered GPU event,[2] this approach does not scale when an application has to run many parallel tasks, which will run many polling threads. Instead, the event polling loop of TensorFlow uses only one CPU thread, which incurs a ∼58.3µs of polling gap on average (see Table 1). Second, it takes time to wake up the CPU thread that invokes the callback function of the triggered event. In TensorFlow, it takes ∼58.7µs for the callback thread to acquire the mutex lock from when it is released by the polling thread. This delay could be reduced to as low as 5µs if both threads are running on the same CPU core, but co-locating the threads or even merging them into a single one would increase the event polling interval as well as the overall processing time. Lastly, it is inefficient for the callback thread to deliver the computation command to GPU B. Delivering the event signal to GPU B would take only 2∼3µs if implemented efficiently,[3] but we need to deliver the callback command binary as well. We can avoid the extra delay if we deliver the GPU command in advance and trigger it later on the CPU side, but this is not supported by commodity GPU.

2.2.2 Asynchronous Control

If the GPUs are under the same NUMA node, CPU can reserve a GPU event to be triggered asynchronously so that GPUs can directly communicate with each other when the event occurs. In this case, one can deliver the callback command to GPU B before the actual event and use the conventional GPU event interface (i.e. `cudaEvent` or a higher-level wrapper such as CUDA Graphs [27]) to trigger the callback command on GPU B with GPU A's event. Ideally, this should take as short as sending a single bit from GPU A to GPU B. However, we find that triggering a GPU event (∼4µs) and waking up a dependent GPU command (10∼20µs) are disappointingly slow – it ends up taking as much as sending the command binary to the GPU at runtime. We suspect that this is due to inefficient hardware implementation on GPU for event handling. In TensorFlow, this overhead contributes to the delay for initiating a transfer that depends on GPU computation as shown in Table 1.

2.3 I/O Overhead of GPU-side Control

Since CPU intervention incurs a large overhead, how about managing the communication with GPU itself? NCCL [32] [4] leverages *GPUDirect* [31] to enable this approach, which exposes the GPU memory space for peer-to-peer access so that GPU threads can read/write data to/from another GPU.[5] As GPU threads can directly invoke data copy, they can handle communication events efficiently without the involvement of CPU. Since commodity GPU hardware disallows GPU threads to initiate its own DMA engine, GPU-controlled communication leverages MMIO, which will implicitly conduct DMA when GPU threads write data on the mapping. Figure 3 compares CPU-controlled and GPU-controlled communication. The former one (Figure 3a) takes the following steps: ① CPU is notified when the data is ready, ② CPU initiates the DMA engine, and ③ DMA copies the data. On the other hand, GPU-controlled communication with MMIO (Figure 3b) follows ① CPU creates a memory map (mmap) of the destination GPU's address space prior to runtime execution, ② the data is ready at runtime, and ③ GPU threads copy the data into the mmap, which implicitly conducts DMA copy.

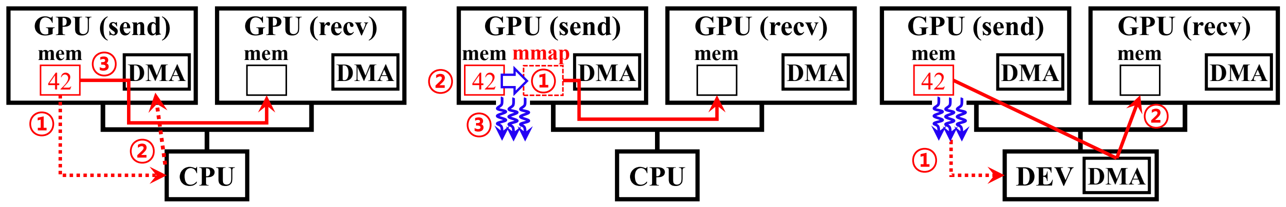
Unfortunately, data copying by GPU threads often heavily interferes with parallel kernel computation, especially due to L2 cache pollution and warp scheduler operations. Specifically, a data-copy GPU thread needs to load the data onto its register file for data transfer, but this pollutes the L2 cache as one cannot bypass the L2 cache when reading from DRAM on commodity GPU [34]. It leads to severe performance degradation over initiating DMA directly, as the latter copies the data on DRAM directly to the I/O bus (PCIe or NVLink). Additionally, the copying threads frequently issue 'load/store'

[2]Please refer to the experiment setup in Section 5.

[3]This is roughly estimated based on that it takes ∼2µs for a GPU thread to read a 4-byte data on the host DRAM and it takes ∼3µs for a busy-waiting CPU to read a GPU event.

[4]Equally applied to RCCL [5] on AMD GPU as well. For convenience, we borrow the terms from CUDA or NVIDIA GPUs, which can be easily converted into corresponding terms in OpenCL or AMD GPUs.

[5]CPU-controlled communication also leverages GPUDirect for efficient `cudaMemcpy` between peer GPUs without crossing the root complex, but its execution path is different from that of GPU-controlled communication.



(a) CPU-controlled. (b) GPU-controlled MMIO. (c) GPU-controlled DMA (this work).

Figure 3: Comparison between CPU-controlled and GPU-controlled communication – the latter has two different approaches, which leverage (b) MMIO (like NCCL) or (c) directly initiated DMA (this work). DEV refers to any kinds of devices that can implement our DMA engine.

instructions that drive warp schedulers busy, which makes other threads for parallel computation yield their clock cycles. Although the affected computation threads are limited to those that co-run warp schedulers with data-copy threads, they delay the entire kernel by falling behind the other threads.

To analyze the impact of the contention, we measure the slowdown of two different GPU kernels that heavily access only a specific type of GPU resources each: L2 cache (1.96 TBps read) and warp schedulers (2.02 IPC),[6] respectively (all numbers measured on a V100 GPU), while running concurrently with NCCL (v2.11.4) 64 MB all-gather[7] kernels using 8x V100 GPUs. We leverage NVIDIA Visual Profiler (NVVP) and Nsight Compute to verify that (1) the L2 cache kernel shows near-zero DRAM access and L1 data cache hit rate and (2) the warp schedulers kernel shows near-zero L2 cache/DRAM throughput. We have also verified the concurrency of computation and all-gather kernels and no other CPU/GPU activities during the experiment. In this experiment, the slowdowns due to L2 cache and warp schedulers contention are up to 2.4x and 2.0x, respectively, where it slows down either the computation or the concurrent NCCL communication (when one side is degraded less, the other side tends to be impacted more). This result shows that heavy contention could arise depending on the GPU resource usage of concurrent computation kernels.

We run a microbenchmark to evaluate the contention of NCCL all-reduce during data-parallel training of a BERT-Large [10] model. This model performs 32 MB of all-reduce at a time, which issues 4 MB data transfer in parallel with eight GPU workers. On a server with 8x V100 GPUs (connected with a single PCIe switch (16x PCIe v3)), the parallel computation throughput drops by 45.0% while all-reduce achieves only 5.0 GBps on average, degraded to 53.6% of the peak throughput without the interference. On a server with 8x A100 GPUs (connected with an NVSwitch (NVLink v3)), the slowdown of all-reduce is even worse – the parallel computation throughput drops by 14.3% while the NCCL all-reduce achieves only 30.9% of the peak throughput (49.0 GBps).

[6]Heavy usage of warp schedulers means frequent instruction fetches, i.e. large instructions per cycle (IPC). $> 99.2\%$ of instructions are FFMA.

[7]We use all-gather as it only performs communication without any extra computation such as reduction in all-reduce.

3 ARK Framework Design

In this section, we present the design of ARK, our approach with the *GPU-driven* code execution that avoids the communication overhead on GPU without CPU intervention.

3.1 GPU-controlled DMA Engine

We claim that a GPU-controlled DMA engine (Figure 3c) can eliminate the communication overhead, which in turn serves as the basis of our GPU-driven system. The GPU-controlled DMA engine enables a GPU thread to directly initiate DMA operations when the data is ready (①), which will immediately push the data into the I/O bus without wasting GPU cycles (②). We leverage existing GPUDirect techniques to expose the GPU's physical address space to our DMA engine.

While GPU-controlled DMA would deliver low-latency communication without the MMIO overhead, it is non-trivial to realize this feature. In fact, an ideal implementation would be to modify the existing DMA engine on GPU to support GPU-controlled DMA, but it is infeasible as we cannot update the GPU hardware. Instead, we consider employing an external device as illustrated in Figure 3c at the cost of extra communication latency from GPU threads.

Despite of performance benefits, adopting new hardware for GPU-controlled DMA engine might be costly in many existing systems. To provide an interim solution, we pursue a general DMA engine design that can be implemented as either software or hardware on any hardware platforms (e.g., CPU, GPU, SmartNIC, FPGA, etc.) or I/O bus types (PCIe, NVLink [33], or Infinity Fabric Link (xGMI) [3]). Regardless of the platform, all implementations need to share the same runtime interface for GPU kernels. Also, the DMA interface should support low latency and flexibility while meeting the different requirements of software and hardware engines.

In this paper, we present both a software implementation and a hardware prototype of GPU-controlled DMA engine. Our software engine works over any existing systems without additional hardware as it leverages host CPU cores – busy-waiting CPU threads read DMA requests from GPU and initiate DMA accordingly. This design is aligned with the principle of GPU-driven system as GPU threads directly initiate the data transfer, while CPU threads only mechanically initiate data copies without any GPU event handling or GPU resource consumption. Our hardware engine prototype is implemented on FPGA, which we present to show the potential benefit of hardware deployment over the software engine. We explain the details of DMA engine implementations in Section 4.1.

3.2 Loop Kernel & Virtual CTA

GPU-controlled DMA engines would be easily adopted by existing systems, e.g., NCCL can replace its MMIO with initiating our DMA engines. However, existing systems would not fully exploit the benefit of GPU-controlled communication as the communication APIs are launched by CPU – the CPU intervention barrier still remains between computation and communication.

To remove this barrier, we propose a GPU-driven code execution system that runs an entire DL application in a single kernel, called a *loop kernel*. Our key observation is that online dynamic scheduling is unnecessary as DL workloads are typically deterministic at runtime. Instead of dynamically launching GPU kernels with CPU at runtime, our GPU-driven system automatically merges all kernels into a loop kernel (one for each GPU) at compile time and launches it only once at application start. Then, the loop kernel runs continuously during the entire lifetime of the application. A loop kernel is generated by our code generator that reads an operational graph of a DL application and automatically assembles corresponding code snippets of GPU operators to build loop kernel code. We call this code generation as *offline scheduling* as all GPU operators are statically distributed across GPU cores, or *Streaming Multiprocessors* (SMs), by the code. Offline scheduling lets GPUs efficiently control the application, which would minimize the event handling overhead for inter-GPU communication. We discuss several technical details of the loop kernel in Section 4.2.

Figure 4 shows that the loop kernel design deviates from the conventional framework for declaring, scheduling, and executing GPU tasks. In both CPU- and GPU-driven systems, a GPU operator is commonly defined as a set of multiple unit operators that each computes a part of the entire output in the SIMD manner. Meanwhile, both systems declare the operator differently in the GPU code. The CPU-driven system declares each unit operator as a Cooperative Thread Array (CTA)[8] and the entire operator as a separate kernel, which requires launching multiple kernels for multiple operators. In contrast, our GPU-driven system disallows multiple kernels as it executes all operators in the single loop kernel. Instead, it exploits intermediate declaration of unit operators that are scheduled as part of the CTAs of the loop kernel, which we call *virtual CTAs* (vCTAs).

vCTA provides the key abstraction for offline scheduling in ARK, which enables *software-defined* SM scheduling. A vCTA declares the code for a unit operator that is affinitized

[8]CTA is conceptually and functionally the same as a thread block in CUDA or a workgroup in OpenCL.

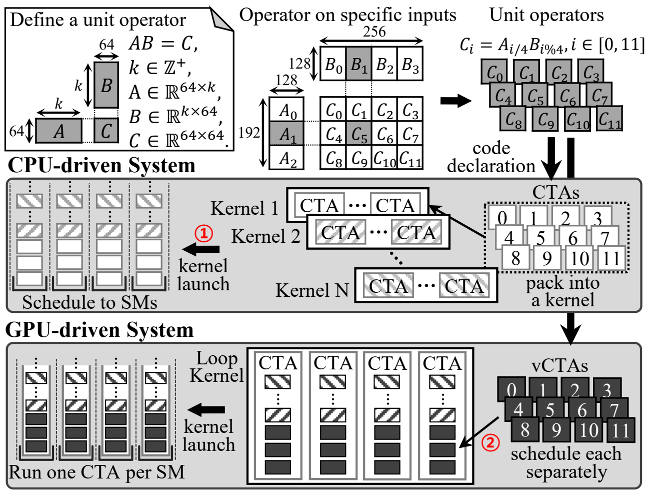


Figure 4: Comparing the procedures for declaring, scheduling, and executing GPU computation tasks between CPU- and GPU-driven systems. For instance, the figure shows a matrix multiplication operator with a 192x256 output, which is split into 12 unit operators that calculate 64x64 outputs each.

to a specific SM inside the loop kernel. While a CPU-driven system relies on the non-programmable hardware scheduler that distributes the CTAs across SMs at kernel launch (① in Figure 4), a GPU-driven system implements a custom logic that distributes vCTAs across CTAs (② in Figure 4). By launching one CTA per SM that assigns each CTA to use the entire resources of an SM, ARK can control the SM-affinity of vCTAs in a programmable manner. This enables fine-grained GPU scheduling, which is useful for the GPU-driven system to implement various computational optimization techniques such as *operator fusion* [17, 24, 35].

Migration of existing code to ARK is straightforward as ARK can reuse existing GPU kernel implementations with minimal modification: replacing the CTA ID (`blockIdx` in CUDA), thread ID (`threadIdx` in CUDA), SM-local memory address (shared memory in CUDA), and synchronization functions (e.g., `__syncthreads()` in CUDA) into corresponding constants or functions provided by the ARK framework. This modification guarantees the correctness of the framework which we have extensively verified.

As shown in Figure 4, offline scheduling writes a code snippet of each vCTA inside the if-branch of the loop kernel that only a particular CTA (or SM) enters. Since each CTA statically executes specific vCTAs that are planned offline, the GPU actually runs a static `while()` loop rather than being controlled dynamically – internal busy-polling loops inside vCTAs handle runtime events. For example, in Figure 5b, each of CTA 0 (`ctaId` is 0) and CTA 1 (`ctaId` is 1) are assigned three vCTAs from the operator `op_0` and two vCTAs from the operator `op_1`. Each CTA uses 256 threads, and vCTAs from `op_0` are executed sequentially by thread 0∼127, while tasks from `op_1` are executed by thread 128∼255 (which im-

```
__device__ void op_0(int vcta_id) {
  Add<...>(&BUF[1024], &BUF[9728], &BUF[1024], vcta_id);
}
__device__ void op_1(int vcta_id) {
  Matmul<...>(
    &BUF[11776], &BUF[9728], &BUF[16384], vcta_id);
}
```

(a) Operators.

```
__global__ void loop_kernel(volatile int *iter) {
  for (;;) {
    // Wait until iteration is requested by the host.
    if (threadId == 0) { while (*iter == 0) {} }
    __syncthreads();
    // Run iterations.
    for (int i = 0; i < *iter; ++i) {
      if (ctaId == 0) {
        if      (threadId < 128) { op_0(0); op_0(2); op_0(4); }
        else if (threadId < 256) { op_1(0); op_1(2); }
      } else if (ctaId == 1) {
        if      (threadId < 128) { op_0(1); op_0(3); op_0(5); }
        else if (threadId < 256) { op_1(1); op_1(3); }
      } ...
    }
    // Inform the host that iterations are done.
    if (threadId == 0) { *iter = 0; }
    __syncthreads();
  }
}
```

(b) Loop kernel.

Figure 5: Example of auto-generated code by the ARK scheduler. Note that the code is simplified for readability.

plies that each vCTA is implemented to use 128 co-working threads). Each vCTA is declared by passing a certain vCTA ID to a GPU function that defines an operator like in Figure 5a. The kernel code library of ARK provides the implementation of common operators (Add or Matmul in the figure) that take the addresses of data chunks and a vCTA ID as runtime arguments.[9] The framework assigns proper offsets to the global GPU buffer (BUF) for each data chunk, and the vCTA ID locates a specific part of the chunk that the vCTA deals with.

3.3 Offline Scheduler

Figure 6 shows the scheduling workflow in ARK. Overall, it reads the DAG of a DL model and generates the corresponding loop kernel code. The ARK scheduler is composed of a high-level scheduler and a profiling module. The high-level scheduler implements operator fusion with profiling results fed by the module. In the initial phase, it builds an *OpGraph* that spots all operators and their dependencies in the model, and generates the code to profile all types of vCTAs that are needed. Then, the high-level scheduler generates its first scheduling decision with the profiling results. The decision may consist of multiple different candidates that need to be profiled to choose the fastest one, then it iterates the overall process to compare against multiple other candidates, which may require additional profiling. The scheduler finally returns the loop kernel when only a single candidate remains.

Reducing compilations in the profiler. Since the code generator conducts deterministic scheduling with static vCTA-SM affinity, it can accurately estimate the performance (i.e. latency and core resource usage) of every scheduling decision by only profiling the performance of vCTAs, which reduces

[9]Other arguments such as input data sizes can be fixed during compilation by passing as template arguments, which we omit here.

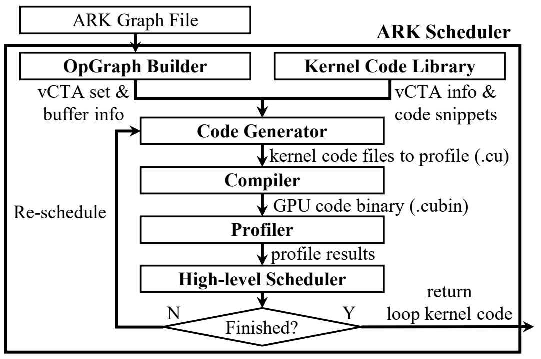


Figure 6: The ARK scheduler workflow.

the compilation for evaluating scheduling strategies. Say there are n parallel operators and each operator has m different implementations of the unit operator (or vCTAs),[10] then up to $O(m^n)$ different kernels should be compiled to find the best fusion decision. Since this number could be unreasonably large, existing works have developed heuristics to focus on only promising candidates [17].

At first glance, this appears to require only $O(nm)$ kernels for vCTA evaluations, but it is more complicated as vCTAs often complete faster when they are run concurrently on the same SM than when executed serially, which we say they have *joint efficiency*. Joint efficiency arises largely due to two causes: (1) because the L1 cache hit ratio improves as they access the common memory space running on the same SM, (2) because the execution of one vCTA hides the memory access of another (and vice versa) that improves simultaneous utilization of ALUs and LSUs. The first case is often found in the vCTAs from the same operator, while the second case is prevalent in most vCTAs, i.e., almost all vCTAs have the joint efficiency with each other.

Considering the joint efficiency, in general, we need to measure the latency when different types of vCTAs co-run on the same SM, which requires one kernel compilation for each. Say up to k vCTAs can run simultaneously in one SM, then the complexity of the number of compilations is $\sum_{i=1}^{k}\binom{n}{i}m^i = O(n^k m^k)$. In practice, this is much smaller than $O(m^n)$ because k is typically a small constant ≤ 4 due to the limitation of SM resources (# of maximum threads, bytes of shared memory, and # of registers).

SM load balancing in the code generator. The code generator automatically maximizes the SM utilization of the loop kernel by distributing parallel vCTAs across SMs to balance their workload. Unfortunately, finding the optimal load balancing is an NP-hard problem due to the joint efficiency. A brute-force searching would take unreasonably long due to the large number of vCTAs to schedule simultaneously.

To tackle this issue, we implement a heuristic load balanc-

[10]It is common to implement multiple different unit operators for the same operator, e.g. cuBLAS [30] implements at least 8 different-sized unit matrix multiplications and choose one depending on the input sizes.

ing on SM by leveraging an existing *graph partitioning* algorithm. Graph partitioning is a popular load balancing problem that splits a graph into a given number of subgraphs by cutting several edges, while achieving two goals: (1) balancing the total node weights of subgraphs and (2) minimizing the total weights of cut edges. We represent the SM load balancing problem into a graph partitioning problem. Specifically, we first group independent vCTAs that need to be distributed across SMs. Each group is represented as a graph where each node represents a vCTA and each edge indicates that the connecting nodes (i.e. vCTAs) have joint efficiency. The node weight is the latency of running the vCTA on an SM, and the edge weight measures joint efficiency, which is calculated as the fraction of latency reduction when we run both vCTAs simultaneously in the same SM compared with when we run both sequentially.

However, it takes too long to run the partitioning because it makes too many edges – since almost all vCTAs have joint efficiency with each other, the graph becomes nearly a mesh connection. To accelerate the algorithm, we adopt *hypergraph* representation [2] instead of an ordinary graph, which represents an equal-weighted mesh connection of multiple nodes as a single edge called *hyperedge*. Fortunately, this representation substantially reduces the time for code generation especially when we use a large batch size (which creates a lot of vCTAs), from tens of hours to only several seconds.

3.4 Limitations

The vCTA-based scheduling takes a whitebox approach that assumes all operators to be open-sourced, thus ARK cannot schedule close-sourced binaries such as cuDNN [28] (similar to Rammer [24]). Also, the offline scheduler of ARK only supports static computational graphs, which is less flexible comparing to e.g. PyTorch's dynamic graph [13]. However, such a limitation is commonly found in many popular frameworks including TensorRT [35] and ONNX Runtime [25].

4 Implementation

This section describes technical details of ARK.

4.1 DMA Engine Implementations

We first present our DMA engine interface, and then introduce our software and hardware DMA engines.

4.1.1 Interface

The key consideration of our interface design is ensuring high communication performance while keeping the interface consistent across software and hardware platforms. One key issue lies in the design of a DMA request message from GPU, which we call a *send request* (SR), as it has significant impact on the performance and the implementation complexity. In terms of hardware, receiving a large SR whose size exceeds the data bus width (64 bits in modern 64-bit processors) will take multiple cycles, which would require SR buffer management, reassembly of segmented SRs, and handling dropped SRs (caused by SR buffer overflow). Implementing them on hardware would significantly complicate the logic and increase the spatial cost. As implementing them on hardware would significantly complicate the logic and increase the spatial cost, we share an 8-byte SR design for both software and hardware engines. While it is challenging to hold the metadata of a general memory copy (two addresses and a copy length) within 8 bytes, we address this by adopting a small number of send/recv buffers, which reduces the address space by replacing general 8-byte addresses with a few bits of buffer indices. This is feasible thanks to the static nature of collective communication where the communicating entities are fixed – it enables offline pre-scheduling of data transfers so that receivers know which data arrives at which buffer without any additional metadata received at runtime. Meanwhile, the DMA requests on different buffers are pipelined for low latency and high throughput.

In terms of software, keeping an SR buffer would be more efficient as it would otherwise require extra control to prevent overwriting a previous SR. That is, unlike a hardware implementation where a fully received SR can immediately trigger the internal DMA pipeline at every cycle, a software thread could overwrite an unread SR unless the sender (GPU) coordinates with the receiver (the DMA stack) prior to sending a new SR. Unfortunately, such coordination would incur an extra delay as the GPU needs to read a remote flag on the DMA stack before sending an SR. We address this issue by maintaining a specialized ring buffer for SR, where the GPU checks only a local replica of the buffer head before sending an SR, and the replica is asynchronously updated by the DMA stack. This removes the coordination delay from the critical path of communication while providing a consistent SR interface for both software and hardware engines.

4.1.2 Software Engine

Our software engine harnesses CPU as the data plane while GPU serves as the control plane. We implement a CPU thread that busy-waits for SRs and invokes `cudaMemcpy` or RDMA writes accordingly, i.e., it leverages the existing hardware DMA engine on the sender GPU. Note that this is different from CPU-controlled communication as we use CPU only for data plane operations while the control plane (event handling) is managed by GPU threads. For high throughput, the busy-waiting loop drains all SRs in the ring buffer and invoke copy once for sending on a continuous memory space. Also, instead of slow `cudaEvent`, we use MMIO for the CPU-GPU communication that delivers SR, SC (Send Completion), and RC (Receive Completion) signals, which takes only 2∼3μs.

Alternatively, the software engine can perform MMIO with CPU threads instead of initiating the hardware DMA engine, which can reduce the `cudaMemcpy` overhead (i.e., sending a copy request from CPU to the DMA engine on GPU). However, this approach fails to achieve the line rate in most host

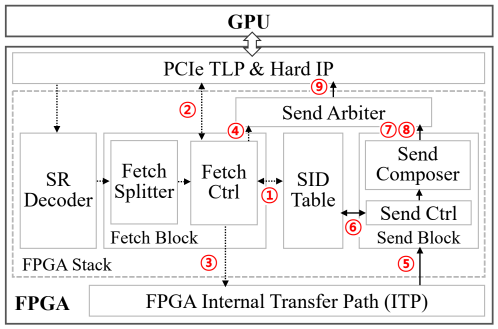


Figure 7: Implementation of the hardware DMA engine.

CPU architectures due to their poor throughput of crossing the PCIe root complex [41, 44]. This issue might be resolved in the future CPU architectures or by leveraging ARM cores on SmartNICs [4], which is left as our future work.

4.1.3 Hardware Engine

We implement a custom hardware with FPGA for DMA operations, which delivers two benefits over our software engine prototype. First, our hardware engine avoids the extra communication delay incurred by the overhead of `cudaMemcpy` as it performs DMA directly. Second, unlike existing hardware DMA engines on GPU, our custom hardware implements pipelining of multiple parallel DMA operations. This helps achieve a high data rate even for sending small data chunks. Table 2 shows resource usage of our implementation on an Intel Arria 10 FPGA.

Note that our FPGA prototype is limited to support the communication between only two GPUs and it does not support NVLink as there is no programmable hardware (or an off-the-shelf device) that can connect to NVLink. Instead, we consider it as a proof-of-concept that demonstrates the ideal benefit rather than a practical device that can be deployed on a large scale. A more practical implementation would be realized by future advances in CPU, GPU, or SmartNICs.

Figure 7 shows the hardware structure of inter-GPU communication stack on the FPGA. Unlike the existing GPU DMA engine, our DMA stack is designed to pipeline multiple DMA requests with different SIDs to be handled simultaneously. This is implemented by splitting a long-length request into multiple short-length sub-requests, which prevents head-of-line blocking and improves the PCIe throughput when GPU sends multiple different data at the same time. We explain how each request is processed by the sender- and the receiver-side stacks, respectively.

Sender side. When the sender stack receives an SR, the Fetch Block reads the decoded SR and retrieves the requested SID, which is translated into the physical source GPU address by looking up the SID Table (①). Using the address, the Fetch Ctrl fetches one sub-request at a time and it may fetch multiple times if the copy length is long. Each sub-request reads the corresponding source data from the GPU and stores it in a FIFO buffer of the Fetch Ctrl (②). When the source data is fully read from the GPU, the stored data and the sub-request are forwarded to the receiver stack through FPGA Internal Transfer Path (ITP). (③). After processing all sub-requests out of an SR, the Fetch Ctrl gives an SC flag to the Send Arbiter, which will be written on the GPU-side SC flag. (④).

Receiver side. The receiver stack receives the sub-request from the sender stack and stores the data into a FIFO buffer of the Send Ctrl (⑤). At the same time, the SID information in the sub-request is translated into the physical destination GPU address (⑥). The Send Ctrl sends the data to the destination address, and when it is done, the Send Composer sends an RC flag to the Send Arbiter, which will be written on the GPU-side RC flag (⑦, ⑧).

Module Name	ALMs		BRAM Blocks	
	#	Capacity	#	Capacity
FPGA Stack	14253	3.34%	188	6.93%
PCIe	1364	0.32%	13	0.48%

Table 2: Resource usage of a single DMA stack.

Resource usage and limitations. We implement the DMA stack on Intel Arria 10 FPGA [16]. Table 2 shows that each stack is implemented at a low cost, using only 14253 ALMs and 188 M20K BRAMs. Note that our current implementation supports communication between only two GPUs by directly connecting the FPGA ITP interfaces of their corresponding FPGA stacks. Our design considers leveraging DUA [41] to support routing between multiple stacks (either intra- or inter-machine), but we leave it as future work.

4.2 Loop Kernel Implementation

This section explains several details of optimizing the loop kernel performance in ARK.

Per-thread register optimization. GPU kernels often fine-tune the number of concurrent threads per SM by evaluating the trade-off between running more threads (gain more parallelism) vs. running fewer threads with more registers per each (gain more computational throughput per thread). So, the loop kernel also needs to tune it. The ARK scheduler generates multiple versions of the loop kernel with a different number of per-thread registers and picks the best-performing one. Actually, in NVIDIA GPUs, only 32, 64, 128, and 256 are available candidates due to hardware limitation.

Dependency on GPU Architecture. Section 3.2 explains that ARK launches one CTA per SM, but it may launch two or more CTAs per SM depending on the GPU architecture. This is because one CTA may be limited to utilize the entire resources of an SM in some architecture. In such cases, we need to launch two CTAs per SM to use the entire SM resources. The ARK scheduler automatically analyzes the resource requirement of the loop kernel and determines the number of CTAs per SM accordingly.

Program size. We reduce the program size of a loop kernel by coalescing multiple identical unit operators, e.g., if a model consists of many convolution operators, only several unique implementations of convolution will be actually defined, which are shared across all operators. Thus, the program size depends only weakly on the number of operators in the model. Instead, it is subject to the aggregate size of operator implementations, which is very limited – e.g., cuBLAS provides only ∼10 instances of a matrix-multiplication implementation on a single GPU architecture, while a loop kernel can accommodate over 5000 instances. This should cover an arbitrary DL program as the size of the matrix-multiplication implementation is one of the largest among the popular operators in DL.

5 Evaluation

We evaluate ARK by comparing it with existing DL frameworks largely in three different aspects. First, the fast inter-GPU communication of ARK contributes to higher end-to-end throughput and lower latency of DL applications. Second, the benefits on communication are obtained without losing the computational throughput of GPU. Third, ARK has flexibility to support various parallelism strategies including data-, tensor-, and pipeline-parallelism.

5.1 Experiment Setup

Software Engine. For experiments that use the software DMA engine, unless specified differently, we use two Intel Xeon Gold 6240R CPUs (48 lcores each, 2.40 GHz) and eight NVIDIA V100 GPUs. We have two NUMA nodes in the machine but only a single NUMA node hosts all GPUs, i.e. node 0 connects two PCIe v3 switches to its PCIe root complex and each switch is directly connected to 4 GPUs. For multi-node experiments, we use four Azure NDv4 SKUs [7] with 32x NVIDIA A100 GPUs in aggregate (8 per node), where each GPU has dedicated 200 Gbps NVIDIA Mellanox HDR InfiniBand connection.

Hardware Engine. For experiments that use the hardware DMA engine, we use an Intel Xeon Gold 5118 CPU (24 lcores, 2.30 GHz), two NVIDIA V100 GPUs, and an Intel Arria 10 FPGA. Both GPUs and the FPGA are behind the same PCIe v3 switch. We use the hardware engine only for experiments in Section 5.2 and Section 5.5.

5.2 DMA Engine Performance

Figure 8 compares the performance of communication between two GPUs with our DMA engines (G-Drv-S and G-Drv-H) over a CPU-controlled communication baseline (C-Drv). C-Drv is our own minimal implementation of a typical CPU-driven system, but unlike TensorFlow, C-Drv leverages asynchronous control using `cudaEvent` when the event is used only by GPUs, which further reduces CPU-GPU synchronizations to accelerate inter-GPU communication.

We measure the throughput by sending many parallel messages at the same time and reporting the maximum throughput

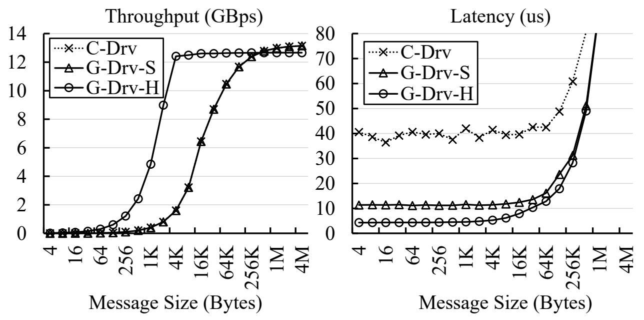


Figure 8: Performance comparison between the CPU-controlled communication (C-Drv) and the GPU-controlled DMA engines (G-Drv-S (software) and G-Drv-H (hardware)) over PCIe v3.

achieved with varying message sizes. For latency measurements, we implement a ping-pong application and report one-way latency – unlike throughput measurements, this includes communication event handling delays. This experiment assumes a favorable scenario for the CPU-controlled baseline where we can adopt the asynchronous control (explained in Section 2.2.2). In this scenario, a one-way trip requires triggering only two GPU events and two stream synchronizations.

In the left graph of Figure 8, our software engine (G-Drv-S) shows the same throughput as that of C-Drv, since both use `cudaMemcpy` for the data-plane. In contrast, our hardware engine (G-Drv-H) shows huge throughput improvement, saturating the bandwidth with only 8 KB messages while G-Drv-S needs 4 MB messages for saturation. This is because the hardware DMA engine pipelines processing multiple DMA requests while `cudaMemcpy` cannot. This improvement would be especially beneficial when GPU sends multiple messages to different destinations at the same time, e.g., all-to-all communication for expert-parallelism, which is popular for scaling out state-of-the-art Transformer-based models [11].

We note that the maximum achieved throughput of G-Drv-H is 3.68% lower than G-Drv-S. This is because an external DMA stack needs to send both read and write requests to sender and receiver GPUs, respectively, while the native DMA engine on the sender GPU needs to send only write requests. However, as the gap is small, it would not affect the end-to-end application performance much.

The right graph of Figure 8 shows that the one-way latency of C-Drv is at least ∼39.3µs on average. In contrast, G-Drv-S and G-Drv-H achieve 3.5x and 9.1x better latency, respectively. This is because our DMA engines handle the communication events directly in GPU threads while C-Drv relies on the `cudaEvent` interface that suffers from large overhead to trigger the events and synchronize streams. This improvement would be especially beneficial when GPUs perform split-and-gather of intermediate results to distribute the workload, as in tensor-parallelism [22, 26]. One thing to note about our DMA engine is that the benefit is obtained with little GPU cycle consumption. We evaluate this in the following section.

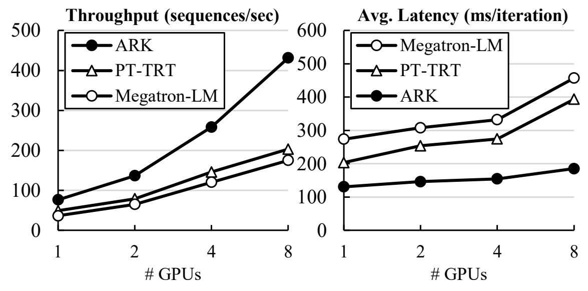


Figure 9: BERT-Large data-parallel training throughput and average latency per iteration with varying numbers of GPUs (sequence length 384, batch size 10, mixed-precision).

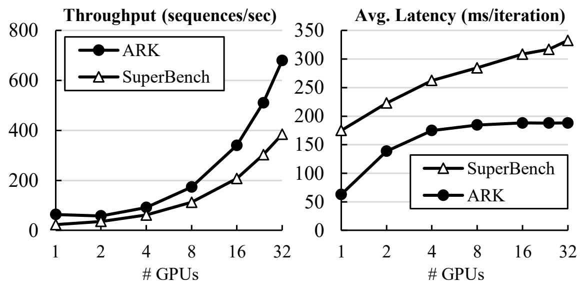


Figure 10: GPT-2 data-parallel training throughput and average latency per iteration with varying numbers of GPUs (sequence length 384, batch size 4, mixed-precision).

5.3 Avoiding Communication Interference

To compare the interference between computation and communication of using NCCL against using our DMA engine, we evaluate data-parallel training throughput of ARK by training representative NLP models.

Baselines. PT-TRT accelerates PyTorch [12] by adopting TensorRT [35], which does not scale out to multiple machines. Megatron-LM [26] is a PyTorch-based framework that supports large-scale training of NLP models but we use only for single-node experiments here. SuperBench [42] provides formal DL benchmarks for system performance evaluation also based on PyTorch, which we use for multi-node experiments. All baselines leverage NCCL [32] for communication.

Single Node. Single-node experiments train BERT-Large [10] model using up to 8x V100 GPUs as shown in Figure 9. The figure shows that ARK outperforms Megatron-LM and PT-TRT respectively by **2.46x** and **2.12x** with 8 GPUs. We find two reasons for the speedup.

First, NCCL adversely affects the computational throughput during back-propagation while ARK does not as it leverages DMA instead of employing GPU threads for data copy. Specifically, 64.5% of the end-to-end gap between ARK and PT-TRT with 8 GPUs is obtained as NCCL operations slow down due to the interference of MMIO with back-propagation computation, showing only 5.0 GBps of all-reduce throughput. We find that NCCL kernels result in 45.0% of slowdown of the overall back-propagation computation, an increase from 107.63 ms to 156.02 ms. On the other hand, our DMA engine suffers near-zero interference by initiating DMA directly instead of using MMIO, achieving 9.10 GBps of all-reduce throughput (**1.82x** faster).

Second, ARK performs more efficient computation on GPU. For example, for about 37.8% of the computation time of PT-TRT, it executes 1.2 thousands of memory-intensive kernels per iteration, such as element-wise arithmetic or intra-GPU data movement. Running these operators as separate kernels would be inefficient because it would incur unnecessary kernel launches and intra-GPU synchronizations. ARK largely reduces such overhead as it schedules all operators in a single loop kernel, similar as operator fusion [17, 24, 35].

Multiple Nodes. Multi-node experiments train the GPT-2 [36] model using up to 32x A100 GPUs as shown in Figure 10. All results use only InfiniBand for communication (no NVLink) and use the ring reduction algorithm. The figure shows that ARK outperforms SuperBench by **1.77x** with 32 GPUs. Furthermore, while per-iteration latency of SuperBench is consistently increasing, the increment in ARK is only marginal. This shows the efficiency of our communication stack over NCCL, which minimizes the interference between communication and computation. We also find a big computational benefit of ARK even without communication (when using a single GPU), which is further explained in the following section.

5.4 Offline Scheduling Evaluation

This section shows that the offline scheduler of ARK can generate comparable or even better GPU kernels comparing with existing DL optimization techniques. Rather than claiming state-of-the-art performance in DL optimization, we intend to show that the communication gain of our GPU-driven system does not come up with any computational performance drop.

We compare the inference performance of popular DL models over different frameworks using a single GPU. The DL models include image classification (ResNet-50 [14] and GoogLeNet [43]), object detection (SSD [23]), and NLP (BERT-Large [10]) models. TensorFlow (TF) is the primary comparison target of ARK because it supports flexible parallelism for DL applications like ARK. We also compare with TensorFlow-XLA (TF-XLA) [1] that implements automatic operator fusion in the TF back-end, but it is not always beneficial to the performance because the fused kernel might perform worse than using vendor-provided kernels (e.g. cuDNN) without fusion. Rammer [24] and TensorRT implement optimized operator fusion that often outperforms TF or TF-XLA, but they support only limited parallelism. For example, TensorRT supports only intra-node data-parallelism by adopting it to accelerate other frameworks like TF and PyTorch, as TensorRT itself does not support distributed execution. Nimble [20] presents careful asynchronous control (or ahead-of-time scheduling) of GPU kernels to reduce runtime

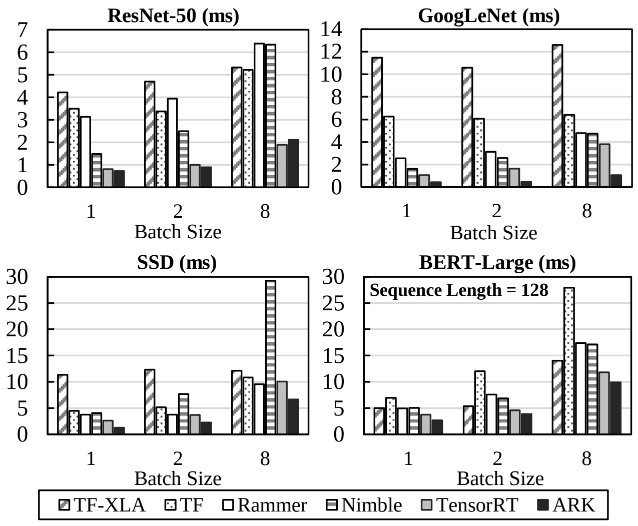


Figure 11: Inference latency comparison of popular DL models over different DL frameworks using a single GPU. All experiments use mixed-precision computation.

overhead of kernel launch and GPU events. As explained in Section 2.2.2, however, asynchronous control is limited to tackle the communication overhead. Nimble also works only on a single GPU at the moment.

Figure 11 shows that ARK achieves faster single-GPU inference against existing frameworks in most cases. For instance, ARK shows **1.11x∼3.56x** lower latency than TensorRT, except the case of ResNet-50 with batch size 8 that is ∼9.90% worse than TensorRT. This is because our matrix multiplication kernel is slower than the cuDNN [28] kernel used in TensorRT in this case (note that we implement convolution via matrix multiplication). ARK currently does not implement vCTAs specialized for large matrix multiplications (one side of the unit operator's output is larger than 256 elements), so it is often slower than existing kernels when the model consists of large matrix multiplications.

We note that the gain of ARK is especially large when the model consists of many parallel operators like GoogLeNet or SSD. This is because our high-level scheduler maximizes overall SM utilization by choosing the best vCTA (or unit operator) for each parallel operators. Specifically, when a lightweight operator runs alone in the GPU, we schedule it to use fine-grained vCTAs so that it utilizes more concurrent SMs. In contrast, when the GPU is overloaded due to other co-running operators, we need to use coarse-grained vCTAs to utilize SMs more efficiently. This is because coarse-grained vCTAs work on more input data at the same time and thus have more opportunities to better utilize the parallelism in an SM. As explained in Section 3.3, the optimization to find the best-performing vCTAs is easy in the ARK framework because it accurately estimates the performance with different vCTAs without running all candidates. We note that other frameworks do not provide a similar optimization like this.

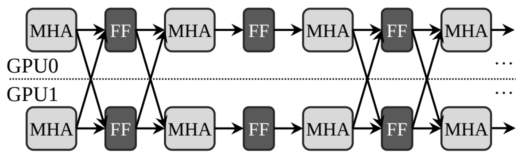


Figure 12: MoE model-parallel execution for Transformer architecture using 2 GPUs, composed of MHA (multi-headed attention) and FF (feed-forward) modules.

Architecture	Message Size (KB)	Time Gap (us)
BERT-Large [10]	256	60.9
GPT-3 XL [8]	512	187.4
T5 3B [37]	256	166.9
M4 [6]	256	60.9

Table 3: The message size and the smallest time gap between transactions for MoE inference. The input sequence length is 128. Time gaps are measured using the ARK framework.

5.5 Tensor-parallel Inference

This section presents the latency improvement with the tensor-parallel approach called mixture-of-experts (MoE) that efficiently scales up the Transformer [45] architecture, which is commonly used in many popular NLP models [6, 8, 11, 37]. This method is suggested to scale NLP models to one trillion of model parameters [11,22], but since we do not have enough GPUs to run the entire model, we evaluate the tensor-parallel inference of the model using two GPUs. In real practice, this is replicated to other GPUs to apply pipeline-parallelism (for training or inference) and data-parallelism (only for training) as well at the same time.

Figure 12 illustrates the MoE execution. The message size and the smallest time gap in-between the exchanges depend on the model hyperparameters, and some examples are shown in Table 3. Even though we present only 2-GPU experiments here, the result would be similar to a larger-scale one because MoE is designed to send each message only up to a small constant number (e.g. two in GShard [22]) of selected GPUs, not to all other GPUs.

We evaluate ARK using the hardware engine with three different comparison baselines – TF, TF-XLA, and C-Drv. Note that TensorRT-accelerated TensorFlow (TF-TRT) does not support model-parallelism, so it is not evaluated here.

Results in Figure 13 shows that ARK outperforms TF and TF-XLA by **1.66x∼3.48x** and **1.25x∼2.31x**, respectively. In terms of only the communication latency, ARK reduces it by 3.68x∼5.65x and 1.77x∼3.31x, respectively. Overall, C-Drv achieves better communication latencies over TF or TF-XLA, but its computation is less efficient because it reuses GPU kernel implementations in ARK but it does not benefit from ARK scheduler optimization. We also find that the GPU-driven communication of ARK delivers a substantial speedup over the CPU-driven communication of C-Drv, as shown in Section 5.2. We note that ARK computation is slower than

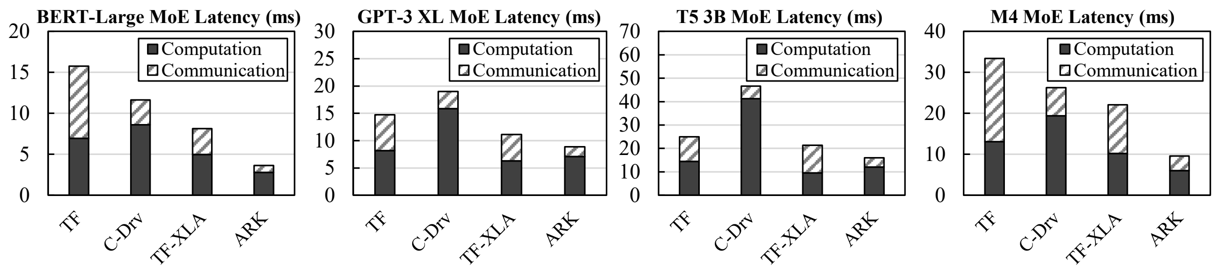


Figure 13: MoE inference latencies with different NLP model architectures (batch size 1, mixed-precision).

TF-XLA in GPT-3 XL and T5 3B. This is because our matrix multiplication kernel performs worse than TF-XLA in these cases, as explained in Section 5.4.

5.6 Pipeline-parallel Training

In this section, we train the GPT-3 [8] 6.7B model, which is the largest variation of GPT-3 that can fit the memory of eight V100 GPUs via pipeline-parallel training. The model consists of 32 sequential layers and each GPU trains 4 layers in the sequential order – GPU 0 reads the input data and runs the forward-pass of layer 0∼3, and the 16 MB output is passed to GPU 1, and so on. When GPU 7 completes the forward-pass, it moves on to the backward-pass of layer 31∼28, and the 16 MB of back-propagating gradient is passed to GPU 6, and so on. We use the mixed-precision computation and set the number of pipeline stages to 5, the batch size of each stage to 1, and the sequence length to 2048. ARK uses the emulated DMA stack in this evaluation.

In this experiment, the training throughputs of TF, TF-XLA, Megatron-LM, and ARK are 0.35, 0.47, 1.69, and 2.38 sequences per second, respectively, i.e. ARK outperforms TF, TF-XLA, and Megatron-LM by 6.80x, 5.06x, and 1.40x, respectively. In this case, most of the improvement of ARK comes from the computational efficiency on GPU, as pipeline-parallel training typically overlaps most of the communication delay with the computation time. This evaluation shows that ARK delivers the gain of operator fusion while supporting flexible parallelism for DL.

6 Future Work & Related Work

We expect that hardware advances in near future would enable more efficient implementations. For example, implementing our software DMA engine on SmartNIC would avoid the throughput issue of the PCIe root complex [44] via direct PCIe connection with GPUs (e.g., NVIDIA H100 CNX [9] combines GPU with SmartNIC), which enables efficient MMIO on SmartNIC. NVIDIA has announced their hardware accelerators for inter-GPU communication on SmartNICs (e.g., all-to-all engine on NVIDIA BlueField-3 [29]), which implies that a similar implementation with our hardware engine might be realized in the future. Additionally, host CPU architectures in the future may fix the root complex issue, which will enable our software DMA engine to replace `cudaMemcpy` with CPU-side MMIO, or even more efficiently, DMA engines on CPU (e.g., Intel I/OAT [15] or AMD PTDMA [21]).

ACE [38] proposes offloading the entire collective communication logic to a hardware accelerator that resides on intra-machine fabric, which cannot be extended to an external network (Ethernet, InfiniBand, etc). Our work differs from ACE as it is generally applicable to any (R)DMA networking and we can reuse most of existing software logic in popular collective communication libraries.

GPUnet [18] presents a network socket API set for GPU threads and leverages CPU intervention to let GPU threads to trigger DMA. This is inefficient as they add a substantial intervention overhead especially for small messages because they do not pipeline processing multiple DMA requests. Its throughput could be suboptimal as it implements a general socket interface on GPU while ARK reduces the overhead by leveraging offline scheduling to remove the metadata to be managed during runtime.

Nimble [20] accelerates DL execution by minimizing runtime scheduling overhead of kernels, but it works only on a single GPU. The proposed methods also cannot help reduce communication event handling overhead as it still relies on the CPU-side control using `cudaEvent` and multi-stream interfaces. ARK tackles this by letting GPU threads fully control all computation and communication tasks.

7 Conclusion

This paper envisions a GPU-driven code execution system that enables autonomous control of GPU throughout the entire lifetime of DL applications. We present the GPU-controlled DMA engine at the heart of the GPU-driven system that enables GPUs to communicate with each other without any external control. To avoid interference between computation and communication, we design our DMA engine and offline GPU scheduling to consume little GPU resources for communication, so that its high communication performance is delivered without sacrificing computational throughput of GPU. While our software engine already shows benefits over commodity hardware, we also present a proof-of-concept of a hardware engine that shows even higher performance, which indicates that our system performance would be further improved with future advances in commodity hardware such as CPU, GPU, or SmartNIC.

Acknowledgements

We appreciate the feedback by our shepherd, Danyang Zhuo, as well as anonymous reviewers of NSDI'23. This work is in part support by the ICT Research and Development Program of MSIT/IITP, Korea, under [2022-0-00531, Development of in-network computing techniques for efficient execution of AI applications] and [2018-0-00693, Development of an ultra low-latency user-level transfer protocol].

References

[1] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2021. [Online; accessed Dec 2022].

[2] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2017.

[3] AMD. Introducing AMD CDNA™ 2 Architecture. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf, 2021. [Online; accessed Dec 2022].

[4] AMD. Alveo SN1000 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html, 2022. [Online; accessed Dec 2022].

[5] AMD. ROCm Communication Collectives Library (RCCL). https://github.com/ROCmSoftwarePlatform/rccl, 2022. [Online; accessed Dec 2022].

[6] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George F. Foster, Colin Cherry, Wolfgang Macherey, Zhifeng Chen, and Yonghui Wu. Massively multilingual neural machine translation in the wild: Findings and challenges. *CoRR*, abs/1907.05019, 2019.

[7] Microsoft Azure. ND A100 v4-series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, 2022. [Online; accessed Dec 2022].

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[9] Charu Chaubal. Build Mainstream Servers for AI Training and 5G with the NVIDIA H100 CNX. https://developer.nvidia.com/blog/build-mainstream-servers-for-ai-training-and-5g-with-the-nvidia-h100-cnx/, 2022. [Online; accessed Dec 2022].

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.

[11] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[12] The Linux Foundation. PyTorch. https://pytorch.org, 2022. [Online; accessed Dec 2022].

[13] The Linux Foundation. How Computational Graphs are Constructed in PyTorch. https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/, 2023. [Online; accessed Jan 2023].

[14] 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 (CVPR)*, 2016.

[15] Intel. Fast memcpy with SPDK and Intel® I/OAT DMA Engine. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html, 2017. [Online; accessed Dec 2022].

[16] Intel. Intel® FPGAs - Intel® Arria® 10 FPGAs. https://www.intel.com/content/www/us/en/products/details/fpga/arria/10.html, 2022. [Online; accessed Dec 2022].

[17] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[18] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for GPU programs.

In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andrés Felipe Cruz-Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models. *CoRR*, abs/2109.10465, 2021.

[20] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[21] Michael Larabel. AMD PTDMA Driver Landing For Linux 5.15 After Two Years In The Works – Phoronix. https://www.phoronix.com/scan.php?page=news_item&px=AMD-PTDMA-For-Linux-5.15, 2021. [Online; accessed Dec 2022].

[22] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[24] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] Microsoft. ONNX Runtime. https://onnxruntime.ai/, 2023. [Online; accessed Jan 2023].

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters. *CoRR*, abs/2104.04473, 2021.

[27] NVIDIA. Using NCCL with CUDA Graphs. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/cudagraph.html, 2020. [Online; accessed Dec 2022].

[28] NVIDIA. CUDA Deep Neural Network (cuDNN). https://developer.nvidia.com/cudnn, 2021. [Online; accessed Dec 2022].

[29] NVIDIA. NVIDIA BlueField-3 DPU – Programmable Data Center Infrastructure On-a-Chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2021. [Online; accessed Dec 2022].

[30] NVIDIA. cuBLAS. https://developer.nvidia.com/cublas, 2022. [Online; accessed Dec 2022].

[31] NVIDIA. GPUDirect. https://developer.nvidia.com/gpudirect, 2022. [Online; accessed Dec 2022].

[32] NVIDIA. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl, 2022. [Online; accessed Dec 2022].

[33] NVIDIA. NVLink & NVSwitch: Fastest HPC Data Center Platform. https://www.nvidia.com/en-us/data-center/nvlink/, 2022. [Online; accessed Dec 2022].

[34] NVIDIA. PTX ISA – Cache Operators. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cache-operators, 2022. [Online; accessed Dec 2022].

[35] NVIDIA. TensorRT SDK. https://developer.nvidia.com/tensorrt, 2022. [Online; accessed Dec 2022].

[36] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[37] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[38] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[39] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[41] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to FPGA. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] SuperBench. SuperBench Documentation. https://microsoft.github.io/superbenchmark/, 2022. [Online; accessed Dec 2022].

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[44] Nathan R Tallent, Nitin A Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. Evaluating on-node gpu interconnects for deep learning workloads. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2017.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017.