PATHWAYS: ASYNCHRONOUS DISTRIBUTED DATAFLOW FOR ML

Paul Barham [1] **Aakanksha Chowdhery** [1] **Jeff Dean** [1] **Sanjay Ghemawat** [1] **Steven Hand** [1] **Dan Hurt** [1]
Michael Isard [1] **Hyeontaek Lim** [1] **Ruoming Pang** [1] **Sudip Roy** [1] **Brennan Saeta** [1] **Parker Schuh** [1]
Ryan Sepassi [1] **Laurent El Shafey** [1] **Chandramohan A. Thekkath** [1] **Yonghui Wu** [1]

ABSTRACT

We present the design of a new large scale orchestration layer for accelerators. Our system, PATHWAYS, is explicitly designed to enable exploration of new systems and ML research ideas, while retaining state of the art performance for current models. PATHWAYS uses a *sharded* dataflow graph of *asynchronous* operators that consume and produce futures, and efficiently gang-schedules *heterogeneous* parallel computations on thousands of accelerators while coordinating data transfers over their dedicated interconnects. PATHWAYS makes use of a novel *asynchronous distributed dataflow* design that lets the control plane execute in parallel despite dependencies in the data plane. This design, with careful engineering, allows PATHWAYS to adopt a single-controller model that makes it easier to express complex new parallelism patterns. We demonstrate that PATHWAYS can achieve performance parity ($\sim 100\%$ accelerator utilization) with state-of-the-art systems when running SPMD computations over 2048 TPUs, while also delivering throughput comparable to the SPMD case for Transformer models that are pipelined across 16 stages, or sharded across two islands of accelerators connected over a data center network.

1 INTRODUCTION

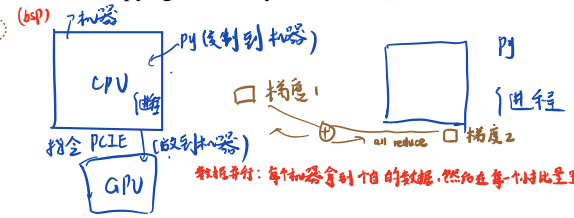
Deep learning has seen remarkable achievements over the last decade, across domains from image understanding (Krizhevsky et al., 2012; He et al., 2016) to natural language processing (Devlin et al., 2019; Brown et al., 2020). This rapid recent progress of machine learning (ML) has been characterized by the co-evolution of ML models, accelerator hardware, and the software systems that tie the two together. This co-evolution poses a danger that systems become over-specialized to *current* workloads and fail to anticipate future needs. In this paper, we describe PATHWAYS, a new system built for distributed ML. PATHWAYS is designed to target specific capabilities that we believe will be needed by future ML workloads (Dean, 2021) – and are therefore needed *today* to support research into those workloads – but which are poorly supported by state-of-the-art systems.

For example, most of today's state-of-the-art ML workloads use a "single program multiple data" (SPMD) model, inspired by MPI (Clarke et al., 1994), where all accelerators run the same computation in lockstep and communication between accelerators is described by collectives like AllReduce. Recently, researchers have begun to run into the limits of SPMD for ML computations. Very large language models have been scaled up using pipelining rather than pure data-parallelism (Narayanan et al., 2019; Rasley et al., 2020; Narayanan et al., 2021), and models such as Mixture of Experts (MoE) (Shazeer et al., 2017) have started to explore computational sparsity that is most naturally expressed using fine-grain control flow and heterogeneous computation across accelerators. System designers have adopted ingenious techniques to execute pipelined (Narayanan et al., 2021; Rasley et al., 2020; Narayanan et al., 2019; Huang et al., 2019) and *homogeneous* MoE (Lepikhin et al., 2020; Fedus et al., 2021) models on MPI-style systems, but as we argue in detail later, the MPI programming model is too restrictive both for users and for the underlying system.

On the other hand, with each new generation of accelerators, ML clusters are becoming increasingly heterogeneous (Jeon et al., 2019; Chaudhary et al., 2020; Weng et al., 2022). Providing exclusive access to large "islands" of homogeneous accelerators connected over high-bandwidth interconnects is expensive, and often wasteful as a single user program must try to keep all of the accelerators continuously busy. Such constraints are further driving researchers towards "multiple program multiple data" (MPMD) computations that allow more flexibility by mapping sub-parts of the overall computation to a collection of more readily available smaller islands of accelerators. To increase utilization, some ML hardware resource management researchers (Xiao et al., 2020; Bai et al., 2020; Yu and Chowdhury, 2020; Wang et al., 2021; Lim et al., 2021; Zhao et al., 2022; Weng et al., 2022)

[1]Google. Correspondence to: PATHWAYS authors <pathways-mlsys@google.com>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022.



multiplex hardware in a fine-grained manner between workloads, enabling workload elasticity, and improving fault tolerance.

Finally, researchers are beginning to standardize on a set of *foundation models* (Bommasani and et. al., 2021; Dean, 2021) that are trained on large data at scale and are adaptable to multiple downstream tasks. Training and inference for such models offers opportunities for improving cluster utilization by multiplexing resources across many tasks, and efficiently *sharing* state between them. For example, several researchers might concurrently fine-tune (Houlsby et al., 2019; Zhang et al., 2021) a foundation model for different tasks, using the same accelerators to hold the fixed foundation model layers. Training or inference over shared sub-models can benefit from techniques that allow examples from different tasks to be combined in a single vectorized batch to get better accelerator utilization (Crankshaw et al., 2017).

This paper describes our system, PATHWAYS, which matches the functionality and performance of state of the art ML systems, while providing the capabilities needed to support future ML workloads. PATHWAYS uses a client-server architecture that enables PATHWAYS's runtime to execute programs on system-managed islands of compute on behalf of many clients. PATHWAYS is the first system designed to transparently and efficiently execute programs spanning multiple "pods" of TPUs (Google, 2021), and it scales to thousands of accelerators by adopting a new dataflow execution model. PATHWAYS's programming model makes it easy to express non-SPMD computations and enables centralized resource management and virtualization to improve accelerator utilization.

In the remainder of the paper we first discuss the limitations of current distributed ML systems and motivate our design choices for PATHWAYS (§2), and next describe the flexible programming model that PATHWAYS supports (§3). We describe PATHWAYS's architecture (§4), highlighting how we have addressed the key limitations of older client-server ML systems using a *sharded dataflow model* and *asynchronous gang-scheduling*. We present both micro-benchmarks and end-to-end evaluations using real ML models that demonstrate we have met the goal of matching the performance of state-of-the-art multi-controllers for realistic workloads (§5), and validate that PATHWAYS's mechanisms are well suited to support the features needed for the research and deployment of novel and efficient ML methods.

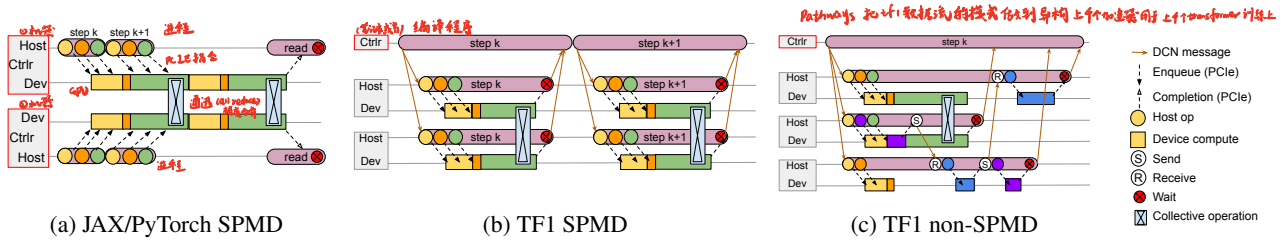
2 DESIGN MOTIVATION

The design choices of distributed ML systems are often driven by the properties of the underlying target hardware accelerators. We refer readers to Appendix A for a discussion on some of these properties and how they typically influence distributed ML systems. Here, we focus on how some of the design and implementation choices of existing distributed ML systems make it hard for them to support large, sparse or irregular models.

Distributed ML systems for training state-of-the-art SPMD models often adopt a *multi-controller* architecture where the same client executable is run directly on all the hosts in the system, taking exclusive ownership of the resources on those hosts for the duration of the program execution. Examples of this architecture include MPI (Clarke et al., 1994), PyTorch (Paszke et al., 2019), JAX (Bradbury et al., 2018), and more recent configurations of TensorFlow (Shazeer et al., 2018; Agrawal et al., 2019). The key advantage of this architecture is the low latency for dispatching accelerator computations (see Figure 1a) since an identical copy of the user's code runs on each of the accelerator hosts and dispatch involves communication only over (relatively) fast PCIe links. All other communication across hosts only happens through collectives that use dedicated interconnects like NVLink (Foley and Danskin, 2017) and ICI (Jouppi et al., 2020) without going via host memory. However, this architecture is a poor match for modern ML workloads that use pipelining or computational sparsity. Any communication beyond standard collectives in multi-controller systems requires users to implement their own coordination primitives. The multi-controller approach also typically assumes exclusive ownership of hardware resources. This not only shifts the responsibility of ensuring high utilization of the expensive accelerators on to the user, but also complicates the design of features like resource virtualization and multiplexing that are needed to build efficient cluster-wide ML infrastructure.

Single-controller systems such as TensorFlow v1 (Abadi et al., 2016) offer a very general distributed dataflow model, including optimized in-graph control flow (Yu et al., 2018). A TensorFlow (TF) Python client builds a computation graph and hands it off to a coordinator runtime, which partitions the graph into a subgraph for each worker and delegates the execution of the subgraphs to local runtimes on workers. Coordination between workers is performed using data- and control-edges passing messages over the datacenter network (DCN). While the single-controller design offers a flexible programming model and virtualization of resources, it presents implementation challenges.

Firstly, while multi-controller systems only require communication over PCIe to dispatch accelerator computations (Figure 1a), clients in single-controller systems are "farther away" and the dispatch latency involves communication over DCN, typically an order of magnitude slower than PCIe (Figure 1b). Secondly, to support concurrent execution of MPMD programs with SPMD sub-computations, each span-



(a) JAX/PyTorch SPMD (b) TF1 SPMD (c) TF1 non-SPMD

Figure 1. Comparison of dispatch overheads and communication patterns between multi-controller and single-controller systems. (a) Jax or PyTorch SPMD independently enqueues accelerator computations asynchronously over fast PCIe; (b) TensorFlow v1 SPMD requires control messages over slower DCN; (c) TensorFlow v1 non-SPMD programs require cross-host coordination or data transfer through explicit send (S) and recv (R) ops.

ning a subset of accelerators drawn from a shared cluster, the runtime must have some mechanism to support gang-scheduling of accelerator computations. Gang-scheduling is essential in the case of TPUs, since they are single-threaded and only run non-preemptible kernels, so the system will deadlock if communicating computations are not enqueued in a consistent order. Even for GPUs or other accelerators that can execute concurrent computations, gang scheduling allows more efficient execution of collectives (Feitelson and Rudolph, 1992). Single-controller systems for ML therefore require a distributed scheduling mechanism to order the computations enqueued on behalf of different programs. Finally, a system for modern ML workloads must be designed to run computations distributed over thousands of accelerators, with first class support for sharded representations and data structures. For instance, a naive dataflow graph representing an edge between an M-way sharded computation and an N-way sharded computation would require $M + N$ nodes and $M \times N$ edges, rapidly becoming unwieldy.

The implementation choices made by TF v1 were over-specialized to assume a single, smallish, exclusively-owned island of accelerators. This over-specialization makes it practically infeasible to use TF for contemporary or future ML workloads. While TF can run computations that require cross-host coordination or data transfer through send and recv ops (Figure 1c), host side work at the destination like dispatching the accelerator computation is triggered only after the transfer is completed. In programs involving many cross-host transfers, for example pipelined models with a large number of stages, these dispatch latencies accumulate, leading to inefficient accelerator utilization. While TF v1 users can (inefficiently) enforce a consistent ordering for gang-scheduling within a single program, by using control edges, the lack of a centralized scheduler in single-controller systems like TF v1 makes it impossible to ensure consistent ordering between computations *across* programs. TF also materializes the full sharded computation graph, which introduces substantial overhead in both graph serialization and execution when the number of shards reaches into the thousands, leading to millions of graph edges between sub-computations.

PATHWAYS combines the flexibility of single-controller frameworks with the performance of multi-controllers. We adopt a single-controller model since we believe it offers much better opportunities than multi-controller for *novel* and *efficient* ML computation, both by exploiting computational sparsity and heterogeneity, and by enabling cluster management systems that promote sharing and virtualizing resources. Our design differs from older single-controller ML systems in that it uses asynchronous dispatch to match the performance of multi-controller systems, supports centralized resource management and scheduling with first-class support for gangs of SPMD accelerator computations, and uses a sharded dataflow system for efficient coordination.

3 PATHWAYS PROGRAMMING MODEL

We have implemented support to target PATHWAYS from source programs written in TensorFlow and JAX, but we concentrate on JAX for the evaluation in this paper. JAX users can explicitly wrap standard Python code with decorators to indicate fragments that should be compiled into (potentially SPMD) XLA computations. These XLA computations are usually characterized by known input and output types and shapes, bounded loops, and with few (if any) conditionals (see Appendix B for more details) making it feasible to estimate the resource requirements of computations in advance. We refer to these computations with known resource requirements as "compiled functions". Each such function maps to a single (sharded) computation node in a PATHWAYS program.

JAX today cannot scale beyond a single TPU pod since JAX programs that run in multi-controller configurations transfer all data using XLA collectives, and these are only currently available over ICI on TPU. PATHWAYS can be used as a plug-in replacement for the JAX backend, allowing JAX code to run unmodified except that SPMD computations now have access not just to the locally connected TPU cores, but to as many cores as are provisioned in the system. And since PATHWAYS can communicate over both ICI and DCN, it allows JAX programs to scale for the first time to multiple TPU pods, containing many thousands of TPU cores.

```python
def get_devices(n):
  """Allocates `n` virtual TPU devices on an island."""
  device_set = pw.make_virtual_device_set()
  return device_set.add_slice(tpu_devices=n).tpus

a = jax.pmap(lambda x: x * 2., devices=get_devices(2))
b = jax.pmap(lambda x: x + 1., devices=get_devices(2))
c = jax.pmap(lambda x: x / 2., devices=get_devices(2))

@pw.program  # Program tracing (optional)
def f(v):
  x = a(v)
  y = b(x)
  z = a(c(x))
  return (y, z)

print(f(numpy.array([1., 2.])))
# output: (array([3., 5.]), array([2., 4.]))
```

Figure 2. Python user code example for PATHWAYS running sharded computations across multiple islands of TPU.

The ability to run unmodified JAX code is convenient but does not unlock the full performance of PATHWAYS. A PATHWAYS user may request sets of "virtual devices", with optional constraints on the device types, locations or interconnect topology, and is then able to place specific compiled functions on those devices (Figure 2). The system will automatically handle all data movement and resharding between dependent computations.

By default, we convert each compiled function into a standalone PATHWAYS program containing just one (sharded) computation, meaning that if a user wants to run many functions back to back, a separate Python call and RPC from client to coordinator is required for each function. We therefore also implemented a new *program tracer* (Figure 2) that a user can wrap around a block of Python code that calls many compiled functions. The tracer generates a single PATHWAYS program where each compiled function is represented by a computation node in a dataflow graph.

JAX's philosophy of supporting *transforms* of traced code is a good match for the research directions we want to explore. For example, JAX has a companion library called FLAX (Heek et al., 2020) that is used to express layered DNN models, and we have written a library that automatically converts a FLAX model into a pipelined PATHWAYS program. In addition, JAX supports transforms to vectorize "per-example" Python functions, producing efficient batched code, and such transforms are a good basis for exploring new forms of data-dependent vectorized control flow, as we briefly describe later (§6.3).

4 PATHWAYS SYSTEM ARCHITECTURE

PATHWAYS builds extensively on prior systems, including XLA (TensorFlow, 2019) to represent and execute TPU computations, TensorFlow graphs and executors (Abadi et al., 2016) to represent and execute distributed CPU computations, and Python programming frameworks including JAX (Bradbury et al., 2018) and TensorFlow APIs. By leveraging these building blocks we are able to focus on the novel coordination aspects of PATHWAYS, while being able to run existing ML models with minimal code changes.

4.1 Resource Manager

A PATHWAYS backend consists of a set of accelerators grouped into tightly-coupled islands that are in turn connected to each other over DCN (Figure 3). PATHWAYS has a "resource manager" which is responsible for the centralized management of devices across all of the islands. A client may ask for "virtual slices" of the island with specific 2D or 3D mesh shapes that suit their communication pattern. Each virtual slice contains "virtual devices" that allow the client to express how computations are laid out on the mesh. The resource manager dynamically assigns physical devices for virtual devices satisfying the desired interconnect topology, memory capacity, etc.

Our initial resource manager implementation uses a simple heuristic that attempts to statically balance load by spreading computations across all available devices, and keeps a one to one mapping between virtual and physical devices. If future workloads require it we can adopt a more sophisticated allocation algorithm, for example taking into account the resource requirements of all client computations and the current state of the system to approximate an optimal allocation of physical devices to computations.

PATHWAYS allows backend compute resources to be added and removed dynamically, with the resource manager tracking available devices. The layer of indirection between virtual and physical devices, as enabled by our single-controller design, will allow us in future to support features like transparent suspend/resume and migration, where a client's virtual devices are temporarily reclaimed or reassigned without the need for cooperation from the user program.

4.2 Client

When the user wants to run a traced program, it calls the PATHWAYS client library which first assigns virtual devices to any computations that have not been run before, and registers the computations with the resource manager, triggering the servers to compile the computations in the background. The client then constructs a device location-agnostic PATHWAYS intermediate representation (IR) for the program, expressed as a custom MLIR (Lattner et al., 2021) dialect. The IR is progressively "lowered" via a series of standard compiler passes, which eventually output a low-level representation that includes the physical device locations. This low-level program takes into account the network connectivity between physical devices and includes operations to

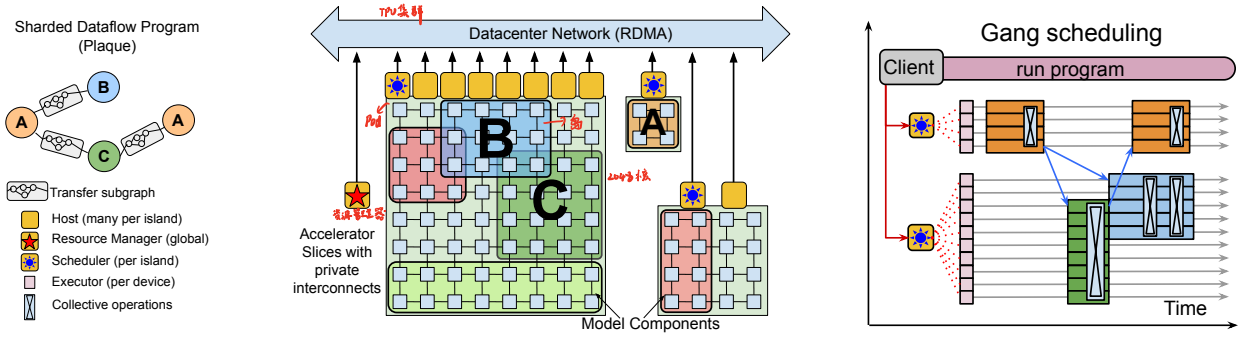


Figure 3. PATHWAYS system overview. (Left) Distributed computation expressed as a DAG where each node represents an individual compiled function, and edges between nodes represent data flows between functions. (Middle) Resource Manager allocates subsets of an island's accelerators ("virtual slices") for each compiled function. (Right) Centralized schedulers for each island gang-schedule computations that are then dispatched by per-shard executors. Red arrows indicate control messages, blue arrows show data-path transfers.

transfer outputs from a source computation shard to the locations of its destination shards, including scatter and gather operations when a data exchange is required. It is efficient to repeatedly run the low-level program in the common case that the virtual device locations do not change, and the program can be re-lowered if the resource manager changes the mapping between virtual and physical devices.

The client in older single controller systems can quickly become a performance bottleneck as it coordinates thousands of individual computations and data buffers corresponding to each shard of computations spread across thousands of accelerators. The PATHWAYS client uses a *sharded buffer* abstraction to represent a logical buffer that may be distributed over multiple devices. This abstraction helps the client scale by amortizing the cost of bookkeeping tasks (including reference counting) at the granularity of logical buffers instead of individual shards.

4.3 Coordination implementation

PATHWAYS relies on PLAQUE for all cross-host coordination that uses DCN. PLAQUE is an existing (closed-source) production sharded dataflow system used at Google for many customer-facing services where high-fanout or high-fanin communication is necessary, and both scalability and latency are important. The low-level PATHWAYS IR is converted directly to a PLAQUE program, represented as a dataflow graph. PATHWAYS has stringent requirements for its coordination substrate, all of which are met by PLAQUE.

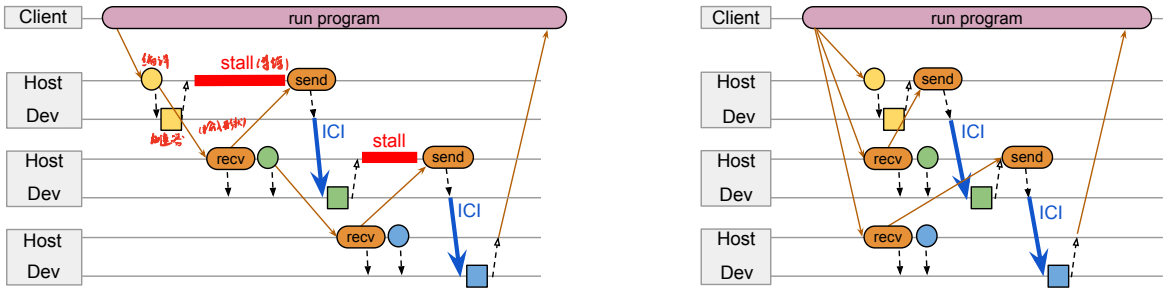
First, the representation used to describe the PATHWAYS IR must contain a single node for each sharded computation, to ensure a compact representation for computations that span many shards, i.e. a chained execution of 2 computations A and B with N computation shards each should have 4 nodes in the dataflow representation: $Arg \rightarrow Compute(A) \rightarrow Compute(B) \rightarrow Result$, regardless of the choice of N. In the PLAQUE runtime implementation each node generates output data tuples tagged with a destination shard, so when performing data-parallel execution N data tuples would flow, one between each adjacent pair of IR nodes.

The coordination runtime must also support *sparse* data exchanges along sharded edges, in which messages can be sent between a dynamically chosen subset of shards, using standard progress tracking mechanisms (Akidau et al., 2013; Murray et al., 2013) to detect when all messages for a shard have been received. Efficient sparse communication is a requirement to avoid the DCN becoming a bottleneck for data-dependent control flow on accelerators, which is one of the key capabilities that we want PATHWAYS to enable.

The coordination substrate is used to send DCN messages that are in the critical path for transmitting scheduling messages and data handles (Figure 4), so it must send critical messages with low latency, and batch messages destined for the same host when high throughput is required.

It is also convenient to use an extensible, general-purpose, dataflow engine to handle DCN communication, since this means that PATHWAYS can also use it for background housekeeping tasks such as distributing configuration information, monitoring programs, cleaning them up, delivering errors on failures, and so on.

We believe that it would be feasible to re-implement the full PATHWAYS design using other distributed frameworks such as Ray (Moritz et al., 2018) rather than PLAQUE to realize the low-level coordination framework. In such an implementation, PATHWAYS executors and schedulers would be replaced by long-running Ray actors that would implement PATHWAYS scheduling on top of the underlying Ray cluster scheduling, and executors could use PyTorch for GPU computation and collectives. Some additions might be required to attain comparable performance (see §5) because Ray lacks, for example, an HBM object store, or primitives to efficiently transfer remote objects over the GPU interconnect.



(a) Sequential dispatch

(b) Parallel dispatch

Figure 4. Sequential vs. Parallel dispatch for a three-node program. When a computation's on-device execution time is *shorter* than the time spent in scheduling, resource allocation, and coordination, the asynchronous pipeline stalls because of host-side work in sequential dispatch. Parallel asynchronous dispatch overcomes this bottleneck by running the host-side work in parallel, exploiting the statically known resource usage of regular compiled functions. Schedulers are omitted for brevity.

4.4 Gang-scheduled dynamic dispatch

As discussed previously (§2), one requirement for supporting SPMD computations on a shared set of accelerators is to support efficient gang-scheduling. The PATHWAYS runtime includes a centralized scheduler per island that consistently orders all of the computations in the island. As PATHWAYS enqueues a program for execution, the PLAQUE dataflow program is responsible for *(i)* enqueueing the execution of local compiled functions at each accelerator, with buffer futures as inputs; *(ii)* enqueueing network sends to remote accelerators for the buffer futures output by function executions; and *(iii)* communicating with the scheduler to determine a consistent order of function executions across all programs running on the island. The scheduler must implement policies for allocating accelerators at a timescale of milliseconds. Our current implementation simply enqueues work in FIFO order, but more sophisticated schedulers might for example reorder computations based on estimated execution times.

4.5 Parallel asynchronous dispatch

When running computations on accelerators, systems can take advantage of asynchronous APIs to overlap computation with coordination (Kwon et al., 2020). Consider the three-node graph in Figure 4a, where the squares correspond to three nodes A, B, and C running on accelerators attached to hosts A, B, and C. All node computations are regular compiled functions. Host A enqueues node A, receives a future for A's outputs, and transmits the future to host B. Host B allocates B's inputs, transmits the input buffer addresses to host A, and performs most of the preparatory work to launch node B's function. When node A completes, its outputs are sent via the accelerator interconnect directly into node B's input buffers, and then host B starts node B. The latency between one node completing and the next node starting can be made to be little more than the data transfer time.

The above design works well when a predecessor node's computation takes *longer* than the time spent in scheduling, resource allocation, and coordination between hosts. However if the computation time is too short, which is the case shown in the figure, the asynchronous pipeline stalls and the host-side work becomes the critical bottleneck for executing the overall sequence of computations. Given that the compiled functions are all regular, a successor node's input shapes can in practice be computed *before* the predecessor computation was even enqueued.

We therefore introduce a novel *parallel asynchronous dispatch* design shown in Figure 4b, which exploits the statically known resource usage of regular compiled functions to run most of the host-side work for a computation's nodes in parallel, rather than serializing the work for a node to happen after its predecessors have been enqueued. Since work can only be scheduled in parallel when functions are regular, PATHWAYS treats parallel scheduling as an optimization and falls back to the traditional model when a node's resource requirements are not known until a predecessor computation has completed (e.g., due to data-dependent control flow). When a subgraph of a computation can be scheduled statically, the program sends a single message (describing the entire subgraph) to the scheduler, which is able to sequence the execution of all the active shards in the subgraph back to back. The use of a single message is designed to minimize network traffic, but does not require the scheduler to actually *enqueue* all the subgraph's shards as a batch: computations may still be interleaved with those submitted by other concurrently executing programs. We evaluate the cost of different dispatch mechanisms in §5.

4.6 Data management

Each host manages a *sharded object store* that is similar to Ray's object stores (Moritz et al., 2018), but extended to also track buffers held in accelerator HBM at each shard. Client programs can hold references to objects in remote host or

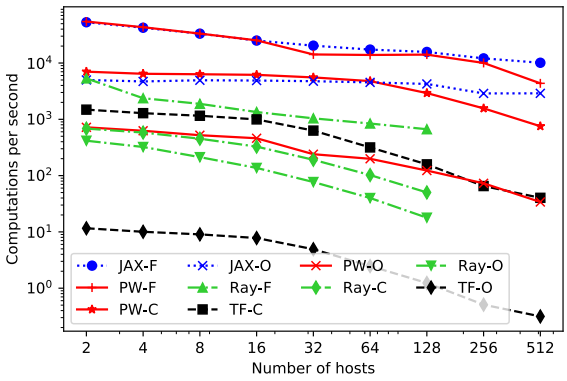


Figure 5. Dispatch overhead of PATHWAYS compared to TF, JAX, and Ray. PATHWAYS outperforms single-controller systems like TF and Ray on all configurations, and matches the performance of multi-controller JAX in Fused (-F) and Chained (-C) configurations for up to 1000 and 256 TPU cores, respectively. Each computation comprises a single scalar AllReduce followed by a scalar addition.

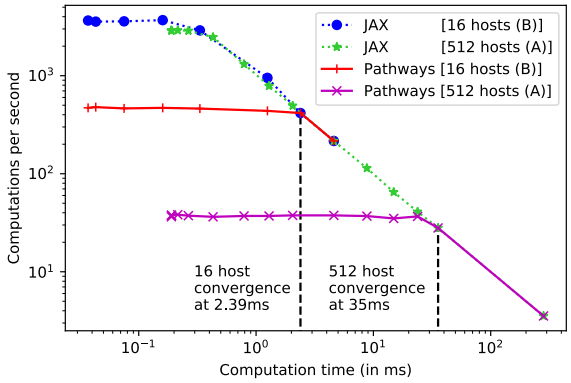


Figure 6. Smallest computation to match throughput between PATHWAYS and JAX, masking the single-controller overhead. PATHWAYS matches JAX throughput for a computation size of at least 2.3 ms for 16 hosts with 128 TPUs on configuration (B), and for a computation size of at least 35 ms for 512 hosts with 2048 TPUs on configuration (A).

accelerator memory, and the client and servers refer to them using opaque handles that allow the system to migrate them if needed. Intermediate program values are also kept in the object stores, for example while the system is waiting to transfer them between accelerators, or pass them to a subsequent computation. The objects are tagged with ownership labels so that they can be garbage collected if a program or client fails. We can use simple back-pressure to stall a computation if it cannot allocate memory because other computations' buffers are temporarily occupying HBM.

5 EVALUATION

For evaluating JAX, PATHWAYS, and TensorFlow on TPU we use three different configurations. Configuration (**A**) has 4 TPUs per host, and the largest instance we report on has 512 hosts, resulting in 2048 total TPUs connected via ICI. Configuration (**B**) has 8 TPUs per host, and the largest instance we report on has 64 hosts, and a total 512 TPUs. Configuration (**C**) uses four islands of TPUs, where each island has 4 hosts and 32 TPUs. We note in the text when experiments use a subset of the TPUs of a particular configuration.

When evaluating Ray on GPU we use Ray v1.3 and PyTorch 1.8.1 running on `p3.2xlarge` VMs[1] with hosts connected via DCN and scheduled using Amazon placement groups.

We mostly compare PATHWAYS against multi-controller JAX, since JAX has demonstrated state of the art performance in industry standard benchmarks (Mattson et al., 2020) and we can easily run JAX and PATHWAYS (PW) on identical hardware configurations. We also compare against TensorFlow (TF) and Ray in micro-benchmarks, to examine specific aspects of PATHWAYS's distributed system performance, and show pipelined performance of a TF model running on PATHWAYS.

5.1 Single-controller dispatch overheads

Our first experiment is a micro-benchmark to compare the overheads of JAX multi-controller with single-controller frameworks. We construct programs that repeatedly run a trivial gang-scheduled computation containing a single AllReduce of a scalar followed by a scalar addition, feeding the output of one computation to the input of the next. We measure the throughput: the number of computations per second that execute on the accelerators. We compare three ways that the user code can enqueue the computations:

- **OpByOp (-O):** The user code contains a separate call for each execution of the computation.
- **Chained (-C):** The user code contains a series of calls each of which executes a chain of 128 nodes, where each node executes the computation. The system executes the entire chain in response to a single client call.
- **Fused (-F):** The user code contains a series of calls each of which executes a single computation node, where the node contains a chain of 128 computations.

For JAX multi-controller, OpByOp means JIT-compiling a function containing one computation and calling it repeatedly from Python, and Fused means JIT-compiling a function containing a chain of computations. There is no analog of Chained for a multi-controller. For PATHWAYS, OpByOp and Fused use the same JAX source as for the multi-controller, and Chained uses the PATHWAYS program tracer to form a multi-node program where each node contains a simple computation. TF is similar to PATHWAYS, where we construct the same TPU computations and ex-

[1]These VMs have 1×V100 GPU and 8×CPU cores.

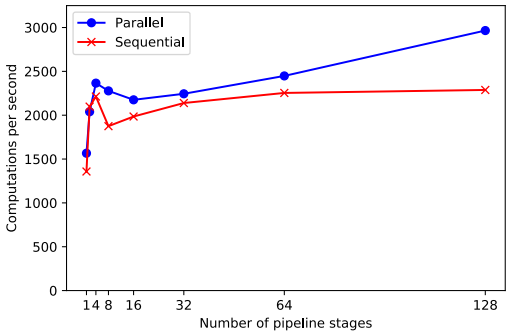


Figure 7. Parallel vs. Sequential Async Dispatch in PATHWAYS. Each pipeline stage runs on a different set of 4 TPU cores (different host) transferring data to next stage via ICI. With parallel async dispatch, PATHWAYS amortizes the fixed client overhead and the scheduling overhead for large number of pipeline stages.

ecute them using TF graphs instead of PATHWAYS. For Ray, OpByOp means executing a separate actor method for each computation which executes a PyTorch AllReduce. Chained means chaining a sequence of actor methods (by passing Ray futures), each of which executes a single PyTorch AllReduce. Fused means executing a single actor method which runs a chain of PyTorch AllReduce commands in a loop.

Figure shows the result. Note that OpByOp is a worst-case experiment that is not idiomatic for any of the frameworks, and it is present merely to stress the underlying systems. As expected, for OpByOp the JAX multi-controller throughput is much better than the single-controller systems, particularly as the number of accelerators increases. Most of PATHWAYS's overhead comes from the fact that the client waits until the coordinator has enqueued one computation and returned its output handles before enqueueing the next. We could eliminate most of this overhead by allowing user code to proceed in parallel with the enqueue RPC, and opportunistically batching multiple small computations into a single PATHWAYS program. We have not focused on optimizing overheads of very small computations since, on real models with computations involving more than scalars, PATHWAYS already matches the performance of multi-controller JAX (see §5.3). Once enough work is Fused into a single node PATHWAYS matches JAX's performance up to 1000 TPU cores, and PATHWAYS Chained outperforms JAX OpByOp up to 256 cores, because PATHWAYS can execute back-to-back accelerator computations directly from C++ while JAX OpByOp transitions to Python for every computation.

TensorFlow and Ray suffer from their lack of a device object store: Ray must transfer the result of a computation from GPU to DRAM before returning the object handle to the client, while TensorFlow transfers the *data* back to the client. This overhead hurts their OpByOp performance but is largely amortized for Chained and Fused. The performance of Ray and PATHWAYS are not directly comparable since they use different hardware, but we interpret the results to suggest that, if the full PATHWAYS design were implemented substituting Ray for PLAQUE, it should be possible to achieve comparable performance. Out of the box, Ray shows about an order of magnitude worse performance per computation than PATHWAYS, but that is unsurprising since Ray can execute general-purpose Python actors and PATHWAYS is specialized to TPU computations launched from C++. With careful attention to engineering, it might be possible to add fast paths to Ray, such as an on-GPU object store and primitives to transfer objects efficiently over the GPU interconnect, that eliminate most of its additional overheads. TensorFlow is slow when running over many cores because it uses a centralized barrier, implemented with control edges, to serialize the gang-scheduled computations.

Figure 6 varies the amount of time spent in each computation to find the smallest computation for which PATHWAYS matches JAX's throughput. For 16 hosts with 128 TPUs on configuration (B), parity is reached with only 2.3 ms, and even for 512 hosts with 2048 TPUs on configuration (A), a computation of at least 35 ms masks all of PATHWAYS's single-controller overhead.

Our next micro-benchmark, also on configuration (B), evaluates the benefit of the parallel asynchronous dispatch mechanism described in §4.5. We construct a more realistic pipeline benchmark in which the simple computations from the earlier benchmark are again chained together, but now each computation runs on a different set of 4 TPU cores, each on a different host, and data output from one computation must be sent via ICI before the next computation can execute. Figure 7 shows three "phases": at first the fixed client overhead is amortized as the number of hosts increases; then the increasing transfer costs of adding more stages begin to dominate; finally the system starts to amortize the fixed scheduling overhead. Eventually we expect that transfer overheads would dominate again. For comparison, we also show the performance when we force the PATHWAYS dataflow execution to use sequential asynchronous dispatch, and wait for one computation to be enqueued before enqueueing the next, to measure the benefit we get from parallel asynchronous dispatch.

5.2 Multi-tenancy

We validate in Figure 8 (performed on configuration (B)) that PATHWAYS is able to time-multiplex accelerators between concurrent programs. PATHWAYS can achieve at least the same aggregated throughput as JAX when multiple clients concurrently submit *different* PATHWAYS programs, i.e., there is no overhead to context switch between programs from different clients, at least when their resources concurrently fit in HBM (traces in Appendix D). As already shown

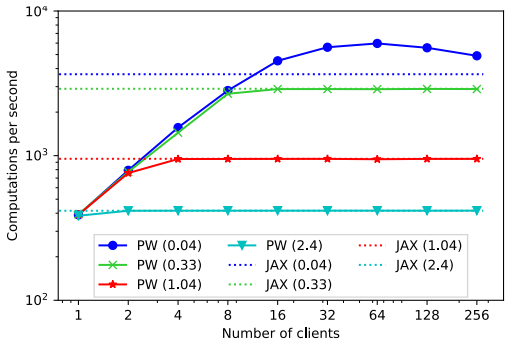


Figure 8. Aggregate throughput of concurrent programs (compute times in ms). PATHWAYS time-multiplexes accelerators between programs efficiently incurring no overhead to context switch.

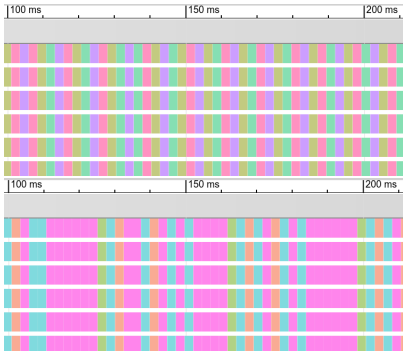


Figure 9. Traces of a sample of cores on PATHWAYS showing interleaving of gang-scheduled concurrent programs with proportional-share ratios of 1:1:1:1 (Upper) and 1:2:4:8 (Lower) between 4 clients.

in Figure 6, the degree of concurrency required to match the throughput is lower for larger computation sizes since the TPU cores reach full utilization sooner. It is noteworthy that the maximum throughput of PATHWAYS exceeds that that of JAX for very small computations, achieving higher TPU utilization. This is because a PATHWAYS worker can accept more computations from remote clients than JAX can dispatch using Python locally.

Figure 9 shows traces of a sample of 128 cores on PATHWAYS for the above workload. This experiment highlights that PATHWAYS performs gang-scheduling of programs submitted by 4 independent clients while controlling allocation of accelerator time for fairness; for example, the scheduler can enforce proportional share in this multi-tenancy setting.

5.3 Large scale model performance

Finally, we show the performance of PATHWAYS in training real machine learning models that can be expressed as SPMD programs. We compared JAX and TF models running on their native systems to the same models running on PATHWAYS, and verified that at numerical results are identical, so we focus only on performance.

We first compare to JAX multi-controller running a Transformer model with an Encoder-Decoder architecture that is used for several text-to-text natural language processing tasks. We use model configurations from (Raffel et al., 2019) and run the experiments on TPUv3s with 16GB memory per accelerator. Table 1 shows the training throughput (tokens/second) for Text-to-text Transformer model with various model sizes (up to 11 billion parameters), training on different number of accelerators. As expected, since the model code is the same, the models trained on JAX and PATHWAYS achieve the same perplexity in the same number of steps. Over all tested model sizes, the two systems show identical performance since realistic computations are large enough to mask single-controller overheads. While we do not report detailed results, we have substantial experience

Table 1. Training throughput (tokens/s) of Text-to-text Transformer model configurations from (Raffel et al., 2019) on JAX multi-controller and PATHWAYS.

Model	Params	TPU cores	JAX	PATHWAYS
T5-Base	270M	32	618k	618k
T5-Large	770M	32	90.4k	90.4k
T5-3B	3B	512	282.8k	282.8k
T5-11B	11B	512	84.8k	84.8k

of running JAX models on PATHWAYS, which corroborates the finding that the performance of the two systems is comparable across a broad range of settings.

Next, we compare the performance of PATHWAYS when training a Transformer-based language model with a Decoder-only architecture on configurations (B) and (C). For this experiment, we use a model expressed in Python using TF. The model consists of 62 Transformer layers with a model dimension of 2048 and a hidden dimension of 8192, which results in 3 billion parameters in total. We compare an SPMD configuration to a pipeline using a GPipe-like schedule (Huang et al., 2019). The pipelined model is split into multiple stages with balanced computation in each stage. Since the first stage has an extra embedding lookup layer and the last stage has an extra softmax layer, we took out one Transformer layer from the first and last stage to balance the amount of compute per stage. Each stage is assigned to a different set of accelerators spanning multiple hosts.

Table 2 shows the training throughput for different numbers of stages (S) and micro-batches (M), while keeping the global batch size and training hyperparameters fixed.[2] The number of examples per micro-batch is fixed at 4 for all cases, and, hence, the global batch size per step is 2048 for the 128-core configurations (8192 for the 512-core one).

[2]Unlike Megatron (Shoeybi et al., 2019), the SPMD-sharded model evaluated here is similar to GShard (Lepikhin et al., 2020) and does *not* have communication proportional to batch size, so it is fair to evaluate pipelined and SPMD with the same batch size.

Table 2. Training throughput (tokens/s) of 3B Transformer language model, using SPMD or multiple pipeline stages, with C TPU cores in PATHWAYS. For pipeline-parallel models, there are S stages and each batch is split into M μ-batches.

Model configuration	TPU cores	PATHWAYS
Model-parallel (SPMD)	128	125.7k
Pipelining, S=4, M=16	128	133.7k
Pipelining, S=8, M=32	128	132.7k
Pipelining, S=16, M=64	128	131.4k
Pipelining, S=16, M=64	512	507.8k

PATHWAYS's training throughput increases proportionally with the number of TPU cores per pipeline stage (Table 2), in line with other systems (Rasley et al., 2020; Narayanan et al., 2021). This result is consistent with Figure 5 showing that the throughput of PATHWAYS linearly scales with the number of hosts. Increasing the number of pipeline stages adds minimal overhead, the throughput being reduced from 133.7k tokens/sec to 131.4k tokens/sec when the number of stages increases from 4 to 16. We compare the pipelined models' performance to an equivalent model expressed using SPMD, and observe that at least in this instance, the pipeline has competitive performance to SPMD, since collective communication within the SPMD computation incurs higher overhead than pipeline bubble overhead.

We also demonstrate that PATHWAYS can efficiently train models over islands of TPUs connected via DCN. In the $S = 16,\ M = 64$ configuration with 128 cores, we measure the same throughput (131.4k tokens/sec) using a single island of 128 cores on configuration (B), or 4 islands of 32 cores each on configuration (C). Figure 10 shows a trace of a sample of cores when the stages are partitioned into islands. DCN transfers occur between every group of 8 rows in the trace, and are not visible in the trace because communication time is effectively overlapped with computation.

Finally, we scale up training of large Decoder-only Transformer models to 64B and 136B parameters using two islands of accelerators. When trained using using PATHWAYS over *two* islands of compute connected over DCN, PATHWAYS achieves $\sim 97\%$ of the throughput as compared to a single island with twice as many devices. For the 136B (64B) LM model, we train over two islands of 1024 (512) cores that uses the fast ICI within island reduction followed by DCN transfer across islands (execution trace available in Appendix D) of 1030GB (457GB) for global reduction.

6 DISCUSSION

6.1 PATHWAYS design vs. implementation

PATHWAYS was designed to target large collections of TPU accelerators. The use of TPU instead of GPU affects many of our low-level design decisions. The biggest difference between TPU and GPU is that far longer-running and more

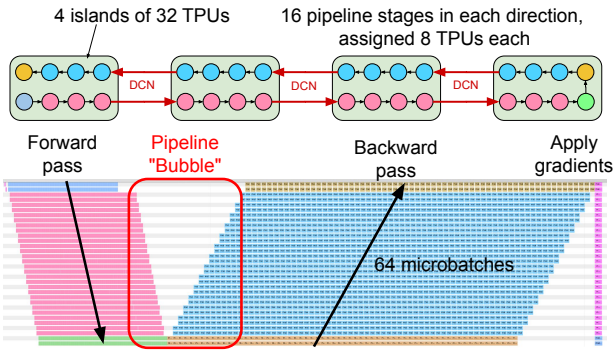


Figure 10. 3B Transformer model pipelined over 128 TPUs: PATHWAYS can efficiently train models over islands of TPUs connected via DCN achieving the same throughput (131.4k tokens/sec) on 4 islands of 32 cores each on configuration (C) as using a single island of 128 cores on configuration (B).

complex computations can be fused into a single TPU kernel, because the TPU supports rich control flow and communication primitives that must instead be executed by driver code on GPU systems. GPUs, by contrast, are more tightly integrated with host memory systems and DCNs (NVIDIA, 2021) (more details in Appendix A.5). TPUs are a good fit for PATHWAYS because XLA can compile high performance functions containing fused collectives, and the large islands of high-performance TPU interconnects allow flexible scheduling of computations of many different sizes. Nevertheless, we believe that most of the high-level architectural choices we made in PATHWAYS and describe in this paper would also be valid for large-scale GPU systems.

6.2 Resource management

PATHWAYS is designed to allow a wide variety of fine-grained dynamic resource-management policies. Our initial research has focused on efficient dynamic time-multiplexing of TPU computations. For more complex future multi-tenancy use cases, PATHWAYS will need to handle more diverse resource types including but not limited to device and host memory, and ICI, DCN, and PCIe bandwidth. PATHWAYS's single-controller model grants the system an extensive ability to track available resources and to allocate resources at large scale. We are planning to explore common multi-tenancy requirements such as priorities, performance isolation, access control, and resource accounting, but at timescales that are significantly smaller than prior work, and for orders-of-magnitude larger pools of resources (e.g., thousands of cores and TBs of accelerator memory).

6.3 Data-dependent vectorized control flow

Almost all ML models currently update *every* model weight based on every training example in every step. We want to enable research that uses fine-grain control flow so that dif-

ferent model weights can be updated per example, or even per sub-example (patch of an image, or word of a sentence). Models like Mixture of Experts (MoE) (Shazeer et al., 2017) and routed capsule networks (Hinton et al., 2018; Barham and Isard, 2019) exploit computational sparsity by "routing" different (sub-)examples to the accelerators hosting different subsets of model weights based on learned functions that are updated as training progresses. This routing requires fine-grain data-dependent data exchanges between nodes. Our ML research colleagues have told us that they would like to use sparsity more effectively when training ever larger models, with ever more tasks, but that current frameworks limit their ability to experiment with novel model architectures. It is the subject of future work to support data-dependent vectorized control flow with both a clean programming model and good performance.

7 Related work

We have examined closely related work in detail in §2. This section expands on related research that addresses ML workloads that need capabilities beyond those offered by SPMD multi-controllers, and validates our PATHWAYS design choices.

Sharing accelerators across multiple tasks is crucial for achieving high resource utilization. Conventional resource sharing is performed in a coarse-grained manner. For example, general-purpose virtualization enables cloud applications to efficiently share multi-tenant resources with performance isolation (Angel et al., 2014; Wentzlaff et al., 2010; Shahrad and Wentzlaff, 2016; Baumann et al., 2009), but cloud providers dedicate accelerators to individual users. Cluster schedulers optimize for heterogeneity of ML workloads (Narayanan et al., 2020) and multi-job, multi-user fairness and performance (Xiao et al., 2018; Ren et al., 2015; Mahajan et al., 2020; Jeon et al., 2018), but resources are still exclusively dedicated to single jobs at long time scales (seconds or more).

Recent work shows that finer-grained sharing can improve resource efficiency further: virtualizing accelerators (Yu et al., 2020; Gupta et al., 2011; Vijaykumar et al., 2016) avoids dedicating a whole accelerator to a single user. Large models (Brown et al., 2020) can be limited by available accelerator memory, requiring GPU memory virtualization (Rhu et al., 2016; Ausavarungnirun et al., 2018) or DRAM offload (Rajbhandari et al., 2021). Concurrent (time-multiplexed or overlapping) ML task execution (Gupta et al., 2018; Xiao et al., 2020; Bai et al., 2020; Yu and Chowdhury, 2020; Wang et al., 2021; Lim et al., 2021) helps harvest idle resources within accelerators. These fine-grained sharing techniques demonstrate opportunities for sharing accelerators that are hard to capitalize on at scale without a single-controller system like PATHWAYS.

Many works have shown that deviating from SPMD computations can improve efficiency on large workloads: pipelining (Huang et al., 2019; Narayanan et al., 2019; Yang et al., 2021) partitions ML models into static heterogeneous computations across accelerators. Graph neural network training (Jia et al., 2020), neural architecture search (Pham et al., 2018), and multi-modal multi-task learning systems (Ma et al., 2018; Lepikhin et al., 2020; Zhao et al., 2019) are examples of inherently heterogeneous and dynamic tasks that do not fit naturally in the SPMD model. We anticipate that upcoming large-scale efficient ML models may form a collection of shared layers and exclusive layers (Bommasani and et. al., 2021), which are natural to express as MPMD.

8 Conclusions

PATHWAYS matches state of the art multi-controller performance on current ML models which are single-tenant SPMD. We have ensured strict compatibility with multi-controller JAX, and as we demonstrate in §5, PATHWAYS matches JAX's performance across very large system scales, for all but the smallest computations.

At the same time, PATHWAYS upends the execution model of JAX programs, pulling user code back into a single-controller model, and interposing a centralized resource management and scheduling framework between client and accelerators. The single-controller programming model allows users simple access to much richer computation patterns. The resource management and scheduling layer permits the reintroduction of cluster management policies including multi-tenant sharing, virtualization and elasticity, all tailored to the requirements of ML workloads and accelerators. Our micro-benchmarks show interleaving of concurrent client workloads, and efficient pipelined execution, convincingly demonstrating that the system mechanisms we have built are fast and flexible, and form a solid basis for research into novel policies to make use of them.

We have shown that careful system design and engineering lets us "get the best of both worlds", matching performance on today's ML models while delivering the features needed to write the models of tomorrow.

Acknowledgements

We gratefully acknowledge contributions to the design and implementation of the PATHWAYS system from many colleagues at Google, and from members of the wider machine learning community. We also thank Martín Abadi, James Laudon, Martin Maas, and the anonymous MLSys reviewers for their helpful suggestions on the presentation of the work.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016. USENIX Association.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. *arXiv preprint arXiv:1903.01855*, 2019.

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2021.

Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. Mask: Redesigning the GPU memory hierarchy to support multi-application concurrency. *ACM SIGPLAN Notices*, 53(2):503–518, 2018.

Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, November 2020.

Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, New York, NY, USA, 2019. Association for Computing Machinery.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

Rishi Bommasani and Drew A. Hudson et. al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2020.

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

Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems*, 1994.

Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

Jeff Dean. Introducing Pathways: A next-generation AI architecture. https://blog.google/technology/ai/introducing-pathways-next-ge

neration-ai-architecture/, 2021. [Online; accessed October-2021].

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

Denis Foley and John Danskin. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro*, 37(2):7–17, March 2017.

Google. Cloud TPU. https://cloud.google.com/tpu, 2021. [Online; accessed March-2021].

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*. PMLR, 10–15 Jul 2018.

Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *2011 USENIX Annual Technical Conference (USENIX ATC)*, volume 31, 2011.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX. http://github.com/google/flax, 2020.

Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *International conference on learning representations*, 2018.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2019.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, 2019.

Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.

Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, Renton, WA, July 2019. USENIX Association.

Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with Roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David A. Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7): 67–78, 2020.

David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, New York, NY, USA, 2007. Association for Computing Machinery.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Advances in Neural Information Processing Systems*, 2020.

Jack Lanchantin, Arshdeep Sekhon, and Yanjun Qi. Neural message passing for multi-label classification. In Ulf Brefeld, Elisa Fromont, Andreas Hotho, Arno Knobbe, Marloes Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases*, Cham, 2020. Springer International Publishing.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021.

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. *arXiv preprint arXiv:2006.16668*, 2020.

Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, July 2021.

Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2018.

Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, November 2013.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Granger, Phil Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

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 using Megatron-LM. *arXiv preprint arXiv:2104.04473*, 2021.

Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in Facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.

NVIDIA. NVIDIA GPUDirect technology. `http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf`, 2021. [Online; accessed February-2021].

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. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*. PMLR, 2018.

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. *arXiv preprint arXiv:1910.10683*, 2019.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, New York, NY, USA, 2020. Association for Computing Machinery.

Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

Mohammad Shahrad and David Wentzlaff. Availability knob: Flexible user-defined availability in the cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016.

Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR (Poster)*, 2017.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-TensorFlow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, 2018.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

TensorFlow. XLA: Optimizing compiler for TensorFlow. https://www.tensorflow.org/xla, 2019. [Online; accessed September-2019].

TensorFlow. TensorFlow Datasets: A collection of ready-to-use datasets. https://www.tensorflow.org/datasets, 2021. [Online; accessed May-2021].

Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Zorua: A holistic approach to resource virtualization in GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient DNN training with Tick-Tock scheduling. In *Proceedings of Machine Learning and Systems*, 2021.

Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, April 2022. USENIX Association.

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, November 2020.

Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel DNN training. In *Proceedings of Machine Learning and Systems*, 2021.

Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, August 2017.

Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. AvA: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2:98–111, 2020.

Yuan Yu, Martin Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of EuroSys 2018*, 2018.

Biao Zhang, Ankur Bapna, Rico Sennrich, and Orhan Firat. Share or not? learning to schedule language-specific capacity for multilingual translation. In *International Conference on Learning Representations*, 2021.

Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training. *IEEE Transactions on Parallel and Distributed Systems*, 33(3), 2022.

Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. Recommending what video to watch next: A multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*, 2019.

A ACCELERATOR DESIGN CONSIDERATIONS

Hardware acceleration is critical to modern deep learning; unfortunately, achieving high performance with accelerators is a non-trivial systems exercise. The following subsections list established techniques commonly employed in deep learning systems to achieve good performance.

A.1 Batching

Given the end of Dennard-scaling, accelerators implement hardware parallelism, often using SIMT (Kirk, 2007) or systolic array (Jouppi et al., 2020) designs. While these hardware architectures remove the arithmetic bottleneck, memory bandwidth quickly becomes the critical resource, necessitating high-bandwidth memory (HBM), an expensive and limited-capacity memory technology. Training schemes for modern neural networks leverage batching to unlock parallelism (good for feeding parallel ALUs) and enable memory re-use (a `float` is read from memory once and used for multiple computations, substantially reducing a computation's memory bandwidth needs). Nevertheless, batching is not a panacea: it puts pressure on the limited HBM memory capacity, and very large batch sizes can slow model convergence rates (Shallue et al., 2018; You et al., 2017; Lanchantin et al., 2020; Anil et al., 2021). While modern GPUs support unified memory—a capability to transparently page memory between accelerators, or from HBM to the host's DRAM—if the user is not careful, an HBM-bandwidth bound computation could slow to PCIe bandwidth, dropping accelerator utilization by an order of magnitude (Lim et al., 2021).

A.2 Asynchronous programming

Accelerator abstractions rely on an asynchronous programming model to achieve performance; a synchronous abstraction wastes too many accelerator computation resources between PCIe latency, kernel scheduling overheads, and interrupt delays. Computations are enqueued on *streams* to be executed on the accelerator at some point in the future. This asynchronous abstraction effectively masks dispatch latency for small operations, so long as a sufficiently large pipeline of work is maintained.

A.3 High performance interconnects

Modern deep neural networks are orders of magnitude larger than the capacity of accelerator (HBM) memory (Lepikhin et al., 2020; Huang et al., 2019). The parallelism within these neural networks is amenable to sharding across multiple accelerators simultaneously, however high speed interconnects between accelerators then become critical for performance. GPUs use interconnects such as NVLink for high-speed communication between "islands" of accelerators on a small number of hosts (Naumov et al., 2020), and use RDMA capabilities of ethernet and Infiniband NICs (GPUDirect) to rapidly communicate between the islands. TPUs have a custom mesh network built directly into the chips, and chips can communicate directly without involving the host or the data-center network. Dedicated GPU and TPU interconnects are typically exposed to applications via 30 year old MPI primitives (e.g., AllReduce) that must be gang-scheduled so that every program enters the same primitive at the same time. As larger computations are run (e.g., training a larger neural network, or training a fixed-size neural network over more accelerators through a form of weak scaling called data-parallel scaling), faster collective operations and thus network bandwidth are required to maintain efficient utilization of aggregate cluster resources. This has prompted significant experimentation with alternate chip-network topologies including hypercubes, and 2-D and 3-D mesh tori (Naumov et al., 2020).

A.4 Single-tenancy

Unlike most resources in a computer, accelerators are not often shared by multiple programs simultaneously. Deep learning models can be easily scaled to use more memory by increasing parameter counts or batch sizes, and thus programs in practice consume most available accelerator (HBM) memory. PCIe bandwidth is much smaller than HBM- or accelerator interconnect-bandwidth. This means that fine-grained context-switching (where much of the data in HBM is paged out to host DRAM over PCIe) results in wasting a significant fraction of accelerator cycles. Thus, when a host program is not fully utilizing an accelerator, the computational resources are stranded and cannot be used productively. Further, preemption of accelerator resources is minimized in practice, resulting in sub-optimal resource scheduling in large, shared clusters serving heterogeneous workloads; it is difficult to allocate large quantities of physically proximate devices to take advantage of network locality.

A.5 Contrasting GPUs and TPUs

While there are many similarities between GPUs and TPUs, there are some important differences. GPU systems tend to have small islands of NVLink-connected devices (e.g., 8 GPUs within one host), with larger aggregations connected over infiniband or data-center networking technology. GPUs are typically programmed by dispatching many small pre-compiled "kernels" to the accelerator, and because they are pre-compiled, the kernels must support dynamic shapes. Any communication between GPUs, whether over NVLink or via DCN, is performed via the NCCL library and initiated by the host.

TPU systems have thousands of devices connected all-to-all, with hundreds of hosts per "island" (Figure 3 Middle). TPUs contain a capable "scalar core" that coordinates the TPU's vector computation units, allowing a TPU to execute long-running functions written in XLA (TensorFlow, 2019) without any host interaction, and these functions may include collective communication across the dedicated ICI network. Consequently, on TPU, an ML framework typically constructs a large XLA program, which is just-in-time (JIT) compiled and dispatched to the accelerator. The fact that a single XLA computation may run for orders of magnitude longer than a GPU kernel justifies increased optimization effort by the compiler such as static buffer assignment and automatic rematerialization of intermediate program values (saving memory capacity). As a consequence of this static buffer assignment, TPUs have only limited support for dynamic shapes, making them a good fit to the PATHWAYS concept of regular compiled functions.

TPUs are restricted to run a single program at a time, with no local pre-emption, mostly because their high-performance RDMA communication implementation between devices makes safe pre-emption difficult without distributed coordination. Because computations are not pre-emptible, it is essential to enqueue communicating computations in a consistent order across devices, or the system will deadlock. This requirement translates to the necessity for PATHWAYS to perform centralized gang-scheduling. As noted in the main text of the paper, however, gang-scheduling is also highly advantageous for GPU efficiency. For an cluster prioritizing ML training workloads, where throughput is more important than latency, it is more efficient to dedicate an entire GPU, or a static fraction of a GPU, to a single carefully sized computation at a time, than to allow the GPU driver and hardware runtime to dynamically multiplex its computational resources across competing concurrent computations. Therefore, even though GPUs can execute concurrent programs without centralized scheduling, there is still a benefit from using a design like PATHWAYS to make more efficient use of resources.

B STRUCTURE OF A TYPICAL ML PROGRAM

This subsection describes a typical contemporary ML computation in terms of the high level structure that maps subcomputations to accelerators, and the lowering of a subcomputation to accelerator kernels.

The computations that are executed by an accelerator running an ML workload are dominated by what we call "compiled functions". These are sub-computations with the following characteristics:

- Input and output types, and the shapes of any input/output tensors, are known before the input data have been computed.
- Bounds of any loops are either known when the node computation is scheduled, or specified as a maximum trip count with potential early termination.
- Conditionals are "functional" where both branches have the same output type, and resources are allocated in advance sufficient for either branch.

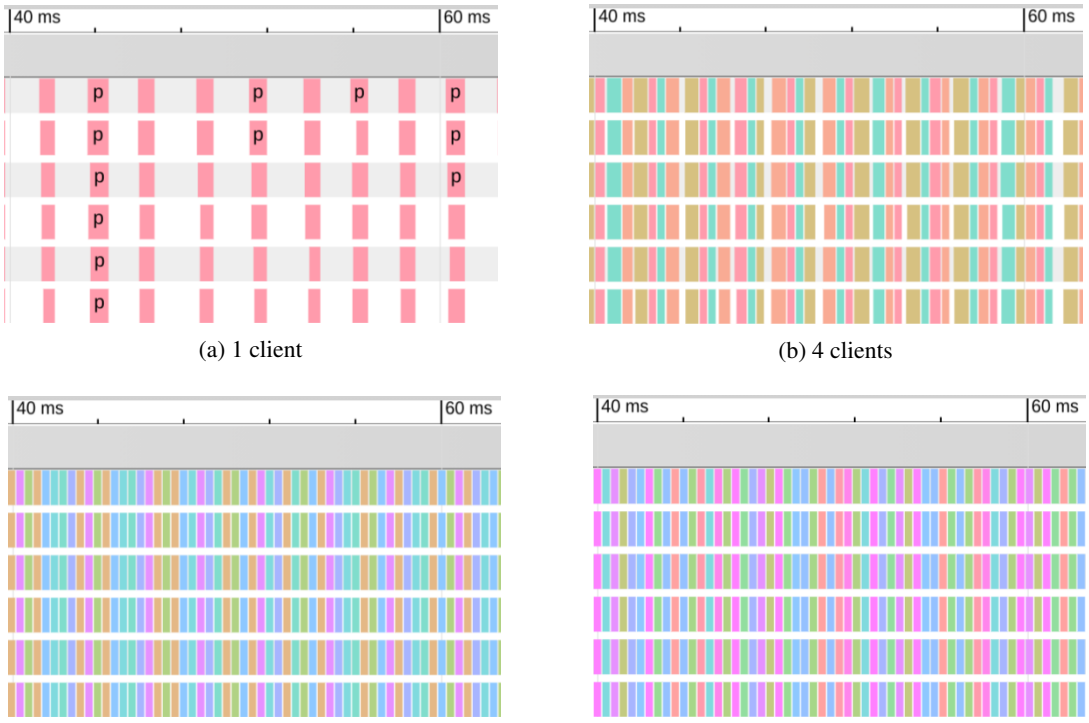
The constraints on compiled functions are mostly due to the co-evolution of ML models with hardware, discussed in detail in §A. Here we discuss some of the implications of the fact that the resource requirements of compiled functions are known in advance.

Almost all of today's high performance ML computations are expressed as long stretches of compiled functions and only occasionally (if ever) branch based on data that is computed by a compiled function. Since the system can perform resource allocation for compiled functions in advance, contemporary ML frameworks exploit this property by enqueueing compiled functions asynchronously before their predecessors have run, allowing host-side work to be done in parallel with accelerator computations (Bradbury et al., 2018; Paszke et al., 2019). Wherever possible the frameworks submit graphs of compiled functions to a "just in time" (JIT) compiler (Chen et al., 2018; TensorFlow, 2019) that is able to exploit optimizations like layout assignment and fusion that can substantially improve the efficiency of the resulting accelerator code.

The need to optimize graphs of compiled functions to achieve peak accelerator performance means that frameworks typically trace the execution of fragments of high level (Python) code that can be lowered to compiled functions. Thus, even though client code may be written in a high level language with complex state bound to the running context at a host, performance-sensitive node computations are often lowered to an internal representation (IR) that is serializable and relatively easy to send to a remote host for execution.

C INPUT DATA PROCESSING

JAX has deliberately avoided re-implementing data loading pipelines, and `tensorflow/datasets` (TensorFlow, 2021) are commonly used for JAX input processing, so it is not difficult for JAX programs to be adapted to offload input processing to the CPU-based TensorFlow executors run on PATHWAYS workers. PATHWAYS instantiates a CPU-based TensorFlow executor on each host, so that user programs can serialize input processing into a TensorFlow graph and distribute it across the workers. We plan to support streaming data protocols so that CPU-based computation can be



(a) 1 client (b) 4 clients

(c) 8 clients (d) 16 clients

Figure 11. Traces of a sample of TPU cores for Figure 8. PATHWAYS showing interleaving of gang-scheduled concurrent programs.

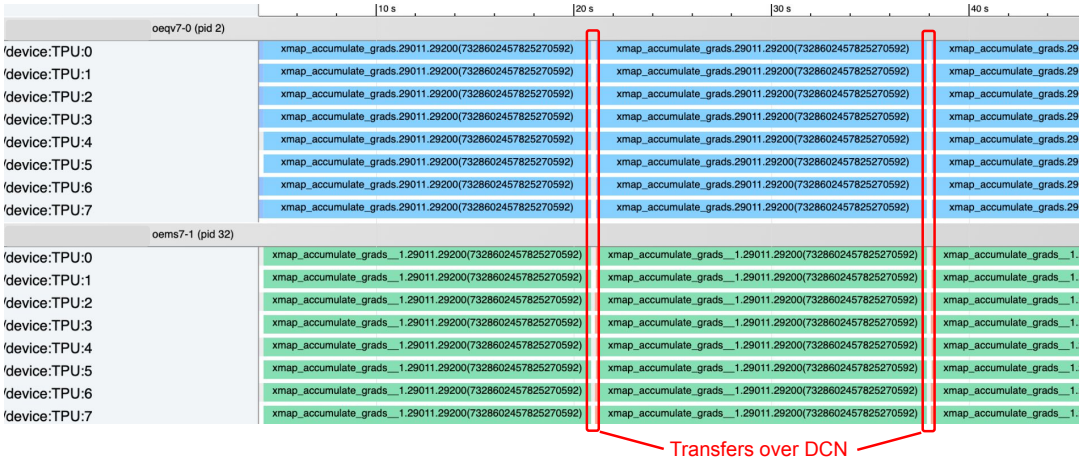


Figure 12. 64B Transformer model training data parallel over two islands of 512 TPUs each. The trace highlights the relatively small overhead of cross-island transfer using DCN.

performed on an independently managed set of servers, thus decoupling the expensive TPU-connected hosts from the CPU resources available for input processing.

D EVALUATION WORKLOAD TRACES

Figure 11 presents the traces for the workload of Figure 8 with a varied number of clients submitting programs concurrently (§5.2). A single client uses a very small per-program compute time of 0.33 ms that is insufficient to saturate accelerators. With PATHWAYS's multi-tenency support, using multiple clients increases the device utilization to $\sim 100\%$. All client programs are gang-scheduled across all cores, and interleaved at a millisecond scale or less, showing little context-switch overhead.

Figure 12 shows a trace profile for multiple training steps when the 64B Decoder only Transformer model is trained data parallel over two islands of accelerators with 512 chips

each (§5.3). The first eight rows (blue) correspond to TPU computations on a host in the first island and the next eight rows (green) correspond to TPU computations on a host in the second island. In this case, each island computes gradients and then enqueues the transfers of these gradients to the other island. When the transfer of gradients is over DCN completes, each island applies the received gradients and starts the next training step. DCN transfers incur minimal overhead even at the scale of pairs of 128 hosts resulting in 97.2% training throughput compared to an SPMD configuration that uses ICI communication over total equivalent number of chips.