The Microarchitecture of DOJO, Tesla's Exa-Scale Computer

Emil Talpes, Debjit Das Sarma, Doug Williams, Sahil Arora, Thomas Kunjan, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Chandrasekhar Poorna, Vaidehi Samant, John Sicilia, Anantha Kumar Nivarti, Raghuvir Ramachandran, Tim Fischer, Ben Herzberg, Bill McGee, Ganesh Venkataramanan, and Pete Banon,
Tesla Motors Inc., Palo Alto, CA, 84304, USA

The Tesla-built DOJO system is a scalable solution targeted towards machine learning training applications. It is based on the D1 custom compute chip which packs together 354 independent processors, resulting in 362 TFLOPS of compute and 440 MB of internal static random-access memory storage. While maintaining full programmability, DOJO emphasizes distribution of resources and an extremely high bandwidth interconnect, allowing it to scale from small systems all the way to exaFLOP supercomputers.

The size of machine learning (ML) models has exploded in the recent years, growing from a few million parameters less than 10 years ago[1,2] to hundreds of millions[3] and even billions of parameters for the most recent models.[4] Training models of this size requires huge datasets, which in turn require massive computational power. Real systems which can train models of this magnitude in a reasonable time already require thousands of compute dies and terabytes of storage.

While DOJO is primarily intended for internal Tesla workloads, it is still subjected to the same general scaling trends. It is inherently a distributed computer like all other large-scale ML training systems but, unlike most other such systems, DOJO relies on components designed specifically for extreme scalability. Typical DOJO builds are expected to target exa-scale systems.

The second goal of DOJO is generality. While we could analyze Tesla's current ML models and design a more efficient system to target these exact requirements, we cannot foresee how the field will evolve in the future. Thus, the DOJO compute model must be general enough to accommodate the evolution of a changing field.

This article describes the microarchitecture of a DOJO system. As for any distributed system, the microarchitecture defines the compute element, how these elements communicate with each other and with the outside world, and the synchronization primitives which make sure all these compute elements work together on a single task.

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3258906
Date of publication 21 March 2023; date of current version 15 May 2023.

MICROARCHITECTURE OF THE COMPUTE NODE

The compute node is the building block sitting at the base of the DOJO system. It is a full-fledged computer with a dedicated processing pipeline, local memory and network interface. To help system scalability and programmability, the node is a relatively powerful, general-purpose CPU. The DOJO node implements a multithreaded, "mostly" in-order pipeline optimized primarily for heavy compute applications rather than control speculation (Figure 1).

Processing Pipeline

The front-end of the pipeline starts with a simple branch predictor capable of producing one prediction every cycle. Each fetch access fills a 32-B fetch window holding up to eight instructions. These instructions are fed to an eight-wide decode stage capable of handling two threads per cycle, followed by a four-wide scalar scheduler. The scalar scheduler can handle four threads per cycle, and it executes all scalar instructions, branches and address generation operations.

Vector instructions proceed further down the pipeline to a dedicated scheduler. They tend to have much longer latency than scalar instructions, so instructions sitting in the vector scheduler typically trail the front-end significantly. This organization allows the front-end of the machine to run ahead and resolve control flow instructions and memory references well ahead of the trailing vector computations.

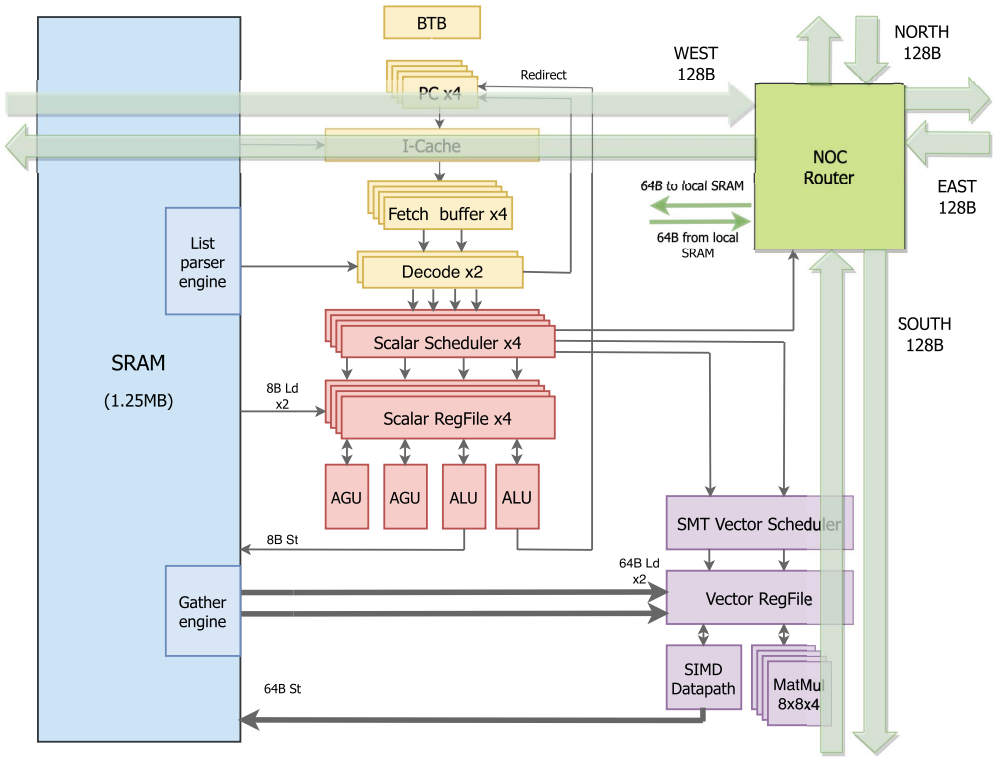

FIGURE 1. Block diagram of a DOJO node.

To some extent, the decoupled microarchitecture of the DOJO node resembles the one proposed by Smith.[7] In DOJO the integration between the scalar and vector pipelines is tighter, with a shared front-end and memory interface servicing both sides of the machine similar in that regard to the coprocessor-based organization employed by many AMD CPUs.[8]

The DOJO node can process four threads in parallel. However, these threads are mostly utilized for conveying the application's parallelism to hardware rather than allowing multiple applications to run independently. DOJO does not implement all the typical thread protection mechanisms and it forces threads running in parallel to collaborate in sharing execution resources. In typical use, a node is expected to run one or two compute threads (e.g., a dot product and a trailing post processing sequence), plus one or two communication streams.

Node Memory

In a scalable distributed system, local memory has two fundamental roles. First, it must provide enough bandwidth to feed the local execution units since the shared memory and interconnect cannot keep the compute fully utilized. Second, it must have enough capacity to cover a significant portion of the working set, reducing the need to reach out to remote resources in the system.

All DOJO nodes have their own 1.25-MB local static random-access memory (SRAM) with 384-GB/s load and 256-GB/s store bandwidth. The memory can service two 64-B loads and one 64-B store per cycle to the local datapath, as well as one load and one store per cycle to the network. Regular load or store instructions are only allowed to access the local memory space. The interconnect can directly access the local memory on behalf of a remote node, without interrupting the instruction flow on the associated CPU. Internally, the node supports both contiguous and gather loads with 8- and 16-B granularities, while remote accesses only support contiguous 64-B accesses.

The memory also provides a side port for a list parsing engine. During instruction decode, the list parser can provide instruction immediates from a precompiled database, essentially allowing the programmer a free level of indirection. The databases are typically compiled together with the application code, but they can move with the data objects during different phases of execution.

DOJO does not implement a virtual memory mechanism to back up the local SRAM with remote storage

available across the network. Such a mechanism would scale poorly at the system sizes we target in our applications. Instead, we rely on software to move data explicitly in and out of the local memory.

Network Interface

All DOJO nodes are connected through a 2-D mesh network which can move two data packets and two request packets per cycle in every direction. Each node has a bidirectional, dedicated interface to the mesh which can move one 64-B data packet per cycle in each direction.

The network supports pull and push operations, and it takes a single cycle to move data between two neighboring nodes. It can execute a single operation per request (either push or pull for a single data packet) or block operations (typically single pull followed by multiple push packets).

While the 2-D mesh has some connectivity downsides compared to other network topologies (e.g., 2-D or 3-D torus), it is a much better match for the integration technologies currently available. DOJO scales the node-to-node bandwidth to take advantage of all available on-die and on-panel communication tracks, with not enough resources available to implement longer reach channels.

Datapath

In a DOJO node, instructions can execute in three separate pipeline stages. Simple primitives like looping, counting, or list-based predication can execute in the eight-way decode. Scalar arithmetic instructions, branches, address generation, and synchronization primitives execute in the four-way scalar engine. Wide instructions like matrix multiplications or 64-B single instruction, multiple data (SIMD) operations execute in the two-way vector engine. This creates a funnel-like pipeline organization with instructions dropping out of the machine as soon as they can be serviced (Table 1).

There are relatively few operations which can execute in the wide decode stage, but they tend to be very common in typical DOJO code. At the other end of the pipeline, the vector unit can only process two instructions per cycle, but it supports a lot more unique opcodes and variants.

TABLE 1. Types of instructions executed in a DOJO node.

Instruction type	Unique opcodes	Variants
Front-end	12	21
Scalar	74	143
Vector	142	1095

AT THE OTHER END OF THE PIPELINE, THE VECTOR UNIT CAN ONLY PROCESS TWO INSTRUCTIONS PER CYCLE, BUT IT SUPPORTS A LOT MORE UNIQUE OPCODES AND VARIANTS.

Within each execution cluster, all instructions are processed in order with regards to other instructions from the same thread. However, across the entire program stream, many instructions can execute out of order. Simple control flow primitives execute during the decode stage and ahead of many older instructions. Similarly, memory references and simple scalar operations execute on the scalar pipeline, ahead of long latency, trailing vector computations.

DOJO nodes implement a full complement of general-purpose scalar and vector instructions. To ensure complete generality, the scalar side of the CPU provides all the capabilities of a modern reduced-instruction-set computer (RISC) processor. Many of the instructions supported there are compatible with the RISC-V ISA,[5] augmented with custom primitives for flow control, network access, and synchronization. On top of that, the vector engine supports specific primitives for ML acceleration, remote communication, and synchronization. The matmul engine consists of four units capable of 8×8 matrix multiplications, with inline support for data loading and gathering, as well as padding, transposing, and expanding compressed operands. Besides the cursory packed integer and floating-point arithmetic instructions, the SIMD datapath also implements specific shuffle, transpose and convert instructions with or without stochastic rounding for data down-sampling. Synchronization primitives include counting semaphores and barriers, with support directly built into the node and network pipelines.

DOJO Arithmetic Formats

While the scalar engine of the DOJO node supports only 64-bit integer data, the vector engine supports all typical integer formats (signed and unsigned byte, word, double word) as well as an assortment of FP32, FP16, and FP8 formats:

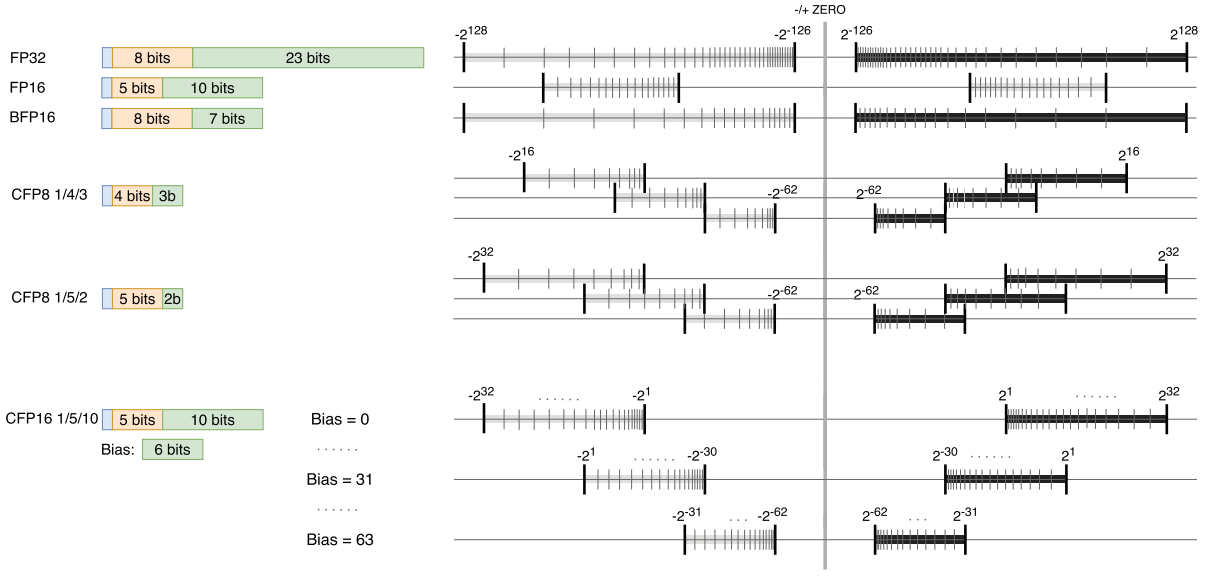

FIGURE 2. Floating point format ranges supported in the DOJO node.

Like all ML specific systems, DOJO provides full support for IEEE FP32 arithmetic. While this data format has sufficient range and precision for all uses, it tends to have more precision than required for many applications. For cases where the application requires similar kind of numerical range but less precision, DOJO offers BFP16 with only seven mantissa bits. Going from FP32 to BFP16 reduces both the memory footprint of the algorithm and the bandwidth required for moving data across the network. In many cases, algorithms can save even more by switching to one of the provided configurable floating point 8-bit data types (CFP8). CFP8 supports either a 4- or 5-bit exponent with a configurable bias, which effectively shifts the representation range of the datatype according to the local application needs. Between all possible bias values, CFP8 has a larger representation range than regular IEEE FP16. The overlapping representation ranges of these data types are shown in Figure 2.

In practice, CFP8 can be used for parameters and activations in most data layers. However, gradients require higher precision, and, in such cases, the application can go to either FP32 or to the intermediate step of CFP16. Using a similar variable bias mechanism, CFP16 has a similar representation range as CFP8 but at much higher precision. When used together with stochastic rounding, CFP16 can effectively replace FP32 as a gradient storage format in many instances.

The DOJO node supports both CFP8 and BFP16 as source operands in the matrix multiplication units, with accumulation in the wider FP32 format. The SIMD unit supports operands in CFP8, BFP16, and FP32 formats, but supports accumulation only in FP32 format. Simple arithmetic instructions such as max, min, compare, and so on, are supported in all the above formats.

DOJO INTEGRATION

Physically, the logic within the DOJO nodes is covered by network wires which can stitch together into a global network. This allows multiple nodes to connect seamlessly, forming a compute plane which can expand in any direction.

The first level of integration for a DOJO machine is the D1 die. It packs together 354 DOJO nodes, surrounded by 144 serial links on each side which allow it to connect directly to other neighboring D1 dies. A D1 die offers 360 TFLOPS of BFP16 or CFP8 compute, as well as 22 TFLOPS of FP32 compute at 2 GHz. It also offers a total of 440 MB of SRAM distributed across all nodes and 4 TB of bandwidth to its neighbors on each edge.

The second integration level is the DOJO training tile. The tile packs together 25 D1 dies within a single mechanical package, with integrated electrical and thermal solutions. Within the package, all die to die interconnects are scaled up to the limit of physics and they occupy all tracks in the X and Y dimensions. To free up space for these horizontal communication links, power delivery and liquid cooling are provided vertically in the Z dimension. Each training tile provides 4.5 TB of bandwidth per edge, with seamless connection capability to other training tiles.

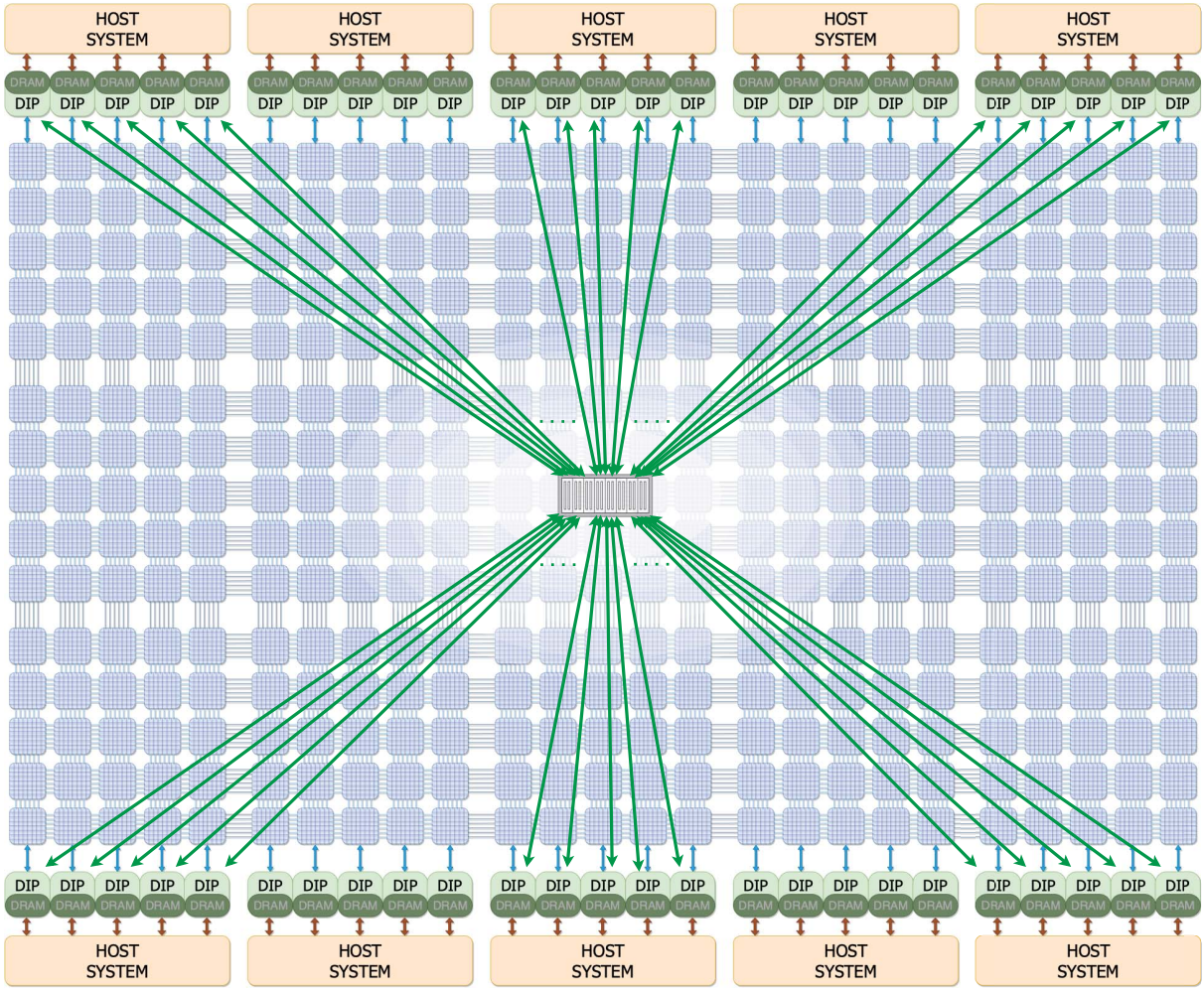

FIGURE 3. Typical organization of a DOJO system.

DOJO SYSTEM TOPOLOGY

A full DOJO system is composed of multiple 5×5 DOJO training tiles, combined to create a bi-dimensional compute plane. Communication within the compute plane uses a Tesla proprietary protocol which connects all components into a flat address space but cannot talk natively with off-the-shelf ×86 hosts or storage servers. To provide connectivity to off-the-shelf components, DOJO interface processors (DIPs) are placed around the edges of the compute mesh as shown in Figure 3.

Besides providing access using off-the-shelf protocol connectivity, the DIPs also provide shared memory support. Each interface processor offers 32 GB of high bandwidth memory, to a total of 160 GB per host. This compares against 11 GB of private SRAM per training tile, distributed across all its 9000 processing nodes.

COMMUNICATION MECHANISMS

Logically, all DOJO components are placed within a single flat address space mapped to the 2-D mesh and communicate with each other using similar primitives. The actual bandwidth, latency, and interface protocols vary depending on where the components are located. Within the D1 die, adjacent nodes are connected using a wide parallel interface with 256-GB/s bidirectional bandwidth and a single cycle latency. Across the entire die, this adds up to a 5-TB/s cross-section bandwidth.

Within a training tile, D1 dies connect together using SerDes links with a combined die-to-die bandwidth of 2 TB/s and an average latency of 100 ns. At the edge of the training tile, D1 dies connect using similar serial links but with a lower density. Across this interface, the die-to-die bandwidth drops to 900 GB/s. At

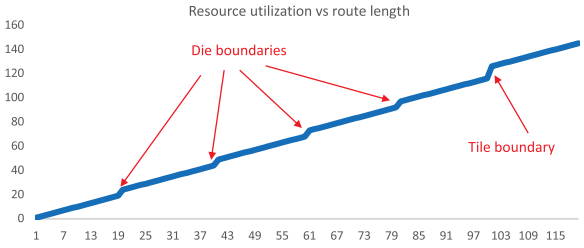

FIGURE 4. Resource occupancy versus route length.

the edge, every DIP connects to an ×86 host system using a PCIe ×16 Gen4 channel. Up to 5 DIPs can be connected to the same host, with a total bandwidth of 160 GB/s per host.

Interface processors also offer ethernet physical links and can communicate with each other using off-the-shelf network switches. DOJO packets sent to far away destinations can be routed to a near-by DIP and from there transferred over an ethernet link to another DIP closer to destination, effectively creating shortcuts through the mesh. Providing shortcuts for long distance packets makes sense because such routes can get very expensive in a 2-D mesh. Figure 4 shows the relative resource occupancy increase with the number of hops traversed by the packet.

While the shortcuts do not always offer lower latency, they offer clear bandwidth benefits by segregating some global routes from the local traffic. In the mesh, a packet which traverses a 10-hops route keeps 10 links occupied for at least a cycle each, making it at least 10 times more expensive than a packet moving between neighboring nodes. This creates strong incentive for software to keep as much as possible of the communication local and to segregate the few global routes onto dedicated networking resources.

To move data from one memory region to another, processing nodes can push data packets directly from their own SRAM to a remote destination or they can issue pull requests to a remote source. For larger transfers, DOJO provides several bulk communication primitives. The interface processors have DMA engines which can sequence transfers between the shared DRAM and various SRAM locations. These DMA engines can service contiguous or strided transfers up to four dimensions.

For SRAM-to-SRAM transfers, the network supports packet broadcasts within the same D1 die, as well as contiguous region transfers. And, for more complex transfer patterns, DOJO nodes offer acceleration primitives using list-based sequences running as asynchronous threads.

SYSTEM NETWORK

The DOJO network topology is geared towards simplicity. It uses a flat addressing space, with no virtualization, and it is completely exposed to software. The compiler needs to know where all data has been placed and must manage the transfers explicitly using the full physical address of the data.

Routing is simple as well. Once a compute die passes the power-up tests all its routers are expected to work, so reaching a destination is simply a matter of following the row and column to the destination node.

For additional flexibility, every D1 die implements a routing table. Once a packet enters the network or reaches a new die on the way to destination, it consults the local routing table to decide what is the best path forward. Based on the information installed in the routing table, the packet can continue along the same row or column, or it can turn to avoid congestion or a malfunctioning network component. The routing tables can also be set up to transfer packets to the nearest DIP in order to utilize the Z-dimension shortcuts (Figure 5).

The DOJO network guarantees packet delivery, but it does not ensure end-to-end packet ordering. This allows software to overwrite the sender's memory buffer as soon as the data has been pushed onto the network. However, the receiving node cannot rely on packets arriving in order to start processing. Packets which belong to the same stream can be routed through different serial links for load balancing and congestion avoidance, or they can be subject to link recovery protocols while in transit. To work around this issue, receiving nodes must count the packets within each block transfer before it can start processing them.

SYSTEM SYNCHRONIZATION

To help synchronize events across multiple CPU nodes, DOJO provides built in support for counting semaphores and barriers. Every node implements 32 semaphore registers, accessible both from within the local node as well as from remote locations through the network. These registers can count events generated by local threads, by remote threads and by data packets being committed to the local SRAM (Figure 6). Threads running anywhere in the DOJO machine can wait on a semaphore, with hardware ensuring atomic updates and starvation avoidance.

Semaphores have several use models in a DOJO system. They are the typical mechanism used for monitoring network transfers, with the consumer thread waiting for the semaphore to reach a value equal to the number of expected packets. They can be used as simple mutexes protecting shared data structures, to

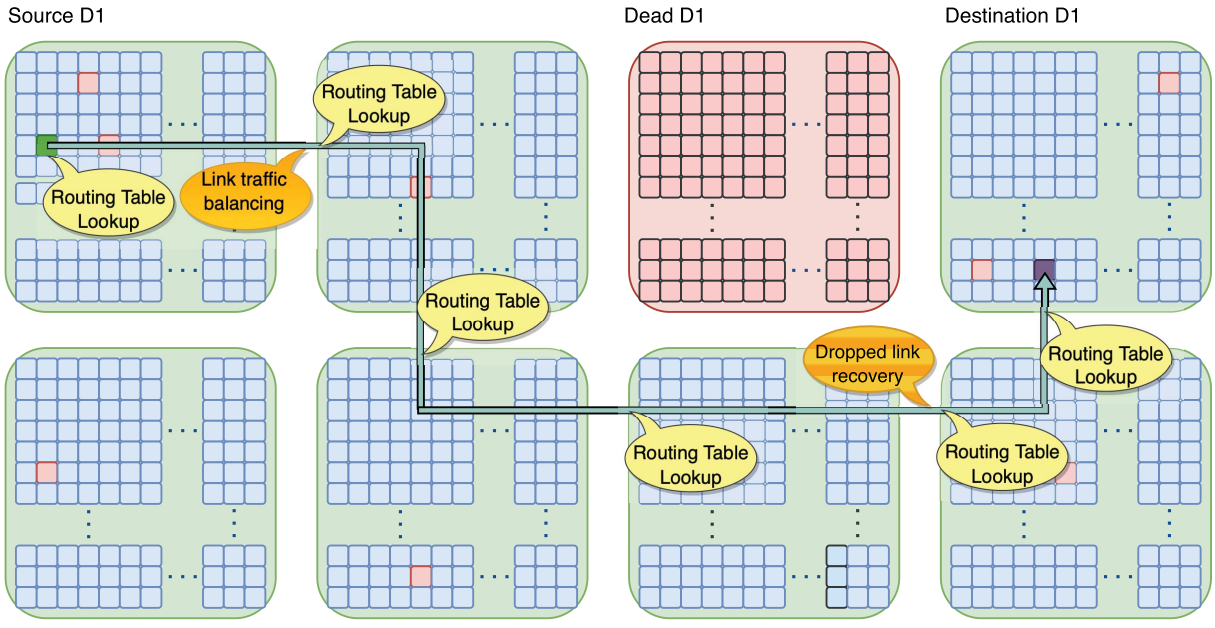

FIGURE 5. Packet routing in the DOJO mesh.

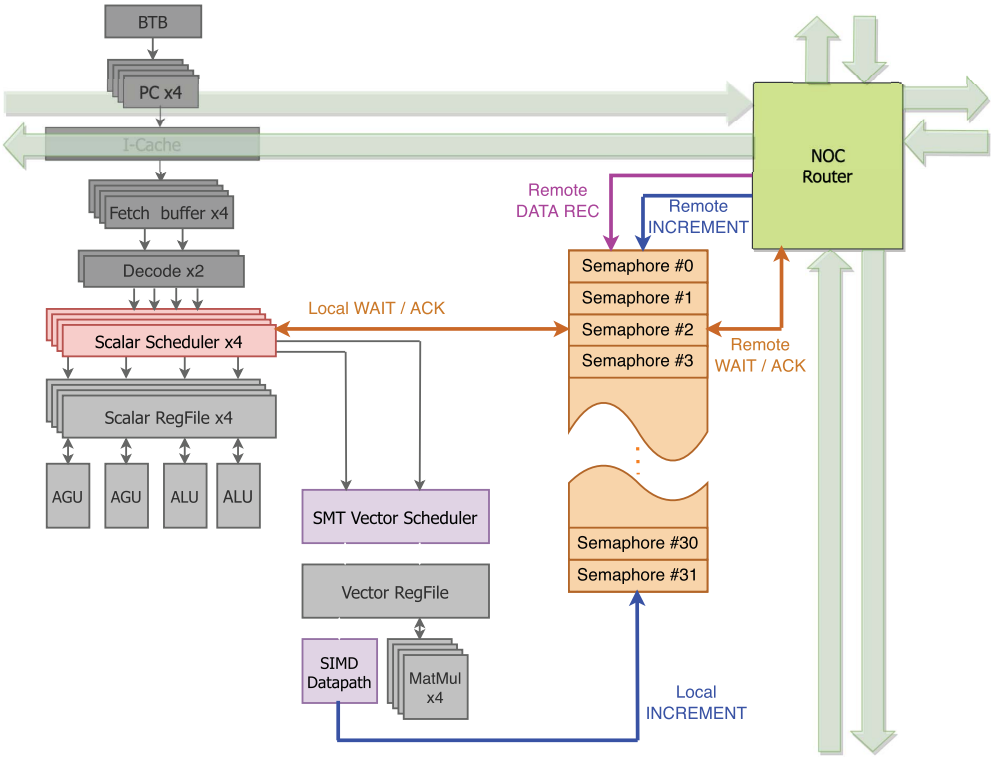

FIGURE 6. Semaphore array connections in the DOJO node.

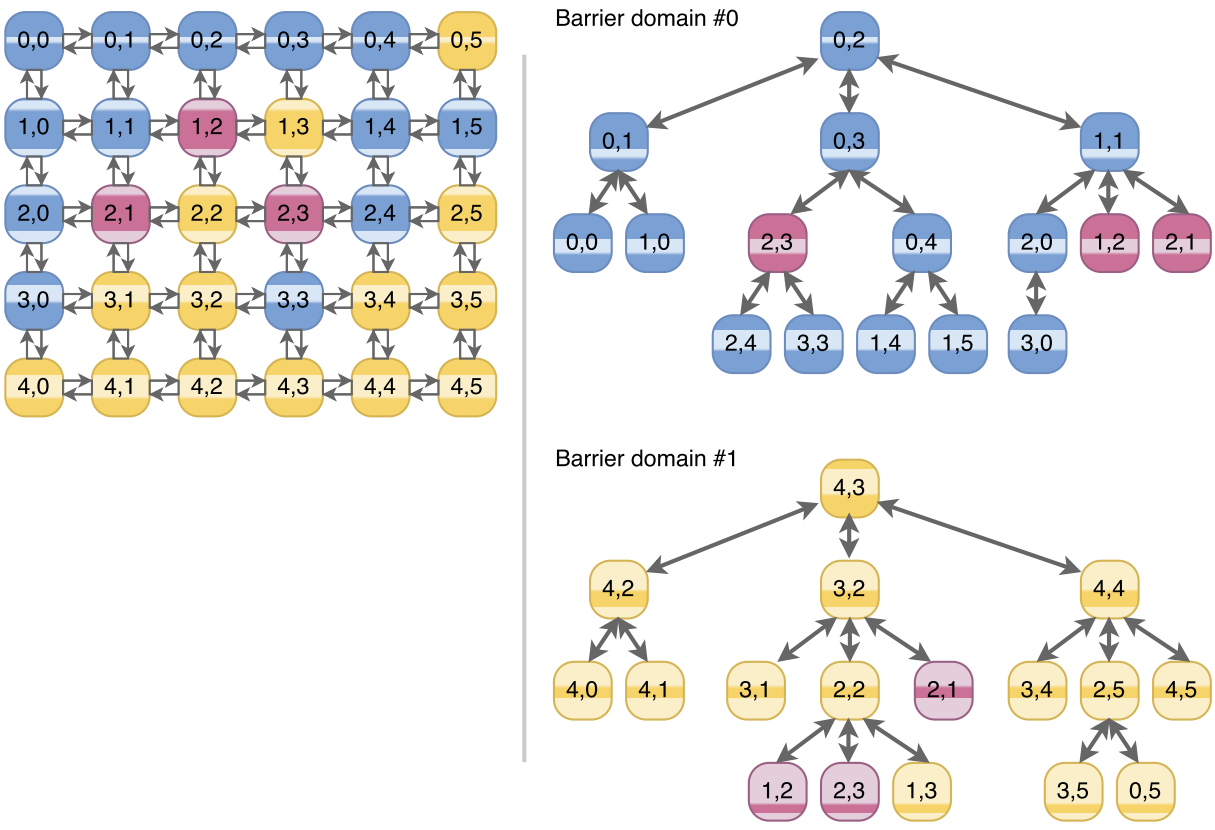

FIGURE 7. Example of barrier trees in DOJO.

synchronize either SMT streams running on the same node or streams running on different nodes. And they can be used as counting semaphores, in producer-consumer type of relations between multiple instruction streams.

The second synchronization mechanism offered in DOJO is the barrier. It implements a version of the tree-based barrier[6] in hardware, with "ARM" messages travelling up the tree towards the root, and "GRANT" messages travelling downstream toward the leaves. In DOJO, a node can be part of up to four barrier trees at the same time, and messages up and down the tree are handled completely in the network routers, without software involvement.

In this example (Figure 7), system nodes are partitioned in three sets. The blue nodes belong to one synchronization domain, the yellow nodes belong to a second domain, and the red nodes belong to both domains. Software creates barrier trees which cover both synchronization domains.

When a node reaches the barrier, it executes a BARM instruction which triggers an "ARM" message upstream. As more nodes in the domain reach the barrier, the "ARM" messages propagate upwards toward the root of the tree. Once the root gets the "ARM" message and it reaches its own BARM instruction, it generates the "GRANT" message towards all downstream nodes. For any member of the synchronization set, the barrier is considered satisfied once the downstream "GRANT" wave reaches that node. The nodes in a barrier tree do not need to be physically adjacent so the application has the flexibility to create and synchronize any node sets.

CONCLUSION

From the very beginning, the fundamental goal of the DOJO project has been to achieve the best wall clock time on real world models, with the ability to accommodate any new models we might need to run in the future. Purposely designed for ML applications, it relies heavily on the inherent parallel nature of these workloads.

The defining characteristic of our system is scalability. It deemphasizes mechanisms like coherency, virtual memory, and global lookup directories which tend to scale poorly as a system grows, relying instead on fast, local storage instead of global memory. This

fast, distributed storage is backed by a high-bandwidth interconnect which is an order of magnitude faster than typical distributed systems.

REFERENCES

1. C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vision Pattern Recognit.*, Jun. 2015, pp. 1–9.
2. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vision Pattern Recognit.*, 2016, pp. 770–778.
3. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Naacl-Hlt*, 2019, pp. 4171–4186.
4. T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 1877–1901.
5. A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V instruction set manual. Volume I: User-level ISA, version 2.0," Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-52, May 2014.
6. J.-S. Yang and C.-T. King, "Designing tree-based barrier synchronization on 2D mesh networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 6, pp. 526–534, Jun. 1998, doi: 10.1109/71.689440.
7. J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, Nov. 1984.
8. S. White, "High-performance power-efficient ×86-64 Server and desktop processors using the core codenamed 'Bulldozer,'" in *Proc. IEEE Hot Chips 23 Symp. (HCS)*, Aug. 2011, pp. 1–32, doi: 10.1109/HOTCHIPS.2011.7477512.

EMIL TALPES is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at etalpes@tesla.com.

DEBJIT DAS SARMA is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at ddassarma@tesla.com.

DOUG WILLIAMS is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at dougwilliams@tesla.com.

SAHIL ARORA is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at saarora@tesla.com.

THOMAS KUNJAN is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at tkunjan@tesla.com.

BENJAMIN FLOERING is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at bfloering@tesla.com.

ANKIT JALOTE is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at ajalote@tesla.com.

CHRISTOPHER HSIONG is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at chsiong@tesla.com.

CHANDRASEKHAR POORNA is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at cpoorna@tesla.com.

VAIDEHI SAMANT is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact her at vsamant@tesla.com.

JOHN SICILIA is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at jsicilia@tesla.com.

ANANTHA KUMAR NIVARTI is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at anivarti@tesla.com.

RAGHUVIR RAMACHANDRAN is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at rramachandran@tesla.com.

TIM FISCHER is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at tifischer@tesla.com.

BEN HERZBERG is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at bhertzberg@tesla.com.

BILL MCGEE is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at bmcgee@tesla.com.

GANESH VENKATARAMANAN is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at gvenkataramanan@tesla.com.

PETE BANON is with Autopilot Hardware, Tesla Motors Inc., Palo Alto, CA, 94304, USA. Contact him at pbannon@tesla.com.