




Compute Substrate for Software 2.0

Jasmina Vasiljevic , Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic , Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, Utku Aydonat , Tony Zhou, Syed Zohaib Gilani, Armond Paiva, Joseph Chu, Djordje Maksimovic, Stephen Alexander Chin, Zahi Moudallal, Akhmed Rakhmati, Sean Nijjar, Almeet Bhullar, Boris Drazic, Charles Lee, James Sun, Kei-Ming Kwong, James Connolly, Miles Dooley, Hassan Farooq, Joy Yu Ting Chen, Matthew Walker, Keivan Dabiri, Kyle Mabee, Rakesh Shaji Lal, Namal Rajatheva, Renjith Retnamma, Shripad Karodi, Daniel Rosen, Emilio Munoz, Andrew Lewycky, Aleksandar Knezevic, Raymond Kim, Allan Rui, Alexander Drouillard, and David Thompson, *Tenstorrent Inc., Toronto, ON, M3C 2G9, Canada*

The rapidly growing compute demands of AI necessitate the creation of new computing architectures and approaches. Tenstorrent designed its architecture (embodied in Grayskull and Wormhole devices) to tackle this challenge via two fundamental and synergistic approaches. The first is via compute-on-packets fabric that is built from ground up for massive scaleout. The second is the ability to execute dynamic computation, built into the compiler, runtime software and hardware architecture. By combining these approaches, TensTorrent will enable continued scaling of AI workloads.

Compute demand of AI is skyrocketing at a rate that far outpaces the compute density improvements that can be gained by Moore's Law alone^{3,4} and approaches based on monolithic shared memory models. We have chosen to attack this challenge via two approaches, dynamic computation and massive scaleout. We design dynamic computation to enable a wide range of techniques that intelligently "forgo unnecessary computation" or "compute only what is relevant to input," akin to what our brains do.

Large clusters are already the norm for training of AI models, while inference for some large models also requires multi-device execution. The shared memory paradigm cannot enable the required scale, which necessitates a paradigm shift to a multicore private-memory model, a foundation in our scaleout architecture. On top of this, we build a push-based data movement in which data transfers are explicitly planned and controlled.

These two approaches can be synergistically combined to take the current steep slope of increasing AI

compute and storage requirements and reduce it down to something much more compatible with Moore's Law.

In the "Hardware" section, we present the details of our hardware, with primary focus on the Grayskull device. The "Software" section presents our software stack. The "Full Stack Performance Optimizations" section deep dives into several full stack performance optimizations enabled by our hardware and software. The "Dynamic Execution" section summarizes various dynamic execution approaches enabled by our architecture. Performance results are presented in the "Results" section. Finally, the "Conclusion" section concludes this article.

HARDWARE

Devices

Over the last four years, Tenstorrent designed three chips, shown in Figure 2 and summarized in Table 1.

Jawbridge is a small 14-nm test-chip, containing six first-generation Tensix cores.

Grayskull, shown in Figure 1, is our first production chip in 12-nm technology, and is currently in evaluation with multiple customers. It is first incarnation of our large cluster-on-a-chip multicore architecture and is composed of 120 compute cores. The physical area of the 10x12 grid of cores is 477 mm². Each core operates independently; it has its own unique instruction queue and progresses at its own pace, in contrast to monolithic chip-scale SIMD, VLIW or single-kernel

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see <https://creativecommons.org/licenses/by/4.0/>
 Digital Object Identifier 10.1109/MM.2021.3061912
 Date of publication 9 March 2021; date of current version 26 March 2021.

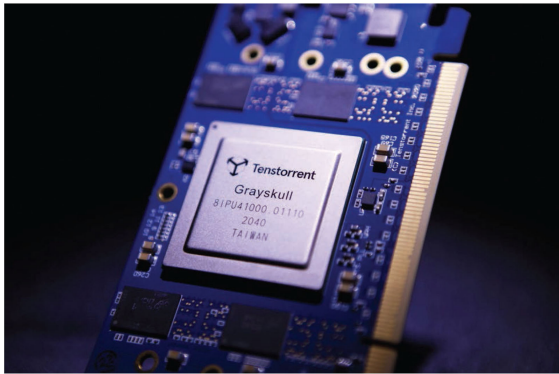


FIGURE 1. Grayskull 75W PCIe board.

GPU models. A network-on-chip is used to transfer data between the cores and to synchronize cores at various points during the program execution.

The Network-on-Chip (NoC) is a 2-D, bi-directional torus, with a bandwidth of 192 GB/s per node. The NoC was architected internally alongside the Tensix core, and it has been optimized for ML workloads, described in more detail in the “Block Sparsity” section. The NoC connects all the cores, as well as off-chip communication and memory controller blocks. As a result, each of the 120 compute cores has access to the PCIe and to the DRAM off-chip memory.

The chip includes a Gen4x16 PCIe block for communication with the host processor and other Grayskull devices. Eight channels of LPDDR4 are located along the north and south edges of the compute grid.

In terms of numerical precision, Grayskull supports:

1. full fp16/BFLOAT at 92 TFLOPs;
2. reduced precision mode of fp16/BFLOAT, at 122 TFLOPs;
3. block-based 8-bit floating point at 368 FLOPs.

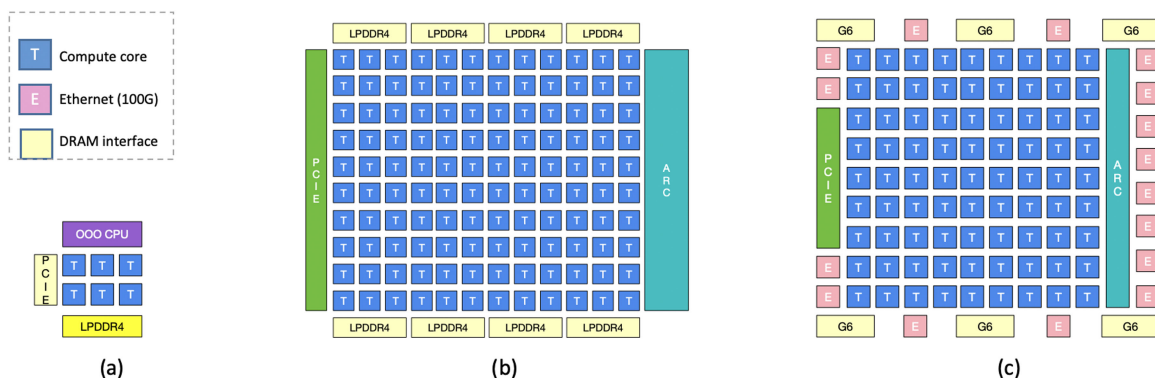


FIGURE 2. TensTorrent devices. (a) Jawbridge (2019). (b) Grayskull (2020). (c) Wormhole (2021).

TABLE 1. TensTorrent devices.

	Jawbridge	Grayskull	Wormhole
Manufactured	2019	2020	2021
Technology	14 nm	12 nm	12 nm
Compute grid	2x3 grid of cores	10x12 grid of cores	10x8 grid of cores
On-chip SRAM	6 MB	120 MB	120 MB
Off-chip IO	1 ch. LPDDR4, PCIe Gen4x4	8 ch. LPDDR4, PCIe Gen4x16	16 ports of 100G Ethernet, 6 ch. GDDR6, PCIe Gen4x16
CPUs	4 core OoO ARC	4 core OoO ARC	4 core OoO ARC

The *Wormhole* device (12 nm) contains 16 ports of 100-Gb Ethernet, an integrated network switch, and six channels of GDDR6. The compute fabric is composed of Tensix cores similar to Grayskull. This communication-oriented architecture realizes our vision of converged networking and accelerated AI compute on a single device. The large number of dedicated communication links on *Wormhole* enable many-device scaleout, by connecting *Wormhole* devices directly to each other, without a central host CPU processor or Ethernet switches.

Tensix Core

TensTorrent architecture operates on the basis of packets. The data units moved between the memories on the NoC are packets. Also, compute is executed directly on packets, shown in Figure 4. A single Tensix core contains a packet processor, a packet manager, SRAM, and five RISC processors. The RISC processors execute the runtime software which dispatches instructions to the packet processor and to the packet manager. On Grayskull, the SRAM has a capacity of 1 MB, with a

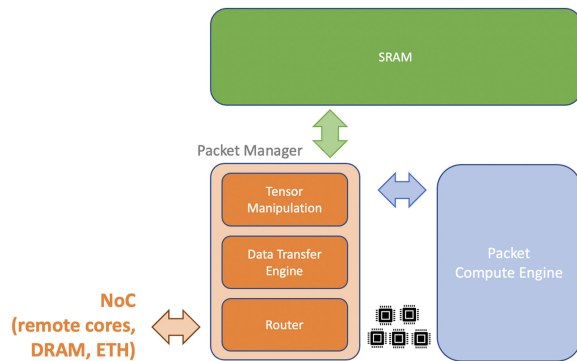


FIGURE 3. Single Tensix core.

384-GB/s read/write bandwidth. The memory space is primarily used by the local core, but it is directly accessible by remote cores as well.

Packet Compute Engine

The packet compute is a SIMD-based matrix and vector engine with a high degree of flexibility and programmability. Peak compute density is 3 TOPs at 8-bit precision, or 0.75 TFLOPs at 16-bit floating-point precision. The packet compute engine is software-programmable via the associated RISC cores, which execute kernels written in standard C language and issue matrix and vector instructions to the engine. A large number of PyTorch and TensorFlow deep learning instructions are supported. The matrix and vector engine support operations on a range of integer and floating-point formats. Most importantly it natively handles sparse computation to achieve speedup, reduce power, and memory footprint.

The packet compute engine does not have a global view of execution across the multicore system. Its operation is driven primarily by the packet manager: incoming packets from the packet manager are computed and returned to the packet manager for storage or data transfer.

Packet Manager Engine

The packet manager, depicted in Figure 3, is composed of data transfer engine, router, and tensor manipulation engine.

The *data transfer engine* is responsible for executing all data movement and synchronization among the compute engines, as well as between on-chip SRAM, off-chip memory and I/O. The packet manager and compute engine each receive their own unique instruction queues from the compiler, and they execute concurrently. The packet manager completely de-burdens

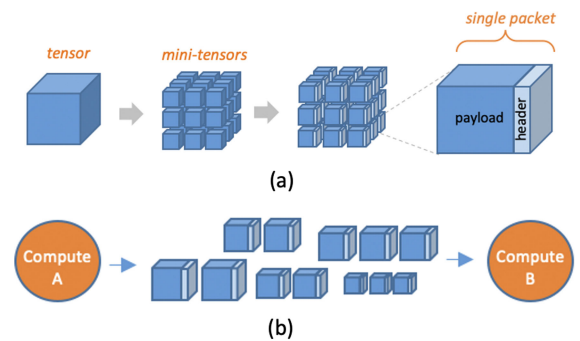


FIGURE 4. Packets flowing between compute operations. (a) Tensor decomposed into packets. (b) Packets flowing between compute operations.

the compute engine from the complexity of data movement and multicore synchronization. These features of the packet manager realize the push-based transfer model, which maximizes the overlap between compute and data transfers.

The *router* moves packets across the NoC. It provides guaranteed ordering, manages flow control and backpressure, and has deadlock-free operation. It is also optimized for the way AI workloads are parallelized across our multicore architecture and has efficient multicast and gather capabilities.

Finally, the *tensor manipulation engine* can perform dynamic packet compression; the smaller memory footprint enabled by compression results in faster data transfers and an increase in data locality. Furthermore, tensor manipulation instructions can be executed by this engine, described in the “Optimization of Tensor Manipulation Instructions” section.

SOFTWARE

The software is composed of three main pieces.

1. The machine learning framework integration and plugin software.
2. The runtime software, executing on the RISC processors.
3. The ahead-of-time graph compiler.

Framework Integration

The Tenstorrent compiler and runtime have been natively integrated into PyTorch, and support both inference and training flows. The user can target execution on a single device, or multidevice cluster. In either case, the hardware is visible to the user as a single device. The multidevice scheduling and parallelization are orchestrated behind the scenes by the software stack. In addition, the software stack can execute

ONNX networks as well, enabling a funnel from the frameworks that export into the ONNX format.

Graph Compiler

The graph compiler is composed of three main components, the front end, optimizer, and back end. The primary role of the front end is to lower a wide range of instructions to a smaller number of optimized instructions supported by the hardware. Instructions are parallelized and scheduled onto the device cores by the optimizer, which maximizes performance by balancing compute, data locality, and data movement. The back-end translates the compiled graph down into instruction queues for each core.

The packet managers and NoC connecting the cores is visible to the software and the data movement and synchronization are both controlled explicitly by the compiler. To schedule the data movement, the compiler packetizes each tensor by splitting it into “mini-tensors,” and each mini-tensor is combined with a packet header. Each packet header contains a unique packet ID, and all data is referenced via unique packet IDs. The header also contains routing information, enabling the packet manager to perform the desired data transfers between the cores across the NoC.

Runtime Software

The runtime software runs concurrently on RISC processors within every core. The compiled executable contains instruction queues for the packet processor and the packet manager of every core. The runtime software manages the queues, and dispatches instructions to the packet compute and the packet manager.

Buffers containing packets are dynamically allocated and de-allocated during runtime. The runtime software works in tight collaboration with the packet manager to store packets into the allocated buffers. The runtime also controls the storage target, allowing for buffers that do not fit into a core’s local SRAM to spill to either remote SRAM, or to an off-chip memory.

The architecture also supports various types of conditional executions such as if-statements, and for and while loops. The runtime software interprets the instruction queues generated for each core and can execute jumps to a specific instruction in the instruction queues to reflect control flow decisions.

FULL STACK PERFORMANCE OPTIMIZATIONS

Optimization of Data Transfers: The Push-Based Model

Traditional multicore devices operate on a pull-based data transfer model. For example, when a compute

core is ready to begin computing, as a first step it issues a request to copy remotely stored data (from another core’s cache, or from DRAM) into its local memory or cache. After the copy has been completed, the compute core starts computing. The read request latency combined with a data transfer through a potentially congested NoC or memory port, could result in the consumer core being idle while waiting for its data to arrive.

In contrast, our architecture operates on a *push-based* data transfer model. A core that produces an output buffer is aware of the consumer core that needs to receive it. Instead of waiting for the consumer core to issue a remote read request, the producer core proactively copies the buffer to the consumer core. This approach minimizes the idle time of the consumer core.

The data transfer engine executes all the required flow control for the push-based data transfer model. It receives an instruction queue from the graph compiler containing information about the producer-consumer connectivity. The instructions enable the data transfer engine to execute data transfers using a number of multicore synchronization instructions, including exchange of data transfer status, such as data-ready or memory-space-ready.

Optimization of Tensor Manipulation Instructions

Instructions that make up deep neural networks fall into two main categories: 1) math instructions, and 2) tensor manipulation (TM) instructions. TM instructions do not modify the data inside the tensor, but simply reshuffle the tensor contents. Common TM instructions in NLP networks include reshape, transpose, flatten, and permute.

The TM reshuffling is performed on the intermediate activation data, and hence must be executed during runtime. One implementation approach is to issue and execute each TM instruction independently to hardware during runtime. This typically involves a specific read/write pattern from/to memory, where the patterns match the particular TM to be implemented. GPUs execute TMs using this approach, shown in Figure 5(c), which idles compute cores while performing potentially complex memory access.

In contrast, our architecture overlaps the execution of math instructions performed by the compute engine and the TM instructions performed by the tensor manipulation engine. This process is facilitated by the graph compiler. The execution trace in Figure 5(b) shows the overlapping of compute and TMs. The

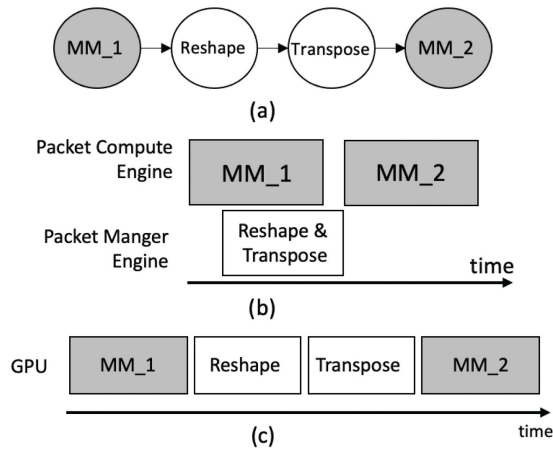


FIGURE 5. Tensor manipulation instructions executed on Grayskull and a GPU. (a) Application graph received from PyTorch. (b) Execution trace on a Grayskull Tensix core. (c) Execution trace on a GPU.

MM₁ compute instruction and the Reshape and Transpose instructions execute concurrently, in a pipelined fashion.

The tensor manipulation engine is programmable – it receives its own unique instruction queue from the compiler. TM instructions are executed using a combination of two methods. First, it contains a small storage that it uses as a scratch pad to load a packet and reshuffle it in place. Second, it can execute complex memory read/write patterns. Using a combination of these approaches, any tensor manipulation instruction can be implemented inline as the packets are being streamed out of the packet compute engine and being written into local SRAM.

Flexible Scheduling and Parallelization

The Tenstorrent architecture unlocks a tremendous amount of concurrency. All building blocks receive their own unique instruction queues from the compiler and can progress at their own pace. As a result, the overlap between compute and data transfers is maximized.

However, any single parallelization approach eventually plateaus, hence the desire to support flexible parallelization approaches along all available dimensions for any given compute layer. Each individual deep learning operation can be parallelized across a variable number of cores, combining a number of parallelization approaches. In addition, operations can be run in parallel, be pipelined, or sequential, across the many cores of a device, shown in Figure 6.

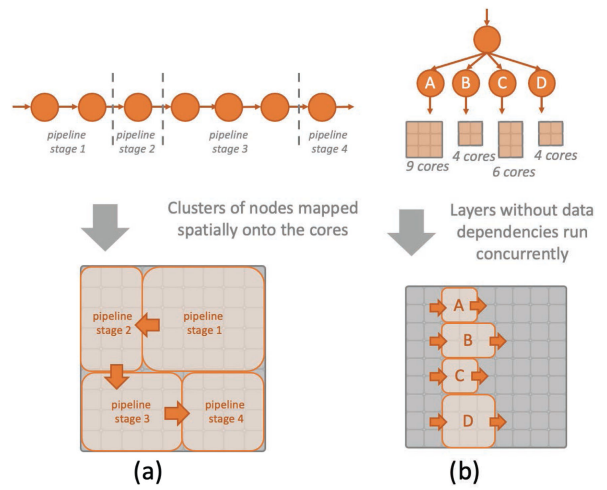


FIGURE 6. Flexible scheduling and parallelization. (a) Pipeline parallelism. (b) Model parallelism.

DYNAMIC EXECUTION

Dynamic execution is an umbrella term representing various approaches that reduce the computational complexity of a network at runtime. Some approaches can be represented within the topology of network itself, such as Mixture-of-Experts (MoE), while others can be used to augment the network execution during runtime. Four approaches enabled by the Tenstorrent architecture are described next.

Block Sparsity

Tensors feeding into math operations within networks contain a variable amount of sparsity within them. Certain models have been tuned to take advantage of sparsity of trained parameters,² or take advantage of block sparsity in model parameters.⁵ However, these approaches do not tap into a large potential of sparsity in activations, which can be inherent or induced at runtime.⁷ To fully utilize this potential, in addition to model parameter sparsity, our architecture supports block sparsity of activations, which enables quadratic gains from run-time activation sparsity.

Dynamic Precision

Similar to scientific computing applications, numerical precision can be traded off for an increase in performance and a reduction in power. The Tenstorrent architecture enables a numerical precision to be set at a fine grain level, per each packet in the neural network. The setting can be specified both ahead-of-time by the compiler, as well as during runtime.

Runtime Compression

Tensors can be compressed and decompressed during runtime via dedicated hardware blocks. Parameters can be compressed at compile time, and decompressed at runtime by the hardware. Similarly, during runtime, the output activations of a math layer can be compressed inline as they are being produced. Both approaches result in reduced memory footprint, in addition to power savings from smaller and faster data transfers.

Conditional Computation

MoE¹ involves the use of conditional computations, where only parts of the network are computed, on a per-input basis. This allows a significant increase in model capacity, without a proportional increase in computational complexity. A host CPU can easily implement such conditional computation, however that approach involves unnecessary data transfers between host and the AI processor and is impractical for large scaleout implementations. The Tenstorrent architecture is the first architecture, to the best of our knowledge, that tightly couples native conditional computation together with the dense compute within an AI processor fabric.

RESULTS

We measured a baseline result for BERT-base in BFLOAT16 precision at 2830 sequences/s. Significant speedup is achievable by applying two optimizations on top of this baseline: dynamic activation sparsity and use of 8-bit floating point. In our experiments we observe that 75% sparsity in activations (induced dynamically at runtime) results in 4x speedup on BERT layers. Similarly, we observe that using block-based 8-bit floating point precision provides a factor of two. The two optimizations can be combined synergistically—they both reduce the activation memory footprint linearly for a total of 8x reduction, and 8-bit floats reduce the model parameter memory footprint by 2x. This enables the majority of the model parameters to fit on chip allowing the sparsified layers to be fed from local SRAM. Realizing this on an entire BERT-base is work in progress and we project a score of 23 345 sequences/s.

CONCLUSION

We solve the private memory parallel computing problem and tensor manipulations in a way that removes communication, synchronization, and data shuffle bottlenecks and enables keeping all the compute units highly utilized.

In comparison to alternative and competing architectures, Tenstorrent's architecture is the only one which has all the features of the next-generation computing architecture, one that allows fusion of Software 2.0 ("neural nets")⁸ and Software 1.0 ("classical programs"):

- › Same programming model for single-chip and multi-chip scaleout. Parallelization for private memory combined with direct compute-on-packets.
- › Flexible parallelization across all tensor dimensions and approaches, including the time dimension.
- › Native ability to intermingle dense-math-heavy nodes of a neural net with sparse nodes, procedural nodes (i.e., "classical programs"), and various dynamic and conditional execution techniques. These are all critical building blocks for a compute substrate that allows fusion of Software 1.0 and Software 2.0., entirely removing the need for host CPU fallback.
- › Fully programmable: kernels and runtime system (firmware) are written in standard C language.
- › Scaleout capability and flexibility: scaleout via standard Ethernet that seamlessly integrates into our NoC.

REFERENCES

1. N. Shazeer *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *Proc. Int. Conf. Learn. Representations*, 2017. [Online]. Available: <https://arxiv.org/abs/1701.06538>
2. V. Sanh *et al.*, "Movement pruning: Adaptive sparsity by fine-tuning," 2020. [Online]. Available: <https://arxiv.org/abs/2005.07683>
3. OpenAI, "AI and compute," 2018. [Online]. Available: <https://openai.com/blog/ai-and-compute/>
4. Stanford HAI. "Artificial intelligence index report 2019," 2019. [Online]. Available: https://hai.stanford.edu/sites/default/files/ai_index_2019_report.pdf
5. T. Gale *et al.*, "Sparse GPU kernels for deep learning," 2020. [Online]. Available: <https://arxiv.org/pdf/2006.10901.pdf>
6. "Fast block sparse matrices for pytorch," 2020. [Online]. Available: https://github.com/huggingface/pytorch_block_sparse
7. Z. Chen *et al.*, "You look twice: Gaternet for dynamic filter selection in CNNs," 2019. [Online]. Available: <https://arxiv.org/pdf/1811.11205.pdf>
8. A. Karpathy, "Software 2.0," 2017. [Online]. Available: <https://medium.com/@karpathy/software-2-0-a64152b37c35>