THEME ARTICLE: HOT CHIPS 2024

FuriosaAI RNGD: A Tensor Contraction Processor for Sustainable AI Computing

Younggeun Choi, Junyoung Park, Sang Min Lee, Jeseung Yeon, Minho Kim, Changjae Park, Byeongwook Bae, Hyunmin Jeong, Hanjoon Kim, and June Paik, *FuriosaAI, Inc., Seoul, 06040, Republic of Korea*

Nuno P. Lopes, *INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal*

Sungjoo Yoo, *Seoul National University, Republic of Korea*

Abstract—Modern AI workloads require architectures capable of efficiently managing diverse tensor contraction patterns. Traditional approaches based on fixed-size matrix multiplications often fall short in scalability and flexibility. RNGD (pronounced "Renegade"), a second-generation tensor contraction processor (TCP), introduces an innovative architecture designed to exploit the parallelism and data locality inherent in tensor computations. Its coarse-grained processing elements (PEs) can operate as a unified large-scale unit or as multiple independent units, providing flexibility for various tensor shapes. Key innovations, such as a circuit switch-based fetch network, input broadcasting, and buffer-based reuse mechanisms, further enhance computational efficiency.
RNGD represents a significant advancement in processor architecture, delivering optimized performance and energy efficiency for sustainable computation of next-generation AI workloads.

The escalating energy demands of AI computing, exacerbated by the significant cooling requirements of kilowatt-consuming GPUs, are steering us towards an unsustainable future. There is an urgent need for a revolutionary AI accelerator tailored to the rigorous demands of large language models (LLMs)—one that provides high memory bandwidth, substantial capacity, and dense compute power, yet dramatically reduces power consumption to allow for cost-effective air-cooling solutions.

Most major commercial AI accelerators, including GPUs, are primarily based on matrix multiplication as the fundamental operation for hardware acceleration.[1,2,3,4] However, tensor contraction is the fundamental operation in machine learning models. Matrix multiplication is just a particular instance of contraction.

Traditional computational architectures implement tensor contraction by mapping or decomposing it into fixed-size matrix multiplication units. This method, however, may not fully leverage the inherent parallelism and data locality specific to tensor contractions. Inference tasks typically involve diverse tensor shapes, necessitating the exploitation of parallelism and data reuse across both tensor shapes and batch sizes. However, chips based on large matrix units face challenges in fully utilizing these units across varied tensor operations due to their fixed sizes.[5,6] Additionally, when the unit size of an accelerator, such as matrix or tile size, is small, the opportunities for data reuse are limited.[2] Spatial accelerators, which consist of numerous small PEs, each with a local memory and interconnected by a network-on-chip (NoC), suffer from a high complexity of parallelizing operations in a NoC-aware manner.

Instead of relying on fixed-size matrix multiplication units, we use tensor contraction as the fundamental computational primitive. Our architecture not only facilitates massively parallel operations but also incorporates temporal pipelining, similar to vector processors. We have engineered large coarse-grained PEs that can be divided into smaller units called *slices*. This design allows for highly flexible configurations to accommodate a variety of tensor shapes. The fetch network, which connects these slices, determines whether

XXXX-XXX © 2025 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

the slices function as a single large PE or as multiple small, independent, and parallel compute units.

The computational units in the tensor contraction processor (TCP)[7] run deterministically, as defined by the software, ensuring performance predictability, which allows us to develop precise cost models for both performance and energy consumption. These cost models allow our compiler to explore the most effective configurations of tensor shapes and contractions.

We propose a domain-specific architecture for AI workloads that elevates the hardware-software interface by treating tensor contractions as a primitive. This approach not only streamlines hardware design for higher performance and energy efficiency, but also provides the flexibility to support a wide variety of AI models. Compared to power-hungry GPUs that target general parallel processing, TCP provides abstractions that target the computing space of deep learning. By integrating various compiler optimizations, while minimizing the complexity of the hardware, TCP makes AI more energy efficient and sustainable.

TENSOR CONTRACTION AS A PRIMITIVE

Tensor contractions, fundamentally reducible to high-dimensional matrix operations, are compute and memory intensive. Executing tensor contractions efficiently requires optimal scheduling of operations across compute units, as well as careful management of data movement and reuse. The performance and energy efficiency of these operations are directly influenced by how the computation of individual units is temporally and spatially scheduled.

The scheduling of operations across multiple compute units determines not only the temporal locality of data but also the overall computational efficiency. Temporal locality, dictated by the sequence in which operations are scheduled, impacts how much data is reused. High data reuse improves energy efficiency. Conversely, the spatial distribution of computations across compute units affects the cost of data movement and the utilization of compute units. Poor spatial scheduling leads to increased data transfers and underutilized compute resources, diminishing both performance and energy efficiency.

Our architecture addresses these challenges by integrating tensor computations and scheduling into a unified approach. Temporal scheduling determines the order of sequential data processing to reuse local data. Spatial scheduling distributes operations across compute units to ensure balanced workloads. It also contributes to reducing memory traffic via multicast capabilities, which distribute data across multiple compute units, while temporal pipelining enables the continuous utilization of spatially distributed compute units.

Figure 1 illustrates a simple example of a tensor contraction operation, where matrices A and B are multiplied to produce C. This operation can be represented as a nested for-loop, as shown in the middle. On actual hardware, this computation is executed in parallel using multiple compute units and SRAM memories.

For simplicity, consider a chip with 4 SRAM blocks and 4 compute units. The outermost loop corresponds to parallel execution along specific axes, the innermost loops represent unit computations performed every cycle, and the intermediate loops handle scheduling across time and space. This unified approach treats tensor contraction as a single primitive, with tensors being distributed across SRAM and computations scheduled spatially and temporally.

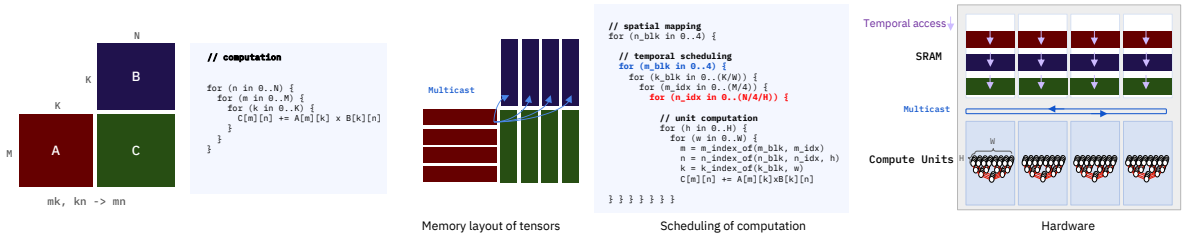
We organize compute units into fetch-operation-commit stages, with each compute unit being paired with an SRAM block, forming a slice (Figure 2). Temporal pipelining enables these slices to process tensor data continuously, with the fetch stage retrieving tensors in a defined order and the fetch network dynamically configured to distribute data to the operation unit.

Note that the dataflow in our architecture is fully streamlined by the compiler. All datapaths, whether fetching multi-dimensional data, supplying inputs to the dot product engine, or routing data through the fetch network, are orchestrated to ensure a unified and well-integrated flow of operations. The fetch unit reads data from SRAM continuously, in an order determined by the compiler. The fetched data is multicast, via the fetch network configured by the compiler, and selectively processed by the contraction, vector, or transpose engines. Finally, the commit unit stores the computation results back to local SRAM in a memory layout that is optimized by the compiler for the following operation.

The operation unit, shown in Figure 2, is designed to maximize data reuse and thus it is equipped with a feed (input) buffer, a register file, and an accumulator. The compiler establishes the streamlined dataflow encompassing local SRAM, the fetch network, and the buffer/register of the operation unit.

The fetch network provides the operation units with a steady, sequential supply of data. The network adopts circuit switching to perform multiple multicasts along fixed routes, e.g., multiple ring connections, during tensor operations.

The reconfigurability of the fetch network is crucial in supporting diverse models and inference scenarios. Figure 3 illustrates how our architecture supports dif-



(a) Conceptual computation (b) Compiled intermediate representation (c) Hardware mapped operation

FIGURE 1. A simple tensor contraction example, consisting of multiplying matrices A and B to produce C.

FIGURE 2. Organization of compute units into slices. Below, we show how spatial-temporal orchestration boosts utilization.

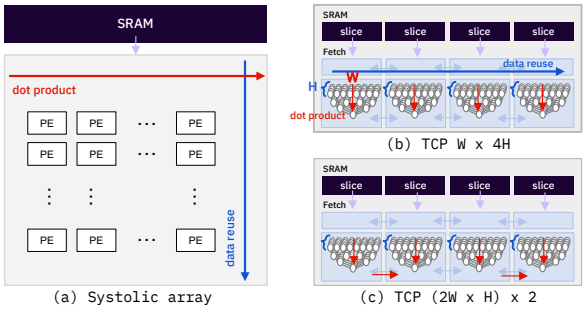


FIGURE 3. Flexible reconfigurability is crucial to support the parallelism and data reuse behavior in diverse models and inference scenarios.

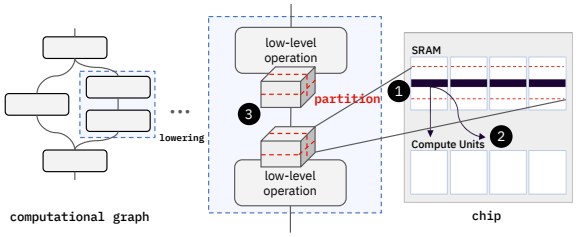


FIGURE 4. Aspects that the compiler must consider when laying out a tensor in memory for a low-level einsum computation: (1) SRAM access performance (e.g., row/column major), (2) data movement of input tensors to compute units for a single low-level operation, (3) data movement across operators.

ferent tensor contractions by reconfiguring the fetch network. When the input data can be heavily reused, the fetch network is configured as a large multicast network. For example, to mimic the behavior of a systolic array as in Figure 3(a), the network can be set to 4 slices (4H), as shown in Figure 3(b). However, when handling large input dimensions in tensor contraction, the fetch network can be dynamically reconfigured to compute a large contraction with multiple contraction engines, for example, through 2 slices (2W), as shown in Figure 3(c). Such reconfigurability is particularly advantageous for inference tasks, where batch sizes can vary significantly. Unlike large matrix multiplication units optimized for static, large batch sizes, our architecture aims at making the best use of the parallelism and data reuse inherent in diverse tensor shapes through optimal scheduling of computations and reconfiguration of the fetch network.

DATA MOVEMENT OPTIMIZATION BY THE COMPILER

Besides scheduling computations, deciding how tensors are laid out in memory is also extremely impor-

tant, as it directly impacts performance and efficiency. Therefore, when the compiler lowers the input computational graph into primitives, it also optimizes the mapping of tensors to on-chip memory.

There are multiple factors to consider when laying out a tensor in memory, as shown in Figure 4. Firstly, the last axis of memory layout impacts memory access performance. For example, whether the stored layout is row-major or column-major and whether it aligns with the way you read it impacts SRAM bandwidth. Secondly, since data is often used across multiple compute units, the memory layout impacts data movement, thereby influencing the performance and energy consumption of the NoC. Finally, the output tensor of one operator is often the input for another operator. If the memory layouts between operators are incompatible, additional data movement may occur. We can eliminate unnecessary data shuffling if we manage to align the memory layouts of the input/output tensors of consecutive operators.

The compiler optimizes code for end-to-end efficiency by extensively exploring a carefully designed space of tactics. For each operator's lowered shape, it searches a tactic space composed of axis permutations to determine the most efficient execution strategy. To maximize effective SRAM utilization, the compiler minimizes redundant data storage across slices and orchestrates data movement through the slice-level network, directing it to compute engines while tracking data lifetimes. By precisely managing when and where data is accessed, the compiler enables a static scheduling approach that ensures efficient execution and resource utilization, allowing multiple operators to be optimally mapped to hardware. Due to the deterministic nature of the hardware, the compiler can accurately predict performance and power consumption for each possible configuration. This predictability allows the compiler to explore the design space effectively, balancing data lifetimes, execution times, and power budgets to select the most efficient execution plan.

Compiling a model involves optimizations across multiple levels of intermediate representation (IR). This optimization process includes operator fusion at the graph level, memory allocation, tensor splitting, merging, and scheduling.

MICROARCHITECTURE

Our processor performs tensor contraction as its core abstraction. The entire SoC was designed to realize tensor contraction for flexibility, to adapt to various tensor shapes, as well as high performance. Each RNGD chip has eight PEs, as illustrated in Figure 5. Each PE can operate independently, much like multi-instance capabilities found in GPUs. Furthermore, up to four PEs can be combined into a larger, unified PE for handling tasks requiring greater computational resources. This modular and dynamic structure ensures adaptability to diverse workloads.

RNGD supports single root I/O virtualization (SR-IOV), enabling multiple virtual machines (VMs) to utilize the chip while maintaining separate address spaces. This makes it particularly effective for multi-tenancy in cloud server environments, where secure and efficient resource sharing is essential.

Processing Element

The processing element (PE) is the foundational building block of RNGD. In order to minimize the operational complexity of the entire chip, we maximized the size of the PE, the unit for programming computations, within the limits of our design flow. Each PE integrates three key components: a CPU core, a tensor DMA engine (TDMA), and a tensor unit (TU). This architecture provides a balance of flexibility, performance, and energy efficiency, making it adaptable to a wide range of computational tasks.

The CPU core supports the TU by managing scalar operations, control flow, and simple vector operations. It acts as the controller for the TU, issuing commands through a queue and managing tensor data movement via the TDMA to ensure efficient data transfers.

As shown in Figure 6, the CPU core initiates PE operations by pushing commands into the tensor unit's command queue, operating asynchronously to drive both the TU and the TDMA. It operates through the PE firmware stored in the scratchpad memory, which also contains the tensor unit commands. The CPU core continuously pushes commands to prevent the command queue from going idle. These commands include load commands to set control registers of the tensor unit, execute commands to perform tensor operations with the loaded registers, DMA commands to drive the TDMA, and wait commands for synchronization. Except for the wait command, all the commands operate asynchronously, enabling the overlap of TDMA and TU operations. TDMA transfers the necessary tensors for the next operation from DRAM to SRAM in an optimized memory layout as determined by the compiler.

The TU serves as the main compute engine of the PE, delivering up to 64 TOPS and 32 MB of SRAM. To enhance scalability and efficiency, the TU's compute pipeline is divided into multiple slices, with each slice functioning as an independent unit. Each slice has

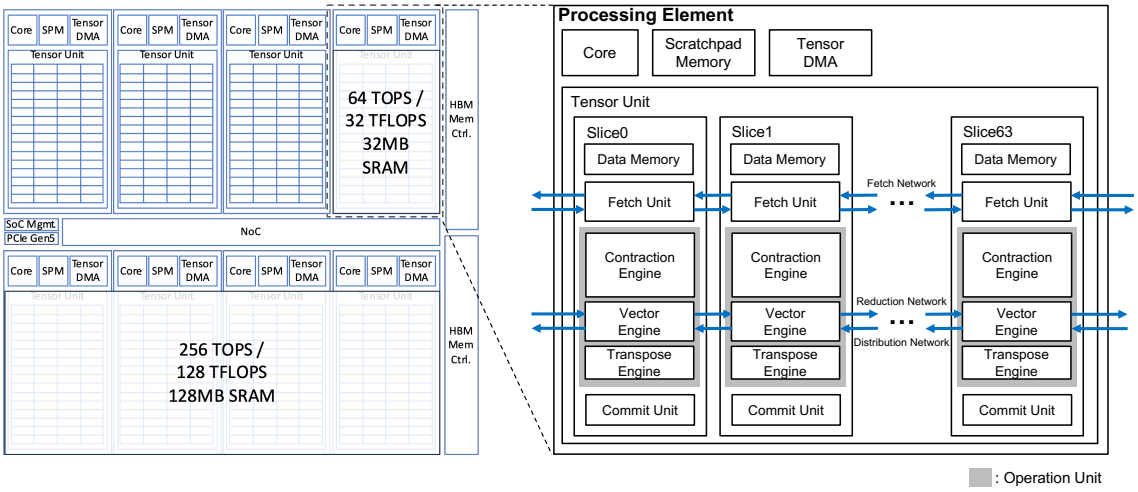


FIGURE 5. Overall architecture of RNGD and the internal components of the Processing Element (PE).

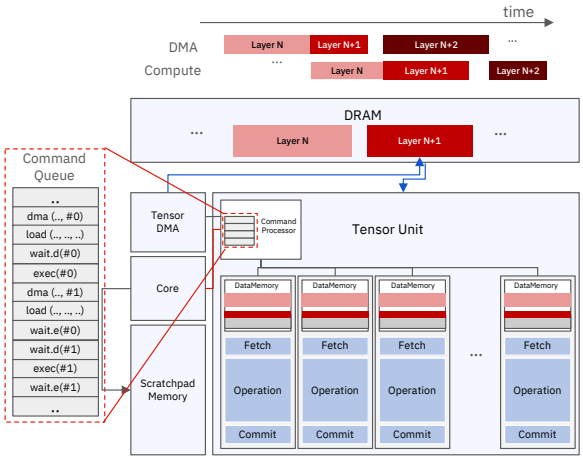


FIGURE 6. The CPU core enqueues commands (e.g., execution of tensor operations and DMA transfers) to be processed asynchronously.

its own SRAM, fetch unit, operation unit, and commit unit, enabling them to handle tensor operations autonomously. This modular design allows slices to operate individually for small-sized tasks like contractions on small tensors or collaborate with each other for larger tasks.

The fetch unit reads data from memory in an N-dimensional loop style, effectively handling the complex data access patterns required by tensor operations. It supplies the loaded data to the operation units by sending and receiving data via the fetch network, coordinating with the fetch units on the other slices. The fetch network ensures that data is delivered to the operation units in the correct sequence needed for tensor computations, enabling efficient data sharing and multicast operations across slices.

The commit unit stores results back to the data memory, also following an N-dimensional loop style. It ensures that the results are stored in optimal locations for subsequent operations, as determined by the compiler, which helps reduce unnecessary data movements.

Within each slice's operation unit, there are three specialized compute engines: the contraction engine (CE), the vector engine (VE), and the transpose engine (TE). These engines can be dynamically connected by the compiler to execute diverse operations in a flexible and efficient manner.

The CE contains multiple dot-product engines (DPEs), each capable of high-throughput element-wise multiplications with configurable reduction trees. To minimize data movement and improve energy efficiency, the architecture incorporates data reuse strategies tailored to different tensor operation requirements. Weights are reused through registers (weight stationary), input tensors can be reused while being retained in feed buffers (input stationary), and intermediate results are stored and reused in accumulators (output stationary). These strategies are optimized by the compiler to match the given workload demands.

The VE is designed for flexible computations, handling element-wise operations, reductions, and type conversions. It features multiple functional units capable of pipelining and executing INT32/FP32 arithmetic, transcendental functions, and various type con-

versions. It also supports intra-slice and inter-slice reductions, achieving 8-way throughput per slice. The reduction/distribution network between VEs enables efficient communication for these reductions across slices. This unit handles arithmetic tasks like softmax (element-wise exponential and sum reduction) and layer normalization. Moreover, it can dynamically chain outputs from the CE, such as fusing expand and activation steps in feed-forward layers (e.g., SiLU), for improved performance and energy efficiency.

The TE rearranges data layouts to meet the requirements of complex tensor computations. It handles data transposition and reshaping operations, which are critical for optimizing memory access patterns and computational efficiency in tensor operations.

The coordination of all these operations within the TU is managed by internal sequencers, which orchestrate data flow, register indexing, and accumulator usage based on control register settings configured by the CPU core prior to computation. This precise control ensures efficient execution of tensor operations across slices, maximizing performance and resource utilization.

Through the integrated networks—the fetch network and the reduction/distribution network between VEs—RNGD can efficiently store data in the desired layout while performing computations expressed in the abstraction of tensor contraction. The fetch network primarily handles inter-slice data movement with sufficient flexibility for typical tensor operations, while the reduction/distribution network allows intermediate results to be placed in layouts that are advantageous for subsequent tensor operations. In addition, hardware hierarchy-aware DMA descriptors are used to manage TDMA transfers, minimizing contention and enabling arbitrary data movements. Collectively, these mechanisms provide efficient communication and data sharing between slices, accommodating the complex data access patterns and computational dependencies that arise in tensor operations.

By combining modularity, flexible data handling, and efficient compute engines, the PE achieves high performance and adaptability. It is well-suited for a wide range of AI workloads, from small-scale inference tasks to large-scale computations.

NoC and Interfaces

RNGD features a hierarchical network-on-chip (NoC) to manage both internal and external data communications. Within each PE, the intra-PE NoC connects its components, including the CPU core, scratchpad memory, TDMA, and TU. Externally, the inter-PE NoC and memory routers handle data transfers between PEs and HBM3 stacks.

TABLE 1. Characteristics of RNGD.

Technology	TSMC 5nm
Freq. & TDP	1GHz, 150W TDP
Dimensions	24.77 mm × 26.38 mm (653.4 mm^2)
DRAM	2x HBM3 stack, 48GB, 1.5 TB/s
On-Chip SRAM	256 MB, 384 TB/s
Host Connectivity	PCIe Gen5 x16 (128 GB/s)
MACs	512 TOPS (INT8), 1024 TOPS (INT4) 256 TFLOPS (BF16), 512 TFLOPS (FP8)
Vector Engine	512 ways per PE transcendental functions (exp, cos, tanh, etc.)

Each chip integrates two stacks of HBM3, offering up to 1.5 TB/s of memory bandwidth, ensuring that all eight PEs can achieve their full bandwidth potential of 256 GB/s. The NoC supports full channel interleaving for efficient HBM utilization and facilitates contention-free data transfers between PEs at the same rate. The signal and power integrity of HBMs has been verified by extensive simulations including the SoC, the interposer, the package, and the PCB.

For workloads exceeding the capacity of a single chip, inter-chip communication is enabled through PCIe Gen5 interfaces with peer-to-peer (P2P) communication, supporting transfer rates of 64 GB/s in each direction. Each PE operates within an independent address space, and unauthorized access is restricted by the address translation unit (ATU). The address translation mechanism abstracts the PE's memory and compute resources, enabling dynamic allocation and seamless inter-PE communication.

Power Management

Table 1 outlines the specifications of RNGD, which is tailored to meet the demanding requirements of AI inference tasks requiring low latency and low power consumption. The chip operates within a thermal design power (TDP) of 150 W, enabling air-cooled deployment in standard data centers.

The SoC's computational backbone consists of CPU cores running at up to 2 GHz with a nominal voltage of 0.75 V, leveraging dynamic voltage and frequency scaling (DVFS) to balance power efficiency and performance. To handle peak current demands of up to 1 kA with significant fluctuation, the design integrates on-die and on-interposer decoupling capacitors (both metal-insulator-metal and deep-trench types). For improved reliability in data center environments, the chip includes timing margin monitors to ensure robust operation under dynamic workloads. Margin

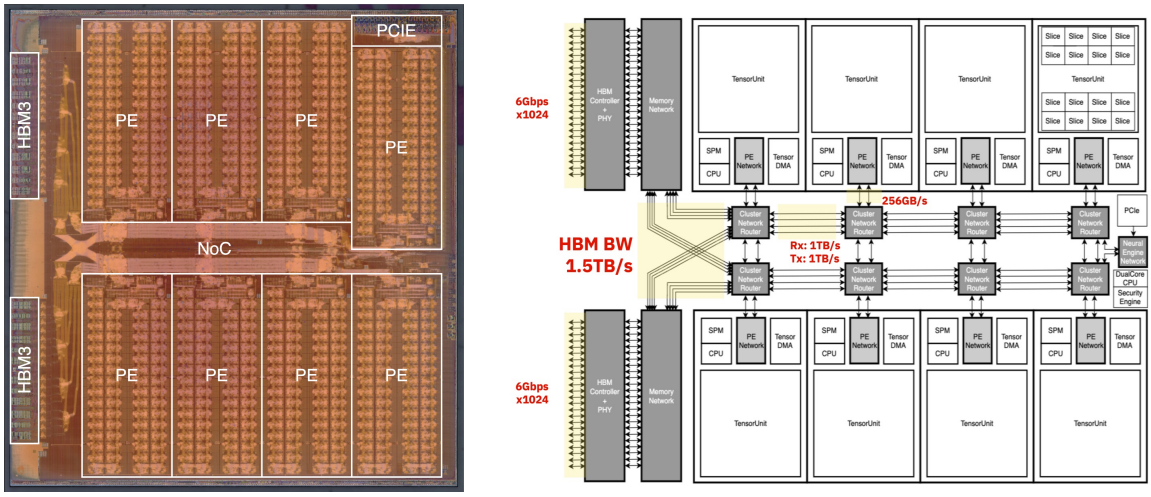


FIGURE 7. Die photo and interconnection networks.

	RNGD	NVIDIA L40S	NVIDIA H100	Intel Gaudi 2	Google TPU v5e
Performance (queries / sec)	12.0 (FP8)	12.3 (FP8)	31.1	10.5	2.5
Power (watt)	185	320	N/A	N/A	N/A
Data source	measured	measured	MLPerf 4.0	MLPerf 3.1	MLPerf 4.0

Disclaimer: Unverified by MLPerf

FIGURE 8. GPT-J 6B MLPerf benchmark scenario (99% accuracy).

monitors enable long term performance drift monitoring even after deployment. The integrated voltage droop sensor monitors instantaneous local voltage drop and generates interrupts. Secure boot capabilities further enhance data integrity and system security.

EVALUATION

To evaluate the proposed system, we conducted a performance and efficiency analysis using the MLPerf benchmark suite.[8] The results are shown in Figure 8. The primary metric for evaluation is performance per watt (Perf/W), which captures the system's computational efficiency. MLPerf was chosen due to its comprehensive benchmarking capabilities, including a workload generator that simulates a variety of input context lengths derived from real-world distributions. This ensures that the performance measurements are representative of actual deployment scenarios. For this study, we used GPT-J, a representative architecture of large language models (LLMs).

Our chip achieved 12.0 queries per second at a measured board power consumption of 185 watts, resulting in a Perf/W improvement of 68.8 % over the L40S and 46.1 % over the H100.[a] This result demonstrates the effectiveness of our approach in enhancing energy efficiency while maintaining high performance. Furthermore, it took only three months to achieve the current level of performance, highlighting the rapid pace of progress in optimizing internal hardware parameters to achieve its full specifications, advancing the compiler's scheduling of tensor operations, and enhancing the runtime software stack for improved efficiency. We anticipate additional improvements in both performance and power efficiency in future iterations, as further refinements are underway. In summary, this result validates the proposed optimizations and their potential applicability to broader LLM workloads, providing a robust foundation for continued advancements in performance per watt efficiency.

In addition to the performance evaluation on the GPT-J benchmark, we are in the early stages of evaluating and improving the performance of the Llama 3.1 8B model. Using the OpenOrca dataset,[9] which is employed by MLCommons for Q&A benchmarking, we measured the offline throughput of the system. At a total power consumption of 181 W (159 W for the chip alone), the system currently delivers 3,265 tokens per second. When expressed as performance per watt, this corresponds to an efficiency of 18.0 tokens per joule. These initial results are very promising and provide a foundation for further optimizations to enhance both

[a]The power consumption of the H100 was assumed to be equal to its TDP (700W).

throughput and energy efficiency.

CONCLUSION

We presented RNGD, a novel accelerator for AI workloads. Recognizing tensor contraction as a foundational concept in deep learning models, RNGD natively supports tensor contractions instead of decomposing them into matrix multiplications. This alignment allows us to exploit inherent data movements and reuse within tensor computations, leading to superior efficiency.

As a result, RNGD delivers 68.8% and 46.1% more performance per watt than L40S and H100, respectively. With a TDP of 150 W, which is compatible with current air-cooled data centers, RNGD avoids the need for complex cooling solutions required by the latest GPUs. By natively supporting tensor contractions, RNGD offers unmatched efficiency and performance for modern AI applications.

REFERENCES

1. K. Chatha, "Qualcomm® cloud AI 100: 12TOPS/W scalable, high performance and low latency deep learning inference accelerator," in *Hot Chips*, 2021. doi: 10.1109/HCS52781.2021.9567417.
2. N. Jouppi *et al.*, "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *ISCA*, 2023. doi: 10.1145/3579371.3589350.
3. S. Knowles, "Graphcore," in *Hot Chips*, 2021. doi: 10.1109/HCS52781.2021.9567075.
4. E. Medina and E. Dagan, "Habana labs purpose-built AI inference and training processor architectures: Scaling AI training systems using standard ethernet with gaudi processor," *IEEE Micro*, vol. 40, no. 2, pp. 17–24, 2020. doi: 10.1109/MM.2020.2975185.
5. Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU platforms for deep learning," 2019, arXiv:1907.10701.
6. D. Zhang *et al.*, "A full-stack search technique for domain optimized deep learning accelerators," in *ASPLOS*, 2022. doi: 10.1145/3503222.3507767.
7. H. Kim *et al.*, "TCP: A tensor contraction processor for AI workloads," in *ISCA*, 2024. doi: 10.1109/ISCA59077.2024.00069.
8. V. J. Reddi *et al.*, "Mlperf inference benchmark," 2019, arXiv:1911.02549.
9. W. Lian, B. Goodson, E. Pentland, A. Cook, C. Vong, and "Teknium", "OpenOrca: An open dataset of GPT augmented FLAN reasoning traces," 2023. [Online]. Available: https://huggingface.co/datasets/Open-Orca/OpenOrca

Younggeun Choi is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at ygchoi@furiosa.ai.

Junyoung Park is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at zenojur@furiosa.ai.

Sang Min Lee is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at sage.lee@furiosa.ai.

Jeseung Yeon is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at jeseung.yeon@furiosa.ai.

Minho Kim is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at minho@furiosa.ai.

Changjae Park is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at changjae.park@furiosa.ai.

Byeongwook Bae is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at lidwoogi@furiosa.ai.

Hyunmin Jeong is a hardware engineer at FuriosaAI, Seoul, Republic of Korea. Contact him at hyunmin.jeong@furiosa.ai.

Hanjoon Kim is the CTO and co-founder of FuriosaAI, Seoul, Republic of Korea. Contact him at hanjoon@furiosa.ai.

June Paik is the CEO and co-founder of FuriosaAI, Seoul, Republic of Korea. Contact him at junepaik@furiosa.ai.

Nuno P. Lopes is an associate professor at Instituto Superior Técnico – University of Lisbon, Portugal, and an advisor at FuriosaAI. Contact him at nuno.lopes@tecnico.ulisboa.pt.

Sungjoo Yoo is a professor at the Seoul National University (SNU), Republic of Korea. Contact him at sungjoo.yoo@gmail.com.