TCP: A Tensor Contraction Processor for AI Workloads

Industrial Product*

Hanjoon Kim, Younggeun Choi, Junyoung Park, Byeongwook Bae, Hyunmin Jeong, Sang Min Lee, Jeseung Yeon, Minho Kim, Changjae Park, Boncheol Gu, Changman Lee, Jaeick Bae, SungGyeong Bae, Yojung Cha, Wooyoung Choe, Jonguk Choi, Juho Ha, Hyuck Han, Namoh Hwang, Seokha Hwang, Kiseok Jang, Haechan Je, Hojin Jeon, Jaewoo Jeon, Hyunjun Jeong, Yeonsu Jung, Dongok Kang, Hyewon Kim, Minjae Kim, Muhwan Kim, Sewon Kim, Suhyung Kim, Won Kim, Yong Kim, Youngsik Kim, Younki Ku, Jeong Ki Lee, Juyun Lee, Kyungjae Lee, Seokho Lee, Minwoo Noh, Hyuntaek Oh, Gyunghee Park, Sanguk Park, Jimin Seo, Jungyoung Seong, June Paik, Nuno P. Lopes†, and Sungjoo Yoo‡
Furiosa AI, Inc.
† INESC-ID / Instituto Superior Técnico, University of Lisbon
‡ Seoul National University

***Abstract*—We introduce a novel tensor contraction processor (TCP) architecture that offers a paradigm shift from traditional architectures that rely on fixed-size matrix multiplications. TCP aims at exploiting the rich parallelism and data locality inherent in tensor contractions, thereby enhancing both efficiency and performance of AI workloads.**

TCP is composed of coarse-grained processing elements (PEs) to simplify software development. In order to efficiently process operations with diverse tensor shapes, the PEs are designed to be flexible enough to be utilized as a large-scale single unit or a set of small independent compute units.

We aim at maximizing data reuse on both levels of inter and intra compute units. To do that, we propose a circuit switch-based fetch network to flexibly connect compute units to enable inter-compute unit data reuse. We also exploit input broadcast to multiple contraction engines and input buffer based reuse to further exploit reuse behavior in tensor contraction. Our compiler explores the design space of tensor contractions considering tensor shapes and the order of their associated loop operations as well as the underlying accelerator architecture.

A TCP chip was designed and fabricated in 5nm technology as the second-generation product of Furiosa AI, offering 256/512/1024 TOPS (BF16/FP8 or INT8/INT4) with 256 MB SRAM and 1.5 TB/s 48 GB HBM3 under 150 W TDP. Commercialization will start in August 2024.

We performed an extensive case study of running the LLaMA-2 7B model and evaluated its performance and power efficiency on various configurations of sequence length and batch size. For this model, TCP is 2.7× and 4.1× better than H100 and L40s, respectively, in terms of performance per watt.

I. Introduction

Most major commercial AI accelerators primarily focus on integrating matrix multiplication as a basic operation of hardware acceleration [2], [9], [10], [12]. Fundamentally, however, the core operation of machine learning models is tensor contraction. Tensor contraction involves summing the elements of certain axes of a tensor. Matrix multiplication is a representative example of tensor contraction. For example, a multiplication of matrices with dimensions $m\times k$ and $k\times n$ can be expressed as a tensor contraction $mk,kn\to mn$. Contraction is done over the (common) k axis. Tensor contraction forms the backbone of machine learning models, handling computations over multi-dimensional data.

Traditional computational architectures often map or divide tensor contractions into matrix multiplication units. This approach may fail to fully utilize the rich parallelism and data locality inherent in tensor contractions themselves. When the unit size of the accelerator (e.g., matrix or tile size) is small, the scope for data reuse is limited, hurting efficiency [7]. In addition, in many spatial accelerators, where a large number of small processing elements (PEs) are equipped with local memory and connected via a network-on-chip (NoC), there is inherently high complexity in parallelizing operations while being NoC-aware. Moreover, generating programs for numerous wimpy cores is known to be more challenging than for a few brawny cores [7]. Except for GPUs, there are few successful commercial chips with a large number of small PEs that provide a software stack capable of compiling arbitrary tensor-manipulating programs into efficient executables.

Inference is often characterized by diverse tensor shapes and thus it is essential to exploit the parallelism and data reuse derived from the tensor shapes as well as the batch size. Thus, in the case of chips with large matrix units, it is challenging to fully utilize the large units across various shapes and types of tensor operations [3], [23], [25].

Instead of matrix multiplications, we use tensor contraction as a primitive. This approach not only enables massively parallel operations but also incorporates pipelining over the time axis, similar to vector processors. We have designed large coarse-grained processing elements (PEs) which can be

*This paper is part of the Industry Track of ISCA 2024's program.

split into smaller compute units called *slices*, as illustrated in Fig. 3, allowing for more flexible configurations for diverse tensor shapes. Depending on the setup of the fetch network connecting the slices, the entire set of slices can function as one large processing element or individual slices can operate as small, independent, and parallel compute units.

For instance, in the case of the attention layer of transformer models [21], a PE's slices can be configured to operate in parallel for each head, e.g., 16 slices per head (see Fig. 10). The data, continuously fetched in a pipelined manner through the fetch network, allows the operation units to be utilized at high throughput. This enables us to adopt various data reuse strategies that efficiently utilize the limited input/weight/output buffers of slices as demonstrated in the case study of the LLaMA-2 7B model execution (Section VI).

Since the operation units perform computations deterministically (as defined by the software), TCP achieves predictable performance. This enables us to develop accurate cost models for performance and energy consumption. Our compiler leverages these models when exploring possible configurations of lowered tensor shapes and their contraction orders.

In the remainder of the paper, we first explain our low-level representations of tensors and operations in Section II. Then, we describe the chip-level architecture and micro-architecture in Sections III and IV. We explain our programming interface and software stack in Section V. Finally, we show how a popular machine learning model such as LLaMA-2 7B can be executed on our chip, discuss the performance results in Section VI, and share our lessons learned during the development of the chip in Section VII.

II. Preliminaries

There are several ways to describe tensor contractions, such as PyTorch [16] or Tensorflow [1]'s matmul function or simply using 'for' loops. In this paper, we use the einsum [4] notation. Supported by frameworks like NumPy and PyTorch, it originates from the Einstein notation used to describe tensor equations. It represents each dimension of a tensor with indices, and tensor contractions are indicated by the common indices in the two input tensors. For example, consider the feed-forward layer of a Transformer model [21]. Assume that the dimensions of the input tensor (`in0`) for the feed-forward layer are $b \times l \times e$, and for the weight (`in1`) are $e \times f$, where b stands for the batch size, l the sequence length, e the embedding size of input, and f the embedding size of feed-forward network. A contraction over the e axis is represented as $ble,ef \rightarrow blf$.

The einsum notation is *declarative*. Thus, it does not designate how tensors are physically stored in memory or how the computation is performed. We extend this notation to additionally represent the key aspects of the TCP architecture. We call our newly augmented notation *low-level einsum notation*. It features the notions of lowered shape and tactic as explained below.

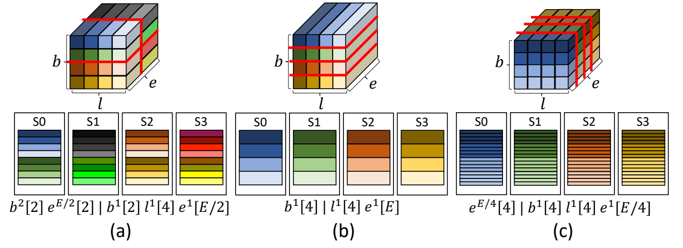


Fig. 1: Examples of lowered shapes of a tensor $(b \times l \times e)$.

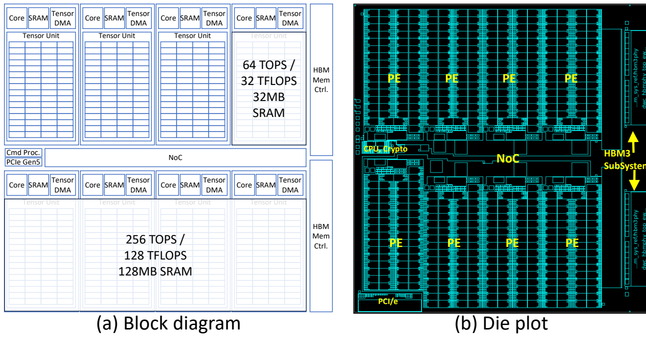
A. Lowered Shape

For describing the complete lowered shape of a given tensor, we use the notation "inter-slice partitioning shape | intra-slice shape". In this section, we use the term *slice* to represent a basic compute unit which contains memory for inputs and outputs and a datapath for computation. In Section III, we will give a more concrete description of a slice in our proposed architecture. Fig. 1 exemplifies three different partitions of a tensor with dimensions $b \times l \times e$, where $b = 4, l = 4, e = E$, into four slices. In Fig. 1 (a), we partition the tensor along the b and e axes. The partitioning along the b axis has a step size or stride of two, thus it is denoted as b^2. The partitioning is done across two slices, thus denoted as $[2]$. The same applies to the e axis. Thus, the e axis has a stride of $E/2$ over two slices thereby having $e^{E/2}[2]$. Each slice stores a quarter of the tensor, and they all have the same shape of $2 \times 4 \times E/2$ and each axis has a stride of one. Thus, the intra-slice shape is $b^1[2]l^1[4]e^1[E/2]$.

B. Tactics

A tactic for contraction describes the order of computing axes, from the outermost to the innermost. As an example, assume that the lowered shape in Fig. 1 (c) is used for the input tensors in the contraction $ble,ef \rightarrow blf$. The first tensor is assumed to be lowered to $e^{E/s}[s] \mid b^1[B]\, l^1[L]\, e^1[E/s]$ and the second one to $e^{E/s}[s] \mid f^1[F]\, e^1[E/s]$, respectively, where B, L, E, F stand for the tensor dimensions, and s the number of slices ($s = 4$ in the figure). We assume the output tensor is lowered to $f^{F/s}[s] \mid b^1[B]\, l^1[L]\, f^1[F/s]$. In this case, one possible tactic is $e^{E/s}[s] \mid b^1[B]\, l^1[L]\, f^1[F]\, e^1[E/s]$. The first term of the tactic, $e^{E/s}[s]$ represents that the outermost loop is on the e axis with a stride of E/s across s slices ($s = 4$ in this case), which means we perform s outermost sub-loops in parallel.

The second group of terms on the tactic (called *slice tactic*), $b^1[B]\, l^1[L]\, f^1[F]\, e^1[E/s]$, describes how each slice processes the remaining loops. In this case, there are four loops on b, l, f, and e (innermost) axes. The last term, $e^1[E/s]$, which is the innermost loop, represents the contraction. Thus, in the innermost loop, each slice performs the contraction, i.e., dot product between two vectors of E/s dimension along the e axis. The second innermost term, $f^1[F]$ is for the f axis, and it means the fetched data, before the f axis loop, can be reused F times, by using the internal input buffer of the slice (see Section IV). The f axis loop is repeated $B \times L$ times on the two outermost loops.



(a) Block diagram (b) Die plot

Fig. 2: Block diagram and die plot of the SoC.

Given a pair of input tensors and a tensor operation, there can be multiple choices of tactics depending on the lowered shapes of input and output tensors and the availability of compute units (e.g., the number of available slices). When the compute units are allocated, the compiler needs to explore the candidate lowered shapes to select the tactic satisfying the given requirement of performance and power consumption. If the lowered shapes between neighboring operations (i.e., the lowered shape of the output of a layer and that of the input of the subsequent layer) do not match, the compiler adds a bridge operator to transform the lowered shape of tensor.

TCP accelerates low-level einsum operations as a primitive, and also supports dynamic tensor shapes through the control registers (to be explained in the following sections) that describe the tensor's shape. In the current example, the b and l axes can be dynamically determined, e.g., depending on the number of new requests and the token generation length of each request, which are dynamically determined when executing large language model (LLM). When a tensor exceeds the memory capacity (of PE or chip), it must be split and the tactic needs to be determined according to the split.

The low-level einsum notation is utilized in our compiler to facilitate the design space exploration of contraction and expose low-level API to the designer (Section V). We will use the low-level einsum notation to describe the execution strategy for LLaMA-2 7B model on the TCP chip in Section VI.

III. System-on-Chip (SoC)

In this section, we describe the overall architecture of the SoC and the PE. TCP consists of eight PEs, as shown in Fig. 2. Each PE can function as an independent device from the host, similar to the multi-instance capabilities of some GPUs [14]. Additionally, up to four PEs can be fused to form a single, larger PE. The SoC also supports SR-IOV, allowing it to be used by multiple virtual machines with separate address spaces. This design allows TCP to provide multi-tenancy in cloud servers, enabling high utilization by multiple users.

Table I shows the chip characteristics. Since inference requires low latency and low power with air cooling [8], TCP was designed from the outset with a TDP of 150 W, enabling dense and scalable use at maximum performance in most existing data centers.

Technology	TSMC 5nm
Frequency	1 GHz
Dimensions	24.59 $\times$ 25.71 mm (632.1 mm^2)
TDP	150 W
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)

TABLE I: Characteristics of TCP.

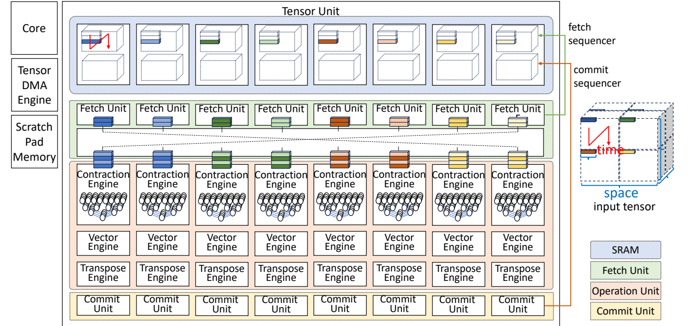


Fig. 3: A processing element of TCP with eight slices.

Each TCP chip is equipped with two stacks of HBM3 [15], the highest bandwidth memory available at the time. The two stacks of HBM3 provide up to 1.5 TB/s of memory bandwidth, and the eight PEs can fully utilize the memory bandwidth, with each PE having a maximum transfer rate of 256 GB/s. Additionally, the NoC supports data transfers between PEs without contention. A PE can transfer data to HBM or another PE on the same chip at up to 256 GB/s.

Large language models (LLMs) exceeding the memory size of one TCP require processing by multiple chips, necessitating inter-chip communication between PEs. TCP is connected via PCIe Gen5 and supports peer-to-peer (or P2P) communication, allowing PEs across multiple chips to transfer data at up to 64 GB/s in each direction.

Each PE has an independent address space and operates independently. Access to unauthorized addresses is restricted by the address translation unit. Address translation abstracts the PE's address space, allowing dynamic allocation of memory space and PE usage at run time. This abstraction forms the basis for inter-PE communication and, combined with P2P capabilities, enables PEs across multiple chips to exchange data using the same abstraction. This design allows for scalable compilation and dynamic resource scheduling.

A. Processing Element

Fig. 3 illustrates the PE, which consists of a CPU core, a tensor unit (TU) for executing large-scale tensor operations, and a tensor DMA engine (TDMA) for transfers of tensors. The TU is a substantial entity, equipped with 32 MB of SRAM and is capable of 64 TOPS. It accelerates tensor operations by continuously fetching the input tensor from SRAM, processing the operations, and committing the results back to SRAM.

The CPU core handles scalar processing, control flow, simple vector processing, and controls the TU (as a co-processor

via a command queue) and the TDMA engine. It includes 64 KB of L1 I/D caches and 256 KB of L2 cache. In order to provide more predictable performance for the accelerator and reduce penalties from cache misses, the CPU core is equipped with a 3.5 MB scratch pad memory. Having a large scratchpad memory space allows all code to be run from scratchpad memory, simplifying the programming model.

Memory transfers are the bottleneck of most of neural networks, especially LLMs. In TCP, they are done asynchronously through the TDMA engine and can be overlapped with TU operations. The compiler schedules TU operations and data transfers in order to maximize the overlap of memory accesses and computation. In order to realize such a compilation result, the CPU core transmits the next command (for a DMA or a new TU operation) to the command queue while the previous tensor operation is underway on the TU, and then the CPU core continues to perform other tasks in parallel. The TU takes a new command in the queue and launches its execution when the required resource, whether memory or compute unit, is available.

The TDMA can index and transfer tensors in any dimension order. This capability allows for optimal lowering of data stored in HBM to multiple slices by the compiler, supporting tensor manipulations like reshape and transpose. It also supports scatter-gather which is useful in improving effective memory bandwidth for memory access patterns like embedding lookups.

B. Tensor Unit

Rather than incorporating a single large SRAM and a massive matrix multiplication unit, in our TU architecture we divide the compute pipeline, including the SRAM and operation unit, into multiple *slices*. Therefore, each slice comprises of SRAM, fetch, operation, and commit units. Each slice, configured via control registers, can independently process (sub-)tensors. Additionally, a fetch network connecting these slices (to be exact, the fetch units on slices) allows us to multicast tensors across multiple slices.

Data fetched from SRAM of each slice is sequentially delivered, via the fetch network, to the operation units, each of which consists of contraction (CE), vector (VE), and transpose (TE) engines. These engines can be chained by the compiler as in a vector processor. The CE includes H ($= 8$ in the chip design) instances of dot-product engines (DPE), which perform element-wise multiplications and can produce a variable number of outputs with reduction trees with W inputs. The depth of each reduction tree is configurable, and the output can be temporally accumulated by an accumulation unit. Each DPE receives two input vectors, one from the CE's register file and the other sequentially delivered vector from the fetch unit.

The architecture supports several data reuse configurations through software, including weight stationary using the register file, input stationary using the input buffer within the CE, and output stationary using accumulator registers [18]. As data is always streamed and sequentially delivered in a

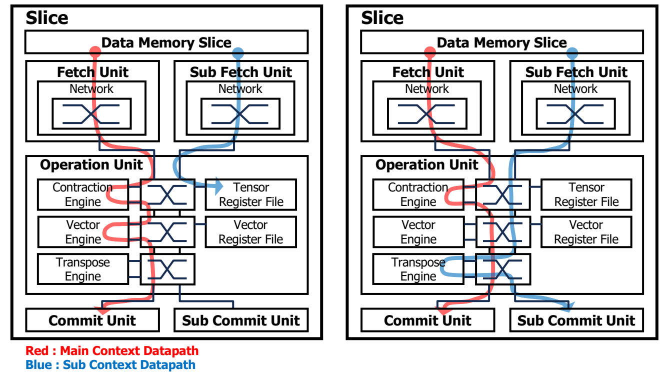


Fig. 4: Two example slices and their contexts.

multi-dimensional form, the operation unit's sequencer logic orchestrates input data feeding and register/accumulator indexing according to the control registers initialized before tensor operations.

While each slice can independently utilize $H \times W$ MACs for contraction, in scenarios where N slices multicast data from N memory slices through the fetch network, all operation units in the slices receive data fetched from N slices. This approach is akin to reading data from N banks of a large SRAM, reusing fetched data in an $H \times N$ format for propagation, and performing dot products in a W format, similar to operations in a large systolic array. We describe the micro-architecture in more detail in Section IV.

The TU includes 64+1 slices, with one reserved as spare to improve chip yields. To control and monitor the slices, the TU includes a TU controller (TUC), which accesses control/status registers on the TU. Control registers of each slice are mapped in the TU's address space. The TUC contains a command queue to which the CPU core writes tensor commands. To configure a tensor operation, the CPU core can either write, via the ARM core's low-latency peripheral port (LLPP), to these registers directly or send tensor commands via the command queue so that the TUC sets the registers on its behalf.

The TU is responsible for performing tensor, vector, and memory operations. While a tensor operation is running, vector and memory operations can be executed in parallel, effectively hiding their execution time within that of tensor operations, which helps improve the utilization of the TU. In order to realize such parallel operations, each slice has two execution contexts (Fig. 4), each being described on a set of control registers. The main context is used for tensor operations and the sub-context for everything else. Contexts are launched asynchronously so as to allow them to run in parallel. For instance, after launching a tensor operation in the main context, several memory operations can be launched sequentially in the sub-context.

The TCP architecture is not optimized for fine-grained sparsity due to its area and power overhead and, instead, focuses on dense models. Especially, in the case of LLMs, low precision has proven more effective than pruning. Thus, our choice of not supporting fine-grained sparsity may not hamper the advantage of TCP on LLMs.

IV. Micro-Architecture

The TU slice consists of a data memory slice and a compute pipeline. The compute pipeline is structured in three stages to perform operations over tensors:

- Fetch unit (FU), which reads tensors from the data memory slice sequentially and feeds it to the operation unit in the required shape.
- Operation unit (OU), which streams and processes data fed from the fetch unit, comprising the contraction engine (CE), vector engine (VE), transpose engine (TE), etc.
- Commit unit (CU), which stores the results from the operation unit back into data memory slice in the specified tensor shape.

A. Data Memory

The data memory (DM) slice functions as a scratch pad and stores tensors. Tensors are partitioned across DM slices according to the lowered shapes as exemplified in Fig. 1. Each DM slice comprises 16 banks, each with an 8B width, providing a maximum SRAM bandwidth of 128GB/s per slice. The DM slice is primarily used for fetching input tensors for operations and committing output tensors after operations. It supports concurrent data transfers, e.g., data movements for the sub-context while handling tensor traffics from or to HBM for the main context in parallel. A DM slice can serve multiple requests by utilizing the bank level parallelism of 16 banks.

The DM slice supports virtual addressing through a page table, converting virtual to physical addresses. This feature allows for dynamic allocation of contiguous large memory spaces, facilitating tensor management by the compiler.

B. Fetch Unit

The fetch unit (FU) accesses tensors from the DM slice sequentially and delivers them to the operation unit. To support concurrent execution of the main and sub-contexts, each slice contains two FUs: the main fetch unit and the sub-fetch unit. Each FU consists of a fetch sequencer for generating address sequences to access the DM slice, a fetch process unit for data preparation like type conversion and padding. The fetch unit is connected to a fetch network for transmitting processed data to operation units on different slices as well as its own slice.

The fetch sequencer generates addresses according to the given tensor shape and specified order, i.e., tactics (described in Section II). The fetch sequencer and process unit not only generate basic N-dimensional loop-style addresses but also support indirect addresses and table lookups for gathers, playing a key role in ensuring flexibility for all types of tensor operations needed by ML models.

Fetched data can be processed in parallel by multiple operation units. The fetch network (Fig. 5) works like a circuit-switched network during tensor operations, maintaining a fixed topology as determined by software based on the lowered shapes and tactics. It ensures data order between multiple sources and within packets from a single source and supports multicast. There is no network congestion since the network is circuit-switched with strictly ordered arbitration between sources and destinations, as determined by the compiler. The circuit-switched network allows for high throughput and reduced hardware complexity in routing, ordering, multicast, and flow control. The fetch network is composed of two independent networks for each of the main and sub-contexts.

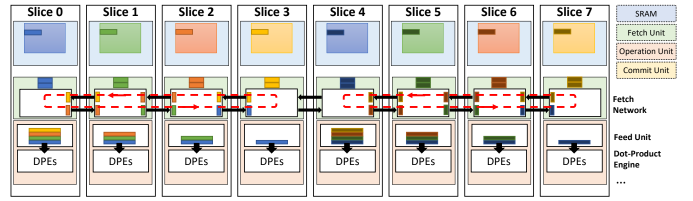


Fig. 5: Fetch network.

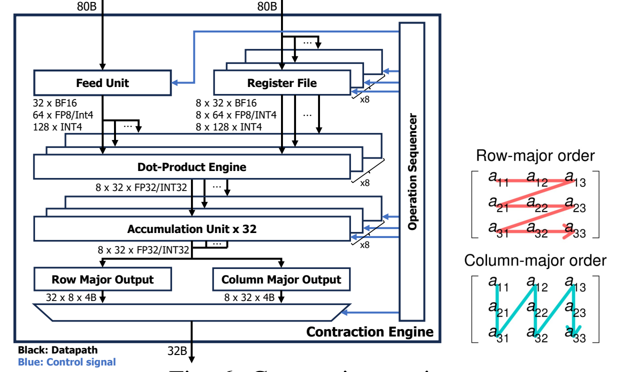


Fig. 6: Contraction engine.

C. Contraction Engine

The contraction engine (CE) performs dot product operations between two tensors. As Fig. 6 shows, the datapath of the CE consists of a feed unit, register files (RFs), dot-product engines (DPEs), and accumulation units. They are controlled by the operation sequencer.

The CE contains eight DPEs each of which performs dot products by spatially summing element-wise multiplication results of two input vectors through a reduction tree. The DPE takes two input vectors from the feed unit and the RF. Each input can hold 32 BF16 values, 64 FP8 values, 64 INT8 values, or 128 INT4 values. The number of input elements also represents the maximum throughput of specified input type.

The eight DPEs share the input from the feed unit (i.e., for data reuse by broadcast), but receive separate inputs from the RFs. The depth of the reduction tree can be configured through control registers, which in turn determines the number of outputs from the DPE. Additionally, the DPE supports max-reduction besides add-reduction to support different types of computations such as max-pooling.

The feed unit processes the packets delivered through the fetch network and, as mentioned above, broadcasts the data to multiple DPEs thereby reusing the fetched data. Controlled by the operation sequencer, the feed unit enables further data reuse by providing the same data multiple times to the eight DPEs in consecutive cycles. As will be exemplified in Section VI-B, the feed unit can provide the same query vector repeatedly to the DPE which performs dot product between the same query vector (from the feed unit) and a key vector

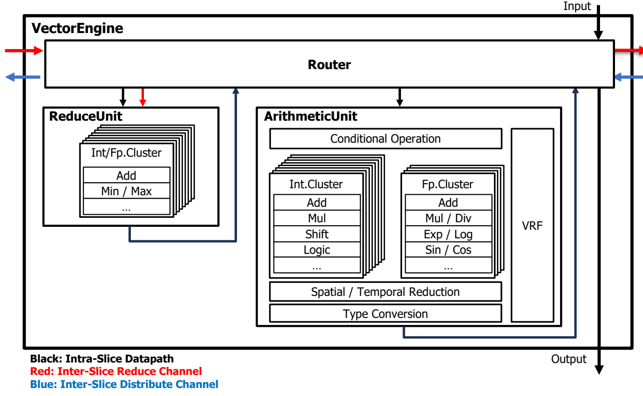


Fig. 7: Vector engine.

(newly fetched from the RFs every cycle). The feed unit can also use shifts for data reuse in operations like convolution, as well as transpose tensors.

The accumulation unit receives the output of DPEs for temporal accumulation. To temporally reuse input from the feed unit, multiple accumulators are necessary to maintain various partial sums. Since increasing the number of accumulators raises read/write costs proportionally, we try to make the best use of a small number of accumulators, four per unit.

The accumulation unit contains a total of 1024 accumulators, which can be indexed differently depending on the output size of the DPE. For example, if 32 BF16 inputs (from the feed unit) are used, and the depth of the reduction tree is 0, each DPE will produce 32 multiplication results. Thus, the total number of outputs on eight DPEs is 256 ($= 8 \times 32$). Considering that 1024 accumulators are available, each output of DPEs can be associated with up to four accumulators for temporal accumulations. However, if the depth of the reduction tree is three, each DPE will produce four outputs (i.e., partial sums) and the eight DPEs give 32 outputs. In this case, each output of the DPEs can be associated with 32 ($= 1024/32$) accumulators for temporal accumulations.

The RFs, which are composed of single port SRAMs (total 80 KB per slice) and a small-sized cache, serves as storage for operands reused in MAC operations. They provide data to each DPE, requiring as many banks as the number of DPEs. The RF handles simultaneous read and write accesses for cases where the next weights have to be loaded by the sub-context while a tensor contraction is running.

D. Vector Engine

Fig. 7 shows the vector engine (VE) which handles non-linear functions, element-wise operations, reductions, and type conversions. It comprises multiple functional units in clusters, which the compiler can pipeline by forming chains. The functional units realize various arithmetic and logical operations on INT32/FP32, transcendental functions like exp or sin, predicated operations, and all kinds of type conversions and quantizations for INT4/8/16/32, FP8, FP16/BF16, and FP32. They also support both intra-slice and inter-slice reductions, routing, or multicasting reduction results to designated slices. Each slice's VE has a throughput of 8-way INT32 and 4-way FP32. While the CE is in charge of lower precision dot products (INT4/8, BF16, and FP8) for area and power efficiency, the VE supports INT32/FP32 dot products as well as other arithmetic operations for flexibility. These facilitate rapid processing of key LLM operations like softmax and layer-norm. For example, the VE handles element-wise exp in hardware, and computes the sum of exponentials using reduction for softmax.

The VE can be chained on-the-fly to the results of the CE. For example, the expand operation of the feed forward layer of LLaMA-2 can be fused with activation functions such as SiLU for enhanced performance and energy efficiency. Moreover, even without contraction, the VE can perform various vector operations alongside the fetch and commit units, functioning like an N-dimensional vector processor. Using multiple contexts, when the main context performs contraction without the VE, another context can utilize the VE in parallel, hiding subsequent element-wise operations following contraction.

E. Transpose Engine

The transpose engine (TE) transposes the last axis of a tensor with other axes within a slice by pushing data for a specified set of rows and then popping them for another set of rows. Note that the fetch/commit sequencers can also be used to perform transposition between the last axis and other axes.

Together with the VE, the TE and the commit unit handle tensor manipulations across multiple slices or within a single slice. Tensor manipulation includes transpose, split, slice, concat, reshape, etc, both explicitly defined by the model and for storing tensors in layouts optimized for subsequent operations as determined by the compiler. Like the VE, these manipulations can be processed on-the-fly after or in parallel with dot product operations.

F. Commit Unit

The commit unit comprises a commit sequencer and a commit process unit. The commit sequencer designates addresses for sequentially transmitted commit data, allowing for manipulation of the storage layout of tensors. Using addresses generated by the commit sequencer, the storage bandwidth consumption can be adjusted (by selecting one of 8B, 16B, 32B/cycle), if necessary, using the commit size parameter. Additionally, the commit process unit supports simple type conversions and can remove padding from data for compaction during storage.

V. Programming Interface and Software Stack

A. Programming Interface

The CPU core controls the TU as a coprocessor through registers. Since there is latency in accessing the coprocessor from the core, the TU was designed to operate asynchronously.

The TU has a large set of control registers that describe the shape of the tensor and tactics. Each slice has its own control register, allowing for individual control, but typically all the slices in a TU are controlled with the same configuration.

The control registers of all the slices can be set at once via broadcast. As such, the contraction operation is exposed directly, including chained processing, through the control register. There is also an enable bit to trigger the execution once all control registers are configured.

The TU receives register read/write inputs from the CPU core, controls the slices, and has a controller to manage its status. We include a simple command processor in the TU controller, allowing tensor operations and movements to be performed asynchronously, similar to a typical DMA Engine, rather than the CPU core directly setting numerous control registers. The CPU core pushes commands, e.g., tensor operations or tensor DMA, into the command queue (of 64 entries), and the command processor executes the commands in the queue sequentially. The most critical role of the CPU core is to continuously utilize the TU by ensuring that the command queue does not become empty. Since the command processor executes commands sequentially, the CPU core is in charge of dynamic control flow.

Some of the TU commands are as follows:

- `dma(addr, id)` - transfer a tensor from/to HBM using DMA based on the `id` that describes source/destination tensor information including shapes
- `load(addr, csr addr, size)` - load from memory to control registers
- `exec(id)` - execute a tensor contraction operation
- $\texttt{wait}_{\texttt{d/e}}$`(id)` - wait for a DMA or exec command

Most commands, including `dma` and `exec`, run asynchronously, which allows the overlap of data transfers and computations. When synchronization is needed, `wait` is called. A typical sequence of commands is `dma(., 0)-load()-`$\texttt{wait}_{\texttt{d}}$`(0)-exec(0)-dma(., 1)-load()-`$\texttt{wait}_{\texttt{e}}$`(0)-`$\texttt{wait}_{\texttt{d}}$`(1)-exec(1)-...`, which first loads tensors with DMA in context 0 and configures the control registers (`load`). After DMA completes, context 0 is executed while the data for the next iteration is loaded in parallel (`dma`). The CPU core can use polling or an interrupt to check the status of the cmd queue.

PEs communicate with each other and with the host through message passing. Each PE exposes an IPC memory area and sets head/tail registers as doorbell registers. We use address translation, which not only protects accesses to different process address spaces, but it also allows multiple PEs to share an address space for communication. PEs on different chips are also mapped to the PE's address space, which simplifies the programming model when using multiple chips.

In multi-node execution, we use communication over TCP with PCIe P2P. PEs can communicate with other PEs using tensor DMA (data) and IPC messages. Accessing a PE within a TCP is identical to accessing a PE in another TCP or memory from the viewpoint of the initiator. If the target address is mapped to a different TCP, the transaction initiated by the PE is transmitted to another TCP via the NoC and PCIe. We enable the synchronization of tensors and the transfer of IPC messages after the execution of a layer, allowing multiple PEs to execute tensor parallel operations.

B. *Various Tactics*

As will be explained in Section V-C, the compiler explores possible choices of tactics and tries to select the best one satisfying the given requirements of performance and power consumption. In order to help understanding the tactic design space, in this section, we provide examples of possible tactics on a given tensor operation.

Fig. 8 shows three examples of lowered shapes and tactics for a tensor unit with s slices ($s = 4$ in this case). On the left, the contraction occurs in $e^{E/2}[2] \mid e^1[E/2]$ across two slices and parallel contraction occurs along the $b^{B/2}[2]$ axis. The case in the middle has a tail axis that is not e, resulting in a DPE depth of 0. Each element of e read from the RFs is broadcast across W, multiplied, and temporally accumulated in the accumulation unit. The partition axis of each slice is either b or l, processed in parallel, and the second operand is multicast to all slices and loaded into the RFs. TCP supports two contexts to parallelize data movement for the next layer while performing tensor operations. It can divide the RFs to use part of it for tensor operations while simultaneously loading tensors for the next layer in another part. On the right, f is partitioned across two slices and loaded into the RFs. The data is loaded once and it is multicast to two slices, allowing parallel execution.

When the compiler explores various tactics, the space also includes tactics where some dimensions of tensor are unaligned. Specifically, within slices, if the last dimension does not align with the width of the dot-product engine, utilization can decrease accordingly. However, the alignment of other axes accessing the tensor along the time axis does not affect utilization. For partitioning axes, we generally divide axes by powers of two for the convenience of setting up the slices. If a specific dimension needs to be divided among slices and it is not a power of two, performance can suffer. In this case, the compiler can choose lowered shapes which assign these axes to in-slice non-last dimension based on the performance estimator.

C. *End-to-End Compiler*

The TCP compiler works by taking the entire model graph as input. The model is transformed into the compiler's intermediate representation (IR), optimized, and then lowered into assembly code. The process consists of the following five stages:

1) Convert the input model into primitive operators.
2) Cluster primitive operators to form kernels that can be executed efficiently on TUs.
3) Select tactics according to a cost function, and convert kernels into low-level operators.
4) Transform low-level operators into command lists that correspond 1:1 to hardware functions.
5) Convert the command lists into an executable binary.

We describe each of the stages in details as follows.

1) Primitive Operator Conversion: ML frameworks like PyTorch have a huge set of operations [5]. However, many

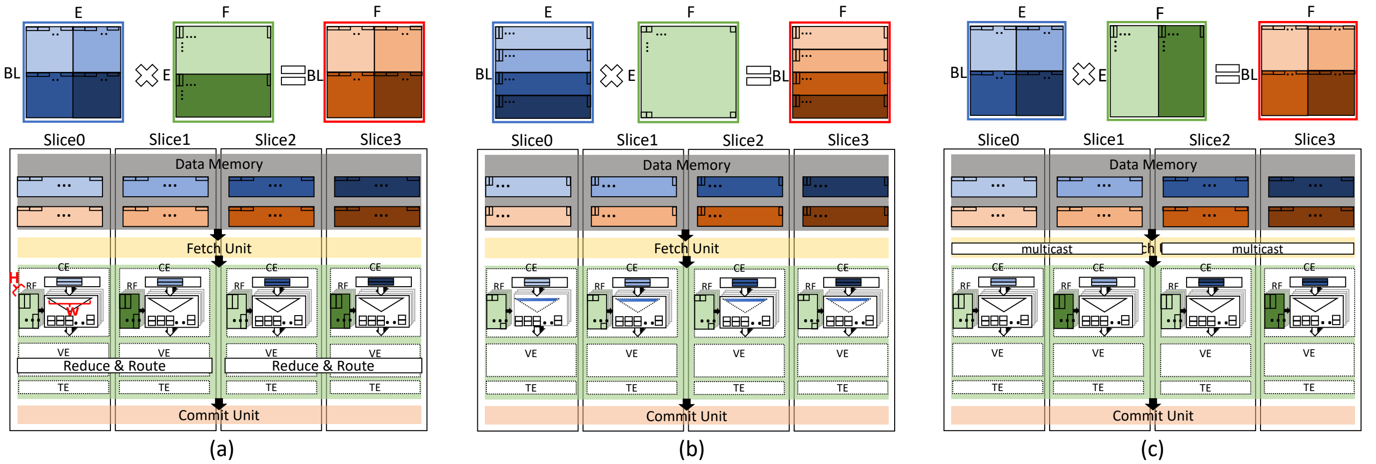


Fig. 8: Three examples of lowered shapes and tactics for the contraction `ble,ef`.

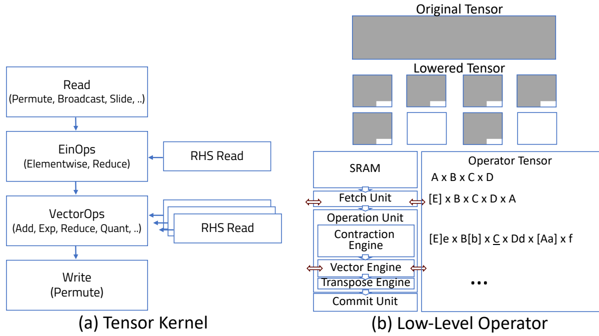


Fig. 9: Low-level compiler IRs.

of these can be decomposed into a small set of primitive operators. Examples of primitive operators include element-wise arithmetic operations, reduction along a specific axis of a tensor, linear algebra operations, reshape/indexing/slice operations, type conversion, and comparison operations. The first stage decomposes the input operators into these primitive operators.

2) Tensor Kernel Generation: The graph of primitive operators can be too large and fine-grained for optimization. Therefore, as exemplified in Fig. 9, we cluster primitive operators to form kernels, which are units of operation structured in a sequence of read, contraction, vector operations, and write. The clustering process aims to maximize data reuse by fusing operators and eliminating redundant tensor usage.

3) Low-Level Operator Generation: Although kernels are executable, not all of them utilize the hardware efficiently. Compiling kernels into low-level operators involves considering multiple possible options of kernels and potential bridge operators (due to tensor shape differences between adjacent layers) and selecting the most cost-effective option. The cost is determined by considering the input/output shapes of the kernels and bridges, the lowered axis shapes, and tactics. The final low-level operators take lowered input/output tensors and have fixed operation sequences, parallelism, and data reuse tactics.

4) Command Generation: A command represents a single tensor unit execution and describes the operations of its various sub-units (fetch unit, contraction engine, vector engine, transpose engine, etc). Although low-level operators mostly correspond to hardware instructions, they are further divided into several commands considering buffer and RF sizes.

5) Binary Creation through Scheduling & Resource Allocation: Finally, we generate an execution plan for the commands, considering resource limits. We need to decide when each command executes, when inputs are prepared, where outputs are stored, and how to operate multiple commands simultaneously within resource limits. The main strategy is to reduce memory pressure, schedule commands to hide memory overhead, and run memory operations concurrently with computations. Our scheduling algorithm uses a mix of heuristics, ILP (integer linear programming), and genetic algorithms. Resource allocation involves mapping tensors to SRAM (i.e., DM slices) and the RFs. After scheduling and resource allocation, the best plan is translated into an executable.

D. Low-Level API

Although most users only interact with AI accelerators through frameworks like PyTorch and TensorFlow, there are use cases that require a low-level API. For example, the user may try running a completely new AI operation for which the compiler may not produce good-enough code. Compiler and runtime researchers also require low-level access to the hardware [19]. We observe that in the case of GPUs, academic researchers have regularly beaten the vendor's software, which then incorporates those novelties [6], [11], [22]. Another important use case is large-scale deployments, where it becomes financially viable to optimize models by hand. For these use cases, we offer an API similar to the low-level einsum notation used in this paper.

VI. Case Study: Running The LLaMA-2 7B Model

In this section, we describe how the LLaMA-2 7B model [20], a representative language model, can be executed on the TCP chip. We explain how TCP achieves high utilization even with small batch sizes, and how energy efficiency is improved through data reuse. We also show how batch size and sequence length impact the choice of the optimal tactic and how computation and DMA transfers overlap efficiently.

To achieve high performance, it is essential to efficiently utilize all PEs as well as HBM bandwidth. In this paper, we focus on parallelism within a single chip, showing how to run LLaMA-2 across multiple PEs.

We follow the notation of [17], [24] to express parallelism. Subscripts are used to describe axes that are divided across multiple PEs. For example, let B be the batch size, L the sequence length, and E the embedding size, BLE_n indicates that the E axis is split into n partitions and that each PE stores a tensor partition with dimensions $(B \times L \times E/n)$.

Distributing weights equally among n PEs using 1D weight stationary is the most straightforward and efficient distribution method for most cases. We use 8-bit weights and activations. We describe next how we partition and run LLaMA-2 and report its performance and power consumption on TCP.

A. Embedding Layer

For the embedding lookup before the first layer, a gather is performed with BL token indices to load a BLE tensor. With n PEs, dividing BL into B_nL or BL_n does not make a significant difference in terms of data movement. After passing through RMS normalization and before computing Q, K, V using 1D weight stationary, all PEs must replicate BLE using an all-gather. Thus, how it is divided and processed before that is irrelevant.

Each PE drives a tensor DMA to perform a gather from the embedding table located in HBM. The DMA unit allows indirect access to specific dimensions of the tensor in HBM, enabling full bandwidth embedding lookups and placing the data on memory in an optimal layout for subsequent layer operations. For example, the data can be stored as $E^{e/s}[s] \mid L[l]E[e/s]$ or $L^{l/s}[s] \mid L[l/s]E[e]$.

During the decode phase, since the previous output token becomes the next input token, the embedding lookup is performed only for the last generated token, and thus the table lookup is done for B tokens only. Instead of dividing B across multiple PEs, each PE performs the same embedding lookup and proceeds to the attention layer, which is more efficient than an all-gather in this case.

B. Attention Layer

In order to explain how the attention layer can be executed on the TU, we use a specific example. We assume $H = 4$ (number of heads), $L = 256$ (sequence length), and $D = 128$ (per-head hidden dimension). In this example, we allocate 16 slices to each head. Fig. 10 illustrates how the attention layer can be executed on multiple slices in parallel. Fig. 10 (a) depicts the multiplication of query (`in0` = Q[0:256][0:128]) and key (`in1` = K[0:256][0:128]) matrices for a head. Fig. 10 (b) illustrates how matrix Q is stored across 16 slices and Fig. 10 (c) shows the complete picture for four heads. Next, we give a detailed description of how the attention layer is executed, focusing on lowered shapes and tactics.

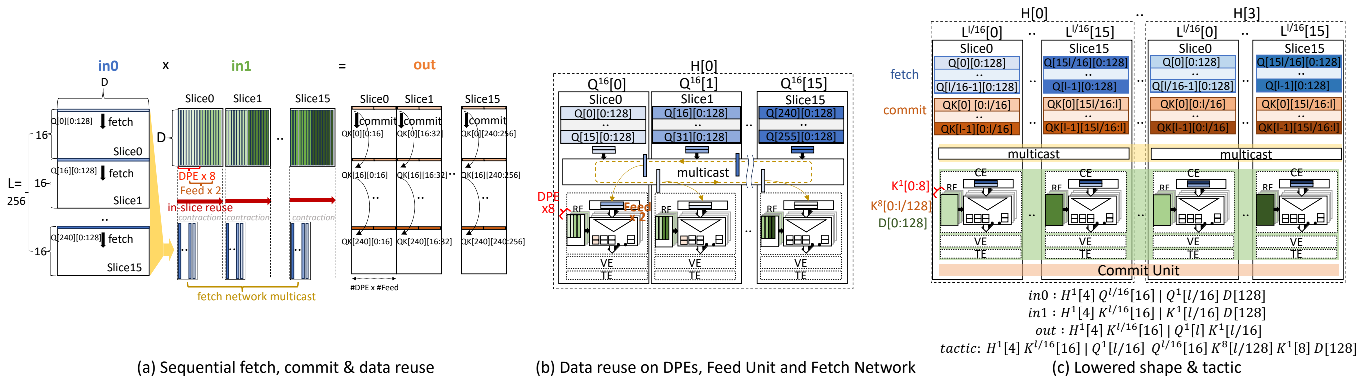
Fig. 10 (a) shows the Q matrix (`in0`), Q[0:256][0:128]. As Fig. 10 (a) and (b) show, we allocate a sub-matrix on a slice. Specifically, the sub-matrix Q[0:16][0:128] is allocated on the DM of slice 0, the next sub-matrix Q[16:32][0:128] on slice 1, and so on. We allocate the K matrix in a similar manner across slices. As Fig. 10 (a) shows, we allocate K[0:128][0:16] on the RFs of slice 0, K[0:128][16:32] on slice 1, and so on. The 16 key vectors on each slice are distributed across the eight RFs of the slice.

After allocating the Q and K matrices across 16 slices, we multiply them on the slices. As shown in Fig. 10 (a), we fetch the first query vector, Q[0][0:128] from the DM of slice 0. The input vector fetched from the DM slice can go through the feed unit and be broadcast to eight DPEs. The query vector is first broadcast to the DPEs of slice 0 to produce eight dot products, i.e., the first eight elements of output matrix QK (`out`), i.e., QK[0][0:8]. The same query vector can be provided to the same DPEs in the subsequent cycle while fetching new key vectors from the RF. Thus, in the next cycle, we obtain the next eight elements of output matrix QK, i.e., QK[0][8:16] on slice 0. Fig. 10 (a) illustrates slice 0 produces a sub-row of output matrix, QK[0][0:16]. In the same manner, each slice produces its own sub-row of the output matrix in parallel.

The query vector fetched from a DM slice needs to be maximally reused. To do that, we multicast the query vector over all the slices. Fig. 10 (b) illustrates how the multicast is performed by the fetch network. A query vector fetched from a DM slice is provided to a slice and then, after it is used 16 times on the current slice, the fetched vector moves, over the fetch network, to the next slice which then uses it 16 times by performing dot products with its key vectors. In this example, once a query vector is fetched from its DM slice, it is reused 256 times (= 16 times, by 16 slices and 16 times on each slice by broadcast to eight DPEs and reuse by the feed unit). As shown in the case of LLaMa-2 model execution, TCP can offer high efficiency on computation and HBM bandwidth consumption by trying to maximally reuse data fetched from HBM.

Fig. 10 (c) shows the low-level einsum representation of this case for the lowered shapes of the Q, K, QK matrices and the tactic where D represents per-head hidden dimension. Note that the output QK matrix is also distributed across slices. Thus, in order to compute the softmax on the QK matrix, each slice first obtains the local max (of QK[0][0:16] on slice 0 for instance) within its slice, and then a max-reduce is performed between 16 slices to obtain the global max. This global max is then broadcast to each slice's RF. On the VE of each slice, we calculate *exp(x - global.max)* for each element of the output QK matrix and accumulate the results to compute the slice sum. The final global sum is obtained through an add-reduce, and then broadcast to the slice's RF, and used to compute the final result of the attention score, which is committed to the DM of each slice.

Before calculating attention, the V tensor is loaded into the RFs. V is stored as $H^1[4]V^{l/16}[16] \mid D^1[128]V^1[l/16]$ and the softmax result as $H^1[4]K^{l/16}[16] \mid Q^1[l]K^1[l/16]$, where V represents the axis of vector index. The attention is the



(a) Sequential fetch, commit & data reuse (b) Data reuse on DPEs, Feed Unit and Fetch Network (c) Lowered shape & tactic

Fig. 10: Example tactic for calculating QK^T in the attention layer, i.e., $HQD, HKD \rightarrow HQK$.

contraction between V and K of the softmax.[1] A possible tactic is $H^1[4]\ V^{l/16}[16] \mid Q^1[l]D^8[16]D^1[8]V^1[l/16]$, which means that each DPE contracts $V^1[l/16]$, and one input data chunk is fed identically to eight DPEs 16 times, resulting in a total of 128 input reuses. The outputs of the CE are transmitted to the VE and to be reduced across slices according to the lowered shape of the output.

C. Feed Forward

The feed forward network (FFN) layer consists essentially of matrix multiplications. Its intermediate dimension, F, is 2.7× larger than that of attention output, which requires a large set of MLP weights thereby incurring a significant amount of HBM read traffic. Additionally, since F is not a power of two, the intermediate vectors must be padded, reducing hardware utilization.

We split the MLP weights into $E \times F/3$. We process each third of the operation while prefetching the next third. The activation function (SiLU) is fused and processed in the vector engine as a chained operation following the previous contraction.

D. Performance Analysis

Table II shows that TCP offers noticeable performance per watt, 2.7× better than H100 and 4.1× better than L40s.[2] In terms of throughput, each chip reaches performance close to its peak, with differences between chips mostly due to peak memory bandwidth. TCP has 1.7× higher peak memory bandwidth than L40s and is 1.76× faster, despite having a 57% lower TDP. H100 has 2.2× larger memory bandwidth than TCP and is 1.72× faster. However, H100 has a 4.7× higher TDP than TCP. TCP shows higher throughput relative to power consumption due to its higher memory bandwidth compared to TDP.

Single batch first token latency ($B = 1$ in the table) represents the compute and memory characteristics of the prefill phase. When sequence length is small ($L = 128$), it tends to be slightly more memory intensive. However, when sequence length is large ($L = 2\text{k}$), it becomes more compute intensive. As shown in Table II, compared with L40s, TCP has much lower TOPS and TDP, but it shows 41% lower latency for $L = 128$ and 11% lower for $L = 2048$, respectively. The better performance for long sequences ($L = 2\text{k}$) demonstrates that TCP achieves higher utilization due to TCP's architecture which makes better use of parallelism and data reuse available in tensor contractions.

Fig. 11 (a) shows the first token latency of the prefill phase over multiple sequence lengths. For sequence lengths smaller than 256, the first token latency is almost linearly proportional

		L40s	H100	TCP
Technology		TSMC 5nm	TSMC 4nm	TSMC 5nm
BF16/FP8 (TFLOPS)*		362/733	989/1979	256/512
INT8/INT4 (TOPS)*		733/733	1979/-	512/1024
Memory Capacity (GB)		48	80	48
Memory Bandwidth (TB/s)		0.86	3.35	1.5
Host I/F		PCIe Gen4 x16	PCIe Gen5 x16	PCIe Gen5 x16
TDP (W)		350	700	150
Latency (msec)	B=1,L=128	14	7	8
	B=1,L=2K	73	36	65
Through-put (tokens /sec)	B=16,IL=2K, OL=2K	531		935
	B=32,IL=2K, OL=2K		2230	1293
Perf/Watt (tokens /sec/W)	B=16,IL=2K, OL=2K	1.52		6.24
	B=32,IL=2K, OL=2K		3.19	8.62

TABLE II: LLaMA-2 7B latency and throughput comparison. B stands for batch size, IL (OL) stands for input (output) sequence length. (* without sparsity) [13]

[1] Note that Q, K, and V all represent the same vector index and thus should be represented as a single axis like L. In this example, we use Q, K and V separately for better readability.

[2] We referred to NVIDIA's public LLaMA results. The 7B models of LLaMA and LLaMA-2 are identical from a computational perspective. While NVIDIA's results are in FP8, TCP is in INT8 since we do not yet have an FPP8 model for LLaMA-2, and TCP provides the same performance for both FP8 and INT8, and power consumption is compared in terms of TDP. While using actual power consumption instead of TDP would be more convincing, but it was difficult to measure the exact power consumption of the GPUs we were comparing with. Therefore, for comparison, we used TDP for both sides. TDP is not exact, but we consider it as a reasonable approximation of actual power consumption.

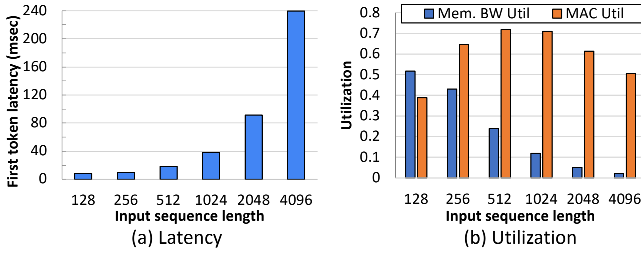


Fig. 11: Latency (a) and utilization (b) for the prefill phases with varying sequence length.

to the sequence length since the computation cost of attention, which is usually proportional to the square of the sequence length, is relatively small. With long sequence lengths ($> 2\text{k}$), the computation cost of attention starts to dominate, and the execution time increases super linearly. Even though the size of QK^T exceeds the SRAM size, TCP is able to hide the latency of tensor transfers within computations, so the total latency increases linearly with the amount of computation.

Fig. 11(b) shows the utilization of memory bandwidth and MAC, where MAC utilization is calculated by dividing the number of contractions by the latency, and does not include element-wise operations like softmax, RMSNorm, add, and mul. Memory bandwidth utilization is calculated by dividing the size of the weights and KV cache by HBM bandwidth, 1.5TB/s. In the case of Fig. 11 (b), when $L = 128$, the overall model is relatively memory bound, thus MAC utilization is low. However, as L increases, MAC utilization also increases. But when L exceeds 2k, attention scores cannot fit entirely into SRAM, and MAC utilization gets slightly decreased due to the increased memory boundedness incurred by the additional memory traffics to deliver split QK^T tensors for softmax. Note that Fig. 11 (b) shows only the memory utilization from the model itself. Thus, the increased memory traffic due to softmax is not reflected.

Fig 12 shows the next token latency and utilization of the decode phase. The decode phase is basically bottlenecked by memory bandwidth due to the read traffic of weights. Thus, as shown in Fig. 12 (a) and (b), the next token latency does not noticeably increase when the sequence length and batch size are small. However, as the sequence length or batch size gets larger than some level, the latency also starts to increase. This is because the increased size of the KV cache incurs additional HBM traffic. As L and B increase, not only does the amount of computation increase, but also the time to load the KV cache, resulting in memory boundedness. Fig. 12 (c) shows as L becomes very large, we observe memory boundedness due to softmax and KV cache which is also reported in Fig. 11 (b).

Fig. 13 shows the trace of TU and TDMA activities for an encoder block during the prefill phase. In Fig. 13 (a), we can see how the computation time can be hidden behind weight transmission when the sequence length is very short. Conversely, in Fig. 13 (b), we can see how weight transmission time can be hidden behind computation time when the sequence length is large and computation dominates.

Fig. 14 shows the performance and power consumption of the attention layer when using different tactics. Power results are based on estimations using Synopsys' SpyGlass and TSMC library. Different tactics exploit different types of data reuse depending on the order of fetch and how buffers are managed, leading to different tradeoffs of performance and power consumption. For performance critical systems, the compiler would select the tactic with the highest performance, i.e., the top-left point (Opt. 1) in the figure. However, a system under stringent peak power constraints may need lower power tactics, e.g., Opts. 2/3.

Fig. 15 shows the power breakdown of six points of Fig. 14: Opt. 1, 2, 3, which represent the most power efficient tactics in their performance category, and Ineff. 1, 2, 3, which represent the least efficient ones. The breakdown reveals that efficient tactics use less DPE power than the inefficient ones with equivalent execution time. The optimal cases commonly utilize all the eight DPEs in parallel. They differ in the usage of the accumulation unit and the data supply from the fetch unit. On the other hand, inefficient cases are characterized by limited usage of DPE parallelism, even though the data supply from the fetch unit is sufficient. Consequently, the balance between the data supply from the fetch unit and parallelism in the DPE contributes to higher performance and lower power consumption.

VII. Lessons Learned

Diversity in Data Reuse Patterns Across Operators Operators in AI models exhibit varied data reuse patterns, requiring a nuanced approach to optimize data handling. Understanding these patterns is essential to make informed decisions on data storage, access, and processing, which in turn affects the overall efficiency of the accelerator.

Optimizing SRAM Locality and Tensor Movement Optimizing SRAM locality between consecutive operators is vital. A common challenge arises when there is a mismatch in the layout of the output of a layer and the input of the subsequent layer. This situation often necessitates costly layout transformations. Overcoming these challenges requires not just local optimizations for each operator but also a global optimizer to manage efficient tensor flow throughout the entire model.

TCP's support for multiple contexts plays a vital role in hiding the overhead associated with data movement, thereby enhancing overall efficiency.

Accurate Cost Model for End-to-End Optimization A precise cost model is critical for optimizing AI programs. Such a model helps predict how different tactics affect performance and power consumption, allowing for more efficient designs.

Supporting Dynamic Shapes and Control Flow Supporting dynamic shapes and dynamic control flow is critical for the optimization of AI accelerators, especially in LLM inference where context length and batch size can vary dynamically. TCP's software supports dynamic shapes by modifying control registers on the fly, and additionally, the CPU core facilitates the creation of dynamic programs easily and efficiently, which

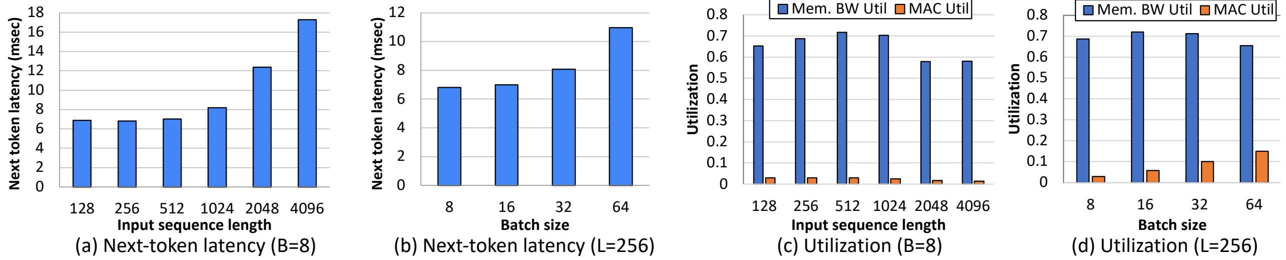


Fig. 12: Latency (a)(b) and utilization (c)(d) for the decode phases with varying sequence length and batch size.

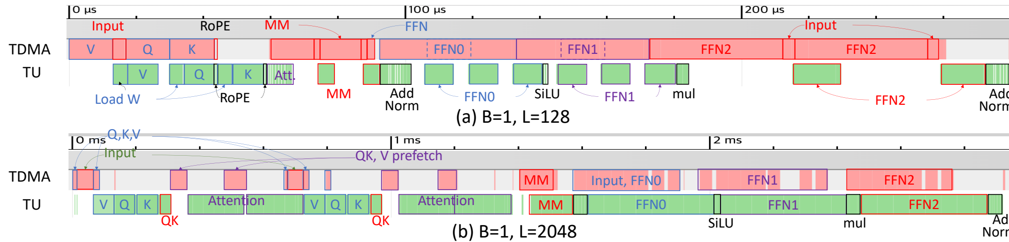


Fig. 13: Traces of TU and TDMA activities of a single decoder block with B=1 and L=128 and 2048.

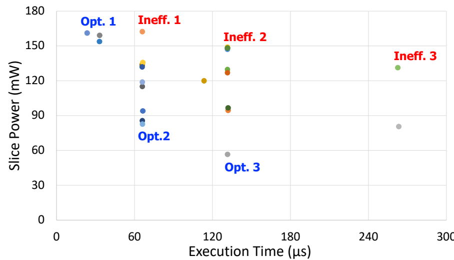


Fig. 14: Performance vs power consumption with various lowered shapes of input/output tensors and tactics.

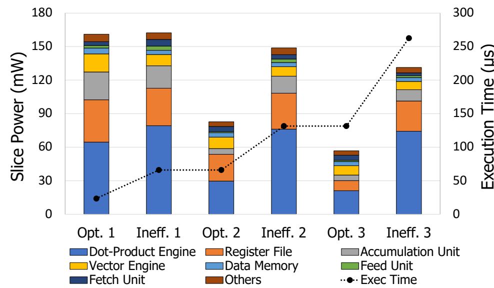


Fig. 15: Slice power breakdown of optimal and inefficient tactics.

enables more efficient program generation according to the dynamically changing requirements of computation, memory and power consumption by allowing PEs to be fused dynamically via the reconfiguration of fetch networks.

Complexities of Mapping Since contraction is possible along any axis, there are challenges faced in navigating this high-complexity space. We divide the problem into optimizing tactics within a given lowered shape and mapping lowered shapes from a perspective of whole model optimization. When exploring numerous tactics available for a given lowered shape, as demonstrated in Fig. 14, the performance estimator practically contributes to narrowing down to meaningful tactics. The design space for minimizing data movement between layers of the model involves 1) various heuristics to prune the space, and 2) several mapping optimization methods including dynamic programming. Models with repetitive blocks, such as transformers, significantly contribute to reducing the search space for global optimization.

VIII. Conclusion

We presented TCP (Tensor Contraction Processor), a novel accelerator for AI workloads. Our proposed SoC consists of eight PEs with two HBM stacks. The PE architecture is characterized by tensor contraction, flexibility, concurrency, and data reuse. Each slice performs tensor contractions as well as vector operations by concurrently running multiple contexts on contraction and vector engines. The fetch network enables data reuse across multiple slices, which renders the PE a coarse-grained processor for large-scale parallel computations. Data reuse is also exploited inside of slices via input broadcast to multiple contraction engines and input reuse on the feed unit. The compiler explores possible choices of tensor shapes and their order in TCP computation and tries to provide the best configuration for the given requirements of performance and power consumption. We demonstrated that, in the case study of running the LLaMA-2 7B model, TCP offers 2.7× and 4.1× better performance per watt than H100 and L40s, respectively.

Acknowledgements

We thank all the members of FuriosaAI who have contributed to the development of the chip as a team. Special thanks go to the Algorithm Team, who assisted with important architectural decisions and prepared the models for performance evaluation, and to the Platform Team, who supported all aspects of software and hardware development, including verification through scalable infrastructure. We also thank the anonymous ISCA reviewers for their feedback and suggestions on earlier drafts of this paper.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *OSDI*.

[2] K. Chatha, "Qualcomm® cloud AI 100: 12tops/w scalable, high performance and low latency deep learning inference accelerator," in *Hot Chips*, 2021.

[3] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[5] H. He, "Where do the 2000+ pytorch operators come from?: More than you wanted to know," 2021. [Online]. Available: https://dev-discuss.pytorch.org/t/where-do-the-2000-pytorch-operators-come-from-more-than-you-wanted-to-know/373

[6] L. Jia, Y. Liang, X. Li, L. Lu, and S. Yan, "Enabling efficient fast convolution algorithms on GPUs via megakernels," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 986–997, 2020.

[7] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles *et al.*, "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *ISCA*, 2023.

[8] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's TPUv4i," in *ISCA*, 2021.

[9] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[10] S. Knowles, "Graphcore," in *Hot Chips*, 2021.

[11] G. Li, Y. Xi, J. Ding, D. Wang, B. Liu, C. Fan, X. Mao, and Z. Zhao, "Easy and efficient transformer: Scalable inference solution for large NLP model," *arXiv preprint arXiv:2104.12470*, 2021.

[12] 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.

[13] "NVIDIA AI inference performance," https://developer.nvidia.com/deep-learning-performance-training-inference/ai-inference, NVIDIA.

[14] "NVIDIA multi-instance gpu user guide," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, NVIDIA.

[15] M.-J. Park, J. Lee, K. Cho, J. Park, J. Moon, S.-H. Lee, T.-K. Kim, S. Oh, S. Choi, Y. Choi *et al.*, "A 192-gb 12-high 896-gb/s HBM3 DRAM with a TSV auto-calibration scheme and machine-learning-based layout optimization," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 256–269, 2022.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[17] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[18] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[19] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *MAPL*, 2019.

[20] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li, "Lightseq: A high performance inference library for transformers," *arXiv preprint arXiv:2010.13887*, 2020.

[23] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[24] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni, R. Pang, N. Shazeer, S. Wang, T. Wang, Y. Wu, and Z. Chen, "GSPMD: General and scalable parallelization for ML computation graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[25] D. Zhang, S. Huda, E. Songhori, K. Prabhu, Q. Le, A. Goldie, and A. Mirhoseini, "A full-stack search technique for domain optimized deep learning accelerators," in *ASPLOS*, 2022.