A Latency Processing Unit: A Latency-Optimized and Highly Scalable Processor for Large Language Model Inference

Seungjae Moon, *HyperAccel, Seoul, 06247, South Korea*

Jung-Hoon Kim, *Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea*

Junsoo Kim, Seongmin Hong, and Junseo Cha, *HyperAccel, Seoul, 06247, South Korea*

Minsu Kim, *Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea*

Sukbin Lim, Gyubin Choi, Dongjin Seo, Jongho Kim, Hunjong Lee, Hyunjun Park, Ryeowook Ko, and Soongyu Choi, *HyperAccel, Seoul, 06247, South Korea*

Jongse Park, *Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea*

Jinwon Lee and Joo-Young Kim, *HyperAccel, Seoul, 06247, South Korea*

The explosive arrival of OpenAI's ChatGPT has fueled the globalization of large language models (LLMs), which consist of billions of pretrained parameters that embody the aspects of syntax and semantics. HyperAccel introduces a latency processing unit (LPU), a latency-optimized and highly scalable processor architecture for the acceleration of LLM inference. The LPU perfectly balances memory bandwidth and compute logic with streamlined dataflow to maximize performance and efficiency. The LPU is equipped with an expandable synchronization link that hides data synchronization latency among multiple LPUs. HyperDex complements the LPU as an intuitive software framework to run LLM applications. The LPU achieves 1.25 ms/token and 20.9 ms/token for the 1.3B and 66B models, respectively, which is 2.09× and 1.37× faster, respectively, than a GPU. The LPU, synthesized using Samsung's 4-nm process, has a total area of 0.824 mm^2 and power consumption of 284.31 mW. LPU-based servers achieve 1.33× and 1.32× energy efficiency over Nvidia's H100 and L4 servers, respectively.

The fundamental goal of artificial intelligence (AI) is to create human-like intelligence. Conventional AI has reached a level of human ability to enable data analysis, decision making, and personalization. It is now advancing at a remarkable pace, even in domains once thought to be uniquely human, such as creativity, but it was yet to replicate the creativity of humans. Generative AI (GenAI) has made a recent breakthrough with transformer models [e.g., the Generative Pre-trained Transformer (GPT) and large language model meta AI (LLaMA)] that are capable of creating original textual and image contents with high sophistication. Specifically, GenAI software platforms based on large language models (LLMs) with multibillion parameters, such as OpenAI, ChatGPT, and Google Bard, are at the forefront of revolutionizing the use of AI. The growing efforts to commercialize LLM models and effectively support these advanced AI platforms highlight the critical need for the development of specialized inference hardware in data centers.

Introduced by Vaswani et al.,[1] LLM model inference is based on the transformer decoder, in which the

0272-1732 © 2024 IEEE
Digital Object Identifier 10.1109/MM.2024.3420728
Date of publication 9 July 2024; date of current version 11 December 2024.

inputs have limited batching capabilities and require sequential processing. As relatively small inputs need to be inferred with large model parameters, the inference incurs memory bottleneck and requires efficient processing of the system's memory bandwidth. Moreover, scalability becomes significant as the ever-increasing compute and memory requirements of LLMs demand multiple devices and communication among them. At the application level, each user makes individual requests and expects the generated output with minimal wait time, making it crucial to have a hardware platform that reduces inference latency. The predominant hardware for inference, a GPU, underperforms for GenAI workloads because it undergoes low hardware utilization for small-batch inputs and high communication overhead during synchronization. Therefore, a new class of processor is required that targets memory-intensive GenAI workloads.

In addition to performance, efficiency and usability are important factors in evaluating inference hardware. Maintaining efficiency across every GenAI application is difficult because each application requires different sizes of LLMs. Although larger LLMs may generate superior response for a given user context, the power and cost overhead of running such models may not be suitable. For instance, the service-level agreement (SLA) is less strict for LLM applications that are used for light interaction with the user (e.g., chatbots), in which smaller LLMs may be adequate, whereas an application that conducts a search or an analysis from a broad context may require larger LLMs, as discussed in Chowdhery et al.[2] However, hardware that considers the acceleration of larger models likely undergoes inefficiencies for smaller models. Therefore, an architecture that fully leverages memory bandwidth to yield maximum performance regardless of the model size would be the most efficient. The hardware must also be accompanied by a comprehensive software framework for a fast speed to market of these various LLMs. A compiler that is both model and hardware aware and automated to output the prerequisite data, such as memory mapping and instructions, is necessary. The runtime software must also adhere to the available application programming interfaces (APIs), such as HuggingFace, so that a hardware solution can be easily integrated to run LLM applications.

In this article, we propose HyperAccel's latency processing unit (LPU), a latency-optimized and highly scalable architecture that accelerates LLM inference for GenAI. The key contributions of the LPU are as follows:

- The LPU introduces streamlined hardware that maximizes effective memory bandwidth use during end-to-end inference regardless of the model size to achieve up to 90% bandwidth utilization for high-speed text generation. It also consists of an expandable synchronization link (ESL) that hides the bulk of the data synchronization latency in a multidevice system to achieve near-perfect scalability, or a 1.75× speedup for doubling the number of devices.
- We propose *HyperDex*, a software framework that enables automated compilation of prerequisite data based on LLM specifications. It also provides a runtime environment based on the widely used HuggingFace API for seamless execution of GenAI applications on LPU hardware.
- LPU achieves 1.25 ms/token for OPT 1.3B, and two LPUs achieve 20.9 ms/token for Open Pretrained Transformers (OPT) 66B, which is 2.09× and 1.37× faster than GPUs with an equal device count. The LPU-based application-specific integrated circuit (ASIC) implemented using a 4-nm process consumes only 0.824 mm^2 in area and 284.31 mW in power.
- We showcase HyperAccel *Orion*, an LPU-equipped server system that is ready to run the GenAI application in cloud and edge data centers. Orion achieves 1.33× and 1.32× energy efficiency over the state-of-the-art Nvidia H100 and L4 GPU server solutions, respectively.

BACKGROUND AND MOTIVATION

Structure of LLMs

LLM inference is largely divided into summarization and generation stages, as shown in Wang et al.[3] Figure 1 shows an overview of LLM inference. LLM inference begins with the summarization stage, in which the input to the decoder layer is a matrix that represents the token embedding of the user context (e.g., a statement or a question). The matrix is inputted to a series of decoder layers, which is based on the transformer decoder. The final decoder layer outputs the captured features from the input context. The result of the final decoder layer enters the language modeling (LM) head. The LM head converts the features into logits that score the candidate tokens from the dictionary based on their likelihood of being appropriate in the given context. With a single execution of the summarization stage, the first output token ($i = 0$) is produced. The first output token then enters the generation stage. Only the keys and values from the masked multihead attention are transferred to the next stage as activations that hold contextual information about the previous

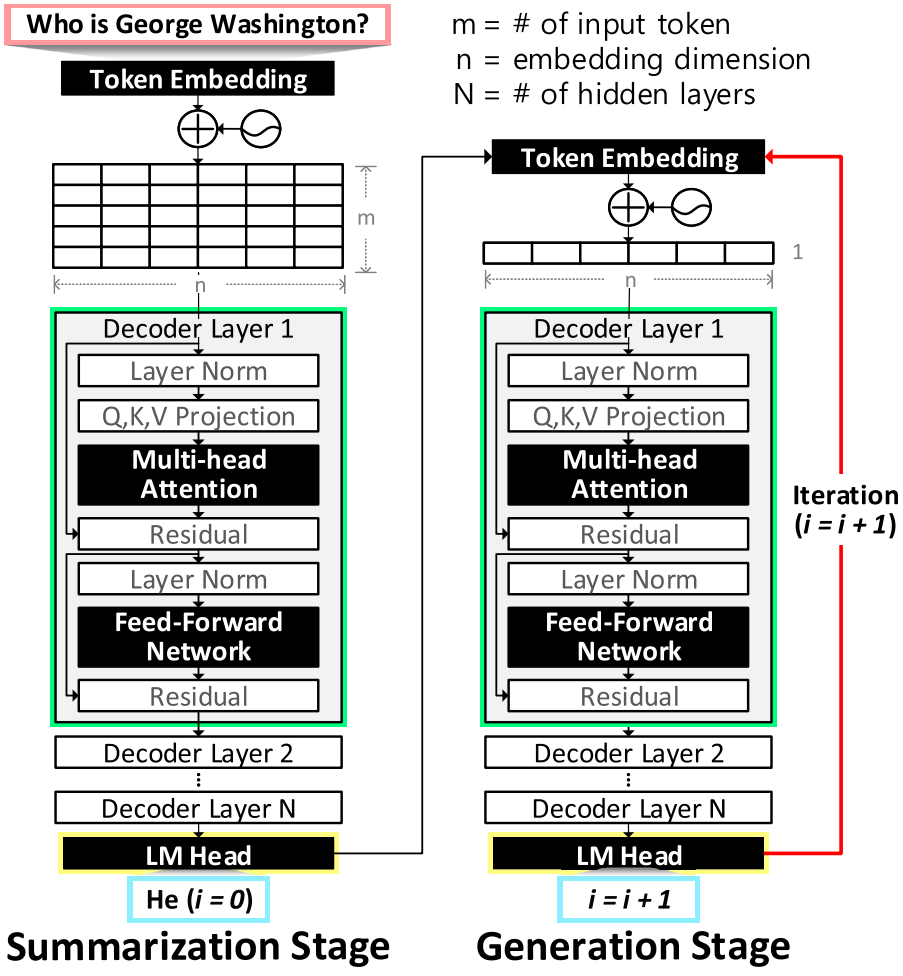


FIGURE 1. Structure of LLMs.

token. In the generation stage, the input to the decoder layer is guaranteed to be a single embedding vector. The processes are repeated to autoregressively output the next output token $(i = i + 1)$. The generation stage iterates until the end-of-sequence token is reached.

Out of all the processes, the most computationally dominant process is the decoder layer. Furthermore, masked multihead attention and feedforward network operations within the decoder layer account for 90.7% of the total inference time for an LLM with 7 billion parameters (e.g., LLaMA 7B). Both operations require the input to be multiplied by a weight matrix, and the input is a matrix in the summarization stage and a vector in the generation stage. Because more iterations are required in the generation stage, an architecture that is optimized for vector-matrix multiplication is necessary for achieving maximum performance.

The sequential characteristic of LLM inference requires constant access to new parameters with minimal reuse, which indicates that LLM inference undergoes significant memory bottleneck. To resolve the memory bottleneck, sufficiently fast memory is required [i.e., high bandwidth memory (HBM)], but more importantly, the efficient use of the given bandwidth is directly proportional to performance.

Diversity of LLMs

The size of LLMs is diverging. The demand for increasingly flexible and accurate LLMs has initiated a competitive push toward larger LLMs, some boasting up to a trillion parameters, whereas the demand for more manageable models have reduced the model size to single-digit billion parameters with optimization efforts in prompt engineering, domain-specific training, and various quantization methods. Depending on the environment, a device with different hardware specifications, especially memory bandwidth, may be adopted due to different SLA and price considerations. For instance, a

device with lower memory bandwidth may be satisfactory if the SLA is less strict or the budget is limited.

To meet the requirements of both diverse-sized LLMs and scalable hardware, a software framework that effectively bridges them is essential. Wolf et al.[4] identifies that models are typically defined and deployed to the acceleration hardware using software frameworks such as HuggingFace Transformers, PyTorch, and TensorFlow in the LLM ecosystem. The software framework performs optimization by considering several system-wide configurations, such as model size, budget, SLA, and hardware specifications, to meet the requirements. These frameworks hide complex details of the underlying acceleration hardware, enabling AI model architects and application developers to leverage the software framework to describe their models and integrate applications into the hardware, respectively.

Inefficiency in Conventional Hardware

Bandwidth

Although the modern GPU features substantial memory bandwidth and computation power, utilizing the full bandwidth of a GPU for LLM inference can be challenging due to the way a GPU is designed. For instance, the architecture of Nvidia GPUs is optimized for parallel processing, where many threads execute simultaneously across multiple cores. Especially in the generation stage, GPUs cannot effectively route the incoming bandwidth to a single core that requires computation with a single vector at a time, which causes underutilization of both compute cores and memory bandwidth, as mentioned in Hong et al.[5] This innate problem is more pronounced in smaller models due to even smaller operands (i.e., input and activations). According to Neubig et al.[6] and Kitaev et al.,[7] myriad software techniques, such as in-flight batching, key-value caching, and other algorithmic optimizations, have been proposed to raise the efficiency of GPUs. Despite these efforts, the utilization when processing real inference workloads is bounded by physical limitations, which leads to inefficiencies when deploying GPUs in practice. On average, Nvidia's H100 GPU achieves as low as 28.5% bandwidth utilization for the smaller OPT 1.3B model, but up to 69.9% for the larger OPT 30B model. Figure 2(a) shows the bandwidth utilization when running an LLM of various sizes.

Power

The high memory bandwidth and compute power of GPUs comes with high power consumption. As LLM inference is a memory-intensive workload, the power consumed during data transfer from memory translates to performance. However, the high operating frequency and the number of unused compute cores result in unnecessary power consumption, along with other power-hungry peripherals. To run the OPT 66B model, two Nvidia H100 GPUs consume an average of 1101 W. Figure 2(b) shows the power consumption of running LLMs of various sizes.

Scalability

The growing size of LLMs has led to the requirement of multiple devices. As the data precision of standard LLM models are half-precision floating-point (FP) 16, the memory requirement of LLMs is approximately the number of model parameters multiplied by two bytes. For instance, the 66B model requires 132 GB and an additional 5 GB for storing key values. As 137 GB exceeds the HBM capacity of Nvidia's H100 GPU with 80 GB, two H100s are required. Therefore, effective communication among devices is essential for efficient LLM inference of large models. Nvidia GPUs support NVLink, a flagship direct GPU-to-GPU interconnect that transfers data up to 900 GB/s. Despite this high-speed interconnection, data synchronization overhead in tensor parallelism is significant because the computation is stalled during the communication. Figure 2(c) shows the scalability of Nvidia's DGX A100 with 3G NVLink (600 GB/s) running GPT3 20B inference. DGX A100 achieves an average of only a 1.38× speedup when doubling the number of devices. The scalability result is based on the released benchmark result of the Nvidia FasterTransformer library. For the most efficient processing, a domain-specific architecture for LLM inference with high bandwidth and core utilization, low power overhead, and high scalability is required.

LPU

The LPU architecture consists of streamlined hardware, a customized instruction set, and LLM-specific dataflow for high-speed LLM inference. The LPU architecture is shown in Figure 3(a).

Hardware Architecture

Streamlined memory access (SMA) is a specialized direct memory access (DMA) that connects all HBM channels to the execution engines to transfer FP16 data at maximum bandwidth. It is preloaded with memory instructions that send continuous read requests for weights, and occasional write requests for key-value write. Our hardware-aware memory mapping removes the need for any data reshaping or switching in the SMA. A matrix transpose is required in the attention mechanism, but SMA utilizes the strobe signal with an algorithmic approach that writes to a specific memory location so that the data are naturally

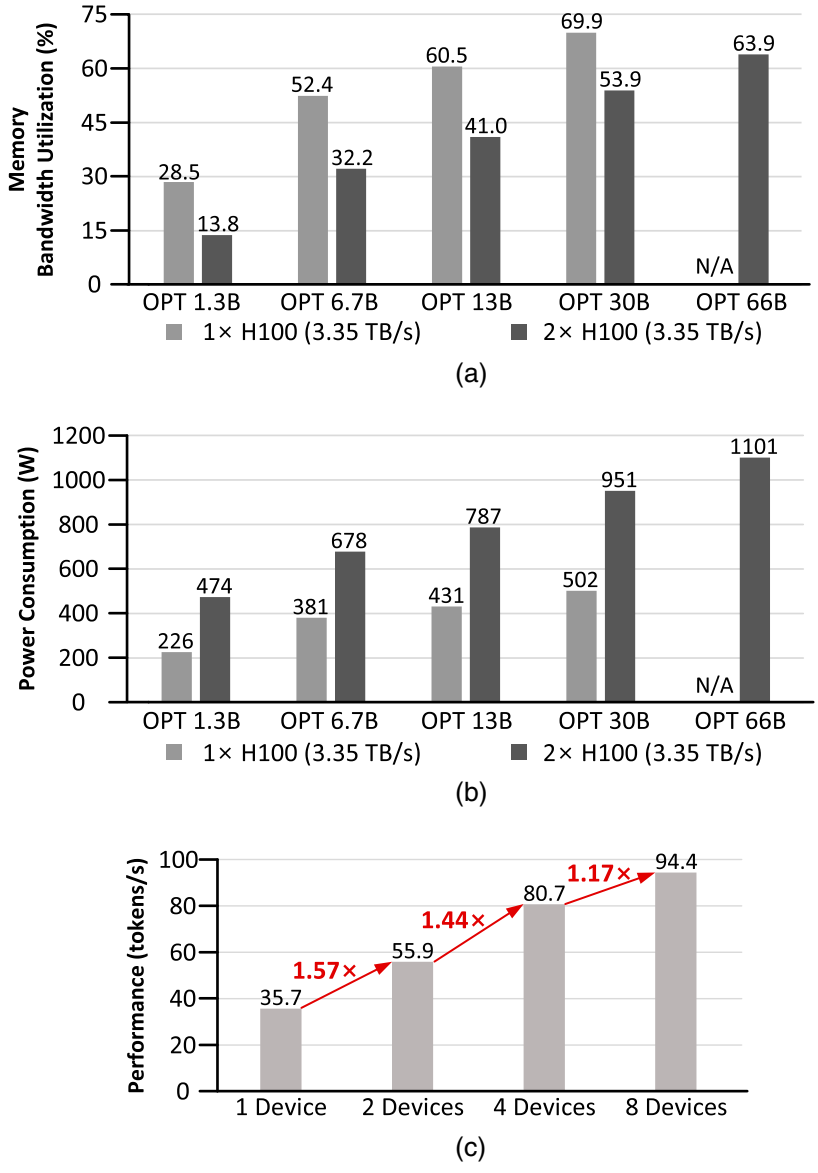
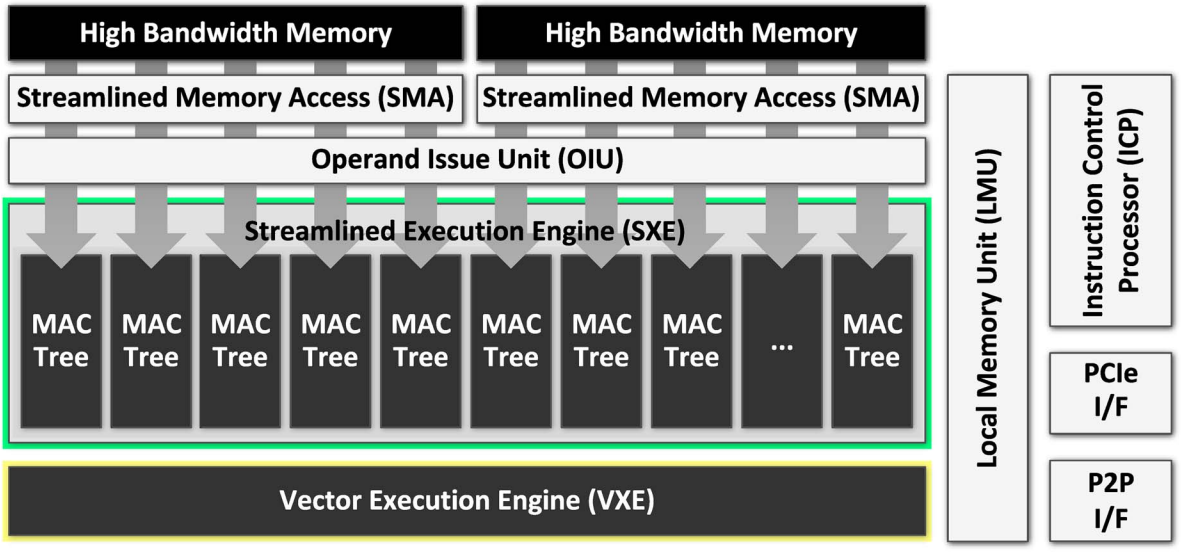


FIGURE 2. GPU analysis when running LLM inference. (a) H100 GPU memory bandwidth utilization. (b) H100 GPU power consumption. (c) DGX A100 performance scalability. N/A: not applicable.

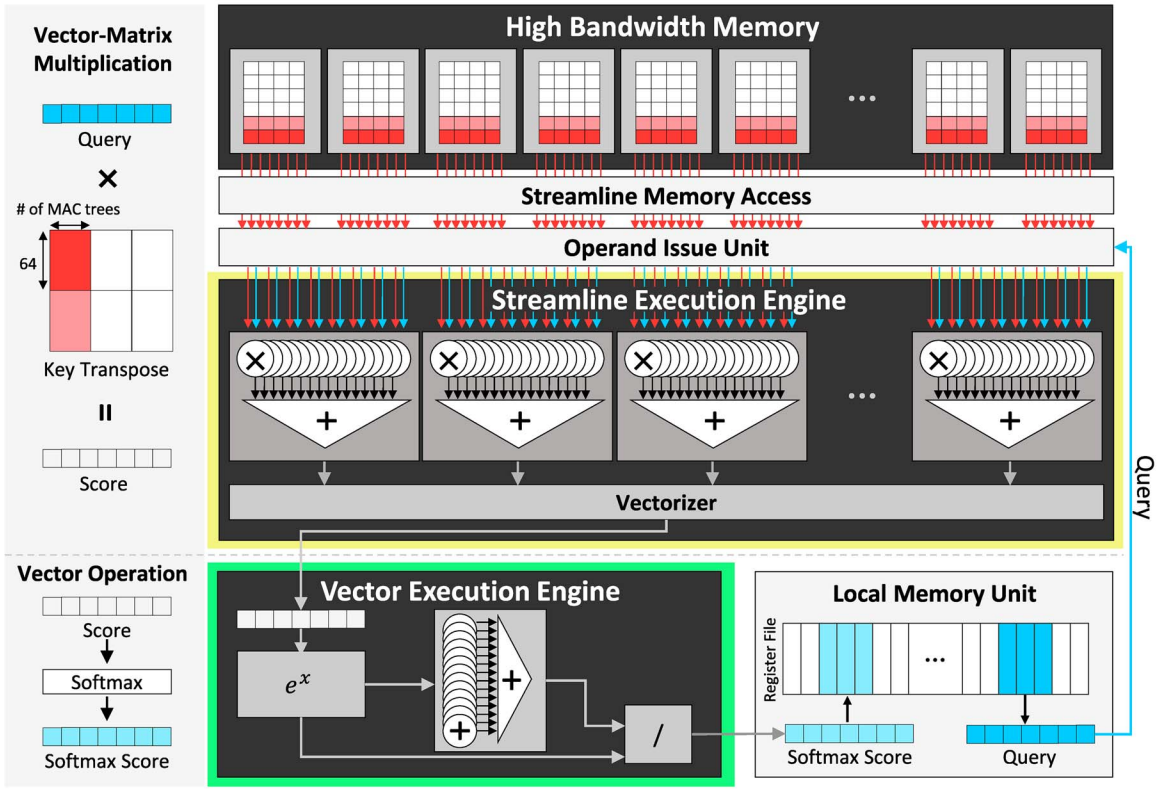
transposed when read without adding latency overhead. Because many compute units are placed to exactly match the total HBM bandwidth, SMA streams the data received at maximum burst size to the execution engines with minimum stalling. The streamed data are parameters for vector-matrix execution (e.g., weight and bias) and other vector-related operations (e.g., gamma/beta and embedding).

The operand issue unit (OIU) arbitrates the streamed data (i.e., the first operand) from the SMA and the input (i.e., second operand) from the on-chip register file before issuing them to the execution engines. Based on the compute instructions, the OIU generates microcodes that configure the execution engines and decide the destination engine for the operands. Appropriate operands are prefetched, ready to be issued to the execution engines immediately.

The streamlined execution engine (SXE) is the LPU's main computing hardware, with customized low-latency multiply-accumulate (MAC) trees, which are primarily designed to execute vector-matrix multiplication for multihead attention and feedforward network



(a)



(b)

FIGURE 3. LPU hardware architecture and its dataflow. (a) Overall LPU hardware architecture. (b) LPU dataflow for multihead attention. I/F: interface; MAC: multiply-accumulate; P2P: peer to peer; PCIe: Peripheral Component Interconnect Express.

operations. Each MAC tree consists of the following features: 1) The preprocessing of the operands based on the exponent and mantissa of the larger FP operand enables fixed-point multiplication and accumulation to reduce the logic area. 2) The fixed-point adder tree for mantissa utilizes a Wallace tree for high-speed addition via parallelization. The SXE has *l* number of MAC trees, each operating on a set of *v* vector elements. To fully utilize memory bandwidth, the incoming bandwidth and the bandwidth of the MAC tree (i.e., $l \times v \times 2\mathrm{B} \times \mathrm{freq}$) must be equal. Note that *freq* refers to the operating frequency of the hardware. Assuming that the target frequency is predetermined, the number of MAC trees is chosen to match the memory speed. Moreover, the SXE is superpipelined to constantly receive operands, which increases throughput and thus lowers overall latency. It also supports model-specific operations such as rotary positional embedding and nonlinear activation functions.

The vector execution engine (VXE) executes vector operations, such as token embedding, softmax, normalization, and residual, with a customized low-latency arithmetic logic unit. The VXE also contains a sampler that sorts logits and selects an output token based on temperature, top-p, and top-k values. As these vector operations occur less frequently, we reduce the fan-in from the OIU to this path to decrease the hardware resource with negligible performance loss.

An instruction control processor (ICP) is an RISC processor that controls the overall execution flow of the LPU. The ICP primarily fetches LPU instructions from the instruction buffer and dispatches them. It also executes basic RISC-type instructions, such as branch and jump, based on the control registers (e.g., the token and layer numbers) for iterative and conditional logic. The dispatcher in the ICP is entirely independent of other LPU modules, so the instructions are continuously prefetched in the other modules for minimum control interference. Moreover, the internal scheduler supports the out-of-order execution of the SXE and the VXE for improved latency and hardware utilization, and a scoreboard is designed to handle data hazards.

The local memory unit (LMU) is a multibank register file with scalar-vector segregation for fast high-bandwidth access to one of the operands. It is also a multiport that supports simultaneous read to the OIU and write from the writeback of the execution engines.

LPU Dataflow

The LPU adopts the output stationary dataflow, in which the activation vector is reused, and weights are streamed to execute vector-matrix multiplication. Rectangular tiles with a length equal to the vector dimension and a width equal to the number of MAC trees are loaded from memory each cycle. The tiles are accessed in a vertical direction, which reduces the number of partial sum buffers and simplifies control compared to other directions (e.g., horizontal and zigzag), because a set of dot products is guaranteed to be finished before the next set begins. The tile is memory mapped so that the data requested from the SMA can be directly wired as inputs to the MAC trees to fully utilize the SXE during the time that the vector-matrix multiplication is being executed. The parameters are also mapped to stream to the VXE for vector-vector operations. Figure 3(b) shows an example of the LPU dataflow when executing multihead attention. The memory mapped key stored in the HBM is read to the SMA based on the tiling. The key and the corresponding portion of the query stored in the LMU are multiplied in the SXE using the customized MAC trees to generate a score. The score is then sent to the VXE to execute the softmax operation while the remaining tile of the key enters the SXE. During parallel execution of the SXE and the VXE, the SMA continuously reads the remaining key from the HBM, and the ICP concurrently calculates the new addresses for the next memory read/write.

Instruction Set Architecture

The LPU operates based on a customized instruction set architecture. Table 1 shows a brief list of the ISA of the LPU. The ISA is largely divided into direct memory access (MEM), compute (COMP), network (NET), and basic RISC-type instructions (CTRL). MEM accommodates memory read/write of different parameters (e.g., weight, key, value, and embedding and normalization data) for data transfer. COMP handles simple vector arithmetic, along with more complex fusion operations for computation. NET handles the transmit/receive of partial results for synchronization. CTRL moves and calculates the program counter and addresses based on scalar register data for controlling the processor. The processes of reading the weight, executing the calculation, and writing the activation are executed concurrently for independent instruction sets and in a streamlined manner for dependent instruction sets to fully utilize the bandwidth, which translates to enhanced performance.

ESL

With the increase in model size, running an LLM on a single device is becoming challenging. Intralayer model parallelism, which distributes both the model parameters and computational load across multiple devices to

TABLE 1. LPU instruction set architecture.

Category	Instruction type	Source		Destination	
MEM	Read embedding	V	HBM	V	LMU
	Read key/value	V	HBM	V	SMA
	Read parameters	V	HBM	V	SMA
	Read from host	V	Host	V	LMU
	Write key/value	V	SMA	V	HBM
	Write to host	V	LMU	V	Host
COMP	Matrix computation	V/S	LMU/ SMA	V/S	LMU/ SMA
	Vector computation	V/S	LMU	V/S	LMU
	Vector fusion computation	V/S	LMU	V/S	LMU
	Sampling with sort	V/S	LMU	V/S	LMU
NET	Transmit	V	LMU	V	P2P
	Receive	V	P2P	V	LMU
CTRL	Scalar computation	S	ICP/ LMU	S	ICP/ LMU
	Branch	S	ICP	S	ICP
	Jump	S	ICP	S	ICP

execute partial vector-matrix multiplications, has become the standard approach. As the entire resulting vector is required before a subsequent multiplication begins, the compute units are stalled until the synchronization of data is completed, which causes significant overhead. A larger model requires even more data and devices to communicate with per synchronization, which further increases overhead. To address this issue, we develop an ESL, a peer-to-peer (P2P) communication technology that performs data synchronization with latency hiding. The ESL also supports reconfiguration to enable optimal network configuration for the given workload.

Expandable Network

The ESL focuses primarily on fostering high scalability that ensures linear performance improvement with the expansion of the number of LPU devices. We devise a customized ESL protocol that effectively overlaps vector-matrix multiplication with synchronization, thereby hiding communication latency. In a typical processor that adopts model parallelism (e.g., tensor parallelism), the communication follows after the computation, and vice versa, leading to the inclusion of the entire communication latency in the overall latency. In an ESL, vector-matrix multiplication is first divided into smaller column-based tasks, in which the result matches the bitwidth of the P2P interface. The destination of the partial products from the SXE is assigned to be a temporary buffer instead of the register file. From the buffer, the supporting ESL dataflow enables the immediate transmission of the partial products to the peer devices while the next operation is ongoing. The dataflow also includes the runtime arbitration between the partial products received from the peer devices and written back from its own SXE. As computing, transmitting, and receiving can all be done concurrently, new computation and the communication pertaining to the previous computation always overlap. This overlap hides all of the communication latency except for a small tail latency. Figure 4(a) shows the ESL dataflow and the operation timeline.

For cases in which two vector-matrix multiplications happen sequentially [e.g., fully connected (FC) layer 1 followed by FC layer 2 in the feedforward network], even the tail latency of the synchronization is hidden. As latency hiding is effective in any type of network topology, we choose the ring topology. A ring minimizes tail latency by simplifying the packet processing on the routers that would otherwise be substantial in a more complex network. In ESL architecture, each device is equipped with two quad small form-factor pluggable ports, supporting full-duplex communication for simultaneous transmission and reception.

Reconfigurable Network

The necessity of a reconfigurable network in the ESL arises from the requirement to efficiently support LLMs across various sizes and use cases. Even in an eight-device configuration, there are scenarios in which operating with only two or four devices is more efficient. For instance, running two different models on a single server can be achieved by switching the model once inference is finished with one of the models on the eight-device configuration. However, performance and efficiency losses occur due to switching overhead (e.g., model loading). Therefore, operating two models with two sets of four devices is more effective. To accommodate diverse model requirements, we implement a reconfigurable network in the ESL to support two-, four-, and eight-device configurations, as shown in Figure 4(b). In an eight-device configuration, a full ring is utilized, whereas in a four-device configuration, it is split into two independent four lines. Similarly, a two-device setup employs four two lines. The router determines the number and direction of hops based on the device ID to formulate a packet header that guarantees the most efficient communication path for

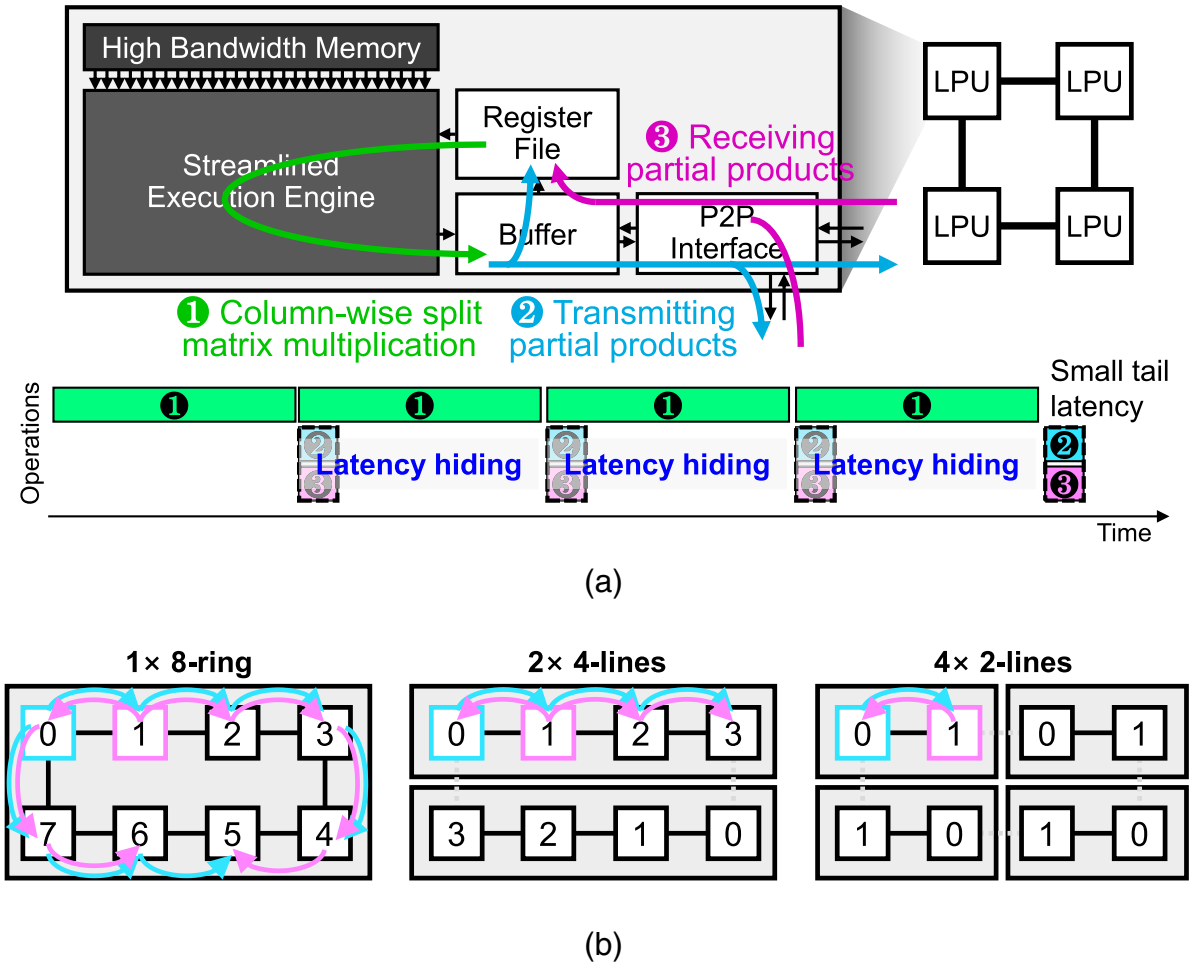


FIGURE 4. Dataflow and timeline of data synchronization in an LPU with an ESL. (a) Data synchronization in an LPU with an ESL. (b) Reconfigurable network in a multiuser environment.

synchronization. This design effectively maximizes the network resource without having to rewire or reload the model. Because each ring is guaranteed not to intersect with a different ring, each configuration can run independently to maximize the network resource within its own ring and thus the overall system. In all the configurations, communication overhead is still the minimal tail latency, thereby achieving high scalability.

HyperDex FRAMEWORK

We develop *HyperDex*, a software framework that consists of both compilation and runtime layers [shown in Figure 5(a)], which is designed to deploy user applications and GenAI models to the LPU. HyperDex's compilation and runtime layers are exposed as APIs to AI model architects and users, respectively. These layers serve as a bridge, enabling both model architects and users to seamlessly integrate GenAI models and applications into LPU-equipped systems without requiring in-depth details about the underlying hardware.

Compilation Layer

The compilation layer within HyperDex framework hides the details of hardware and provides an abstracted interface of LPU to model architects, simplifying the deployment of GenAI models on LPU-equipped systems with requiring minimal effort. We design the compilation layer to enable developers to import existing HuggingFace models and perform memory mapping, instruction generation, and compilation to generate a binary program for the LPU hardware. Moreover, model architects can customize, program, and compile their own models with an API provided by HyperDex's compilation layer.

HyperDex's model and memory mapper analyzes the given model architecture and parameters, determining the most optimal memory allocation and

alignment of each model parameter for maximum burst and streamlined processing within the LPU. The HyperDex mapper considers system setups, such as number of devices and topology of the network, to partition the model parameters across multiple devices based on the intralayer model parallelism, a type of model parallelism that divides the model parameters of parallelizable operations into multiple devices. As the ESL innately complements model parallelism, its dataflow can stay transparent to the HyperDex mapper. Therefore, the mapper does not require further consideration to support the ESL. The HyperDex model and memory mapper also takes into account memory (e.g., the number of HBM channels and the burst size) and compute (e.g., the number of compute units) configurations to determine optimal memory mapping, tiling, and padding size. The mapper divides the multihead attention weights with headwise tiles and the feedforward network weights with columnwise tiles, in which their dimensions are dependent on hardware specifications. The result is memory mapping of the tiled weights that perfectly matches the memory channel bitwidth and the order of operation to enable maximum utilization of memory bandwidth. The mapped parameters are fed to the instruction generator and loaded to the LPU via the HyperDex runtime layer.

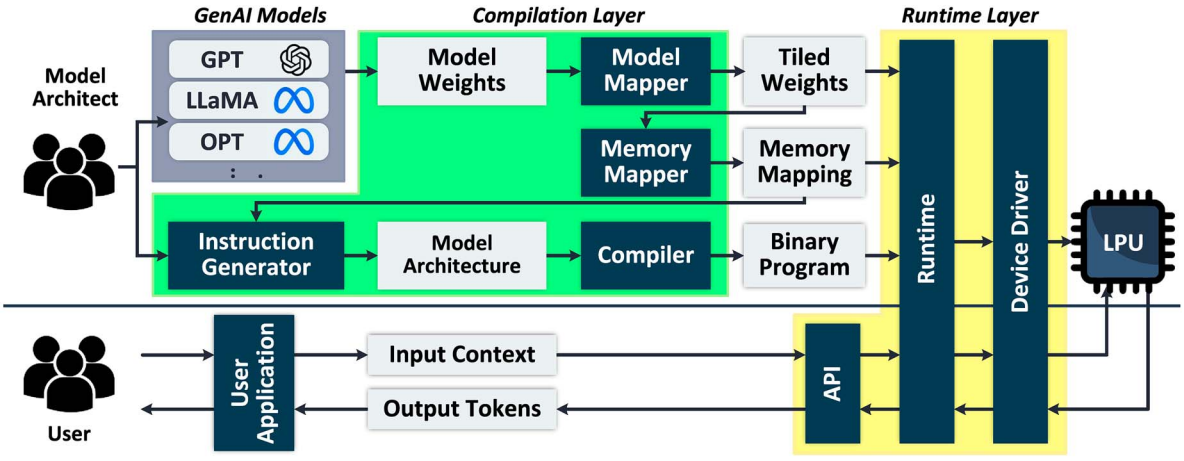
HyperDex's instruction generator creates a series of instructions for the LPU that describes the GenAI model's architecture based on the memory mapping information generated by the HyperDex model and memory mapper. The HyperDex instruction generator provides a front end that converts the Open Neural Network Exchange (ONNX) format into Python API calls of predefined instruction blocks for popular LLM models such as GPT, OPT, and LLaMA. For instance, Figure 5(b) demonstrates how the model architecture code for the LLaMA model is converted into Python with the HyperDex instruction generator. In this code snippet, the model architecture of LLaMA is defined through the following code blocks: 1) `input_load` loads input user context via DMA and stores it in the designated variable; 2) `token_embed`, `decoder`, and `lmhead` are predefined blocks that encapsulate the core logic of the model; 3) `output_store` returns the inferred output tokens via DMA; and 4) `hlt` marks the termination of the program. Subsequently, this model architecture code is compiled with the `do_compile` function and is converted into a binary program through the `fwrite` function. For communication between devices, the `sync` block, composed of `transmit` and `receive` instructions, is utilized. As it is the role of the P2P interface to leverage the ESL directly, the instruction generator operates independently of the ESL.

For foundational models, the generation of the instruction blocks is automated, in which the parameters (e.g., the source, destination, and their sizes) are parsed from the ONNX model. For custom models, HyperDex provides an API for model architects to program their own custom model architecture, and for corresponding operators using the instruction blocks. As the LPU consists of an RISC-type processor and programmable engines, customized instruction blocks can be programmed and inserted to accommodate future operations and model architectures. All of these blocks are translated into a list of LPU instructions (i.e., machine codes).

HyperDex's compiler then compiles the instructions generated by HyperDex's instruction generator into a binary program for the LPU hardware with several optimizations. The *register allocator* of the compiler tracks the lifetime of all variables and automatically allocates and releases the hardware registers at the compiler level. It eliminates the necessity for the instruction generator and GenAI model architects to manually manage registers, offering them an abstracted view of the register file. *Instruction chaining* strategically divides the operations into a series of dependent instructions that can be executed back to back without any control overhead after initialization. Our optimization for instruction chaining further separates instructions utilizing independent hardware modules and separating them into distinct groups (e.g., MEM, COMP, NET, and CTRL) of instruction chains. The compiler organizes a set of instructions that perform specific functions from the distinct groups and interleaves them so that the execution of each instruction can be overlapped. It enables parallel execution of instruction chains across these groups, effectively minimizing control overhead and latency.

Runtime Layer

Once the information about the weights and architecture of the LLaMA model is loaded into the LPU, HyperDex's runtime layer initializes the system. This initialization process occurs offline, and upon completion, other users can proceed to conduct model inference with the LPU-equipped system. HyperDex's runtime layer provides a collection of APIs for user applications so that users can seamlessly integrate their GenAI applications with the LPU-based hardware. The runtime layer provides APIs that align with the interfaces found in HuggingFace, including text generation, sampling, and streaming, allowing existing user applications to be integrated with LPU hardware with minimal code modifications. Figure 5(b)



(a)

```python
# Instruction Generation Example
model = InstgenLLaMAModel()
compiler.add_inst([
    input_load(dest=input_val),
    model.token_embed(src=input_val,
        dest=[embed_val, rot_val]),
    model.decoder(src=[embed_val, rot_val],
        dest=logit_val),
    model.lmhead(src=logit_val, dest=output_val),
    output_store(src=output_val),
    hlt()
])
compiler.do_compile()
compiler.fwrite()
```

```python
# Runtime Example
inputs = "Who is George Washington?"
input_ids = tokenizer.encode(
    inputs, return_tensors="np")[0]

output_ids = model.generate(
    input_ids,
    max_length=1024,
    do_sample=True,
    top_p=0.7,
    top_k=10,
    temperature=1.1,
    repetition_penalty=1.2)

outputs = tokenizer.decode(
    output_ids, skip_special_tokens=True)
```

(b)

FIGURE 5. (a) HyperDex software stack for the LPU. (b) Example use of the HyperDex framework.

demonstrates an example text generation application implemented using the API. Note that our design approach for the runtime API aligns with the interfaces of HuggingFace. In this example code, `tokenizer` and `model` have the same interface as HuggingFace's `AutoTokenizer` and `AutoModelForCausalLM`, respectively.

HyperDex's runtime layer incorporates a device driver beneath the runtime API to perform low-level operations and hide complex hardware details from developers. This device driver extracts user-specified per-request and per-core arguments (e.g., core number, input and output token length, and sampling parameters) via the runtime API and transfers these data to the control registers of the LPU. Whenever users send the requests through the runtime API, the LPU receives these requests with user-specified arguments via the device driver, performs GenAI model inference, and returns the inference result. Furthermore, HyperDex's runtime layer offers monitoring tools that provide hardware-level statistics, such as power consumption, LPU utilization, and HBM use, which are obtained from the device driver. These tools are crucial for managing LPU-equipped systems at the data center level.

EXPERIMENTAL SETUP

ASIC Implementation

We develop a register transfer-level implementation of the LPU using System Verilog, and synthesize the

proposed LPU architecture at a Samsung 4-nm library and Synopsys Design Compiler. To demonstrate the scalability of our architecture, we synthesize the LPU with three different HBM configurations: 819 GB/s (24 GB), 1.64 TB/s (48 GB), and 3.28 TB/s (96 GB). The vector dimension to fixed to 64 because the embedding dimension of most LLMs is multiples of 64, thus, the number of MAC trees is set to eight, 16, and 32 to exactly match the memory bandwidths given the operating frequency of 1 GHz. An alternative is to scale down the vector dimension and proportionally scale up the number of MAC trees, but this configuration would halve the area of the VXE at the cost of doubling its latency. We place and route using Synopsys IC Compiler II and measure the power and area through Synopsys's PrimePower and Design Compiler tool, respectively. Figure 6(a) shows the overall LPU chip layout.

ASIC Simulator

For performance evaluation, we implement an in-house cycle-accurate simulator, written in C++, to measure the LPU's latency. It also simulates the ESL to support any number of desired devices. We integrate Ramulator with our simulator to simulate Samsung's HBM3 Icebolt.[8] We configure a single stack of HBM3 to have a speed of 819 GB/s and a capacity of 24 GB. In the evaluation, one, two, and four stacks of HBM3 are used for each HBM configuration. Also with the simulator, we observe that the LPU occurs no accuracy loss on popular datasets (e.g., LAMBADA and the Winograd Schema Challenge) with the open source Megatron-LM GPT2, as the LPU supports standard FP16 data precision.

Field-Programmable Gate Array Implementation

Before completing the fabrication of the LPU ASIC, we implement the LPU architecture on Xilinx Alveo U55C field-programmable gate array (FPGA) accelerator cards, where each device consists of the HBM2 with 460-GB/s memory bandwidth and 16-GB capacity. Our FPGA implementation utilizes 46.4% of lookup table, 39% of flip-flops, 57% of BRAM, 36.9% of ultra random-access memory, and 27.5% of digital signal processors running at a 220-MHz kernel operating frequency. To match the memory bandwidth, the LPU is configured with 16 MAC trees, which have a total bandwidth of $16 \times 64 \times 2\text{B} \times 220\ \text{MHz} \approx 460\ \text{GB/s}$.

Server-Scale Integration

For initial commercialization, we productize HyperAccel Orion, a data center rack server based on the FPGA implementation, in two configurations: 1) Orion Cloud, with eight LPU-equipped acceleration cards in a 2-U server chassis with a 128-GB and 3.3-TB/s HBM, and 2) Orion Edge, with two LPU-equipped acceleration cards in an edge-server chassis with a 32-GB and 960-GB/s HBM. For P2P communication, the devices are connected via the ESL, a ring network connected by dual-QSFP28 cables capable of 2×100 Gb/s. The software stack of Orion consists of the Xilinx Vitis 2022.2 platform and the HyperDex framework with a HuggingFace-like API for running multibillion-parameter LLMs. Figure 6(b) shows the implementation and server details.

Methodology

LPU Performance

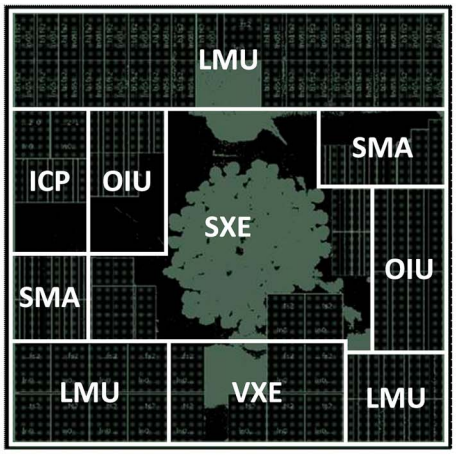
We measure the latency per output token of the LPU with a 3.28-TB/s HBM configuration. As publicly available models on HuggingFace mostly consist of up to 70 billion parameters, we scale out to two LPUs to provide a total of 6.56 TB/s and 192 GB, respectively. For the model, we run widely benchmarked OPT 1.3B, 6.7B, 30B, and 66B. Note that the LPU supports other LLMs, such as GPT, LLaMA, and their variants, but the latency is largely affected by model size, not type. For a fair performance comparison, we compare the HyperAccel LPU with the state-of-the-art Nvidia H100 GPU, which has a comparable memory bandwidth of 3.35TB/s. We run LLM inference with a focus on generative tasks (e.g., article, code, and other text generation), in which we fix the number of input and output tokens to 32 and 2016, respectively. Note that the number of input tokens would barely affect the latency of the GPU as it can process inputs in parallel.

Server Efficiency

The efficiency of Orion is based on a real-system performance achieved while running an end-to-end text generation application. The number of tokens generated in 1 s per kilowatt of power consumption for each server is measured. We compare Orion Cloud and a GPU server equipped with two Nvidia H100s running OPT 1.3B to 66B, and compare Orion Edge and a GPU server equipped with two Nvidia L4s running OPT 1.3B and 6.7B. The chosen models are variations of the LLMs that fit into the given system. Note that memory space is labeled in a decimal prefix (gigabytes) but has physical capacity based on the binary prefix [gibibytes (GiB)], so that the 66B model can fit into the 128-GB Orion Cloud system (e.g., 1.074 GB = 1 GiB). The compared systems have similar memory bandwidth specifications and thermal design power.

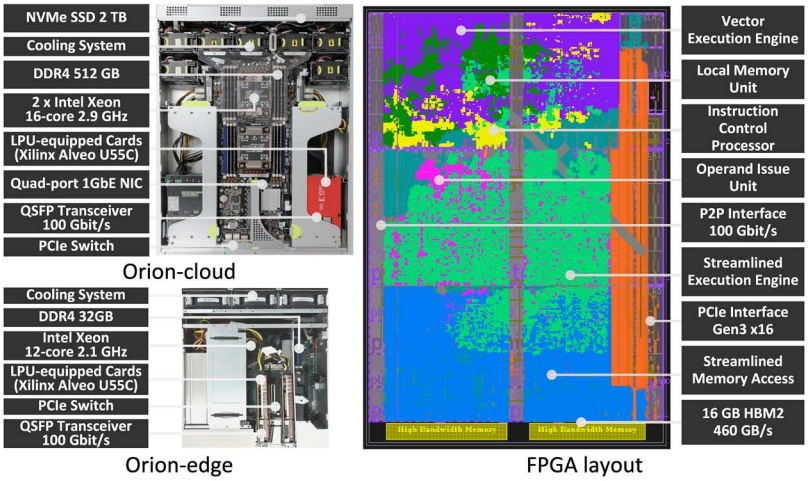
Scalability

We compare the scalability of the LPU and the GPU in a multidevice setting. The GPU results are based on the open benchmark numbers of Nvidia's DGX A100



	LPU (819 GB/s)	LPU (1.64 TB/s)	LPU (3.28 TB/s)
Process Node	Samsung LPP 4nm		
Frequency	1.0 GHz		
# of MAC Trees	8	16	32
On-Chip SRAM	812 KB	910 KB	1.107 MB
Memory Type	HBM3, 1 stacks	HBM3, 2 stacks	HBM3, 4 stacks
Memory Capacity	24 GB	48 GB	96 GB
Memory Bandwidth	819 GB/s	1.64 TB/s	3.28 TB/s
Chip Area	0.548 mm^2	0.646 mm^2	0.824 mm^2
Chip Power	81.10 mW	149.70 mW	284.31 mW
System Power	22 W	43 W	86 W

(a)



(b)

FIGURE 6. LPU implementation and specification. (a) LPU ASIC chip layout and specification. (b) Orion, the LPU-based data center server, and a field-programmable gate array (FPGA) layout. DDR: double data rate; LPP: low power plus; NIC: network interface card. NVMe: non-volatile memory express; QSFP: quad small form-factor pluggable; SRAM: static random-access memory; SSD: solid-state device.

mentioned earlier. DGX A100 is a server system that supports eight Nvidia A100 GPUs on a 6-U server chassis and a 3G NVLink with a GPU-to-GPU bandwidth of 600 GB/s. The scalability of the LPU is validated using Orion Cloud with ESL enabled.

EVALUATION

Chip Area and Power Analysis

Figure 6(a) shows the area and power estimation breakdown of the LPU in three different configurations.

We specifically focus on the LPU with 32 MAC trees and a 3.28-TB/s HBM, which has comparable memory bandwidth to the baseline Nvidia H100 GPU with a 3.35-TB/s HBM. The area and power consumption of the LPU chip is 0.824 mm^2 and 284.31 mW, respectively. The SXE dominates the area and power consumption of the LPU for housing a majority of the compute logic, followed by SMA and the LMU with mostly static random-access memories (SRAMs) for buffering and storing data. Including four HBM3 stacks, the total power consumption of the LPU system is 86 W. Compared to the H100 GPU, the LPU system requires only 15.2% of the power consumption when running OPT 30B. The areas are not compared because the GPU is a general-purpose processor and the LPU is a domain-specific processor.

LPU Performance

Latency

Figure 7(a) shows the simulated latency of the LPU. Notably, the LPU achieves 1.25 ms/token with the 1.3B model, and 4.62 ms/token with the 6.7B model. For the bigger OPT 66B model, two LPUs generate one token in just 22.2 ms. When compared with equal numbers of H100, one LPU achieves a 2.09× speedup on the 1.3B model, and two LPUs achieve a 1.37× speedup on the 66B model. The streamlined architecture of the LPU effectively uses the given memory bandwidth during end-to-end inference compared to the GPU. On the OPT 30B and 66B models, one LPU and two LPUs use up to 90.2% and 90.6% of the memory bandwidth, respectively, whereas one GPU and two GPUs use up to 70.8% and 64.9%, respectively. The advantage is more drastic for the smaller OPT 1.3B model, in which one LPU achieves 63.3% memory bandwidth utilization, whereas one GPU achieves only 28.9%.

Server Efficiency

Figure 7(b) shows the efficiency analysis between HyperAccel Orion and GPU servers with comparable hardware specifications. For the cloud server, HyperAccel Orion Cloud with eight LPUs achieves 1.33× energy efficiency over the GPU server with 2× the Nvidia H100 when running the OPT 66B model. Orion Cloud consumes 608 W, whereas H100 consumes 1100 W. For the edge server, Orion Edge with two LPUs achieves 1.32 × energy efficiency over the GPU server with 2× Nvidia L4 when running the OPT 6.7B model. Note that the efficiency advantage of a device with an LPU-based ASIC over the GPU would be significantly greater.

Scalability

The LPU is specifically optimized to fully utilize its streamlined core for the small-batch computing required by LLM inference, whereas the GPU suffers from severe underutilization for such a workload. However, the GPU still requires multiple devices for the best performance because 1) memory bandwidth is the bottleneck and 2) additional memory capacity is required to support larger models. The GPU underutilization is accentuated with additional devices. Moreover, the LPU devises a customized protocol to hide a majority of synchronization latency, while the GPU undergoes significant overhead, as shown in the strong scaling of the two corresponding processors in Figure 7(c). We analyze the scaling efficiency of up to eight devices when running GPT3-20B. The LPU achieves a 5.43× speedup for the output token generation compared to a single device, which is significantly better than the a 2.65× speedup of DGX A100. The LPU achieves, on average, a 1.75× speedup for doubling the number of devices due to the high scalability of ESL technology, whereas the GPU achieves only a 1.38× speedup when the number of devices doubles due to the inability to hide the designated synchronization latency after the matrix multiplication. Scalability of the LPU is verified with the Orion Cloud product.

RELATED WORK

Recently, there has been a surge in the development of AI processors that accelerate LLM inference, with competitive performance. Intel's Habana Gaudi2 is a specialized accelerator that is designed for deep learning training and inference.[9] It excels in handling small tensor operations, which facilitates simultaneous computation and network communication among diverse components. This capability also helps to reduce the bandwidth demands on its memory subsystem.

Groq's LPU is based on the tensor streaming processor, with optimizations to support LLM inference.[10] It integrates 16 chip-to-chip interconnects and 230 MB of SRAM, offering versatile options for embedded applications. However, the absence of external memory, such as HBM, requires Groq to integrate hundreds of cores to effectively accelerate practical LLMs with tens of billions of parameters (e.g., 512 chips for LLaMA2-70B), which incurs substantial communication overhead.

Although these processors consist of architectures tailored for general AI computing to effectively handle LLMs, HyperAccel's LPU boasts an LLM-specific streamlined dataflow with high memory bandwidth and compute utilization to achieve unprecedented efficiency for

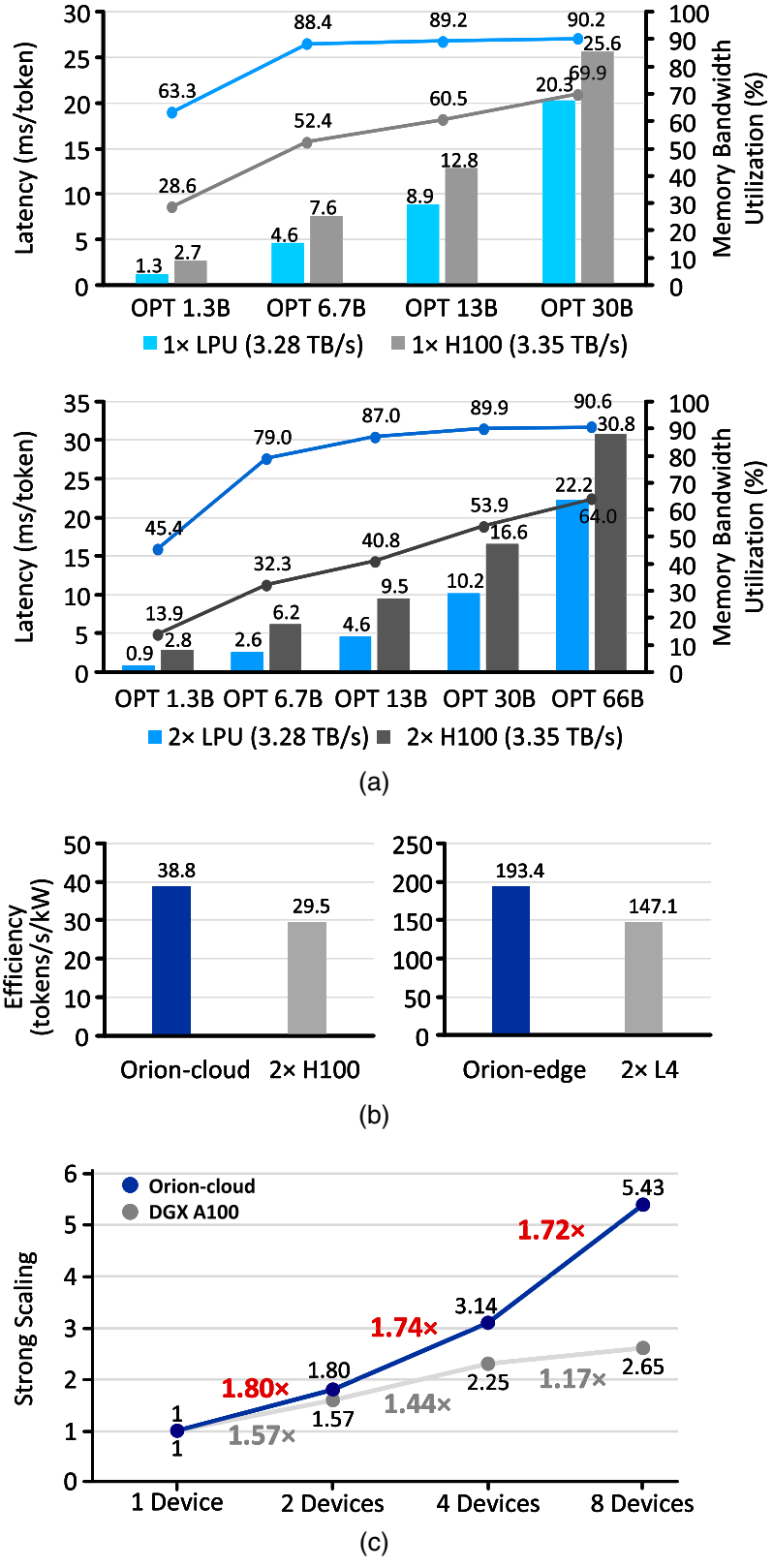


FIGURE 7. Performance of the LPU compared to the GPU. (a) Latency and memory bandwidth utilization. (b) Server efficiency analysis. (c) Scalability.

LLM inference. Moreover, the LPU features an efficient interconnect system that overlaps computation and communication between processors, which is beneficial for current LLMs that require systems ranging from node scale to rack scale.

CONCLUSION

We presented a new class of processing unit, the LPU, a latency-optimized and highly scalable architecture that accelerates LLM inference for GenAI. It introduces both the streamlined processor and ESL that maximize bandwidth use for agile processing and hide synchronization overhead in P2P computing, respectively. The HyperDex software framework assists the LPU to provide an optimized end-to-end solution.

The LPU-based ASIC achieved a token generation latency of 1.25 ms/token for the 1.3B model, and two LPUs achieved 20.9 ms/token for the 66B model, while having a total area of 0.824 mm^2 and power consumption of 284.31 mW. In addition, we implemented the LPU on cloud and edge FPGA servers to achieve 1.33×- and 1.32×-higher energy efficiency over Nvidia's H100 and L4 GPU server solutions, respectively.

Increasing the number of reused parameters would alleviate the memory bottleneck and proportionally increase performance. Therefore, our future work includes developing an architecture that exploits the use of identical weights for different input contexts and batches, under the assumption that the operations are synchronized by layer. With additional sets of the SXE and the VXE, the LPU can support two modes for parameter reuse. First, the multitoken mode that supports the simultaneous execution of multiple input tokens would speed up the initial summarization stage. This mode can reduce latency significantly for user requests with long input tokens. Second, the batch mode that supports different user requests simultaneously would greatly improve throughput, which is essential in high-traffic data centers. The two modes would further increase LPU performance while maintaining its outstanding efficiency and scalability.

In addition, we also considered a hybrid system composed of both the GPU and LPU to cover a broader workload scope. As the GPU excels in other compute-intensive AI workloads (e.g., image processing) and the LPU specializes in the sequential generation of textual content, combining the two systems should effectively handle multimodal workloads, such as text-to-image generation. By overcoming the communication latency between the GPU and the LPU with dataflow optimizations, we expect that the LPU and its area of application can expand to more diverse domains.

REFERENCES

1. A. Vaswani et al., "Attention is all you need," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 6000–6010, doi: 10.48550/arXiv.1706.03762.
2. A. Chowdhery et al., "Palm: Scaling language modeling with pathways," *J. Mach. Learn. Res. (JMLR)*, vol. 24, no. 1, pp. 11,324–11,436, 2023, doi: 10.48550/arXiv.2204.02311.
3. H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Seoul, Korea (South), 2021, pp. 97–110, doi: 10.1109/HPCA51647.2021.00018.
4. T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process., Syst. Demonstrations (EMNLP)*, 2020, pp. 38–45, doi: 10.48550/arXiv.1910.03771.
5. S. Hong et al., "DFX: A low-latency multi-FPGA appliance for accelerating transformer-based text generation," in *Proc. 55th IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2022, pp. 616–630, doi: 10.1109/MICRO56248.2022.00051.
6. G. Neubig, Y. Goldberg, and C. Dyer, "On-the-fly operation batching in dynamic computation graphs," in *Proc. 31st Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 3974–3984, doi: 10.48550/arXiv.1705.07860.
7. N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," 2020, *arXiv:2001.04451*.
8. Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan./Jun. 2015, doi: 10.1109/LCA.2015.2414456.
9. 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, Mar./Apr. 2015, doi: 10.1109/MM.2020.2975185.
10. D. Abts et al., "A software-defined tensor streaming multiprocessor for large-scale machine learning," in *Proc. 49th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2022, pp. 567–580, doi: 10.1145/3470496.3527405.

SEUNGJAE MOON is a cofounder of and a hardware engineer at HyperAccel, Seoul, 06247, South Korea. His research interests include hardware architecture for machine learning model inference and dataflow optimization for memory-intensive applications. Moon received his M.S. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at sj.moon@hyperaccel.ai.

JUNG-HOON KIM is a Ph.D. student at the Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea. His research interest is AI chip design. Kim received his M.S. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at wjdgns7737@kaist.ac.kr.

JUNSOO KIM is a cofounder of and a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI software system. Kim received his M.S. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at js.kim@hyperaccel.ai.

SEONGMIN HONG is a cofounder of and a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Hong received his Ph.D. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at sm.hong@hyperaccel.ai.

JUNSEO CHA is a cofounder of and a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Cha received his M.S. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at js.cha@hyperaccel.ai.

MINSU KIM is a Ph.D. student at the Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea. His research interest is AI software system. Kim received his M.S. degree in computer science from the Korea Advanced Institute of Science Technology. Contact him at mskim@casys.kaist.ac.kr.

SUKBIN LIM is a hardware engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Lim received his M.S. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at sb.lim@hyperaccel.ai.

GYUBIN CHOI is a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI software system. Choi received his B.S. degree in electrical engineering from Purdue University. Contact him at gb.choi@hyperaccel.ai.

DONGJIN SEO is a hardware engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Choi received his M.S. degree in electrical engineering from Sungkyunkwan University. Contact him at dongjin.seo@hyperaccel.ai.

JONGHO KIM is a hardware engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Kim received his B.S. degree in electrical engineering from Seoul National University. Contact him at jongho.kim@hyperaccel.ai.

HUNJONG LEE is a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI software system. Lee received his M.S. degree in computer science from Korea University. Contact him at hj.lee@hyperaccel.ai.

HYUNJUN PARK is a software engineer at HyperAccel, Seoul, 06247, South Korea. His research interest is AI software system. Park received his M.S. degree in electrical engineering from Yonsei University. Contact him at hj.park@hyperaccel.ai.

RYEOWOOK KO is a hardware intern at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Ko received his B.S. degree in electrical engineering from Konkuk University. Contact him at kru0109@hyperaccel.dooray.com.

SOONGYU CHOI is a hardware intern at HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Choi received his B.S. degree in electrical engineering from Yonsei University. Contact him at soongyu1291@hyperaccel.dooray.com.

JONGSE PARK is an associate professor in the School of Computing, Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea. His research interest is AI software system. Park received his Ph.D. degree in computer science from Georgia Institute of Technology. Contact him at jspark@casys.kaist.ac.kr.

JINWON LEE is CTO of HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Lee received his M.S. degree in electrical engineering from Seoul National University. Contact him at jw.lee@hyperaccel.ai.

JOO-YOUNG KIM is CEO of HyperAccel, Seoul, 06247, South Korea. His research interest is AI chip design. Kim received his Ph.D. degree in electrical engineering from the Korea Advanced Institute of Science Technology. Contact him at jy.kim@hyperaccel.ai.