(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
01 May 2025 (01.05.2025)

WIPO | PCT

(10) International Publication Number
WO 2025/090955 A1

(51) International Patent Classification:
G06F 9/50 (2006.01) *G06F 3/045* (2006.01)
G06N 3/063 (2023.01)

(21) International Application Number:
PCT/US2024/053102

(22) International Filing Date:
25 October 2024 (25.10.2024)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
63/593,413 26 October 2023 (26.10.2023) US

(71) Applicant: **GOOGLE LLC** [US/US]; 1600 Amphitheatre Parkway, Mountain View, California 94043 (US).

(72) Inventors: **PIQUERAS, Enrique**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US). **BRADBURY, James Edward Kahn**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US). **SAETA, Brennan**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US). **DOUGLAS, Sholto Francis Alexandre**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, CA 94043 (US). **LEVSKAYA, Anselm**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US). **SCHUH, Parker Edward**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US). **POPE, Reiner**; c/o Google LLC, 1600 Amphitheatre Parkway, Mountain View, California 94043 (US).

(54) **Title:** EFFICIENTLY SERVING MACHINE-LEARNED MODEL COMPUTATIONS WITH HIGH THROUGHPUT AND LOW LATENCY

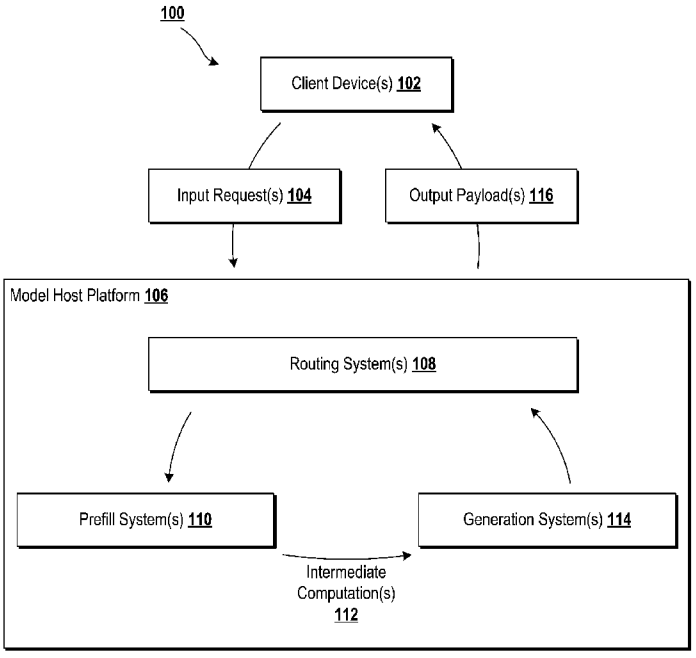


FIG. 1

(57) **Abstract:** An example method includes receiving input requests to process a plurality of input sequences using the machine-learned sequence processing model to generate a plurality of output sequences respectively corresponding to the plurality of input sequences; generating a plurality of initial attention tensors respectively for the plurality of input sequences, wherein: one or more respective initial attention tensors are generated for each respective input sequence in parallel over input elements of the respective input sequence; and the one or more respective initial attention tensors are generated in one or more batches having a first batch size using a prefill system that comprises one or more prefill computing devices and executes one or more layers of the machine-learned sequence processing model; and autoregressively generating, using the plurality of initial attention tensors, a plurality of output elements for each of the plurality of output sequences in one or more batches having a second batch size, wherein: the plurality of output elements are generated

WO 2025/090955 A1

[Continued on next page]

(74) **Agent: NOEL, James H.** et al.; Dority & Manning, P.A., P.O. BOX 1449, Greenville, South Carolina 29602-1449 (US).

(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

using a generation system that comprises one or more generation computing devices.

EFFICIENTLY SERVING MACHINE-LEARNED MODEL COMPUTATIONS WITH HIGH THROUGHPUT AND LOW LATENCY

PRIORITY

[0001] This application claims priority to U.S. Provisional Patent Application No. 63/593,413 (filed October 26, 2023), which is hereby incorporated by reference herein in its entirety.

FIELD

[0002] The present disclosure relates generally to machine learning processes and machine-learned devices and systems. More particularly, the present disclosure relates to improved techniques for hosting and serving machine-learned models for executing computations.

BACKGROUND

[0003] A computer can receive input(s). The computer can execute instructions to process the input(s) to generate output(s) using a parameterized model. The computer can obtain feedback on its performance in generating the outputs with the model. The computer can generate feedback by evaluating its performance. The computer can receive feedback from an external source. The computer can update parameters of the model based on the feedback to improve its performance. In this manner, the computer can iteratively "learn" to generate the desired outputs. The resulting model is often referred to as a machine-learned model.

SUMMARY

[0004] Aspects and advantages of embodiments of the present disclosure will be set forth in part in the following description, or can be learned from the description, or can be learned through practice of the embodiments.

[0005] Example aspects of the present disclosure provide an example method. In some implementations, the example method can include receiving input requests to process a plurality of input sequences using the machine-learned sequence processing model to generate a plurality of output sequences respectively corresponding to the plurality of input sequences. The example method can include generating a plurality of initial attention tensors

respectively for the plurality of input sequences. In the example method, one or more respective initial attention tensors can be generated for each respective input sequence in parallel over input elements of the respective input sequence. In the example method, the one or more respective initial attention tensors can be generated in one or more batches having a first batch size using a prefill system that comprises one or more prefill computing devices and executes one or more layers of the machine-learned sequence processing model. The example method can include autoregressively generating, using the plurality of initial attention tensors, a plurality of output elements for each of the plurality of output sequences in one or more batches having a second batch size. In the example method, the plurality of output elements can be generated using a generation system that includes one or more generation computing devices. In the example method, at least a portion of the machine-learned sequence processing model can be partitioned over the one or more generation computing devices along a batch dimension.

[0006] Example aspects of the present disclosure provide one or more example non-transitory computer-readable media storing instructions that are executable by one or more processors to cause a computing system to perform example operations. In some implementations, the example operations can include implementations of the example method.

[0007] Example aspects of the present disclosure provide an example computing system that includes one or more processors and one or more example non-transitory computer-readable media storing instructions that are executable by one or more processors to cause a computing system to perform example operations. In some implementations, the example operations can include implementations of the example method.

[0008] Other example aspects of the present disclosure are directed to other systems, methods, apparatuses, tangible non-transitory computer-readable media, and devices for performing functions described herein. These and other features, aspects, and advantages of various implementations will become better understood with reference to the following description and appended claims. The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate implementations of the present disclosure and, together with the description, help explain the related principles.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] Figure 1 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0010] Figure 2 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0011] Figure 3 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0012] Figure 4 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0013] Figure 5 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0014] Figure 6 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0015] Figure 7 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0016] Figure 8 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0017] Figure 9 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0018] Figure 10 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0019] Figure 11 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0020] Figure 12 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0021] Figure 13 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0022] Figure 14 is a block diagram of aspects of an example system for implementing example aspects of the present disclosure;

[0023] Figure 15 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0024] Figure 16 is a diagram of aspects of an example partitioning layout of an example system for implementing example aspects of the present disclosure.

[0025] Figure 17 is a diagram of aspects of an example partitioning layout of an example system for implementing example aspects of the present disclosure.

[0026] Figure 18 is a diagram of aspects of an example partitioning layout of an example system for implementing example aspects of the present disclosure.

[0027] Figure 19 is a diagram of aspects of an example partitioning layout of an example system for implementing example aspects of the present disclosure.

[0028] Figure 20 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0029] Figure 21 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0030] Figure 22 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0031] Figure 23 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0032] Figure 24 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0033] Figure 25 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0034] Figure 26 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0035] Figure 27 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0036] Figure 28 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0037] Figure 29 is a chart reporting example measurements of a case study of an example system for implementing example aspects of the present disclosure.

[0038] Figure 30 is a flow chart diagram illustrating an example method for implementing example aspects of the present disclosure;

[0039] Figure 31 is a flow chart diagram illustrating an example method for implementing example aspects of the present disclosure;

[0040] Figure 32 is a flow chart diagram illustrating an example method for training a machine-learned model according to example implementations of aspects of the present disclosure;

[0041] Figure 33 is a block diagram of an example processing flow for using machine-learned model(s) to process input(s) to generate output(s) according to example implementations of aspects of the present disclosure;

[0042] Figure 34 is a block diagram of an example sequence processing model according to example implementations of aspects of the present disclosure;

[0043] Figure 35 is a block diagram of an example technique for populating an example input sequence for processing by a sequence processing model according to example implementations of aspects of the present disclosure;

[0044] Figure 36 is a block diagram of an example model development platform according to example implementations of aspects of the present disclosure;

[0045] Figure 37 is a block diagram of an example training workflow for training a machine-learned model according to example implementations of aspects of the present disclosure;

[0046] Figure 38 is a block diagram of an inference system for operating one or more machine-learned model(s) to perform inference according to example implementations of aspects of the present disclosure;

[0047] Figure 39 is a block diagram of an example networked computing system according to example implementations of aspects of the present disclosure;

[0048] Figure 40 is a block diagram of an example computing device according to example implementations of aspects of the present disclosure; and

[0049] Figure 41 is a block diagram of an example computing device according to example implementations of aspects of the present disclosure.

DETAILED DESCRIPTION

[0050] Generally, the present disclosure is directed to techniques for enhanced parallelization of machine-learned models, such as an autoregressive machine-learned sequence processing model. Example implementations of the present disclosure provide techniques for distributing an execution task over multiple hardware accelerators, which can improve the scalability of machine-learned models and systems. The distribution of the execution task can be adapted for different workloads based on the demands of each workload.

[0051] In an aspect, example implementations of the present disclosure can improve the efficiency and speed of executing and serving results from autoregressive machine-learned sequence processing models (e.g., such as Large Language Models or LLMs) by decoupling different workloads and optimizing the serving configuration for each. Some applications, including interactive workloads like chatbots, involve tight latency constraints. Others, including offline inference for scoring or distillation, emphasize high throughput and low cost per token at any latency. By decoupling different workloads and optimizing the

serving configuration for each, example implementations of the present disclosure can optimize for different targets (e.g., throughput, latency, cost, etc.).

[0052] Traditional infrastructure generally couples prefill and generation workloads together and forces them to run on the same server at the same batch size. More generally, traditional approaches generally failed to appreciate the differences between prefill and generation workloads and the corresponding impacts on the computational resource usage. Combining this with a tight latency constraint on the overall inference can force prefill to run at batch size 1, and this coupling causes decode to also run at batch size 1, resulting in relatively poor processor utilization and low generation throughput.

[0053] Example implementations of the present disclosure can separate the workloads based on a stage of processing input requests. For example, a request to perform inference can request that a model process an input sequence containing an initial set of input elements and generate an output sequence (with one or more output elements) based on the input sequence. Processing the input sequence to understand the input context can be called a "prefill" task. For example, the internal reasoning system(s) of the model can be "prefilled" over this initial set of context before beginning to generate new output elements. Processing the context to generate new output elements can be called a "generation" or "decode" task.

[0054] Example implementations of the present disclosure can separate the workloads of prefilling and autoregressive generation. This decoupling can potentially lead to a significant reduction in inference costs by optimizing a configuration of the model serving architecture for each workload, thus increasing the overall efficiency of the system. The decoupling of workloads can allow for the optimization of overall throughput by changing the serving configuration as needed for each task, leading to improved efficiency and cost savings. For instance, using different batch sizes for prefill and generation can increase efficiency and throughput and reduce latency.

[0055] An example implementation of the present disclosure can interleave prefill and decode programs on the same server, with prefill for each new session occurring during a first interval at a first batch size (e.g., 1) and generation occurring during a second interval at a larger second batch size. The system can alternate between prefill computations and generation computations. This configuration can improve overall throughput while decreasing a latency, in some circumstances. To batch together different sessions, the server could maintain a batch of inference intermediates (e.g., KV caches) in the form of a circular buffer that's updated during prefill and generation. The buffer could maintain a "cursor" representing the current position of decoding. In some examples, this approach can keep

effective batch size—and hence throughput—high throughout prefill and generation. The approach can include splitting the inference job into two tasks that are batched separately and run dynamically based on arrival of requests. This approach can include maintaining decoding intermediates (e.g., a KV cache circular buffer) in accelerator memory between these computations.

[0056] Another example implementation of the present disclosure can use disaggregated prefill and generation systems. This approach can execute prefill tasks and generation tasks on separate pools of servers. An example life cycle of an input request can include arriving at a router, being forwarded to an available prefill server that computes internal context states for the model (e.g., attention tensors, such as a KV cache), and then having the context states sent to a generation server for generation based on the context. The generation system(s) could employ a circular buffer similar to that described above with respect to the interleaving approach, except that instead of pausing generation to perform prefill locally, the prefill updates to the buffer could be received from a prefill system. The generation system(s) can employ a different batch size from the prefill system(s) (e.g., 32, 64, 256, 512, etc.).

[0057] Separating the prefill and generation workloads can also facilitate using different model architectures or model configurations for prefill and generation. For instance, different model partitioning schemes, quantization schemes, etc. can be used in the prefill and generation workflows.

[0058] Further, separating the prefill and generation workloads can allow for different hardware to be used for each task. For example, the hardware selections for each of the prefill tasks and the generation tasks can be selected to optimize resources for the different demands of prefill and decode.

[0059] For example, a prefill task can have a compute bottleneck. For instance, due to high innate parallelism, prefill (e.g., over a number of inputs in parallel) can be performed using a single forward pass through the model, which can demand less memory bandwidth for loading or unloading intermediate computations or model weights. In contrast, a generation task can have a memory bottleneck. For instance, due to autoregressive generation involving multiple forward passes through the model to sample a series of outputs (e.g., that are based on previously-sampled outputs), generation can rely heavily on memory bandwidth for loading and offloading model weights and intermediate computations from compute core(s) (e.g., for each forward pass).

[0060] In this manner, for instance, a pool of hardware devices can be provisioned for optimally servicing input requests. A pool of hardware devices can be homogenous or heterogeneous. For instance, a pool of hardware devices can include computing devices with various specifications. In some cases, some hardware devices can have higher memory bandwidth than others. Some hardware devices can have faster or more compute cores than others.

[0061] Example implementations also provide techniques for partitioning large models across pools of computing devices. For example, large models can have a large memory footprint both due to the trained model parameters as well as the transient state of intermediate computations used during prediction/generation. The model parameters might not fit in the memory of a single accelerator chip. For some models that attend over a context sequence when generating next elements in the sequence, attention tenors for one or more layers can be cached or stored in memory for the duration of generation (e.g., to avoid re-computation at each generation step). This footprint too might not fit within the memory of a single accelerator chip. Such a large memory footprint can give rise to a large amount of memory traffic to load the parameters and intermediate computations from high-bandwidth memory (HBM) into the compute cores for each step, and hence a large total memory bandwidth used to meet a given latency target. For instance, in some implementations, a system can store tensors such as weights and attention tensors in HBM. These tensors can be transferred from HBM to the compute cores of the chip once per forward pass (prefill or generation step) of the model. This takes a certain amount of time. At small batch sizes and sequence lengths, the time to load weights can dominate. At larger batch sizes and sequence lengths (e.g., 2048+ tokens with batch size 512+), the time to load the attention tensors can dominate. For instance, the attention tensor size can be quadratic in sequence length.

[0062] Accordingly, larger models can be partitioned across many accelerator chips to fit in memory. This divides the memory and compute costs over all the chips, but can introduce chip-to-chip communication costs. Generally, model partitioning introduces communication between chips, and different partitioning strategies for a given model involve different patterns and amounts of communication. For a given partitioning layout, the time needed for chip-to-chip communication can decrease less quickly (or not at all) with the number of chips used. Chip-to-chip communication can become a bottleneck as the chip count grows.

[0063] Different partitioning strategies can involve different amounts of memory bandwidth loads. Selecting a partitioning strategy for a given model size can depend on

specific application requirements. A multi-axis tensor partitioning strategy, batch size(s), and chip configuration for a particular application can be based on latency and throughput targets. For example, lower latency can often be achieved with smaller batch sizes, but smaller batch sizes can also result in worse processor utilization, which can result in a higher total cost (in terms of chip-seconds or dollars) per token. If an application requires generating text with long attention contexts, it can substantially increase the inference time. For example, for a 500 B+ parameter transformer model with multihead attention, the attention KV cache grows large: for batch size 512 and context length 2048, the KV cache totals 3 TB, which is 3 times the size of the model's parameters. In some generation workflows, the on-chip memory can load this KV cache from off-chip memory once for every token generated, during which the computational core of the chip can be essentially idle. If an application uses offline inference (e.g., latency is not a concern), the primary goal can be to maximize per-chip throughput (e.g., minimize total cost per token). It can be efficient to increase the batch size because larger batches typically result in better processor utilization. Certain partitioning strategies that are less efficient for small batch sizes can become more efficient as the batch size grows larger.

[0064] In an aspect, example implementations of the present disclosure provide technical solutions for parallel layout and memory optimizations to unlock the scalability of machine-learned models for efficient, low-latency inference of large (e.g., high parameter count) models. Specifically, example aspects of the present disclosure provide a powerful partitioning framework to facilitate model parallel scaling. This framework can identify a best partitioning strategy for a given model size with specific application requirements. This enables intelligent selection of the best multi-axis tensor partitioning strategy, batch size and chip configuration for a given application, in contrast to traditional black-box exhaustive searches over partitioning strategies, which often involve the consumption of large amounts of compute simply to explore the search space. For example, tradeoffs in computational efficiency, latency, and communication cost may change as model size, sequence length, and application requirements for latency and throughput targets change, so having a framework that enables expression of and optimization over different strategies and choices can facilitate improved system selection and implementation on different hardware configurations without necessarily requiring experimental execution of each such different hardware configuration first. For example, the modeling framework provided herein can provide solutions for the technical problem of achieving a desired latency target given a throughput specification, a number of chips, and a particular chip topology.

[0065] For instance, in an example case study, a 540 B parameter dense model running on 64 TPU v4 chips achieved a low-batch-size latency of 29 ms per token during generation (with int8 weight quantization) and a 76% MFU during large-batch-size processing of input tokens while supporting a large context length of 2048 tokens. For an interactive application such as a chatbot running on PaLM 540 B with int8 weights, the case study implementation on 64 TPU v4 chips can process 64 tokens of text from a user, consult a cached conversation history of 1920 tokens, and generate a 64-token response in a total of 1.9 seconds. For an offline throughput-oriented application, the case study implementation can process 1984 tokens of input and generate 64 tokens of output, for huge numbers of examples, with an overall FLOPS efficiency of 73%. More details are provided herein that document the technical improvements achieved using example partitioning techniques described herein.

[0066] Example implementations of the present disclosure can achieve numerous technical effects and benefits related to the enhanced parallelization of an autoregressive or nonautoregressive machine-learned sequence processing model. For instance, the technology can lead to increased energy efficiency during the operations of machine-learned models, contributing to an improvement in the function of computers implementing such models. This improved energy efficiency can manifest in several ways, such as in the form of more energy-efficient runtime execution or inference. In practical terms, this could mean less energy is required to maintain the model in memory or to perform calculations within the model. Additionally, increased energy efficiency could allow for more tasks to be completed within a given energy budget, such as processing larger quantities of tasks, dealing with more complex tasks, or performing the same task but with a higher degree of accuracy or precision.

[0067] Another example can be improved efficiency and speed of executing and serving results from machine-learned models. By decoupling different workloads and optimizing the serving configuration for each, the technology can reduce inference costs and increase the overall efficiency of the system. For example, the present disclosure can separate the tasks of prefilling and autoregressive generation, which are traditionally coupled in autoregressive machine-learned model inference. This decoupling can potentially lead to a reduction in inference costs by optimizing a configuration of the model serving architecture for each workload. This results in a more efficient use of computational resources, leading to the ability to process larger amounts of data in a shorter time, which can be particularly beneficial in time-sensitive applications such as real-time language processing or prediction models.

[0068] Another benefit of example implementations described in the present disclosure is its potential to make better use of hardware resources. By separating prefill and generation workloads, different hardware can be used for each task. For instance, a prefill task that has a compute bottleneck can be allocated to hardware with more compute cores, while a generation task that has a memory bottleneck can be assigned to hardware with faster memory or higher memory bandwidth. This improved hardware utilization can lead to higher performance and cost savings.

[0069] Furthermore, example implementations disclosed in the present disclosure can also enable new functionality in a computing system. For example, through its use of disaggregated prefill and generation systems, example implementations can execute prefill tasks and generation tasks on separate pools of servers. This can allow for more dynamic and flexible system configuration, potentially enabling more complex or large-scale tasks to be performed that would be challenging or impossible with a traditional coupled approach.

[0070] Another technical effect of example implementations of the present disclosure is increased energy efficiency in performing operations using machine-learned models, thereby improving the functioning of computers implementing such models. For instance, example implementations can provide for more energy-efficient runtime execution or inference. In some scenarios, increased energy efficiency can provide for less energy to be used to perform a given task (e.g., less energy expended to maintain the model in memory, less energy expended to perform calculations within the model, etc.). In some scenarios, increased energy efficiency can provide for more task(s) to be completed for a given energy budget (e.g., a larger quantity of tasks, more complex tasks, the same task but with more accuracy or precision, etc.).

[0071] In another example aspect, example implementations can provide for more energy-efficient training operations or model updates. In some scenarios, increased energy efficiency can provide for less energy to be used to perform a given number of update iterations (e.g., less energy expended to maintain the model in memory, less energy expended to perform calculations within the model, such as computing gradients, backpropagating a loss, etc.). In some scenarios, increased energy efficiency can provide for more update iterations to be completed for a given energy budget (e.g., a larger quantity of iterations, etc.).

[0072] In this manner, for instance, the improved energy efficiency of example implementations of the present disclosure can reduce an amount of pollution or other waste associated with implementing machine-learned models and systems, thereby advancing the field of machine-learning and artificial intelligence as a whole. The amount of pollution can

be reduced in toto (e.g., an absolute magnitude thereof) or on a normalized basis (e.g., energy per task, per model size, etc.). For example, an amount of CO2 released (e.g., by a power source) in association with training and execution of machine-learned models can be reduced by implementing more energy-efficient training or inference operations. An amount of heat pollution in an environment (e.g., by the processors/storage locations) can be reduced by implementing more energy-efficient training or inference operations.

[0073] Various example implementations are described herein with respect to the accompanying Figures.

[0074] Figure 1 is a block diagram of an example ecosystem implementing a disaggregated machine-learned sequence processing model architecture. One or more client devices 102 can send one or more input request(s) 104 to a model host platform 106. Model host platform 106 can include one or more routing system(s) 108 that can orchestrate processing of the input requests 104. For example, routing system(s) 108 can pass data from input request(s) 104 to one or more prefill system(s) 110 to perform a prefill task (e.g., to prefill latent context data for generating content based on the latent context data). Prefill system(s) 110 can pass these intermediate computation(s) 112 (e.g., directly or via routing system(s) 108) to one or more generation system(s) 114 to perform a generation task. Generation system(s) 114 can process the intermediate computation(s) 112 (e.g., and other data) to generate output(s) for servicing input request(s) 104. Model host platform 106 can serve output payload(s) 116 to client device(s) 102 in response to input request(s) 104.

[0075] Client devices 102 can be or include any type of computing devices that are capable of interacting with the model host platform 106. These devices could include, but are not limited to, personal computers, servers, virtual machines, laptops, tablets, smartphones, wearable technology, or any other device capable of sending and receiving data. Client devices 102 can use various applications or interfaces to send input requests 104 to the model host platform 106. These input requests can be related to a variety of tasks, such as interactive workloads like chatbots or offline inference for scoring or distillation.

[0076] Client devices 102 can also receive output payloads 116 from the model host platform 106, which include the results of the processed requests. A client device can be used by an individual user or multiple users to send input requests to the machine-learned sequence processing model for processing input sequences and generating corresponding output sequences.

[0077] Client device(s) 102 can include software or applications that facilitate the interaction with the model host platform 106. This software can handle the formation of input

requests, the transmission and reception of data to and from the model host platform 106, and the presentation of the output payloads to the user. Client device(s) 102 can be a part of model host platform 106 or can be implemented on a same platform or system that also implements model host platform 106.

[0078] Input request(s) 104 can be a series of commands or data sent by one or more client devices 102 to the model host platform 106. These requests can be sent via a variety of protocols or methods, such as HTTP requests, API calls, or any other form of data transmission. They represent the initial interaction between the client device(s) and the model host platform, initiating the processing sequence in the system.

[0079] Each input request 104 can contain an input sequence that is intended for processing by the machine-learned sequence processing model. The input sequence may contain various forms of data, such as text, numbers, images, or any other type of data that the model is trained to process. For instance, in the context of a language model, the input sequence could represent a string of text.

[0080] Input request(s) 104 may be generated as a result of user interaction with an application or interface on the client device(s) 102. For example, a user could input a text-based query into a search engine, which could then form the basis of an input request. Alternatively, input request(s) 104 could be generated automatically by software on the client device(s) without direct user input. A client device 102 could capture an image, record a sketch, record audio, etc. that can be processed and sent in input request(s) 104.

[0081] The nature and format of input request(s) 104 can depend on the specific application or function being performed. For instance, in the context of a chatbot, the input request 104 could include the text of the user's query or statement. For offline inference tasks, the input request 104 could contain a batch of data to be processed or scored.

[0082] Input request(s) 104 can also contain additional parameters or metadata that can guide the processing of the input sequence. These could include, for instance, the desired output format, the level of detail or complexity required in the output, or any other user preferences or requirements. This information can be used to customize the processing of the input sequence, thereby allowing the system to adapt to the specific needs of different users or applications.

[0083] In some instances, multiple input request(s) 104 can be sent in parallel, allowing for the processing of multiple input sequences simultaneously. This can be useful in scenarios that require high throughput or efficiency, such as in large-scale data processing tasks or real-time applications.

[0084] Each input request 104 can be associated with a unique identifier that allows the model host platform 106 to track the processing of the request and to associate the results of the processing with the correct client device 102. The unique identifier can also be used to manage the processing of multiple input requests 104, for example, by allowing the model host platform 106 to prioritize certain requests or to handle requests in a specific order.

[0085] Input request(s) 104 can be generated by an application or interface running on the client device 102. This application or interface can provide a user-friendly way for users to generate requests, for example by allowing users to enter data through a graphical user interface, or by automatically generating requests based on user actions or system events. The application or interface can also handle the communication with the model host platform 106, including sending the input request(s) 104 and receiving the output payload(s) 116.

[0086] Input request(s) 104 can be provided via an application programming interface (e.g., API) that enables software to communicate with model host platform 106 in a standardized manner.

[0087] Model host platform 106 can be a computing platform or system that hosts and manages the execution of a machine-learned model, such as a machine-learned sequence processing model. This platform can be implemented on one or more servers or other computing devices, which could be located in a single physical location or distributed across multiple locations. Model host platform 106 can be implemented on a single computer, a network of computers, or across multiple data centers. Model host platform 106 can use various forms of hardware, such as CPUs, GPUs, TPUs, or other types of processing units. Model host platform 106 can be designed to handle high volumes of input requests and to process these requests quickly and efficiently. The platform can host one or more machine-learned sequence processing models, such as large language models (LLMs), image processing models, or any type of machine-learned models (e.g., autoregressive machine-learned models).

[0088] The model host platform 106 can include various systems and components designed to efficiently process the input requests. For example, it can include one or more routing systems 108 that manage the flow of data between different components of the platform. These routing systems can receive the input requests, forward them to the appropriate processing systems (such as prefill systems 110 and generation systems 114), and route the resulting outputs back to the client devices.

[0089] Model host platform 106 can be configured to handle a wide variety of input request types and to generate a range of different outputs. The configuration of the platform

can vary based on the nature of the machine-learned sequence processing model that it hosts. For example, if the model is a language model, the platform could be configured to handle input requests containing text data and to generate text-based outputs. If the model is a multimodal model, then the platform could be configured to handle input requests containing multiple different modalities of data. The platform could be configured to assemble multiple different modalities of data to augment data received in input requests 104.

[0090] Model host platform 106 can be configured to optimize the processing of different types of workloads. For instance, it could be configured to prioritize latency-sensitive tasks for interactive workloads like chatbots, to maximize throughput for large-scale offline inference tasks, maximize efficient computation to decrease costs, or some combination of such objectives. To achieve this, model host platform 106 can use different serving configurations for different types of workloads. For example, model host platform 106 can separate the tasks of prefilling and autoregressive generation, allowing these tasks to be optimized independently based on the specific requirements of each workload.

[0091] Model host platform 106 can also be configured to manage the hardware resources used by the system, such as capabilities for selecting the appropriate hardware for each task and for partitioning large models across multiple computing devices. Partitioning a model can involve different partitioning strategies that involve different patterns and amounts of communication between the devices. For instance, larger models that use high memory amounts (e.g., to store weights, activations, intermediate computations) can be partitioned across many accelerator chips to fit in memory, dividing the memory and compute costs over all the chips.

[0092] Model host platform 106 can include a variety of computing devices with different capabilities, such as processors with different numbers of cores or memory devices with different bandwidths. Model host platform 106 can also include software components designed to manage and optimize the use of these hardware resources, such as load balancers, task schedulers, or memory management systems.

[0093] Model host platform 106 can use a coupled configuration where prefill and generation workloads run on the same server at the same batch size. This traditional configuration can achieve low latency, sometimes at the cost of lower processor utilization and throughput.

[0094] Model host platform 106 can use disaggregated serving infrastructure configurations. For instance, model host platform 106 can separate the prefill tasks and generation tasks onto different pools of servers (e.g., prefill systems 110, generation systems

114). This could allow for more efficient use of hardware resources, as different tasks could be assigned to servers that are optimized for those specific tasks. For example, prefill tasks could be assigned to servers with more compute cores, while generation tasks could be assigned to servers with faster memory or higher memory bandwidth. In an example disaggregated configuration, a router system 108 can forward data from an incoming input request to an available prefill server that computes internal context states for the model, such as attention tensors or KV caches. These context states can then be sent to a generation server for content generation based on the context. This disaggregated approach can increase overall throughput and allow for different hardware to be used for each task, leading to improved efficiency and cost savings.

[0095] Model host platform 106 can use interleaved serving infrastructure configurations. In this case, prefill and decode programs can be interleaved on the same server, with prefill for each new session occurring during a first interval at a first batch size, and generation occurring during a second interval at a larger second batch size.

[0096] Model host platform 106 can use a combination of interleaved and disaggregated serving configurations, depending on the specific requirements of each workload. For instance, model host platform 106 can use disaggregated configurations for some tasks and interleaved configurations for others. Model host platform 106 can dynamically switch between different configurations based on the current workload.

[0097] Model host platform 106 can include various security features designed to protect the data and resources of the system. These could include, for example, encryption capabilities for securing the data transmitted between the platform and client devices 102, access control features for managing who can access the system and what actions they can perform, and intrusion detection systems for identifying and responding to potential security threats.

[0098] Routing system(s) 108 can be applications, sub-systems, or services within model host platform 106 that can distribute input request(s) 104 to the appropriate processing systems, such as prefill system(s) 110 and generation system(s) 114. Routing system(s) 108 can also control the flow of data within the platform and manage the responses from the processing systems. Routing system(s) 108 can serve as the primary point of coordination by which requests from client device(s) 102 are processed and then sent to the appropriate destinations.

[0099] Routing system(s) 108 can be responsible for orchestrating the processing of input request(s) 104 using the correct prefill systems and generation systems based on the

nature and requirements of each request. When an input request arrives, routing system(s) 108 can forward the request to an available prefill server that computes internal context states for the model, such as attention tensors or KV caches. Once the prefill task is completed, the resulting context states can be sent, directly from the prefill server or via the routing system(s), to a generation server for content generation based on the context.

[0100] Routing system(s) 108 can implement scheduling algorithms to queue prefill tasks and generation tasks. These algorithms can take into account various factors, such as the current load on the different processing systems, the expected processing times for the different tasks, the priority levels of the different tasks, and the specific resource requirements of the tasks. For example, if a prefill task requires more computational power (e.g., long input sequence length), it might be queued for a server with more computing power, a larger on-processor memory, etc. Similarly, if a generation task is expected to use more resources (e.g., to service a request for long-form content generation as compared to answering a yes/no question), it can be queued for a server with a high-memory bandwidth.

[0101] In some examples, sequence lengths can be or include lengths of 64, 128, 256, 512, 1024, 2048, 8192, up to 32768 or higher, such as up to 43,000. Various example sequence lengths are presented herein.

[0102] In some examples, batch sizes can be or include 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, etc. Various example sequence lengths are presented herein.

[0103] In some examples, number of chips used can include 2, 4, 8, 16, 32, 64, 128, 256, or more, such as up to thousands of chips or more.

[0104] Routing system(s) 108 can maintain queues of prefill and generation tasks together or separately. For instance, when an input request 104 arrives, routing system(s) 108 can add a new prefill task to the prefill queue. Similarly, when a prefill task completes, routing system(s) 108 can add a new generation task to the generation queue. Routing system(s) 108 can optimize the queuing of these tasks based on various factors, such as the current load on the prefill and generation systems, the priority of different tasks, or the latency requirements of different client device(s) 102 (e.g., different service accounts associated with client device(s) 102).

[0105] Routing system(s) 108 can queue prefill tasks based on the order of arrival of input requests or according to a priority scheme. For instance, some requests may have a higher priority due to their time-sensitive nature, and routing system(s) 108 can push these requests to the front of the queue for immediate processing. Generation tasks can also be queued based on the order of arrival or according to a priority scheme.

[0106] Routing system(s) 108 can work in continuous dialogue with prefill system(s) 110 and generation system(s) 114 to monitor the status of tasks, receive updates, and redistribute tasks as needed. In the case of multiple input requests, routing system(s) 108 can maintain a client connection while the input requests are queued and processed.

[0107] In addition, routing system(s) 108 can incorporate system conditions, such as a backlog of requests or system overload, into task distribution and prioritization processes. For example, when model host platform 106 is overloaded, routing system(s) 108 can prioritize urgent or high-priority tasks, delay less urgent tasks, or initiate load shedding if the system capacity is exceeded.

[0108] Routing system(s) 108 can handle error scenarios, such as the failure of a prefill or generation server. In such cases, the routing system could reroute the pending tasks from the failed server to another available server in order to ensure the continued processing of tasks.

[0109] Routing system(s) 108 can also handle the task of sending the generated output sequences back to client devices 102. This can involve packaging the output sequences into an appropriate format, sending the output sequences over the network, and ensuring that the output sequences are correctly received by the client devices.

[0110] The flexible configuration provided by routing system(s) 108 can help to optimize the utilization of system resources, improve the system's response time, and make it more resilient to failures or changes in the workload, enhancing the overall efficiency and performance of model host platform 106.

[0111] Prefill system(s) 110 can be a component or subsystem of model host platform 106 tasked with the initial processing of input sequences received in the form of input requests. Prefill system(s) 110 can be designed to perform prefill tasks, which can involve processing input sequences (e.g., from input request(s) 104) to generate intermediate computations 112 so that the internal reasoning system of the model can understand the context before beginning to generate new output elements.

[0112] Prefill system(s) 110 can include one or more generation servers, each of which can be implemented on separate computing devices or clusters of devices. These devices can be equipped with specific hardware, such as CPUs, GPUs, TPUs, or other types of processors necessary for effectively executing prefill tasks.

[0113] A prefill task can involve processing an input sequence (e.g., a string of text, a sequence of image patches, a series of multimodal datatypes, etc.) to generate an initial set of attention tensors. These attention tensors can capture the relationships between different parts

of the input sequence and can be used by the model to understand the contextual information of the input sequence. For example, some attention tensors within transformer model architectures can be referred to as a Key-Value (KV) cache.

[0114] For instance, in an autoregressive transformer-based content generation model, prefill system(s) 110 can generate the KV cache of one or more transformer layers in parallel over an input prompt sequence. The generated KV cache can include information about every token in the input sequence and its relation to every other token. This processing stage can be done using a single forward pass through the transformer model and can be highly parallelizable due to high innate parallelism. The KV cache can be computed by the self-attention layer of the transformer model. A self-attention layer can create a set of key and value pairs for each token in the input sequence. These key and value pairs can be used to calculate an attention score for each possible pair of tokens, reflecting how much each token should attend to every other token when generating a next output token(s).

[0115] Generating the KV cache in parallel over the prompt sequence can be advantageous in terms of efficiency, as this computation can occur independently for each token, allowing prefill system(s) 110 to process the entire sequence at once with one forward pass through the model. This parallelized approach can significantly reduce the amount of time required to prefill the model, resulting in faster response times for the input request. For instance, model parameters, weights, activations, etc. can be loaded to a processing core once per forward pass. By performing prefill in one forward pass, prefill system(s) 110 can perform more computations with fewer loads/unloads. In this manner, for instance, memory bandwidth might not be a bottleneck for prefill system(s) 110.

[0116] Once intermediate computations 112 are generated, intermediate computations 112 can be transferred from the prefill device(s) to generation device(s) for generation tasks based on intermediate computations 112. This transfer can be initiated and managed by routing system(s) 108. Intermediate computations 112 can be stored or cached in memory for later use during subsequent generation process or subsequent prefill tasks. For instance, for incremental prefill, intermediate computations 112 can be used to jump-start prefill such that prefill system(s) 110 can simply perform prefill on an incremental addition to an input sequence and concatenate an incremental additional intermediate computation set with intermediate computations 112 to obtain an updated set of intermediate computations 112. For example, in a multi-turn session environment (e.g., for a chatbot application), incremental prefill can improve efficiency of prefill system(s) 110. For incremental prefill configurations, the machine-learned sequence processing model(s) can be configured for causal prefill (e.g.,

prefill performed with attention mask blocking attention over later elements in an input sequence).

[0117] Prefill system(s) 110 can be configured to optimize prefill tasks based on workload requirements, server specifications, properties of the input data, and other considerations. For instance, for workloads that prefer high throughput, prefill system(s) 110 can be configured to process larger batches of data at once. Similarly, in situations where minimal latency is preferred, prefill system(s) 110 can be configured to prioritize the quick processing of smaller data batches.

[0118] Prefill system(s) 110 can operate on separate computing devices (e.g., chips, servers, virtual instances, etc.) than those performing generation tasks. This disaggregation can allow for the independent optimization of each workload, potentially leading to cost savings and improved performance. For instance, prefill system(s) 110 could be located on servers with high computational power, given that prefill tasks could have a compute bottleneck because of inherent parallelism, while generation tasks, which can have a memory bandwidth bottleneck, can be performed on servers equipped with faster memory or higher memory bandwidth.

[0119] Prefill system(s) 110 can operate with a prefill batch size. The prefill batch size can be selected for optimal performance of prefill system(s) 110. The prefill batch size can indicate how many different input sequences are processed by the model or model partitions in each forward pass.

[0120] Prefill system(s) 110 can operate with a prefill batch size of one. For example, prefill tasks can be highly parallelizable, such that prefill system(s) 110 can perform prefill tasks in parallel over the length of an input sequence (e.g., in parallel for each element in the sequence). In this manner, for example, an "effective batch size" can be a length of the input sequence. Accordingly, even when processing with a prefill batch size of one (e.g., one input sequence per forward pass), the prefill task can efficiently use computing resources by operating in parallel over the length of the input sequence. In some circumstances, the prefill task can be compute-bound even at batch sizes of one, so increasing the batch size might not significantly increase efficiency or throughput, depending on the hardware used. It is to be understood, however, that larger prefill batch sizes can be used as desired to optimize for efficiency, throughput, latency, etc. for a particular hardware configuration.

[0121] Intermediate computation(s) 112 can be data or information that is generated as a result of prefill system(s) 110 processing input sequences obtained based on input request(s) 104. These intermediate computations can serve as the basis for subsequent

processing tasks, such as the autoregressive generation of new output elements in the sequence.

[0122] One example of intermediate computation(s) 112 is a Key-Value (KV) cache that can be produced by prefill system(s) 110 when processing data in a transformer-based machine-learned sequence processing model. The KV cache can include vectors that represent an understanding of the input sequence at each step in the sequence. The Key (K) and Value (V) pairs in the KV cache can represent self-attention scores and feature representations for each token (for instance, a word or a word piece in a text sequence) in the input sequence. These Key-Value pairs can be used during the generation task by the transformer model to compute the attention scores for obtaining a new output element, which aid in determining the relevance or context of one token to another within the input sequence. The KV cache can be updated dynamically as each new token is generated in the sequence.

[0123] Other examples of intermediate computation(s) 112 can include feature maps in convolutional neural networks (CNNs), hidden states in recurrent neural networks (RNNs), latent vectors in autoencoders, or any other type of data or information that a model generates as it processes an input sequence.

[0124] Intermediate computation(s) 112 can be stored in memory during the processing of the input sequences and can be used in various ways to support the generation of output sequences. For example, they can be used as input to further layers of the model, combined with other intermediate computations, or used to update the model's internal state. Intermediate computation(s) 112 can also be sent to different computing devices for further processing, routed back to the client device, saved for future use, or used in various other ways depending on the requirements of the specific machine-learned sequence processing model and the specific use case.

[0125] Furthermore, intermediate computation(s) 112 can be used to manage and optimize the processing of the model. For instance, they can be used to determine a batch size or an expected output length for processing tasks. Intermediate computation(s) 112 can be stored in a buffer (e.g., a circular buffer) for reusing across different steps of the generation procedure. An efficient management of intermediate computations can reduce the number of computations, saving time and resources. Management of intermediate computation(s) 112 can include buffering strategies, efficient memory usage strategies, and compute-reuse strategies that can lead to efficient use of hardware resources and reduction of the computational overhead.

[0126] Generation system(s) 114 can be or include components or subsystems of model host platform 106 that use intermediate computation(s) 112 generated by prefill system(s) 110 to generate output sequences. For example, generation system(s) 114 can perform the task of autoregressively generating output elements for each output sequence based on intermediate computation(s) 112, which can include a plurality of initial attention tensors and other pre-computed information.

[0127] Generation system(s) 114 can include one or more generation servers, each of which can be implemented on separate computing devices or clusters of devices. These devices can be equipped with specific hardware, such as CPUs, GPUs, TPUs, or other types of processors for effectively executing generation tasks.

[0128] In some example implementations, generation tasks can involve a multitude of forward passes through the model to sample a series of outputs. As such, generation tasks can thus use large amounts of memory bandwidth for loading and offloading model weights and intermediate computations from the compute cores of the device.

[0129] Generation system(s) 114 can use various different strategies to generate the output sequence. Generation system(s) 114 can use intermediate computation(s) 112, such as a KV cache in transformer-based models, to generate the output elements one at a time.

[0130] Although some generation tasks can be serial (e.g., autoregressive generation that has data dependencies on prior iterations), multiple generation tasks can be batched together for efficiency. The chosen batch size for generation tasks can be optimized based on a variety of factors. For instance, larger generation batch sizes can be selected to maximize throughput and computational efficiency for some hardware configurations. However, when latency below a particular threshold is desired (e.g., for real-time interactive applications like chatbots), generation tasks can be run on smaller generation batch sizes or even on individual sessions to meet these requirements.

[0131] The generation batch size(s) can be different from a prefill batch size. The generation batch size(s) can be larger than a prefill batch size.

[0132] Generation system(s) 114 can be configured to use different model architectures or configurations for generating output based on the pre-filled context. For instance, in a text-based model, they might generate a continuation of a text prompt, complete a sentence, generate a paragraph, or write an entire document. Depending on the specific use case and the characteristics of the input sequence, the generated output can vary significantly in length and complexity.

[0133] Generation system(s) 114 can be scalable for handling various workloads. For example, if a large number of generation tasks are predicted or observed, additional generation servers can be provisioned to handle the load, allowing model hosting platform 106 to adapt dynamically to changes in demand.

[0134] Once the output sequence has been generated, it can be returned to client device(s) 102 via routing system(s) 108, often packaged into an appropriate format as a part of output payload(s) 116. Payloads can be streamed piece-by-piece or delivered upon completion.

[0135] Output payload(s) 116 can be the result or output generated by model host platform 106 in response to input request(s) 104. Output payload(s) 116 can be data that is sent back to the client devices 102 after input requests 104 have been processed by the machine-learned model. Output payload(s) 116 can include the output sequences generated based on input sequences or any other type of data produced by the model during its processing of input requests 104. Examples of output sequences could be text sequences, image sequences, or sequences of multiple or other types of data, depending on the type of model being used.

[0136] Output payload(s) 116 can vary in size and format, depending on the specific characteristics of the processed requests. For example, a text-generation task might produce an output payload that contains a sequence of words, while an image-processing task might produce an output payload that contains image data.

[0137] Output payload(s) 116 can be filtered before being sent to the client device(s) 102 based on various considerations. For instance, an output filter can be applied to determine whether content generated by the model follows certain guidelines or standards, such as content moderation rules, user-specific preferences, or legal regulations. Similarly, an output filter can be applied to the generation system output to optimize the output for the client device's requirements or limitations, such as screen size, processing power, or bandwidth. For example, the generation system(s) 114 might generate text with intricate detail, but a summarization filter might be applied to the content to return a condensed version in the output payload(s) 116. Intermediary buffering can be implemented for review of generation outputs before sending as part of output payload(s) 116.

[0138] Output payload(s) 116 can be communicated to client devices 102 in various formats. For example, output payload(s) 116 can be sent as a single block of data, split into multiple chunks, or streamed in real time as they are generated by the model. Streaming can be useful to provide low latency responses (e.g., low latency to start of output sequence),

such as interactive chat bots or real-time content generators. Multiple output payload(s) 116, generated based on multiple input requests, can be communicated simultaneously or in staggered manner depending on the processing and communication capabilities of client device(s) 102 and the specific application or use case.

[0139] Output payload(s) 116 can include additional information along with the generated output sequences, such as confidence scores, attention maps, or other types of metadata. Additional data can include metadata about the processing of the input request, status or error messages, or other information. This additional information can provide further insight into the operations of the machine-learned model, facilitating better understanding of the generated output.

[0140] Figure 2 is a block diagram depicting additional details of example implementation(s) of model host platform 106. Routing system(s) 108 process data from input request(s) 104 and can pass input element(s) 202 to prefill system(s) 110. Prefill system(s) 110 can implement the machine-learned sequence processing model using prefill computing device(s) 204 that execute model partition(s) 206. Prefill system(s) 110 can process input elements 202 to generate intermediate computation(s) 208. Routing system(s) 108 can pass intermediate computation(s) 208 directly to generation system(s) 114. Routing system(s) 108 can shard/reshard intermediate computation(s) 208 to obtain (re)sharded intermediate computation(s) 210. (Re)sharded intermediate computation(s) 210 can pass to generation system(s) 114. Generation system(s) 114 can process (re)sharded intermediate computation(s) 210 using the machine-learned sequence processing model executing on generation computing device(s) 212 in one or more model partition(s) 214. Generation system(s) 114 can return output element(s) 216.

[0141] Input element(s) 202 can be the individual pieces of data within an input sequence that a machine-learned sequence processing model processes. These input elements can vary widely, depending on the nature of the model and the specific task it is designed to perform.

[0142] For instance, in the context of a natural language processing model, input element(s) 202 can be or represent individual words, sentences, or paragraphs in a text document. Input element(s) 202 can be smaller units of text such as characters, morphemes, or n-grams. These input elements can be processed and analyzed by the model to understand the structure and meaning of the text.

[0143] For an image processing model, input element(s) 202 can be tokens that represent individual pixels or regions (e.g., patches) of an image. Each pixel can be

represented as a combination of red, green, and blue (RGB) color values, or in other color spaces such as YUV or HSV. Regions of an image can be represented using various techniques such as convolutional patches (e.g., patches of a feature map generated by one or more layers of a convolutional neural network), superpixels, or other segmentation methods.

[0144] In the case of a speech recognition model, input element(s) 202 can be or represent individual phonemes, syllables, or words in a spoken language. These elements can be extracted from audio data using various feature extraction techniques such as Fourier transforms, Mel-frequency cepstral coefficients (MFCCs), or deep neural network-based methods.

[0145] For a multimodal model, which processes multiple types of data simultaneously, input element(s) 202 can be a combination of different types of data elements. For example, in a model designed to generate captions for images, the input elements could be a combination of text data (such as existing captions or metadata) and image data (such as pixels or image features).

[0146] In the context of a time series analysis model, input element(s) 202 can be individual data points in a time series. These data points can represent various types of information, such as stock prices, weather conditions, or sensor readings, recorded at specific points in time.

[0147] For a model designed to process genetic data, input element(s) 202 can be individual nucleotides, codons, or genes in a DNA or RNA sequence. These elements can be analyzed by the model to perform tasks such as gene prediction, variant identification, or phylogenetic analysis.

[0148] In some cases, input element(s) 202 can be represented using various data types or formats, such as integers, floating-point numbers, strings, arrays, tensors, or other data structures. For example, input element(s) 202 can be embedded representations of data, such as embedded representations of text tokens, image tokens, audio tokens, other data, etc.

[0149] Input element(s) 202 can be derived or otherwise obtained from input request(s) 104. For instance, an input request could contain a text document, and the input elements could be the individual words, sentences, or paragraphs within that document. In another example, if the input request contains an image, the input elements could be the individual pixels or sections of the image. In cases where the input request involves audio data, the input elements might be segments of the audio waveform or features extracted from it. Input request(s) 104 can contain multiple types of data, such as, for example, text and images together, in which case input element(s) 202 can include elements from each type of

data. Furthermore, these elements could undergo preprocessing steps, such as tokenization, normalization, feature extraction, or encoding, before being fed into the model.

[0150] Prefill computing device(s) 204 can be or include a range of hardware systems that are used to perform prefill tasks in a machine-learned sequence processing model. Prefill computing device(s) 204 can include specific hardware accelerators or other components that are designed to improve the performance of prefill tasks. These could include, for example, memory devices with high bandwidth, storage devices with fast read/write speeds, or specialized processing units for tasks like matrix multiplication or convolution. Prefill computing device(s) 204 can include CPUs, GPUs, TPUs, ASICs, FPGAs, or combinations thereof.

[0151] Prefill computing device(s) 204 can be standalone devices, or they can be part of a larger system. For example, prefill computing device(s) 204 can be installed in a server, a workstation, a laptop, mobile device, or another type of computer. Alternatively, they could be part of a distributed computing system, where tasks are distributed across multiple devices in a network.

[0152] Model partition(s) 206 can be various sections or subsets of the machine-learned sequence processing model that are independently implemented and executed on different computing devices. These partitions can be created based on various characteristics of the model, such as the layer structure, the data flow, the computational requirements, or other factors. In the case of models that are too large to fit into the memory of a single device, model partition(s) 206 can be created to distribute the model across multiple devices. This could involve partitioning the model's parameters, its intermediate computations, or both. The partitioning could be done in a way that decreases or minimizes communication costs between devices, such as by ensuring that each device only needs to communicate with a small number of neighboring devices.

[0153] For example, in a deep neural network model, model partition(s) 206 could correspond to individual layers or groups of layers or subportions of layers within the network. Each layer could be implemented and executed on a separate computing device, allowing for parallel processing of different parts of the input data. Different portions of a layer can be implemented and executed on a separate computing device.

[0154] For a convolutional neural network (CNN), model partition(s) 206 can contain various sets of model weights for convolutional layers, pooling layers, fully connected layers, or other components of the model. Each partition could be responsible for a computation or

part of a computation in the processing of input data, such as applying convolutional filters, pooling features, or making final predictions.

[0155] For a transformer-based model, model partition(s) 206 can be sets of model weights for individual transformer blocks or layers. Each transformer block could contain components such as attention heads, feed-forward networks, normalization layers, or other components of the model. Each partition could be responsible for a computation or part of a computation of a stage in the processing of the sequence data, such as attending to different parts of the input sequence, transforming the attended features, or normalizing the outputs.

[0156] In another example, model partition(s) 206 can be created based on the data flow in the model. For instance, in a pipeline parallelism approach, different stages of the processing pipeline could be implemented on different devices. For example, one device could handle the pre-processing of the data, another device could implement the main processing tasks, and yet another device could handle post-processing or output generation.

[0157] Model partition(s) 206 can also be defined along different axes of the model's architecture. For instance, a model can be partitioned along the batch axis, the sequence axis, the feature axis, or a combination thereof. This type of partitioning can allow the model to handle larger batches of data, longer sequences, or higher-dimensional features than would be possible with a single device.

[0158] Model partition(s) 206 could also be created based on the computational requirements of different parts of the model. For instance, if certain parts of the model require high computational power while others require less, those parts could be partitioned and allocated to different devices according to their computational needs. This could lead to more efficient use of the available hardware resources. In some cases, model partition(s) 206 could be dynamically adjusted based on the current workload or other factors.

[0159] Model partition(s) 206 can be implemented using data parallelism. Example data parallelism approaches can include different partitions of the model processing different partitions of the data (e.g., input data, intermediate activations, etc.) simultaneously. In the context of a machine-learned sequence processing model, data parallelism could involve splitting the input sequence into several smaller sequences and assigning each smaller sequence to a different model partition 206 (which can be replicas). Each model partition can process its assigned data independently of the others. After all partitions have completed their processing, the results can be aggregated to produce the final output. This approach can increase the throughput of the model and enable it to handle larger input sequences. For instance, if the model is a neural network, each partition could correspond to a copy of a

given portion of the network, and each partition could process a different subset of the input data.

[0160] Model partition(s) 206 can be implemented using tensor parallelism. Example tensor parallelism approaches can involve splitting the model's computations across multiple devices. Rather than each device processing a different subset of the data with a full copy of the model, each device can process a portion of the computations for the entire dataset. This can be done by partitioning the tensors that the model uses in its computations and distributing these partitions across multiple devices. For instance, if the model is a neural network, tensor parallelism can involve splitting the network's weight tensors across multiple devices. Each device could then perform a portion of the network's computations using its subset of the weights. After all devices have completed their computations, the results can be aggregated to produce the final output. This approach can be helpful when the model is too large to fit on a single device, as it allows the model's computations to be distributed across multiple devices.

[0161] Both data parallelism and tensor parallelism can be used together in a mixed parallelism approach to further increase the scalability and efficiency of the model. In such a setup, the model's computations can be partitioned across multiple sets of devices using tensor parallelism, and each of these sets of devices can contain individual devices that each process a different subset of the data using data parallelism.

[0162] Prefill system(s) 110 can use one or multiple different partitioning techniques. Prefill system(s) 110 can use one or multiple partitions 206 over one or multiple prefill computing device(s) 204 for feedforward portion of a machine-learned sequence processing model.

[0163] For example, a 1D weight-stationary partitioning strategy can be applied. This partitioning strategy can involve dividing each weight matrix for the feedforward network across a number of chips along one axis of the weight matrix (e.g., a network depth axis, an embedding depth axis). This partitioning method can allow each weight shard to be multiplied with the appropriate activation shard on each chip. The results can then be aggregated between the chips with an all-gather or reduce-scatter operation.

[0164] Another strategy (e.g., for a larger numbers of chips) can be a 2D weight-stationary partitioning strategy. This strategy can involve partitioning each weight matrix along both a network depth and an embedding depth axis, such that each shard can be roughly square. This method can reduce chip-to-chip communication time. Communication can be more efficient when multiplying an activation matrix through a set of consecutive

weight matrices by alternating which of the two axes on which to perform the activation aggregation between each multiplication. Each chip can always have the necessary activation shard to multiply with its weight shard, without ever having a fully replicated copy of the activation tensor.

[0165] For large batch sizes, the size of the activations can become larger than the size of the weights. In these cases, it can be more economical to keep the activations stationary on each chip, and instead transfer the weights between chips. For a distributed computational system having devices distributed over three axes, X, Y, and Z, this approach can be called XYZ weight-gathered. For moderate batch sizes, it can be useful to use a "hybrid" approach where both weights and activations are partially transferred along different axes. These approaches can be referred to as X-weight-gathered and XY-weight-gathered. The weight-gathered layout can use an all-gather operation for the weights and reduces the communication on activations. Therefore, this layout can decrease a total communication volume by adjusting the relative sizes of the X, Y, and Z axes. This can trade off weight communication against activation communication.

[0166] In some examples, the weight-stationary strategy can minimize communication at low tokens per batch, while different weight-gathered layouts can be optimal at larger number of tokens per batch. The weight-gathered layout can become cheaper or more efficient when the batch size and prefill sequence length are sufficiently large.

[0167] Prefill system(s) 110 can use one or multiple partitions 206 over one or multiple prefill computing device(s) 204 for an attention-computing portion of a machine-learned sequence processing model. This attention-computing portion can include elements such as the Query (Q), Key (K), and Value (V) activations used in multihead attention processes. These elements can be partitioned over the number of attention heads (nheads) dimension into a number of partitions equal to the number of chips (nchips) when nheads is a multiple of nchips.

[0168] For instance, in a scenario where the number of chips (nchips) is greater than the number of attention heads (nheads), the attention heads can be partially replicated across the chips. This distribution can allow for each chip to have a portion of the Q, K, and V activations, enabling parallel processing of the attention computations and efficient utilization of computational resources.

[0169] Each partition 206 can handle a subset of the attention computations, which can be processed independently on the associated prefill computing device(s) 204. This

arrangement can allow the prefill tasks to be distributed across multiple devices, improving the throughput and efficiency of the system.

[0170] The partitioning layout can match a two-dimensional weight-stationary feedforward layout, which can provide an efficient organizational structure for distributing the computations across multiple devices. In this weight-stationary layout, the weights of the model stay stationary on each device while the data moves, which can reduce the communication overhead between devices and improve the speed and efficiency of the computations.

[0171] Intermediate computation(s) 208 can be the same as or different from intermediate computation(s) 112. Intermediate computation(s) 208 can be the results produced by prefill system(s) 110 after processing the input element(s) 202. These computations can represent the model's understanding of the input data and can be used as a basis for further processing tasks, such as generating output sequences.

[0172] (Re)sharded intermediate computation(s) 210 can include intermediate computation(s) 208 in a format compatible with generation system(s) 114. For example, generation system(s) 114 can implement the machine-learned sequence processing model partitioned differently from a partitioning used by prefill system(s) 110. Thus, to use intermediate computation(s) 208, model host platform 106 can shard or re-shard intermediate computation(s) 208 to obtain (re)sharded intermediate computation(s) 210 that align with a partitioning scheme used by generation system(s) 114.

[0173] Generation computing device(s) 212 can be a range of hardware systems that can be used to perform generation tasks in a machine-learned sequence processing model. These computing devices can include specific hardware accelerators or other components that are designed to improve the performance of generation tasks. These could include, for example, memory devices with high bandwidth, storage devices with fast read/write speeds, or specialized processing units for tasks like matrix multiplication or convolution. Generation computing device(s) 212 can include CPUs, GPUs, TPUs, ASICs, FPGAs, or combinations thereof.

[0174] Generation computing device(s) 212 can be standalone devices, or they can be part of a larger system. For example, generation computing device(s) 212 can be installed in a server, a workstation, a laptop, mobile device, or another type of computer. Alternatively, they could be part of a distributed computing system, where tasks are distributed across multiple devices in a network.

[0175] Generation computing device(s) 212 can be equipped with a large(r) amount of memory or high(er) memory bandwidth to handle the memory utilization of generation tasks (e.g., as compared to prefill computing device(s) 204). For instance, generation tasks can involve multiple forward passes through the model to sample a series of outputs, which can use large amounts of memory bandwidth for loading and offloading model weights and intermediate computations from the compute cores of the device.

[0176] Generation system(s) 114 can use one or multiple different partitioning techniques. Generation system(s) 114 can use one or multiple partitions 214 over one or multiple generation computing device(s) 212 for feedforward portion of a machine-learned sequence processing model.

[0177] For example, a 1D weight-stationary partitioning strategy can be applied. This partitioning strategy can involve dividing each weight matrix for the feedforward network across a number of chips along one axis of the weight matrix (e.g., a network depth axis, an embedding depth axis). This partitioning method can allow each weight shard to be multiplied with the appropriate activation shard on each chip. The results can then be aggregated between the chips with an all-gather or reduce-scatter operation.

[0178] Another strategy (e.g., for a larger numbers of chips) can be a 2D weight-stationary partitioning strategy. This strategy can involve partitioning each weight matrix along both a network depth and an embedding depth axis, such that each shard can be roughly square. This method can reduce chip-to-chip communication time. Communication can be more efficient when multiplying an activation matrix through a set of consecutive weight matrices by alternating which of the two axes on which to perform the activation aggregation between each multiplication. Each chip can always have the necessary activation shard to multiply with its weight shard, without ever having a fully replicated copy of the activation tensor.

[0179] For large batch sizes, the size of the activations can become larger than the size of the weights. In these cases, it can be more economical to keep the activations stationary on each chip, and instead transfer the weights between chips. This approach can be called XYZ weight-gathered. For moderate batch sizes, it can be useful to use a "hybrid" approach where both weights and activations are partially transferred along different axes. These approaches can be referred to as X-weight-gathered and XY-weight-gathered. The weight-gathered layout can use an all-gather operation for the weights and reduces the communication on activations. Therefore, this layout can decrease a total communication

volume by adjusting the relative sizes of the X, Y, and Z axes. This can trade off weight communication against activation communication.

[0180] In some examples, the weight-stationary strategy can minimize communication at low tokens per batch, while different weight-gathered layouts can be optimal at larger number of tokens per batch. The weight-gathered layout can become cheaper when the batch size and prefill sequence length are sufficiently large.

[0181] Generation system(s) 114 can use one or multiple partitions 214 over one or multiple prefill computing device(s) 212 for an attention-computing portion of a machine-learned sequence processing model. This attention-computing portion can include elements such as the Query (Q), Key (K), and Value (V) activations used in multihead attention processes. These elements can be partitioned over the number of attention heads (nheads) dimension into a number of partitions equal to the number of chips (nchips) when nheads is a multiple of nchips.

[0182] For instance, in a scenario where the number of chips (nchips) is greater than the number of attention heads (nheads), the attention heads can be partially replicated across the chips. This distribution can allow for each chip to have a portion of the Q, K, and V activations, enabling parallel processing of the attention computations and efficient utilization of computational resources. Each partition 214 can handle a subset of the attention computations, which can be processed independently on the associated generation computing device(s) 212. This arrangement can allow the prefill tasks to be distributed across multiple devices, improving the throughput and efficiency of the system.

[0183] The partitioning layout can match a two-dimensional weight-stationary feedforward layout, which can provide an efficient organizational structure for distributing the computations across multiple devices. In this weight-stationary layout, the weights of the model stay stationary on each device while the data moves, which can reduce the communication overhead between devices and improve the speed and efficiency of the computations.

[0184] Different model partitioning schemes can be used for prefill system(s) 110 and generation system(s) 114. For example, prefill system(s) 110 can partition the computation of attention tensor(s) (e.g., KV values) across a head dimension. Generation system(s) 114 can partition the computation of attention tensor(s) across a batch dimension.

[0185] For example, in a context of a transformer model, the Q, K, and V matrices can be partitioned over the batch (B) dimension into nchips partitions. This can reduce the

memory cost of loading the KV cache per chip by a factor of nchips, thereby reducing the memory time by the same factor.

[0186] To facilitate using different model partitioning schemes for prefill system(s) 110 and generation system(s) 114, the system can reshard the input activation tensors using an all-to-all collective. For example, during autoregressive generation, there might be only one token per example of Q, K, and V tensors, whereas the KV cache can have many tokens. Since the KV cache can be orders of magnitude larger than the Q, K, and V tensors, it can be profitable to spend the all-to-all communication time on the small tensors to save the memory time on the large tensors.

[0187] Further example partitioning schemes and test results can be found in Pope et al., *Efficiently Scaling Transformer Inference*, arXiv:2211.05102v1 [cs.LG] 9 Nov 2022, which is hereby incorporated by reference herein in its entirety. Various sharding and resharding techniques are described in Rink et al., *Memory-Efficient Array Redistribution Through Portable Collective Communication*, arXiv:2112.01075v2 [cs.DC] 28 Nov 2022, which is hereby incorporated by reference herein in its entirety.

[0188] Output Element(s) 216 can be the individual pieces of data that make up the final output sequence generated by the machine-learned sequence processing model. These elements can vary widely depending on the nature of the model and the specific task it is designed to perform.

[0189] For example, in the context of a natural language processing model, output element(s) 216 can be or represent individual words, sentences, or paragraphs in a generated text document. These elements can be produced by the model as it processes and generates text based on the input sequence and the learned patterns or structures in the data.

[0190] In an image processing model, output element(s) 216 can be tokens that represent individual pixels or regions of a generated image. Each pixel can be represented as a combination of red, green, and blue (RGB) color values, or in other color spaces such as YUV or HSV. Regions of an image can be represented using various techniques such as convolutional patches, superpixels, or other segmentation methods.

[0191] For a speech synthesis model, output element(s) 216 can be or represent individual phonemes, syllables, or words in generated speech. These elements can be extracted from audio data using various feature extraction techniques such as Fourier transforms, Mel-frequency cepstral coefficients (MFCCs), or deep neural network-based methods.

[0192] In the context of a recommendation system, output element(s) 216 can be or represent individual items that the system recommends. For example, these could be products, movies, songs, or any other type of item that the system is designed to recommend.

[0193] Output Element(s) 216 can also include additional information or metadata related to the generated output. For instance, in a classification task, each output element could be accompanied by a confidence score that indicates the model's certainty in its prediction. In a generative task, each output element could be accompanied by information about the generation process, such as the specific model parameters or random seeds used to generate the output.

[0194] In some cases, output element(s) 216 can be represented using various data types or formats, such as integers, floating-point numbers, strings, arrays, tensors, or other data structures.

[0195] Figure 3 is a block diagram of an example configuration of prefill system(s) 110. Prefill system(s) 110 can include a plurality of groups of computing devices. A group 302 can include M prefill computing devices 302-1, 302-2, . . . , 302-M. A group 304 can include N prefill computing devices 304-1, 304-2, . . . , 304-N. A group 306 can include O prefill computing devices 306-1, 306-2, . . . , 306-O.

[0196] Each group can be configured with the same or different partitioning scheme(s). Each group can be configured with the same or different hardware device(s). Prefill system(s) 110 or routing system(s) 108 can select an execution group for processing a particular input sequence based on one or more attributes of the input sequence. For instance, a length of the input sequence, a complexity of the input sequence, a priority of the input sequence, a deadline associated with the input sequence, or other factors can be considered when selecting an execution group.

[0197] For instance, a sequence length can be determined for the input sequence, and a group can be selected based on the sequence length. For instance, longer sequences can be routed to a group with a larger buffer size, while shorter sequences can be routed to a group with a smaller buffer size. This can optimize the use of computational resources and memory, leading to increased efficiency and throughput.

[0198] In some scenarios, the selection of a group for processing an input sequence can be dynamic and can change over time. For instance, if a sequence initially routed to a group starts to exceed the group's buffer size, the sequence could be dynamically rerouted to a different group with a larger buffer size. This can ensure that the processing of the sequence continues smoothly without exceeding the memory capacity of the devices.

[0199] Each group can implement different partitioning schemes for the machine-learned sequence processing model. For example, one group may implement a weight-stationary partitioning layout where the model's weights stay stationary on each device while the data moves. Another group could implement a weight-gathered partitioning layout where the data stays stationary and the weights are gathered from different devices. Yet another group might implement a hybrid partitioning layout combining aspects of both weight-stationary and weight-gathered layouts. These different partitioning schemes can be chosen based on the specific requirements of the tasks that each group is expected to handle.

[0200] For example, different groups can have different levels of processing efficiency. For instance, a model flops utilization (MFU) can vary based on a partitioning layout. Some layouts can achieve high speed with low efficiency. The different groups can be configured to make different speed/throughput and efficiency tradeoffs. Based on a priority or cost target associated with a given input request 104, routing system(s) 108 or prefill system(s) 110 can assign a given request to be processed by a particular group.

[0201] Groups can have varying levels of hardware specification. For example, group 302, for instance, can be configured with devices that have high-speed processors. Group 304 can contain less powerful (e.g., more cost-effective) processing units. Group 306 can slower but very efficient (e.g., energy efficient) processors. Based on a priority or a target cost/speed associated with a given request 104, routing system(s) 108 or prefill system(s) 110 can assign a given request to be processed by a particular group.

[0202] Prefill system(s) 110 or routing system(s) 108 can select an execution group based on the current workload or capacity of each group. For example, a group with fewer ongoing tasks or more available resources might be selected to process a new input sequence. This could help to balance the load across the different groups and maximize the overall throughput of the system.

[0203] Prefill system(s) 110 or routing system(s) 108 can select an execution group based on the specific requirements of the input sequence. For instance, if the input sequence contains text data, a group that is optimized for processing text data could be selected. If the input sequence contains image data, a group that is optimized for processing image data could be selected.

[0204] Prefill system(s) 110 or routing system(s) 108 can dynamically reconfigure the groups based on the current workload or other factors. For instance, if the workload increases, additional computing devices could be added to a group to increase its capacity. Conversely, if the workload decreases, some computing devices could be removed from a

group to save resources. This dynamic reconfiguration can allow the system to adapt to changes in the workload and optimize the use of the available resources.

[0205] The configuration of the groups can also vary the topology of the devices. For instance, devices in a group can be connected in a specific way to optimize data transfer between them. This could involve connecting the devices in a ring, mesh, tree, or other topologies (e.g., a toroidal grid), depending on the communication requirements of the tasks.

[0206] Figure 4 is a block diagram of an example configuration of generation system(s) 114. Generation system(s) 114 can include a plurality of groups of computing devices. A group 402 can include P generation computing devices 402-1, 402-2, . . . , 402-P. A group 404 can include Q generation computing devices 404-1, 404-2, . . . , 404-Q. A group 406 can include R generation computing devices 406-1, 406-2, . . . , 406-R.

[0207] Each group can be configured with the same or different partitioning scheme(s). Each group can be configured with the same or different hardware device(s). Each group can be configured with the same or different generation batch size(s). Generation system(s) 114 or routing system(s) 108 can select an execution group for processing a particular output sequence based on one or more expected or actual attributes of the output sequence. For instance, a length of the output sequence, a complexity of the output sequence, a priority of the output sequence, a deadline associated with the output sequence, or other factors can be considered when selecting an execution group.

[0208] Attributes of the output sequence can be expected or actual. Expected attributes can be determined by predicting a likely attribute associated with the output sequence based on a context associated therewith (e.g., the input sequence, the corresponding input request, the client device, etc.). For example, an output sequence for a yes/no question can be expected to be short and be processed accordingly. An output sequence for a request for long-form content can be expected to be long and be processed accordingly. An output sequence for an input request 104 that contains a request of imagery can be expected to involve multimodal data types, calls to external APIs, etc., and be processed accordingly.

[0209] Expected attributes as described herein can be obtained by predicting an attribute value using a predictive model trained to predict attribute values based on logged usage data that records one or more features of a processing task and the eventual ground truth attribute value. Expected attributes can be implicitly used by directly outputting a prediction that controls a task (e.g., directly outputs a prediction that controls a route for an input), such that an explicit attribute may not be regressed.

[0210] Actual attributes can be monitored and detected during generation of the output sequence. Model host platform 106 can monitor a state of generation of the output sequence and determine a current length, a current mixture of modalities, etc. and respond accordingly to optimize processing (e.g., shifting to a different group, etc.).

[0211] For instance, a sequence length can be determined for the output sequence, and a group can be selected based on the sequence length. For instance, longer sequences can be routed to a group with a larger buffer size, while shorter sequences can be routed to a group with a smaller buffer size. This can optimize the use of computational resources and memory, leading to increased efficiency and throughput.

[0212] In some scenarios, the selection of a group for processing an output sequence can be dynamic and can change over time. For instance, if a sequence initially routed to a group starts to exceed the group's buffer size, the sequence could be dynamically rerouted to a different group with a larger buffer size. This can ensure that the processing of the sequence continues smoothly without exceeding the memory capacity of the devices.

[0213] Each group can implement different partitioning schemes for the machine-learned sequence processing model. For example, one group may implement a weight-stationary partitioning layout where the model's weights stay stationary on each device while the data moves. Another group could implement a weight-gathered partitioning layout where the data stays stationary and the weights are gathered from different devices. Yet another group might implement a hybrid partitioning layout combining aspects of both weight-stationary and weight-gathered layouts. These different partitioning schemes can be chosen based on the specific requirements of the tasks that each group is expected to handle.

[0214] For example, different groups can have different levels of processing efficiency. For instance, a model flops utilization (MFU) can vary based on a partitioning layout. Some layouts can achieve high speed with low efficiency. The different groups can be configured to make different speed/throughput and efficiency tradeoffs. Based on a priority or cost target associated with a given input request 104, routing system(s) 108 or generation system(s) 114 can assign a given request to be processed by a particular group.

[0215] Groups can have varying levels of hardware specification. For example, group 402, for instance, can be configured with devices that have high-speed processors. Group 404 can contain less powerful (e.g., more cost-effective) processing units. Group 406 can include slower but very efficient (e.g., energy efficient) processors. Based on a priority or a target cost/speed associated with a given request 104, routing system(s) 108 or generation system(s) 114 can assign a given request to be processed by a particular group.

[0216] Generation system(s) 114 or routing system(s) 108 can select an execution group based on the current workload or capacity of each group. For example, a group with fewer ongoing tasks or more available resources might be selected to process a new output sequence. This could help to balance the load across the different groups and maximize the overall throughput of the system.

[0217] Generation system(s) 114 or routing system(s) 108 can select an execution group based on the specific requirements of the input sequence. For instance, if the output sequence contains or is expected to contain text data, a group that is optimized for processing text data could be selected. If the input sequence contains or is expected to contain image data, a group that is optimized for processing image data could be selected.

[0218] Generation system(s) 114 or routing system(s) 108 can dynamically reconfigure the groups based on the current workload or other factors. For instance, if the workload increases, additional computing devices could be added to a group to increase its capacity. Conversely, if the workload decreases, some computing devices could be removed from a group to save resources. This dynamic reconfiguration can allow the system to adapt to changes in the workload and optimize the use of the available resources.

[0219] The configuration of the groups can also vary the topology of the devices. For instance, devices in a group can be connected in a specific way to optimize data transfer between them. This could involve connecting the devices in a ring, mesh, tree, or other topologies, depending on the communication requirements of the tasks.

[0220] Figure 5 is a block diagram of an example interleaved processing flow. Interleaved system(s) 500 can process a plurality of sessions in parallel. Interleaved system(s) 500 can sequentially process batches of prefill and batches of generation or decode, such as by using a common pool of computing device(s) (e.g., hardware accelerators).

[0221] In Figure 5, it is to be understood that a numbered "interval" as drawn can represent a single forward pass through the model.

[0222] In an example illustrated in Figure 5, a batch size for prefill can be 1 session. For example, in interval 2, prefill for session 1 can be completed (e.g., in one forward pass). A batch size for generation can be 3 sessions. However, in intervals 3, 4, 5, (e.g., three forward passes) only the prefill for session 1 had been completed, so generation proceeds only for session 1.

[0223] Interleaved system(s) 500 can then pause to perform prefill for session 2 during interval 6. Thus, in a next set of decode intervals, generation for sessions 1 and 2 can proceed in intervals 7 and 8. Interleaved system(s) 500 can then pause to perform prefill for

session 3 during interval 9. Thus, in a next set of decode intervals, generation for sessions 1, 2, 3 can proceed in interval 10. Upon processing of an end-of-sequence token in session 1, prefill for session 4 can proceed during the next pause in decode (e.g., interval 11). After prefill for session 4, generation can proceed again with a full batch size of 3.

[0224] Interleaved systems 500 can be used in addition to or alternative to a disaggregated serving system.

[0225] An example hardware configuration for an implementation of interleaved system(s) 500 can include a pool of hardware accelerators that can be leveraged for both prefill and decode.

[0226] Figure 6 is a block diagram of an example model host platform 106 that can be used for interleaved processing. Model host platform 106 can include hardware accelerator(s) 602. Model host platform 106 can be associated with a plurality of configurations 604-1, 604-2, . . . , 604-N. Model host platform 106 can control one or more or all of hardware accelerator(s) 602 to execute a machine-learned model forward pass, and the model can be partitioned differently based on the selected configuration of the plurality of configurations 604-1, 604-2, . . . , 604-N. Although described herein with respect to interleaved processing it is to be understood that the examples described with respect to Figure 6 can include example attributes of other implementations described elsewhere herein.

[0227] Hardware accelerator(s) 602 can correspond to prefill computing device(s) 204 or generation computing device(s) 212. Hardware accelerator(s) 602 can include CPU(s), GPU(s), TPU(s), ASIC(s), FPGA(s), or combinations thereof. Hardware accelerator(s) 602 can be interconnected in a mesh, tree, or other topologies. An example topology includes a torus, such as 3D toroidal topology.

[0228] Model host platform 106 can engage hardware accelerator(s) 602 with different configurations (e.g., different model partitioning layouts) based on a current task being performed using hardware accelerator(s) 602.

[0229] Figure 7 is an example communication timeline diagram illustrating a workflow cycle that includes hot swapping between different configurations for prefill and decode. Model host platform 106 can receive input request(s) 104 and engage hardware accelerator(s) 602 to load a first configuration at operation 702. Using the first loaded configuration, hardware accelerator(s) 602 can execute prefill operations at 704. Model host platform 106 can engage hardware accelerators 602 to load a second configuration at operation 706. Using the second loaded configuration, hardware accelerator(s) 602 can execute decoding operations at 708 and return generated outputs (e.g., final outputs,

intermediate output(s)) to model host platform 106. Model host platform 106 can store the generated outputs in a buffer or can generate an output payload containing all or part of the generated output.

[0230] In an example, hardware accelerator(s) 602 can be initialized with an initial weight partition. Loading a first or second configuration can include using the initial weight partition (e.g., loading all or part of the initial weight partition into a processor core memory). Loading a first or second configuration can include using a different weight portion, such as gathering model weights across an array of hardware accelerators to populate a local memory for each hardware accelerator.

[0231] In an example, the initial weight partition can correspond to a weight-stationary partition layout. In this manner, the hardware accelerator(s) 602 can be used in a weight stationary configuration without further reconfiguring. The initial weight partition can also be selected to correspond to an initial state for gathering weights across multiple hardware accelerators in a at least partially activation-stationary or weight-gathered configuration. This can enable hot-swapping between weight stationary and weight-gathered configurations by selecting whether to use the initial weight partition or to gather weights from the respective initial weight partitions of other devices.

[0232] Figure 8 is a block diagram of an example circular buffer 800. A cursor can identify a position in the buffer for writing the results of a time step t. The cursor can overwrite past-time values that have since been offloaded from the buffer.

[0233] Various references are made herein to different example model partitioning techniques and model weight layouts. Further detailed examples of model partitioning configurations are provided herein to illustrate example implementations of the various partitioning configurations referenced throughout with respect to various different systems and devices.

[0234] Example notation is adopted herein to aid the understanding of various example implementations. Consider an example machine-learned sequence processing model M with a number of parameters n_{params} (e.g., learned weights). The parameters can be partitioned over a number of chips n_{chips}. Model M can have an embedding dimension referred to as E or d_{model}. Model M can have a feedforward network having an intermediate dimension referred to as F or d_{ff}. Model M can have a number of output heads n_{heads} or H.

[0235] Each example in a batch of B sequences can have L_{input} tokens of input text and generate L_{gen} tokens of output text. Since the input tokens can all be present at the start of the inference, model M can execute over all $B \times L_{\text{input}}$ tokens in parallel, in a single forwards

pass over some or up to all the tokens. This step can be called prefill. The output tokens can be generated autoregressively, with a sequential loop of L_{gen} steps. Each step can include a single forwards pass through the model, after which a new token can be sampled for each of the B examples in the batch. This loop can be referred to as generation or decoding.

[0236] Since prefill can run in parallel over L_{input}, but autoregressive decode runs sequentially over L_{gen}, the two phases may have different performance characteristics.

[0237] This framework can be used for exploring a number of optimization tradeoffs. For example, scaling up model sizes can unlock new capabilities and applications but can introduce tradeoffs in terms of inference cost. Inference cost can be measured in terms of the following metrics: latency, throughput, and model FLOPS utilization. Latency may be or include the total time for an inference and can be broken down into the time to process the input tokens present at the start of the inference and the time to autoregressively generate one or more output tokens. The decode latency can also be measured per step, divided by the number of tokens in each sequence. The throughput of prefill or decode can be or include the number of tokens processed or generated per second. The model FLOPS utilization (MFU) is the ratio of the observed throughput to the theoretical maximum throughput if the benchmarked hardware setup were operating at peak FLOPS with no memory or communication overhead.

[0238] Communication overhead in this framework can arise when larger models do not fit on a single accelerator chip and need to be partitioned across many accelerator chips to fit in memory.

[0239] Memory costs in this framework can correspond to, for instance, the storage of tensors such as weights and the attention value caches in on-device high-bandwidth memory (HBM). In some examples, such as for large transformer-based models, weight tensors and attention value tensors are the dominant memory cost. While there are other tensors that pass may through the HBM, their memory footprint may generally be much smaller.

[0240] For a forward pass through the example model M, these tensors are transferred from HBM to the compute cores of the chip once per forward pass (prefill or decode step) of the model. This takes a certain amount of time, which can be called memory time. In some instances, at small batch sizes and sequence lengths, the time to load weights dominates. In some instances, at larger batch sizes and sequence lengths, the time to load the attention vale tensors dominates.

[0241] Compute costs in this framework can include matrix multiplication and accumulation operations. For instance, a decoder-only model with N parameters can require

2N matmul FLOPs in the forward pass per token seen because each matmul performs one multiplication and one addition per pair of input token and parameter values in the forward. If all chips were running at peak FLOPS, these matmuls would take a certain amount of time, which can be referred to as compute time. In some instances, matmuls in an attention of a model mechanism may add a much smaller number of FLOPs per token for large models and can often be excluded. Even though the computational cost of attention is relatively small, it can still account for a significant fraction of memory capacity and bandwidth costs, since (unlike the weights) the attention cache is unique for each sequence in the batch (as it is generated based on the input(s)).

[0242] Both the weight loading part of the memory time and the non-attention compute time may be proportional to the model size and inversely proportional to the number of chips. However, for a given partitioning layout, the time needed for chip-to-chip communication may decrease less quickly (or not at all) with the number of chips used and can become an increasingly important bottleneck as the chip count grows.

[0243] For instance, reducing latency may be achieved by engaging more chips and partitioning the model in as many ways as sensible. Lower latency can often be achieved with smaller batch sizes, but smaller batch sizes also generally result in worse MFU, resulting in a higher total cost (in terms of chip-seconds or dollars) per token.

[0244] Increasing an attention context, and thereby sequence length, can increase inference time. For instance, for a 500B+ model with multihead attention, the attention KV cache grows large: for batch size 512 and context length 2048, the KV cache totals 3 TB, which is 3 times the size of the model's parameters. The on-chip memory may need to load this KV cache from off-chip memory once for every token generated, during which the computational core of the chip is essentially idle.

[0245] If latency is lower priority, and the primary goal is to maximize per-chip throughput (e.g., minimize total cost per token), may be efficient to increase the batch size. Larger batches may result in better MFU, and certain partitioning strategies that are not efficient for small batch sizes may become efficient as the batch size grows larger.

[0246] Example partitioning layouts are described herein with respect to a 3D torus topology X by Y by Z. Subscripts are used to specify the tensor dimension that is partitioned. For example, notation BLE_{xyz} indicates that the last dimension E of a tensor of logical shape [B, L, E] is split into X times Y times Z partitions, where *x*, *y*, and *z* refer to the physical axes along with hardware accelerators are interconnected. The per-chip tensor may be of a shape [B, L, E / (XYZ)]. If a tensor is replicated over an axis *x*, that axis is omitted from the

notation. A suffix "partialsum-" indicates that a given tensor has been contracted (summed) locally on each chip (over some axis not represented in the shape), but still needs to be summed across the chips in the indicated axis (creating a tensor replicated over the indicated axis) before the result is meaningful.

[0247] Several communication collectives are used. All-reduce(x) sums a partialsum tensor such as BLE_{yz} (partialsum-x) across sets of chips in the x axis of the torus and broadcasts the sum back to all the involved chips, returning output of shape BLE_{yz}. All-reduce may be split into two phases: a reduction phase and a broadcast phase. The reduction phase is called reduce-scatter(x), and it sums BLE_{yz} (partialsum-x) tensors across sets of chips in the x axis but produces an output that is sharded rather than replicated over the chips in that axis, in a layout such as B_xLE_{yz} or BLE_{xyz}. The broadcast phase is called all-gather(x), and it broadcasts and concatenates the tensor BLE_{xyz} to all chips in the x axis, producing an output X times larger than its input, replicated over the x axis: BLE_{yz}. The all-to-all collective shifts sharding from one tensor dimension to another, e.g. BLH_x Q to B_xLHQ by using direct communication between every (source, destination) pair.

[0248] Figure 9 illustrates the collectives.

[0249] Figure 10 illustrates one example weight-stationary partitioning layout for a feedforward layer of a machine-learned model. Each [E, F] weight matrix is partitioned (or sharded) among n_{chips} along the E or F axis. Each weight shard is multiplied by the appropriate activation shard on each chip, and the results are aggregated between the chips with an all-gather and/or reduce-scatter. Additionally, when computing two consecutive matrix multiplications (as in a Transformer MLP block), the first matmul may be partitioned by the output axis, and the resulting activation shard on each chip may be the exact one needed to compute the second matmul partitioned by the input axis.

[0250] As illustrated in Figure 10, the weights and activations of the feedforward layer are partitioned over n_{chips} along the F axis. On an example 3D torus topology the partition layout for weights is EF_{xyz} and $F_{xyz}E$. For instance, they are partitioned in to XYZ = n_{chips} partitions, with X, Y, and Z partitions across physical hardware accelerator communication axes. The weights are kept stationary in each chip, and the activations are transferred between chips to match the weight layout, involving one all-gather and one reduce-scatter. In this 1D weight-stationary partitioning strategy, each chip gets inputs and outputs of shape BLE in the reduce-scatter and all-gather respectively. The resulting communication time can be expressed as

$$T_{\text{comm}} = \frac{2BLE}{\text{network bandwidth}}$$

[0251] Another strategy involves partitioning each [E, F] weight matrix along both the E and F axes, such that each shard is roughly square. For example, if E = 1024, F = 4096, and $n_{\text{chips}} = 64$, then we would shard 4-ways among E and 16-ways among F, so that each of the 64 chips stores a 256-by-256 chunk of the weight matrix, and activations are transferred between chips. This may be called 2D weight-stationary. The total compute cost may be the same as 1D weight-stationary, but communication may be more efficient: when multiplying an activation matrix through a set of consecutive weight matrices, we can alternate which of the two axes we perform the activation aggregation on between each multiplication. With the a selection of partitioning, each chip may always have the necessary activation shard to multiply with its weight shard, without ever having a fully replicated copy of the activation tensor.

[0252] Since each axis is partitioned on $O((n_{\text{chips}})^{0.5})$, the communication time scales as $O((n_{\text{chips}})^{-0.5})$ rather than remaining constant. In some implementations, even if the 2D layout is communication-limited at a certain chip count and batch size, we can continue to reduce latency by adding more chips, because communication time continues to reduce. In some implementations, 2D weight-stationary becomes more communication-efficient than 1D weight stationary when

$$\sqrt{n_{\text{chips}}} > \frac{F}{E}.$$

In some implementations, F = 4E, in which the crossover occurs when $n_{\text{chips}} = 16$.

[0253] Figure 11 illustrates an example 2D weight stationary partitioning layout. Whereas the example 1D weight-stationary layout of Figure 10 runs its all-gather and reduce-scatter with unsharded shape BLE per chip, this 2D weight-stationary layout partitions E so that the communication volume for F partitioning is reduced from BLE to BLE/X. This comes at the cost of introducing a second pair of reduce-scatter and all-gather operations, whose cost may be balanced with the existing communication.

[0254] The partitioning layout for weights in Figure 11 is $E_X F_{YZ}$, as they are partitioned along the E dimension into X partitions and along the F dimension into YZ partitions. The partitioning layout for the input activations is the same as the 1D example.

[0255] In an example, the optimization framework provided herein can provide optimal values of X, Y and Z to minimize total communication time in. Assuming F = 4E, an

example implementation may achieve the minimum communication time with $X=0.5\sqrt{n_{\text{chips}}}$ and $YZ=2\sqrt{n_{\text{chips}}}$. The resulting total communication time may be expressed:

$$T_{\text{comm}} = \frac{8BLE}{\sqrt{n_{\text{chips}}} \cdot \text{network bandwidth}}$$

[0256] In the previously described example weight-stationary strategies, each chip stores one shard of each weight matrix in memory, and that chip is responsible for multiplying its stationary weight shard with each corresponding activation shard. The output of each per-chip matrix multiplication is then aggregated between chips to be used as input to the subsequent operations.

[0257] However, as the batch size (and sequence length) grows larger, the size of the output activations may become significantly larger than the size of the weights. When this happens, it can become more economical to keep the activations stationary on each chip, and instead transfer the weights between chips. For very large batch sizes, it may be beneficial to keep the activations fully stationary between sequential matrix multiplications and fully transfer the weights between all chips.

[0258] When at least some weights are moved from chip to chip, it can be referred to as a "weight gathered" layout. Weight gathering can be 1D, 2D, 3D, etc. based on the axes over which weights are gathered. For example, gathering weights over all axes (e.g., retrieving a full copy of weights from partitions over all the axes) can be called XYZ-weight gathered for three axes XYZ. Hybrid approaches can include gathering weights over two axes to generate, at each chip, replica weights for the partitions over those two axes (the weights may still be distributed over other axes). For instance, gathering weights from an initial distribution over the X and Y axes can be called XY-weight-gathered. Gathering weights from an initial distribution over the X axis can be called X-weight-gathered.

[0259] Figure 12 illustrates an example XY-weight-gathered layout. An aspect of the specific layout illustrated here is that weights start in the same E_xF_{yz} layout as in 2D weight-stationary, so that we can use the same initial weight layout for weight-gathered (e.g., during prefill) and weight-stationary (e.g., during decoding). Just before the einsums, the weight tensors are all-gathered over the X and Y axes, with communication volume EF/Z. This is additional communication relative to weight-stationary layout, but in return we reduce the communication on activations: one reduce-scatter/all-gather pair for activations is skipped, and the communication volume on the other pair drops from BLE/X to BLE/XY.

[0260] By changing the relative sizes of the X, Y, and Z axes, we can trade off weight communication against activation communication, and thereby minimize the total communication volume.

[0261] In some examples, the initial weight layout can be selected to share the weights between weight-stationary and weight-gathered layouts. This enabled selection from variants of the weight-gathered layout. The variant shown in Figure 12 uses all-gather(xy) for the weights and $B_{xy}LE_z$ partitioning of batch for the activations.

[0262] Figure 13 illustrates an X-weight-gathered layout that uses all-gather(x) for the weights and B_xLE_{yz} for partitioning of the activations.

[0263] Figure 14 illustrates an XYZ-weight-gathered layout that uses all-gather(xyz) for weights, and correspondingly uses $B_{xyz}LE$ partitioning for the activations.

[0264] Figure 15 shows how the communication-optimal configuration switches between these layouts as batch size grows for some example implementations. Communication volumes estimated for X=Y=Z=4, E=16384, and F=65536. While the 2D weight-stationary strategy minimizes communication at low tokens per batch, different weight-gathered layouts are optimal at larger number of tokens per batch.

$$T_{\text{comm}} = \frac{4E\sqrt{BLF}}{\sqrt{n_{\text{chips}}} \cdot \text{network bandwidth}}$$

[0265] The communication time for the weight-stationary layout may be linear in BL, while the communication time for the weight-gathered layout may be linear in sqrt(BL). Therefore, the weight-gathered layout may become more efficient or cheaper when the batch size and prefill sequence length are sufficiently large.

[0266] Although described with respect to feedforward network layer(s), the example layouts described above can be used for various model components. Attention mechanisms can be parallelized using similar techniques.

[0267] For instance, multihead attention can be parallelized similarly, by replacing F with n_{heads}. But inference with multihead attention can involve significant memory capacity and bandwidth costs to store and load the KV cache, and these costs can dominate the rest of the inference at large batches or long context lengths.

[0268] An alternative approach, called multiquery attention, generally emits n_{heads} for the query tensor, but only a single head for the key and value tensors. This key and value head is shared across the n_{heads} query heads. This reduces the size of the KV cache tensors by

a factor of n_{heads} and hence the memory time spent loading them. But it also removes an axis otherwise used for parallelism.

[0269] Figure 16 illustrates an example comparison of multi-head and mutli-query tensor shapes.

[0270] The partitioning layout of projection matrices that have a n_{heads} dimension (W_Q and W_O in multiquery attention, and those two plus W_K and W_V in multihead attention) may match the layout used in the feedforward layer.

[0271] Figure 17 at (A) shows an example partitioning layout for multihead attention, matching the 2D weight stationary feedforward layout. Here the Q, K, and V activations are partitioned over the n_{heads} dimension into n_{chips} partitions when n_{heads} is a multiple of n_{chips}. For n_{chips} greater than n_{heads}, the attention heads are partially replicated. The most similar partitioning layout for multiquery attention (shown in Figure 17 at (B)) treats the KV cache the same as in multihead attention. Even though the key and value tensors are shared across all heads, they are replicated on each chip and the memory cost savings of multiquery attention are lost.

[0272] Advantageously, partitioning multiquery attention mechanisms (e.g., the Q, K, V tensors) over a batch dimension B as shown in Figure 17 at (C) may reduce the memory cost of loading the KV cache per chip by a factor of n_{chips}, thereby reducing the memory time by the same factor.

[0273] Figure 18 provides an example layout for a sharded-over-heads approach.

[0274] Figure 19 provides an example layout for a sharded-over-batch approach.

[0275] Sharding over batch can incur an additional communication cost of resharding the input activation tensors using an all-to-all collective in comparison to the multiquery attention sharding strategy where the Q, K, and V matrices are partitioned over the heads dimension. During autoregressive generation, there is generally only one token per example of Q, K, and V tensors (e.g., per instance in a batch), whereas the KV cache has many (e.g., 2048 or more) tokens. Since the KV cache may be orders of magnitude larger than the Q, K, and V tensors, it may be profitable to spend the all-to-all communication time on the small tensors to save the memory time on the large tensors.

[0276] In some cases, such as during prefill, it may be less impactful to shard attention over batch. The Q tensor may have many tokens, all of which may be queried against the same K and V tensors. The memory load of the K and V tensors can be amortized over all tokens in the Q tensor, and so this memory load might not be a bottleneck during

prefill. In such cases, a sharded-over heads layout may be used to spare the additional communication cost.

[0277] With the proposed partitioning layout, multiquery attention can enable using larger batch sizes and sequence lengths, thereby increasing throughput in addition to the latency reduction from reduced memory time. The measured savings can be an order of magnitude compared to multihead attention.

[0278] Parallel transformer blocks can be used. In an example parallel transformer block, the feedforward layer and attention layer can be computed in parallel from the layernormed input and summed to get the output. The benefits from an example parallel formulation are as follows. First, there may be only one layernorm per layer instead of two, which reduces latency at small batch sizes. Second, the input matrices of the feedforward layer may be be fused with the query projection matrix W_Q of the attention layer, the key/value projection matrices W_K and W_V may be be fused in the attention layer, and the output matrix of the feedforward layer may be fused with the output projection matrix W_O of the attention layer. This fusion can help facilitate higher FLOPS utilization because larger matrix-multiplications run more efficiently on accelerators. It may also eliminate one of the two all-reduce operations in each Transformer layer needed for F/n_{heads} parallelism, cutting communication time over this axis in half.

[0279] We use the Looped CollectiveEinsum technique to run communication concurrently with computation. This allows us to partially or fully hide the communication time of most of the reduce-scatter and all-gather operations. For reduce-scatter operations, we can select from whether to reduce-scatter into a batch or sequence dimension (B or L) or into the hidden dimension (E or F). For the case study that follows, we chose the latter.

[0280] In the case study that follows, the CollectiveEinsum loops are the overwhelming majority of the inference latency, so we invested considerable effort to maximize their performance. First, optimized CollectiveEinsum for different scenarios: latency versus throughput, different numbers of torus axes, fusing with different input/output collectives. Second, we explicitly match up communication collectives with the matrix multiplies that they should be fused with, to maximize the potential for overlap. Through such optimizations, we achieved about 1.4 times better performance than the simpler compiler-partitioned-and-scheduled implementation that we started with. Some of the weight-gathered layouts would exhaust memory without these optimizations.

[0281] We also included the following low-level optimizations: better in-memory layout of tensors to minimize padding and copying during matrix multiplies, faster top-k/top-

p implementations for decode sampling, faster log-base-2 implementations of Softmax and Swish, and support for incremental processing of sequences during prefill.

[0282] We use the AQT library to reduce the memory cost of 16-bit weights by converting them to int8 without noticeable quality loss. This enables memory time savings from weight loading, which is especially helpful in the low batch size regime, and it reduces communication volume in weight-gathered layouts. Activation quantization could reduce compute time in large-batch configurations and reduce communication volume of activations in weight-stationary layouts but was not used in the case study that follows.

[0283] A case study was conducted to evaluate the efficiency of various implementations of techniques described herein. What follows is an empirical study of our techniques on the PaLM family of models, which we select since the model architecture incorporates the techniques of multiquery attention and parallel attention and feedforward layers.

[0284] The Case Study inference framework is based on JAX and XLA~\cite{XLA}, and the Case Study original high-level implementation was based on T5X. The Case Study uses up to 256 TPU v4 chips for our benchmarks. Each TPU v4 chip can run bfloat16 matrix arithmetic at 275 TFLOPS, has 32 GiB of High Bandwidth Memory (HBM) at 1200 GB/s of bandwidth, and has 270 GB/s of interconnect bandwidth in a 3D torus topology. For the PaLM 540B model we padded the number of attention heads up from 48 to 64 in order to partition more effectively on 64+ chips. This adds 18B parameters to the model, which comes at a 3% MFU cost, which was more than recovered by being able to partition more effectively.

[0285] The Case Study evaluates the relative performance of our feedforward layer partitioning strategies. First we evaluate performance of decoding. We use batch size 512 to balance latency and MFU.

[0286] Figure 20 shows the performance of 1D and 2D weight-stationary layouts as we increase the chip count. Both layouts start to become communication-limited, but the 2D layout performs better because of its asymptotically better scaling with chip count.

[0287] Next we consider the prefill phase. We consider batch sizes from 2048 tokens (1 example, 2048 tokens) to 1 million tokens (512 examples, 2048 tokens per example).

[0288] Figure 21 shows that the optimal partitioning layout for this Case Study switches from the 2D weight-stationary layouts to the weight-gathered layouts as the batch size increases. The weight-gathered layouts are less efficient at low batch sizes, but eventually they become the most efficient at high batch sizes, achieving 76% MFU when the

communication overhead is almost negligible. Such large batch sizes would fail from memory exhaustion without multiquery attention on this hardware.

[0289] An example strategy, then for dynamically selecting a partitioning layout follows: during the prefill phase, the system can select from weight-stationary and weight-gathered layouts based on the current number of tokens in the batch. During the generate phase, the system can select the 2D weight-stationary layout because the batch size in tokens is small.

[0290] The Case Study further evaluated the partitioning layouts for multiquery attention. We consider PaLM with multiquery attention in both the baseline layout that partitions by attention heads and the optimized layout that partitions by batch. We also create a modified variant of PaLM 540B which uses multihead attention instead of multiquery. To keep parameter count in the attention layer constant, we shrink a head dimension from 256 in the multiquery variant to 128 in the multihead variant.

[0291] At large batch sizes and context lengths, the KV cache can become very large, putting us at the risk of running out of memory. Table 1 shows that the optimized multiquery layout can fit up to 32--64 times longer context lengths than the multihead and baseline multiquery variant into a fixed memory size.

Table 1

Model	Head dimension	Context length; B=128	Context length; B=512
Multihead	128	1320	330
Baseline multiquery	256	660	165
Case Study multiquery	256	43,000	10700

[0292] During prefill, multiquery and multihead attention may incur similar inference latencies because we compute many attention queries in parallel and the attention computation becomes compute-limited on the attention matrix multiplies.

[0293] Figure 22 shows that the optimized multiquery layout improved speed during generation in the Case Study. The speed improvement is small when the context length is short because almost all of the time is spent on the feedforward layer. As the context length grows longer, the time to load the KV cache in the attention layer becomes a much larger portion of overall inference time. For instance, when Multiquery attention scales up to

sequence lengths of 8192--32,768 tokens (batch sizes 512 and 128 respectively), attention may take only 8--31% of total runtime.

[0294] The Case Study considered a variant of PaLM 540B with the parallel formulation of Transformer block replaced by serial attention/feed-forward layers. During generation, we use 2D weight-stationary layout, 64 chips, and batch size 512. The serial formulation incurs 14% higher inference latency per step than the parallel version because of the increased communication time for activations. In the prefill phase, this difference shrinks because the weight-gathered layouts incur less activation communication.

[0295] The Case Study evaluated the Pareto frontier between efficiency and latency as we scale the model size for the PaLM family of models: 8B, 62B and 540B, with weights in either bfloat16 or int8. We use a context length 2048 and sweep over the batch size and chip count.

[0296] To meaningfully compare throughput across multiple model sizes with different chip count and batch sizes, we report the cost of an inference in terms of chip-seconds per token calculated as

$$\text{chip-seconds per token} = \frac{n_{chips} \cdot time}{BL}$$

[0297] This can be directly proportional to operational cost and inversely proportional to MFU.

[0298] Figure 23 shows the relationship between model size, latency, and cost in the generate phase, at the Pareto frontier of optimal batch size, chip count, and partitioning strategy. The lowest cost is achieved at batch sizes larger than about 512, where the cost is proportional to the number of parameters. As we decrease the batch size, we improve the latency but incur higher cost per token. The minimum latency for generation is 3 times lower than the batch-512 latency. We observe that int8 weight quantization achieves the minimum latency in Figure 23: for example, we achieve 28.5 ms/token with int8 weights at batch size 64 on PaLM 540B, while we achieve 36.9 ms/token with bfloat16 weights. At low latency targets the cost is improved just over a factor of 2, because low-batch-size cost is dominated by weight loading time. At large batch size, cost is more neutral between int8 and bfloat16, because large-batch cost is dominated by the compute time and the matmuls still use bfloat16 arithmetic. We believe that quantization of activations to int8 could enable a further cost improvement.

[0299] Figure 24 shows the relationship between model size, latency, and cost in the prefill phase. The tradeoff between batch size and latency is less severe in the prefill phase

than the generate phase and even batch size 1 runs with fairly low cost. Further, the cost of batch-512 prefill is 2 times lower than batch-512 generate because of the increased MFU of the weight-gathered layouts we use during prefill. More details on the relationship between model size and MFU are presented in Figures 25 and 26.

Table 2. Example configurations for PaLM 540B, in the same setting as Figures 23 and 24. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D) and Weight Gathered XYZ (WG XYZ).

	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	64	64	64	64
Batch	1	64	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	43%	14%	76%	33%
Latency	0.29s	1.82s	85.2s	6.0s

Table 3. Example configurations for PaLM 62B, in the same setting Figures 23 and 24. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D) and Weight Gathered XYZ (WG XYZ).

	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	16	16	32	8
Batch	1	32	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	36%	8%	73%	37%
Latency	0.16s	0.73s	20.2s	5.1s

[0300] Tables 2 and 3 show some configurations from the Pareto frontiers of Figures 23 and 24 on PaLM 540B and PaLM 62B. In the low-latency scenarios we combine batch-1 prefill with batch 32-to-64 decode: batch size 1 achieves best latency in the prefill phase, but for the generate phase we can increase the batch size up to 64 with negligible latency impact, and doing so is dramatically better for generate MFU. This mixture of batch sizes is possible in practice either by generating multiple samples from the same input text, or by pipelining a batch-1 prefill server into a batch-64 decoding server.

[0301] In the high-throughput scenarios of Tables 2 and 3, we use larger batch sizes and we switch partitioned layouts between prefill and decode. We use bfloat16 weights for high-throughput scenario, because the weight-loading time is unimportant at large batch sizes, and because our software is missing some optimizations for large-batch int8 mode.

[0302] Comparing 62B (Table 3) vs.\ 540B models (Table 2), we find that we use more chips for the 540B model, but similar batch sizes and the same partitioned layouts. High-throughput MFUs are similar between the model sizes. The low-batch-size latencies grow sublinearly with model size at the Pareto frontier of this Case Study: even though larger models load proportionally more weights from memory, we can partition them across more

chips before becoming communication-limited. We estimate an approximately square-root relationship between model size and latency based on Figure 23.

[0303] Figure 27 records compared performance measurements of our Case Study against the FasterTransformer benchmarks across a wide range of batch sizes and configurations of prefill and generate. There are multiple differences between our benchmark setup and the FasterTransformer benchmark. In particular, we use different types of chips and chip counts -- FasterTransformer uses 16--32 NVIDIA A100s with 80GiB HBM, while we use 64 Google TPU v4 chips with 32GiB HBM. Therefore, we report throughput numbers in terms of MFU, which normalizes for both chip count and chip FLOPS.

[0304] We benchmark the Megatron 530B model and the similarly-sized PaLM 540B model, which has architectural optimizations including multiquery attention and parallel attention/feedforward layers.

Table 4

	PaLM 540B	**Megatron 530B**
n_{params}	540B	530B
n_{layers}	118	105
d_{model}	18432	20480
d_{ff}	73728	81920
n_{heads}	48	128
d_{head}	256	160
Attention	Multiquery	Multihead
Parallel ffn/attn	Yes	No

[0305] In this section, we report the latency and MFU of our implementations of both the PaLM 540B model and the MegatronTuring NLG 530B model run on 64 TPU v4 chips, in comparison to FasterTransformer baselines. We first note the model architecture differences above. Then, we report the full set of comparisons for the three configurations in the FasterTransformer benchmarks: 20 input tokens and 8 output tokens, 60 input tokens and 20 output tokens, and 128 input tokens and 8 output tokens. For each table we report the Pareto frontier of latency and MFU with bold font (frontier across all 500B-class results) and underline (frontier across MT-NLG specifically). This frontier is not a per-row comparison, but instead is defined globally across the table. It is defined as follows: a benchmark result (latency, MFU) is on the Pareto frontier if, for all other benchmark results (latency2 , MFU2), either latency $\leq$ latency2 or MFU $\geq$ MFU2 (or both) is true.

	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)							
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total	
batch	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU
1	565	1%	431	1%	842	0%	-	-	-	-	-	-	-	-
2	598	2%	455	1%	860	1%	-	-	-	-	-	-	-	-
4	616	4%	493	2%	867	2%	34	14%	255	1%	**289**	**2%**	**289**	**2%**
8	660	7%	523	5%	929	3%	40	25%	226	2%	**265**	**5%**	304	4%
16	730	13%	575	8%	1049	6%	58	34%	234	3%	**292**	**9%**	339	8%
32	865	22%	672	14%	1283	10%	99	40%	235	7%	**334**	**16%**	420	13%
64	1191	32%	942	20%	1722	15%	186	42%	265	12%	**451**	**24%**	532	20%
128	1862	41%	1431	27%	2124	24%	356	44%	312	20%	**668**	**33%**	740	29%
256	**3341**	**46%**	2483	31%	3140	32%	668	47%	415	30%	**1083**	**41%**	1151	38%
512	-	-	-	-	-	-	1366	46%	671	37%	**2037**	**43%**	2151	40%
1024	-	-	-	-	-	-	2785	45%	1257	40%	4041	44%	4082	42%

Table 5. Results for the 20-input-token, 8-output-token benchmark. All times in milliseconds. The bold and underline annotations are not per row, but instead show the Pareto frontier of time vs. MFU.

batch	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)							
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total	
	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU
1	1379	1%	1037	1%	2085	1%	-	-	-	-	-	-	-	-
2	1515	2%	1110	2%	2122	1%	-	-	-	-	-	-	-	-
4	1512	4%	1198	3%	2184	2%	50	29%	640	1%	690	3%	678	3%
8	1631	8%	1295	5%	2367	4%	80	37%	574	2%	**653**	**6%**	728	5%
16	1868	15%	1454	9%	2753	7%	153	39%	602	3%	**755**	**10%**	838	9%
32	2361	23%	1804	15%	3543	10%	270	44%	626	6%	**896**	**18%**	1058	15%
64	3383	32%	2646	21%	4117	18%	501	47%	717	11%	**1218**	**26%**	1275	24%
128	5406	40%	4099	27%	5319	27%	985	48%	829	19%	**1814**	**35%**	1902	32%
256	OOM	-	7203	30%	8318	35%	2041	46%	1114	28%	**3155**	**40%**	3189	39%
512	-	-	-	-	-	-	4167	45%	1743	36%	**5910**	**43%**	6210	40%
1024	-	-	-	-	-	-	8349	45%	3260	39%	11608	43%	12390	40%

Table 6 Results for the 60-input-token, 20-output-token benchmark. All times in milliseconds. The bold and underline annotations are not per row, but instead show the Pareto frontier of time vs. MFU.

	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)									
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total			
batch	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU		
1	585	5%	451	3%	866	2%	-	-	-	-	-	-				
2	667	9%	508	6%	932	4%	-	-	-	-	-	-				
4	765	15%	606	10%	1097	7%	81	39%	258	1%	343	10%	**338**	**10%**		
8	990	23%	766	15%	1434	11%	149	42%	234	2%	403	17%	**384**	**16%**		
16	1377	34%	1074	22%	2104	15%	287	44%	253	3%	586	23%	**540**	**23%**		
32	2251	41%	1741	27%	2623	23%	536	47%	263	6%	**796**	**34%**	799	33%		
64	4002	46%	3114	30%	3578	34%	1056	48%	317	10%	**1329**	**40%**	1372	39%		
128	OOM	-	5784	32%	5512	45%	2202	46%	381	17%	**2343**	**46%**	2583	45%		
256	OOM	-	11232	33%	**9614**	**51%**	4479	45%	431	29%	4710	45%	4911	45%		
512	-	-	-	-	-	-	8913	45%	734	34%	9673	44%	9647	43%		
1024	-	-	-	-	-	-	17766	45%	1370	37%	19723	43%	19136	43%		

Table 7 Results for the 128-input-token, 8-output-token benchmark. All times in milliseconds. The bold and underline annotations are not per row, but instead show the Pareto frontier of time vs. MFU.

[0306] The Case Study implementation of PaLM 540B achieves the best absolute latency in the comparison, and the Case Study implementation also offers the best MFU for the Megatron model for all but one latency target. FasterTransformer reports results with 8-, 16-, and 32-way tensor parallelism. Their 32-way tensor parallelism achieves a maximum of 33% MFU across all reported benchmarks, compared to 46% MFU in their 16-way tensor parallel configuration. This likely indicates a communication bottleneck of scaling tensor parallelism beyond this point. In contrast, our implementation is able to scale up to 64-way tensor parallelism while still achieving 44% MFU. We also note that our benchmarks throughout the Case Study include more challenging inference scenarios, such as context lengths in the range 1024--4096, and report the inference latency for the generate phase and the prefill phase separately (since they have different characteristics).

[0307] Figure 28 shows the Pareto frontier of cost vs. latency for the Case Study as we sweep sequence length from 32 to 1024 at batch size 1.

[0308] Figure 29 depicts a flowchart of a method 2900 for improved parallelization of an autoregressive machine-learned sequence processing model, according to aspects of the present disclosure.

[0309] One or more portion(s) of example method 2900 can be implemented by a computing system that includes one or more computing devices such as, for example, computing systems described with reference to the other figures. Each respective portion of example method 2900 can be performed by any (or any combination) of one or more computing devices. Moreover, one or more portion(s) of example method 2900 can be implemented on the hardware components of the device(s) described herein, for example, to train one or more systems or models. Figure 29 depicts elements performed in a particular order for purposes of illustration and discussion. Those of ordinary skill in the art, using the disclosures provided herein, will understand that the elements of any of the methods discussed herein can be adapted, rearranged, expanded, omitted, combined, or modified in various ways without deviating from the scope of the present disclosure. Figure 29 is described with reference to elements/terms described with respect to other systems and figures for exemplary illustrated purposes and is not meant to be limiting. One or more portions of example method 2900 can be performed additionally, or alternatively, by other systems.

[0310] At 2902, example method 2900 can include receiving input requests to process a plurality of input sequences using the machine-learned sequence processing model to

generate a plurality of output sequences respectively corresponding to the plurality of input sequences. For example,

[0311] At 2904, example method 2900 can include generating a plurality of initial attention tensors respectively for the plurality of input sequences. In example method 2900, one or more respective initial attention tensors can be generated for each respective input sequence in parallel over input elements of the respective input sequence. In example method 2900, the one or more respective initial attention tensors can be generated in one or more batches having a first batch size using a prefill system that comprises one or more prefill computing devices and executes one or more layers of the machine-learned sequence processing model.

[0312] At 2906, example method 2900 can include autoregressively generating, using the plurality of initial attention tensors, a plurality of output elements for each of the plurality of output sequences in one or more batches having a second batch size. In example method 2900, the plurality of output elements can be generated using a generation system that includes one or more generation computing devices. In example method 2900, at least a portion of the machine-learned sequence processing model can be partitioned over the one or more generation computing devices along a batch dimension.

[0313] Example method 2900 can include orchestrating the inputs to the prefill system and the outputs from the generation system with a router system that is configured to receive the input requests.

[0314] In example method 2900, at least one of the prefill computing devices can be different from at least one of the generation computing devices.

[0315] In example method 2900, the prefill system can be configured for a workflow having a compute bottleneck.

[0316] In example method 2900, the prefill system can be configured with a lower memory bandwidth as compared to the generation system.

[0317] In example method 2900, the generation system can be configured for a workflow having a memory bandwidth bottleneck.

[0318] In example method 2900, the generation system can include faster high-bandwidth memory (HBM) as compared to the prefill system.

[0319] In example method 2900, the generation system can include larger static random-access memory as compared to the prefill system.

[0320] In example method 2900, the generation computing devices can include a higher memory bandwidth than the prefill computing devices.

[0321] In example method 2900, the generation system can implement more processing cores than the prefill system.

[0322] In example method 2900, the generation computing devices can be optimized for a training workflow.

[0323] In example method 2900, the second batch size can be larger than the first batch size.

[0324] In example method 2900, the first batch size can be one.

[0325] In example method 2900, the one or more respective initial attention tensors can describe a relatedness of the respective plurality of input elements to one another.

[0326] In example method 2900, autoregressively generating the plurality of output elements can include processing, using the generation system, the initial attention tensors to determine a likely next output element. In example method 2900, autoregressively generating the plurality of output elements can include selecting the likely next output element to include in the output sequence. In example method 2900, autoregressively generating the plurality of output elements can include updating the initial attention tensors based on the selected output element.

[0327] Example method 2900 can include routing, based on a length of an input sequence, the input sequence to a first group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-gathered partitioning layout.

[0328] Example method 2900 can include routing, based on a length of an input sequence, the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-stationary partitioning layout.

[0329] Example method 2900 can include routing, based on a length of an input sequence, the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout in which weights are stationary along at least one parallelization axis and transferred along at least one other parallelization axis.

[0330] Example method 2900 can include, for a length above a threshold, routing the input sequence to a second group of prefill computing devices that implement the machine-

learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a first number of axes along which weights are transferred.

[0331] Example method 2900 can include, for a length below a threshold, routing the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a second number of axes along which weights are transferred. In example method 2900, the first number can be greater than the second number.

[0332] Example method 2900 can include selecting, based on an expected length of the one or more output sequences, a first group of generation computing devices to generate the one or more output sequences, wherein the first group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-gathered partitioning layout.

[0333] Example method 2900 can include selecting, based on an expected length of the one or more output sequences, a second group of generation computing devices to generate the one or more output sequences, wherein the second group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-stationary partitioning layout.

[0334] Example method 2900 can include selecting, based on an expected length of the one or more output sequences, a group of generation computing devices to generate the one or more output sequences, wherein the group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout in which weights are stationary along at least one parallelization axis and transferred along at least one other parallelization axis.

[0335] Example method 2900 can include, for an expected length above a threshold, selecting a group of generation computing devices to generate the one or more output sequences, wherein the group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a first number of axes along which weights are transferred.

[0336] Example method 2900 can include, for an expected length below a threshold, selecting a group of generation computing devices to generate the one or more output sequences, wherein the group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a second number of axes along which weights are transferred. In example method 2900, the first number can be greater than the second number.

[0337] Example method 2900 can include implementing a weight-stationary partitioning layout using a plurality of computing devices arranged in a three-dimensional toroidal configuration. In example method 2900, a chip count along a first dimension can be one-fourth the chip count of a product of the other two dimensions.

[0338] Example method 2900 can include implementing a weight-gathered partitioning layout using a plurality of computing devices arranged in a three-dimensional toroidal configuration. In example method 2900, a chip count along a first dimension can be one-fourth the chip count of a product of the other two dimensions.

[0339] Example method 2900 can include implementing a hybrid partitioning layout in which both weights and activations are partially transferred along different axes.

[0340] Example method 2900 can include sharing a weight layout between the weight-stationary layout and the weight-gathered layout.

[0341] In example method 2900, a feed-forward network of the machine-learned sequence processing model can be partitioned according to a weight-stationary partitioning layout. In example method 2900, a feed-forward network of the machine-learned sequence processing model can be partitioned according to a weight-gathered partitioning layout. In example method 2900, a feed-forward network of the machine-learned sequence processing model can be partitioned according to a combination thereof.

[0342] Example method 2900 can include determining an expected ratio of input elements and output elements. Example method 2900 can include determining a ratio of the number of prefill computing devices and a number of generation computing devices decode servers based on the expected ratio of input elements and output elements.

[0343] Example method 2900 can include routing a processing task to a group of generation computing devices based on a buffer size of the group and an expected output length.

[0344] Example method 2900 can include shifting a processing task to a different group of generation computing devices with a larger buffer size mid-completion when an actual output length is determined to exceed an expected output length.

[0345] Example method 2900 can include generating an initial attention tensor incrementally by receiving a prior attention tensor. Example method 2900 can include generating an initial attention tensor incrementally by computing, in parallel for an incremental set of input elements, an incremental attention tensor component. Example method 2900 can include generating an initial attention tensor incrementally by combining the prior attention tensor with the incremental attention tensor component to obtain the initial attention tensor.

[0346] In example method 2900, the initial attention tensor can be obtained incrementally over one or more turns in a multi-turn content generation session.

[0347] In example method 2900, the multi-turn content generation session can include a chat conversation. In example method 2900, the multi-turn content generation session can include a series of instructions to a tool (e.g., an image generation tool, a text generation tool, etc.).

[0348] Figure 30 depicts a flowchart of a method 3000 for improved parallelization of an autoregressive machine-learned sequence processing model, according to aspects of the present disclosure.

[0349] One or more portion(s) of example method 3000 can be implemented by a computing system that includes one or more computing devices such as, for example, computing systems described with reference to the other figures. Each respective portion of example method 3000 can be performed by any (or any combination) of one or more computing devices. Moreover, one or more portion(s) of example method 3000 can be implemented on the hardware components of the device(s) described herein, for example, to train one or more systems or models. Figure 30 depicts elements performed in a particular order for purposes of illustration and discussion. Those of ordinary skill in the art, using the disclosures provided herein, will understand that the elements of any of the methods discussed herein can be adapted, rearranged, expanded, omitted, combined, or modified in various ways without deviating from the scope of the present disclosure. Figure 30 is described with reference to elements/terms described with respect to other systems and figures for exemplary illustrated purposes and is not meant to be limiting. One or more portions of example method 3000 can be performed additionally, or alternatively, by other systems.

[0350] At 3002, example method 3000 can include receiving, by a computing system comprising a plurality of hardware accelerator devices, an input.

[0351] At 3004, example method 3000 can include executing, using the plurality of hardware accelerator devices, a forward pass of an attention mechanism of the network, wherein the attention mechanism is partitioned over a batch dimension of the network.

[0352] At 3006, example method 3000 can include returning an output generated based on the forward pass.

[0353] For example, the attention mechanism can be a multiquery attention mechanism of a machine-learned transformer block.

[0354] Figure 31 depicts a flowchart of a method 3100 for improved parallelization of an autoregressive machine-learned sequence processing model, according to aspects of the present disclosure.

[0355] One or more portion(s) of example method 3100 can be implemented by a computing system that includes one or more computing devices such as, for example, computing systems described with reference to the other figures. Each respective portion of example method 3100 can be performed by any (or any combination) of one or more computing devices. Moreover, one or more portion(s) of example method 3100 can be implemented on the hardware components of the device(s) described herein, for example, to train one or more systems or models. Figure 31 depicts elements performed in a particular order for purposes of illustration and discussion. Those of ordinary skill in the art, using the disclosures provided herein, will understand that the elements of any of the methods discussed herein can be adapted, rearranged, expanded, omitted, combined, or modified in various ways without deviating from the scope of the present disclosure. Figure 31 is described with reference to elements/terms described with respect to other systems and figures for exemplary illustrated purposes and is not meant to be limiting. One or more portions of example method 3100 can be performed additionally, or alternatively, by other systems.

[0356] At 3102, example method 3100 can include receiving, by a computing system comprising a plurality of hardware accelerator devices, an input. In some implementations, the plurality of hardware accelerator devices communicate along a topology characterized by one or more axes, wherein each respective hardware accelerator device of the plurality of hardware accelerator devices comprises one or more respective processor chips and a respective memory storing an initial weight partition of a weight tensor of a network of a machine-learned model according to an initial partitioning of the weight tensor.

[0357] At 3104, example method 3100 can include executing, using the plurality of hardware accelerator devices, a first forward pass of the network using a first layout of the weight tensor. The first layout can be a weight-stationary layout or a weight gathered layout.

[0358] At 3106, example method 3100 can include executing, using the plurality of hardware accelerator devices, a second forward pass of the network using a second layout of the weight tensor. The second layout can be a weight-stationary layout or a weight gathered layout. The second layout can be different from the first layout. The second layout can be a weight stationary layout. The second layout can correspond to the initial partitioning of the weight tensor.

[0359] At 3108, example method 3100 can include returning an output generated based on the first forward pass and the second forward pass.

[0360] In some implementations, a weight-gathered layout is generated by broadcasting weights across a partitioning axis over which the weight tensor is initially partitioned. In some implementations, a weight stationary layout is generated by broadcasting weights across a partitioning axis over which the weight tensor is initially partitioned.

[0361] In some implementations, executing the first forward pass includes obtaining a first input activation tensor of the network based on the input, the first input activation tensor distributed over the plurality of hardware accelerator devices, wherein the first input activation tensor is partitioned along one or more first partition axes of the one or more axes. In some implementations, executing the first forward pass includes broadcasting, to the plurality of hardware accelerators, respective copies of weights gathered over the one or more first partition axes. In some implementations, executing the first forward pass includes computing, based on processing the first input activation tensor using the respective copies of gathered weights at the plurality of hardware accelerators, a first updated activation tensor partitioned over the one or more first partition axes.

[0362] In some implementations, executing the second forward pass includes obtaining a second input activation tensor of the network, the second input activation tensor partitioned over the first slice of hardware accelerator devices. In some implementations, executing the second forward pass includes computing, based on processing the second input activation tensor using the respective partitions of weights at the plurality of hardware accelerators, a second updated activation tensor partitioned over the first slice.

[0363] Figure 32 depicts a flowchart of a method 3200 for training one or more machine-learned models according to aspects of the present disclosure. For instance, an

example machine-learned model can include a machine-learned sequence processing model implemented on a model host 106.

[0364] One or more portion(s) of example method 3200 can be implemented by a computing system that includes one or more computing devices such as, for example, computing systems described with reference to the other figures. Each respective portion of example method 3200 can be performed by any (or any combination) of one or more computing devices. Moreover, one or more portion(s) of example method 3200 can be implemented on the hardware components of the device(s) described herein, for example, to train one or more systems or models. Figure 32 depicts elements performed in a particular order for purposes of illustration and discussion. Those of ordinary skill in the art, using the disclosures provided herein, will understand that the elements of any of the methods discussed herein can be adapted, rearranged, expanded, omitted, combined, or modified in various ways without deviating from the scope of the present disclosure. Figure 32 is described with reference to elements/terms described with respect to other systems and figures for exemplary illustrated purposes and is not meant to be limiting. One or more portions of example method 3200 can be performed additionally, or alternatively, by other systems.

[0365] At 3202, example method 3200 can include obtaining a training instance. A set of training data can include a plurality of training instances divided between multiple datasets (e.g., a training dataset, a validation dataset, or testing dataset). A training instance can be labeled or unlabeled. Although referred to in example method 3200 as a "training" instance, it is to be understood that runtime inferences can form training instances when a model is trained using an evaluation of the model's performance on that runtime instance (e.g., online training/learning). Example data types for the training instance and various tasks associated therewith are described throughout the present disclosure.

[0366] At 3204, example method 3200 can include processing, using one or more machine-learned models, the training instance to generate an output. The output can be directly obtained from the one or more machine-learned models or can be a downstream result of a chain of processing operations that includes an output of the one or more machine-learned models.

[0367] At 3206, example method 3200 can include receiving an evaluation signal associated with the output. The evaluation signal can be obtained using a loss function. Various determinations of loss can be used, such as mean squared error, likelihood loss, cross entropy loss, hinge loss, contrastive loss, or various other loss functions. The evaluation

signal can be computed using known ground-truth labels (e.g., supervised learning), predicted or estimated labels (e.g., semi- or self-supervised learning), or without labels (e.g., unsupervised learning). The evaluation signal can be a reward (e.g., for reinforcement learning). The reward can be computed using a machine-learned reward model configured to generate rewards based on output(s) received. The reward can be computed using feedback data describing human feedback on the output(s).

[0368] At 3208, example method 3200 can include updating the machine-learned model using the evaluation signal. For example, values for parameters of the machine-learned model(s) can be learned, in some embodiments, using various training or learning techniques, such as, for example, backwards propagation. For example, the evaluation signal can be backpropagated from the output (or another source of the evaluation signal) through the machine-learned model(s) to update one or more parameters of the model(s) (e.g., based on a gradient of the evaluation signal with respect to the parameter value(s)). For example, system(s) containing one or more machine-learned models can be trained in an end-to-end manner. Gradient descent techniques can be used to iteratively update the parameters over a number of training iterations. In some implementations, performing backwards propagation of errors can include performing truncated backpropagation through time. Example method 3200 can include implementing a number of generalization techniques (e.g., weight decays, dropouts, etc.) to improve the generalization capability of the models being trained.

[0369] In some implementations, example method 3200 can be implemented for training a machine-learned model from an initialized state to a fully trained state (e.g., when the model exhibits a desired performance profile, such as based on accuracy, precision, recall, etc.).

[0370] In some implementations, example method 3200 can be implemented for particular stages of a training procedure. For instance, in some implementations, example method 3200 can be implemented for pre-training a machine-learned model. Pre-training can include, for instance, large-scale training over potentially noisy data to achieve a broad base of performance levels across a variety of tasks/data types. In some implementations, example method 3200 can be implemented for fine-tuning a machine-learned model. Fine-tuning can include, for instance, smaller-scale training on higher-quality (e.g., labeled, curated, etc.) data. Fine-tuning can affect all or a portion of the parameters of a machine-learned model. For example, various portions of the machine-learned model can be "frozen" for certain training stages. For example, parameters associated with an embedding space can be "frozen" during fine-tuning (e.g., to retain information learned from a broader domain(s) than present

in the fine-tuning dataset(s)). An example fine-tuning approach includes reinforcement learning. Reinforcement learning can be based on user feedback on model performance during use.

[0371] Figure 33 is a block diagram of an example processing flow for using machine-learned model(s) 1 to process input(s) 2 to generate output(s) 3. Any feature described herein with respect to machine-learned model(s) 1 is to be understood as descriptive of various attributes and feature of the models described herein with respect to any of the Figures, with machine-learned model(s) 1 being used as a representative example for the sake of clarity and concision.

[0372] Machine-learned model(s) 1 can be or include one or multiple machine-learned models or model components. Example machine-learned models can include neural networks (e.g., deep neural networks). Example machine-learned models can include non-linear models or linear models. Example machine-learned models can use other architectures in lieu of or in addition to neural networks. Example machine-learned models can include decision tree based models, support vector machines, hidden Markov models, Bayesian networks, linear regression models, k-means clustering models, etc.

[0373] Example neural networks can include feed-forward neural networks, recurrent neural networks (RNNs), including long short-term memory (LSTM) based recurrent neural networks, convolutional neural networks (CNNs), diffusion models, generative-adversarial networks, or other forms of neural networks. Example neural networks can be deep neural networks. Some example machine-learned models can leverage an attention mechanism such as self-attention. For example, some example machine-learned models can include multi-headed self-attention models.

[0374] Machine-learned model(s) 1 can include a single or multiple instances of the same model configured to operate on data from input(s) 2. Machine-learned model(s) 1 can include an ensemble of different models that can cooperatively interact to process data from input(s) 2. Machine-learned model(s) 1 can employ a mixture-of-experts structure. *See, e.g.*, Zhou et al., *Mixture-of-Experts with Expert Choice Routing*, ARXIV:2202.09368v2 (Oct. 14, 2022).

[0375] Input(s) 2 can generally include or otherwise represent various types of data. Input(s) 2 can include one type or many different types of data. Output(s) 3 can be data of the same type(s) or of different types of data as compared to input(s) 2. Output(s) 3 can include one type or many different types of data.

[0376] Example data types for input(s) 2 or output(s) 3 include natural language text data, software code data (e.g., source code, object code, machine code, or any other form of computer-readable instructions or programming languages), machine code data (e.g., binary code, assembly code, or other forms of machine-readable instructions that can be executed directly by a computer's central processing unit), assembly code data (e.g., low-level programming languages that use symbolic representations of machine code instructions to program a processing unit), genetic data or other chemical or biochemical data, image data, audio data, audiovisual data, haptic data, biometric data, medical data, financial data, statistical data, geographical data, astronomical data, historical data, sensor data generally (e.g., digital or analog values, such as voltage or other absolute or relative level measurement values from a real or artificial input, such as from an audio sensor, light sensor, displacement sensor, etc.), and the like. Data can be raw or processed and can be in any format or schema.

[0377] In multimodal inputs 2 or outputs 3, example combinations of data types include image data and audio data, image data and natural language data, natural language data and software code data, image data and biometric data, sensor data and medical data, etc. It is to be understood that any combination of data types in an input 2 or an output 3 can be present.

[0378] An example input 2 can include one or multiple data types, such as the example data types noted above. An example output 3 can include one or multiple data types, such as the example data types noted above. The data type(s) of input 2 can be the same as or different from the data type(s) of output 3. It is to be understood that the example data types noted above are provided for illustrative purposes only. Data types contemplated within the scope of the present disclosure are not limited to those examples noted above.

[0379] Figure 34 is a block diagram of an example implementation of an example machine-learned model configured to process sequences of information. For instance, an example implementation of machine-learned model(s) 1 can include machine-learned sequence processing model(s) 4. An example system can pass input(s) 2 to sequence processing model(s) 4. Sequence processing model(s) 4 can include one or more machine-learned components. Sequence processing model(s) 4 can process the data from input(s) 2 to obtain an input sequence 5. Input sequence 5 can include one or more input elements 5-1, 5-2, . . . , 5-*M*, etc. obtained from input(s) 2. Sequence processing model 4 can process input sequence 5 using prediction layer(s) 6 to generate an output sequence 7. Output sequence 7 can include one or more output elements 7-1, 7-2, . . . , 7-*N*, etc. generated based on input sequence 5. The system can generate output(s) 3 based on output sequence 7.

[0380] Sequence processing model(s) 4 can include one or multiple machine-learned model components configured to ingest, generate, or otherwise reason over sequences of information. For example, some example sequence processing models in the text domain are referred to as "Large Language Models," or LLMs. *See, e.g.*, PaLM 2 Technical Report, GOOGLE, https://ai.google/static/documents/palm2techreport.pdf (n.d.). Other example sequence processing models can operate in other domains, such as image domains, *see, e.g.*, Dosovitskiy et al., *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, ARXIV:2010.11929v2 (Jun. 3, 2021), audio domains, *see, e.g.*, Agostinelli et al., *MusicLM: Generating Music From Text*, ARXIV:2301.11325v1 (Jan. 26, 2023), biochemical domains, *see, e.g.*, Jumper et al., Highly accurate protein structure prediction with AlphaFold, 596 Nature 583 (Aug. 26, 2021), by way of example. Sequence processing model(s) 4 can process one or multiple types of data simultaneously. Sequence processing model(s) 4 can include relatively large models (e.g., more parameters, computationally expensive, etc.), relatively small models (e.g., fewer parameters, computationally lightweight, etc.), or both.

[0381] In general, sequence processing model(s) 4 can obtain input sequence 5 using data from input(s) 2. For instance, input sequence 5 can include a representation of data from input(s) 2 in a format understood by sequence processing model(s) 4. One or more machine-learned components of sequence processing model(s) 4 can ingest the data from input(s) 2, parse the data into pieces compatible with the processing architectures of sequence processing model(s) 4 (e.g., via "tokenization"), and project the pieces into an input space associated with prediction layer(s) 6 (e.g., via "embedding").

[0382] Sequence processing model(s) 4 can ingest the data from input(s) 2 and parse the data into a sequence of elements to obtain input sequence 5. For example, a portion of input data from input(s) 2 can be broken down into pieces that collectively represent the content of the portion of the input data. The pieces can provide the elements of the sequence.

[0383] Elements 5-1, 5-2, . . . , 5-*M* can represent, in some cases, building blocks for capturing or expressing meaningful information in a particular data domain. For instance, the elements can describe "atomic units" across one or more domains. For example, for textual input source(s), the elements can correspond to groups of one or more words or sub-word components, such as sets of one or more characters.

[0384] For example, elements 5-1, 5-2, . . . , 5-*M* can represent tokens obtained using a tokenizer. For instance, a tokenizer can process a given portion of an input source and output a series of tokens (e.g., corresponding to input elements 5-1, 5-2, . . . , 5-*M*) that represent the portion of the input source. Various approaches to tokenization can be used. For

instance, textual input source(s) can be tokenized using a byte-pair encoding (BPE) technique. *See, e.g.*, Kudo et al., *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*, PROCEEDINGS OF THE 2018 CONFERENCE ON EMPIRICAL METHODS IN NATURAL LANGUAGE PROCESSING (System Demonstrations), pages 66–71 (October 31–November 4, 2018), https://aclanthology.org/D18-2012.pdf. Image-based input source(s) can be tokenized by extracting and serializing patches from an image.

[0385] In general, arbitrary data types can be serialized and processed into input sequence 5. It is to be understood that element(s) 5-1, 5-2, . . . , 5-*M* can be the tokens or can be the embedded representations thereof.

[0386] Prediction layer(s) 6 can predict one or more output elements 7-1, 7-2, . . . , 7-*N* based on the input elements. Prediction layer(s) 6 can include one or more machine-learned model architectures, such as one or more layers of learned parameters that manipulate and transform the input(s) to extract higher-order meaning from, and relationships between, input element(s) 5-1, 5-2, . . . , 5-*M*. In this manner, for instance, example prediction layer(s) 6 can predict new output element(s) in view of the context provided by input sequence 5.

[0387] Prediction layer(s) 6 can evaluate associations between portions of input sequence 5 and a particular output element. These associations can inform a prediction of the likelihood that a particular output follows the input context. For example, consider the textual snippet, "The carpenter's toolbox was small and heavy. It was full of ___." Example prediction layer(s) 6 can identify that "It" refers back to "toolbox" by determining a relationship between the respective embeddings. Example prediction layer(s) 6 can also link "It" to the attributes of the toolbox, such as "small" and "heavy." Based on these associations, prediction layer(s) 6 can, for instance, assign a higher probability to the word "nails" than to the word "sawdust."

[0388] A transformer is an example architecture that can be used in prediction layer(s) 4. *See, e.g.*, Vaswani et al., *Attention Is All You Need*, ARXIV:1706.03762v7 (Aug. 2, 2023). A transformer is an example of a machine-learned model architecture that uses an attention mechanism to compute associations between items within a context window. The context window can include a sequence that contains input sequence 5 and potentially one or more output element(s) 7-1, 7-2, . . . , 7-*N*. A transformer block can include one or more attention layer(s) and one or more post-attention layer(s) (e.g., feedforward layer(s), such as a multi-layer perceptron).

[0389] Prediction layer(s) 6 can include other machine-learned model architectures in addition to or in lieu of transformer-based architectures. For example, recurrent neural networks (RNNs) and long short-term memory (LSTM) models can also be used, as well as convolutional neural networks (CNNs). In general, prediction layer(s) 6 can leverage various kinds of artificial neural networks that can understand or generate sequences of information.

[0390] Output sequence 7 can include or otherwise represent the same or different data types as input sequence 5. For instance, input sequence 5 can represent textual data, and output sequence 7 can represent textual data. Input sequence 5 can represent image, audio, or audiovisual data, and output sequence 7 can represent textual data (e.g., describing the image, audio, or audiovisual data). It is to be understood that prediction layer(s) 6, and any other interstitial model components of sequence processing model(s) 4, can be configured to receive a variety of data types in input sequence(s) 5 and output a variety of data types in output sequence(s) 7.

[0391] Output sequence 7 can have various relationships to input sequence 5. Output sequence 7 can be a continuation of input sequence 5. Output sequence 7 can be complementary to input sequence 5. Output sequence 7 can translate, transform, augment, or otherwise modify input sequence 5. Output sequence 7 can answer, evaluate, confirm, or otherwise respond to input sequence 5. Output sequence 7 can implement (or describe instructions for implementing) an instruction provided via input sequence 5.

[0392] Output sequence 7 can be generated autoregressively. For instance, for some applications, an output of one or more prediction layer(s) 6 can be passed through one or more output layers (e.g., softmax layer) to obtain a probability distribution over an output vocabulary (e.g., a textual or symbolic vocabulary) conditioned on a set of input elements in a context window. In this manner, for instance, output sequence 7 can be autoregressively generated by sampling a likely next output element, adding that element to the context window, and re-generating the probability distribution based on the updated context window, and sampling a likely next output element, and so forth.

[0393] Output sequence 7 can also be generated non-autoregressively. For instance, multiple output elements of output sequence 7 can be predicted together without explicit sequential conditioning on each other. *See, e.g.*, Saharia et al., Non-Autoregressive Machine Translation with Latent Alignments, ARXIV:2004.07437v3 (Nov. 16, 2020).

[0394] Output sequence 7 can include one or multiple portions or elements. In an example content generation configuration, output sequence 7 can include multiple elements corresponding to multiple portions of a generated output sequence (e.g., a textual sentence,

values of a discretized waveform, computer code, etc.). In an example classification configuration, output sequence 7 can include a single element associated with a classification output. For instance, an output "vocabulary" can include a set of classes into which an input sequence is to be classified. For instance, a vision transformer block can pass latent state information to a multilayer perceptron that outputs a likely class value associated with an input image.

[0395] Figure 35 is a block diagram of an example technique for populating an example input sequence 8. Input sequence 8 can include various functional elements that form part of the model infrastructure, such as an element 8-0 obtained from a task indicator 9 that signals to any model(s) that process input sequence 8 that a particular task is being performed (e.g., to help adapt a performance of the model(s) to that particular task). Input sequence 8 can include various data elements from different data modalities. For instance, an input modality 10-1 can include one modality of data. A data-to-sequence model 11-1 can process data from input modality 10-1 to project the data into a format compatible with input sequence 8 (e.g., one or more vectors dimensioned according to the dimensions of input sequence 8) to obtain elements 8-1, 8-2, 8-3. Another input modality 10-2 can include a different modality of data. A data-to-sequence model 11-2 can project data from input modality 10-2 into a format compatible with input sequence 8 to obtain elements 8-4, 8-5, 8-6. Another input modality 10-3 can include yet another different modality of data. A data-to-sequence model 11-3 can project data from input modality 10-3 into a format compatible with input sequence 8 to obtain elements 8-7, 8-8, 8-9.

[0396] Input sequence 8 can be the same as or different from input sequence 5. Input sequence 8 can be a multimodal input sequence that contains elements that represent data from different modalities using a common dimensional representation. For instance, an embedding space can have P dimensions. Input sequence 8 can be configured to contain a plurality of elements that have P dimensions. In this manner, for instance, example implementations can facilitate information extraction and reasoning across diverse data modalities by projecting data into elements in the same embedding space for comparison, combination, or other computations therebetween.

[0397] For example, elements 8-0, . . . , 8-9 can indicate particular locations within a multidimensional embedding space. Some elements can map to a set of discrete locations in the embedding space. For instance, elements that correspond to discrete members of a predetermined vocabulary of tokens can map to discrete locations in the embedding space that are associated with those tokens. Other elements can be continuously distributed across

the embedding space. For instance, some data types can be broken down into continuously defined portions (e.g., image patches) that can be described using continuously distributed locations within the embedding space.

[0398] In some implementations, the expressive power of the embedding space may not be limited to meanings associated with any particular set of tokens or other building blocks. For example, a continuous embedding space can encode a spectrum of high-order information. An individual piece of information (e.g., a token) can map to a particular point in that space: for instance, a token for the word "dog" can be projected to an embedded value that points to a particular location in the embedding space associated with canine-related information. Similarly, an image patch of an image of a dog on grass can also be projected into the embedding space. In some implementations, the projection of the image of the dog can be similar to the projection of the word "dog" while also having similarity to a projection of the word "grass," while potentially being different from both. In some implementations, the projection of the image patch may not exactly align with any single projection of a single word. In some implementations, the projection of the image patch can align with a combination of the projections of the words "dog" and "grass." In this manner, for instance, a high-order embedding space can encode information that can be independent of data modalities in which the information is expressed.

[0399] Task indicator 9 can include a model or model component configured to identify a task being performed and inject, into input sequence 8, an input value represented by element 8-0 that signals which task is being performed. For instance, the input value can be provided as a data type associated with an input modality and projected along with that input modality (e.g., the input value can be a textual task label that is embedded along with other textual data in the input; the input value can be a pixel-based representation of a task that is embedded along with other image data in the input; etc.). The input value can be provided as a data type that differs from or is at least independent from other input(s). For instance, the input value represented by element 8-0 can be a learned within a continuous embedding space.

[0400] Input modalities 10-1, 10-2, and 10-3 can be associated with various different data types (e.g., as described above with respect to input(s) 2 and output(s) 3).

[0401] Data-to-sequence models 11-1, 11-2, and 11-3 can be the same or different from each other. Data-to-sequence models 11-1, 11-2, and 11-3 can be adapted to each respective input modality 10-1, 10-2, and 10-3. For example, a textual data-to-sequence model can subdivide a portion of input text and project the subdivisions into element(s) in

input sequence 8 (e.g., elements 8-1, 8-2, 8-3, etc.). An image data-to-sequence model can subdivide an input image and project the subdivisions into element(s) in input sequence 8 (e.g., elements 8-4, 8-5, 8-6, etc.). An arbitrary datatype data-to-sequence model can subdivide an input of that arbitrary datatype and project the subdivisions into element(s) in input sequence 8 (e.g., elements 8-7, 8-8, 8-9, etc.).

[0402] Data-to-sequence models 11-1, 11-2, and 11-3 can form part of machine-learned sequence processing model(s) 4. Data-to-sequence models 11-1, 11-2, and 11-3 can be jointly trained with or trained independently from machine-learned sequence processing model(s) 4. Data-to-sequence models 11-1, 11-2, and 11-3 can be trained end-to-end with machine-learned sequence processing model(s) 4.

[0403] Figure 36 is a block diagram of an example model development platform 12 that can facilitate creation, adaptation, and refinement of example machine-learned models (e.g., machine-learned model(s) 1, sequence processing model(s) 4, etc.). Model development platform 12 can provide a number of different toolkits that developer systems can employ in the development of new or adapted machine-learned models.

[0404] Model development platform 12 can provide one or more model libraries 13 containing building blocks for new models. Model libraries 13 can include one or more pre-trained foundational models 13-1, which can provide a backbone of processing power across various tasks. Model libraries 13 can include one or more pre-trained expert models 13-2, which can be focused on performance in particular domains of expertise. Model libraries 13 can include various model primitives 13-3, which can provide low-level architectures or components (optionally pre-trained), which can be assembled in various arrangements as desired.

[0405] Model development platform 12 can receive selections of various model components 14. Model development platform 12 can pass selected model components 14 to a workbench 15 that combines selected model components 14 into a development model 16.

[0406] Workbench 15 can facilitate further refinement and adaptation of development model 16 by leveraging a number of different toolkits integrated with model development platform 12. For example, workbench 15 can facilitate alignment of the development model 16 with a desired performance profile on various tasks using a model alignment toolkit 17.

[0407] Model alignment toolkit 17 can provide a number of tools for causing development model 16 to generate outputs aligned with desired behavioral characteristics. Alignment can include increasing an accuracy, precision, recall, etc. of model outputs. Alignment can include enforcing output styles, schema, or other preferential characteristics of

model outputs. Alignment can be general or domain-specific. For instance, a pre-trained foundational model 13-1 can begin with an initial level of performance across multiple domains. Alignment of the pre-trained foundational model 13-1 can include improving a performance in a particular domain of information or tasks (e.g., even at the expense of performance in another domain of information or tasks).

[0408] Model alignment toolkit 17 can integrate one or more dataset(s) 17-1 for aligning development model 16. Curated dataset(s) 17-1 can include labeled or unlabeled training data. Dataset(s) 17-1 can be obtained from public domain datasets. Dataset(s) 17-1 can be obtained from private datasets associated with one or more developer system(s) for the alignment of bespoke machine-learned model(s) customized for private use-cases.

[0409] Pre-training pipelines 17-2 can include a machine-learned model training workflow configured to update development model 16 over large-scale, potentially noisy datasets. For example, pre-training can leverage unsupervised learning techniques (e.g., de-noising, etc.) to process large numbers of training instances to update model parameters from an initialized state and achieve a desired baseline performance. Pre-training pipelines 17-2 can leverage unlabeled datasets in dataset(s) 17-1 to perform pre-training. Workbench 15 can implement a pre-training pipeline 17-2 to pre-train development model 16.

[0410] Fine-tuning pipelines 17-3 can include a machine-learned model training workflow configured to refine the model parameters of development model 16 with higher-quality data. Fine-tuning pipelines 17-3 can update development model 16 by conducting supervised training with labeled dataset(s) in dataset(s) 17-1. Fine-tuning pipelines 17-3 can update development model 16 by conducting reinforcement learning using reward signals from user feedback signals. Workbench 15 can implement a fine-tuning pipeline 17-3 to fine-tune development model 16.

[0411] Prompt libraries 17-4 can include sets of inputs configured to induce behavior aligned with desired performance criteria. Prompt libraries 17-4 can include few-shot prompts (e.g., inputs providing examples of desired model outputs for prepending to a desired runtime query), chain-of-thought prompts (e.g., inputs providing step-by-step reasoning within the exemplars to facilitate thorough reasoning by the model), and the like.

[0412] Example prompts can be retrieved from an available repository of prompt libraries 17-4. Example prompts can be contributed by one or more developer systems using workbench 15.

[0413] In some implementations, pre-trained or fine-tuned models can achieve satisfactory performance without exemplars in the inputs. For instance, zero-shot prompts can

include inputs that lack exemplars. Zero-shot prompts can be within a domain within a training dataset or outside of the training domain(s).

[0414] Prompt libraries 17-4 can include one or more prompt engineering tools. Prompt engineering tools can provide workflows for retrieving or learning optimized prompt values. Prompt engineering tools can facilitate directly learning prompt values (e.g., input element values) based one or more training iterations. Workbench 15 can implement prompt engineering tools in development model 16.

[0415] Prompt libraries 17-4 can include pipelines for prompt generation. For example, inputs can be generated using development model 16 itself or other machine-learned models. In this manner, for instance, a first model can process information about a task and output a input for a second model to process in order to perform a step of the task. The second model can be the same as or different from the first model. Workbench 15 can implement prompt generation pipelines in development model 16.

[0416] Prompt libraries 17-4 can include pipelines for context injection. For instance, a performance of development model 16 on a particular task can improve if provided with additional context for performing the task. Prompt libraries 17-4 can include software components configured to identify desired context, retrieve the context from an external source (e.g., a database, a sensor, etc.), and add the context to the input prompt. Workbench 15 can implement context injection pipelines in development model 16.

[0417] Although various training examples described herein with respect to model development platform 12 refer to "pre-training" and "fine-tuning," it is to be understood that model alignment toolkit 17 can generally support a wide variety of training techniques adapted for training a wide variety of machine-learned models. Example training techniques can correspond to the example training method 3200 described above.

[0418] Model development platform 12 can include a model plugin toolkit 18. Model plugin toolkit 18 can include a variety of tools configured for augmenting the functionality of a machine-learned model by integrating the machine-learned model with other systems, devices, and software components. For instance, a machine-learned model can use tools to increase performance quality where appropriate. For instance, deterministic tasks can be offloaded to dedicated tools in lieu of probabilistically performing the task with an increased risk of error. For instance, instead of autoregressively predicting the solution to a system of equations, a machine-learned model can recognize a tool to call for obtaining the solution and pass the system of equations to the appropriate tool. The tool can be a traditional system of equations solver that can operate deterministically to resolve the system of equations. The

output of the tool can be returned in response to the original query. In this manner, tool use can allow some example models to focus on the strengths of machine-learned models—e.g., understanding an intent in an unstructured request for a task—while augmenting the performance of the model by offloading certain tasks to a more focused tool for rote application of deterministic algorithms to a well-defined problem.

[0419] Model plugin toolkit 18 can include validation tools 18-1. Validation tools 18-1 can include tools that can parse and confirm output(s) of a machine-learned model. Validation tools 18-1 can include engineered heuristics that establish certain thresholds applied to model outputs. For example, validation tools 18-1 can ground the outputs of machine-learned models to structured data sources (e.g., to mitigate "hallucinations").

[0420] Model plugin toolkit 18 can include tooling packages 18-2 for implementing one or more tools that can include scripts or other executable code that can be executed alongside development model 16. Tooling packages 18-2 can include one or more inputs configured to cause machine-learned model(s) to implement the tools (e.g., few-shot prompts that induce a model to output tool calls in the proper syntax, etc.). Tooling packages 18-2 can include, for instance, fine-tuning training data for training a model to use a tool.

[0421] Model plugin toolkit 18 can include interfaces for calling external application programming interfaces (APIs) 18-3. For instance, in addition to or in lieu of implementing tool calls or tool code directly with development model 16, development model 16 can be aligned to output instruction that initiate API calls to send or obtain data via external systems.

[0422] Model plugin toolkit 18 can integrate with prompt libraries 17-4 to build a catalog of available tools for use with development model 16. For instance, a model can receive, in an input, a catalog of available tools, and the model can generate an output that selects a tool from the available tools and initiates a tool call for using the tool.

[0423] Model development platform 12 can include a computational optimization toolkit 19 for optimizing a computational performance of development model 16. For instance, tools for model compression 19-1 can allow development model 16 to be reduced in size while maintaining a desired level of performance. For instance, model compression 19-1 can include quantization workflows, weight pruning and sparsification techniques, etc. Tools for hardware acceleration 19-2 can facilitate the configuration of the model storage and execution formats to operate optimally on different hardware resources. For instance, hardware acceleration 19-2 can include tools for optimally sharding models for distributed processing over multiple processing units for increased bandwidth, lower unified memory requirements, etc. Tools for distillation 19-3 can provide for the training of lighter-weight

models based on the knowledge encoded in development model 16. For instance, development model 16 can be a highly performant, large machine-learned model optimized using model development platform 12. To obtain a lightweight model for running in resource-constrained environments, a smaller model can be a "student model" that learns to imitate development model 16 as a "teacher model." In this manner, for instance, the investment in learning the parameters and configurations of development model 16 can be efficiently transferred to a smaller model for more efficient inference.

[0424] Workbench 15 can implement one, multiple, or none of the toolkits implemented in model development platform 12. Workbench 15 can output an output model 20 based on development model 16. Output model 20 can be a deployment version of development model 16. Output model 20 can be a development or training checkpoint of development model 16. Output model 20 can be a distilled, compressed, or otherwise optimized version of development model 16.

[0425] Figure 37 is a block diagram of an example training flow for training a machine-learned development model 16. One or more portion(s) of the example training flow can be implemented by a computing system that includes one or more computing devices such as, for example, computing systems described with reference to the other figures. Each respective portion of the example training flow can be performed by any (or any combination) of one or more computing devices. Moreover, one or more portion(s) of the example training flow can be implemented on the hardware components of the device(s) described herein, for example, to train one or more systems or models. The Figure depicts elements performed in a particular order for purposes of illustration and discussion. Those of ordinary skill in the art, using the disclosures provided herein, will understand that the elements of any of the methods discussed herein can be adapted, rearranged, expanded, omitted, combined, or modified in various ways without deviating from the scope of the present disclosure. The Figure is described with reference to elements/terms described with respect to other systems and figures for exemplary illustrated purposes and is not meant to be limiting. One or more portions of the example training flow can be performed additionally, or alternatively, by other systems.

[0426] Initially, development model 16 can persist in an initial state as an initialized model 21. Development model 16 can be initialized with weight values. Initial weight values can be random or based on an initialization schema. Initial weight values can be based on prior pre-training for the same or for a different model.

[0427] Initialized model 21 can undergo pre-training in a pre-training stage 22. Pre-training stage 22 can be implemented using one or more pre-training pipelines 17-2 over data from dataset(s) 17-1. Pre-training can be omitted, for example, if initialized model 21 is already pre-trained (e.g., development model 16 contains, is, or is based on a pre-trained foundational model or an expert model).

[0428] Pre-trained model 23 can then be a new version of development model 16, which can persist as development model 16 or as a new development model. Pre-trained model 23 can be the initial state if development model 16 was already pre-trained. Pre-trained model 23 can undergo fine-tuning in a fine-tuning stage 24. Fine-tuning stage 24 can be implemented using one or more fine-tuning pipelines 17-3 over data from dataset(s) 17-1. Fine-tuning can be omitted, for example, if a pre-trained model as satisfactory performance, if the model was already fine-tuned, or if other tuning approaches are preferred.

[0429] Fine-tuned model 29 can then be a new version of development model 16, which can persist as development model 16 or as a new development model. Fine-tuned model 29 can be the initial state if development model 16 was already fine-tuned. Fine-tuned model 29 can undergo refinement with user feedback 26. For instance, refinement with user feedback 26 can include reinforcement learning, optionally based on human feedback from human users of fine-tuned model 25. As reinforcement learning can be a form of fine-tuning, it is to be understood that fine-tuning stage 24 can subsume the stage for refining with user feedback 26. Refinement with user feedback 26 can produce a refined model 27. Refined model 27 can be output to downstream system(s) 28 for deployment or further development.

[0430] In some implementations, computational optimization operations can be applied before, during, or after each stage. For instance, initialized model 21 can undergo computational optimization 29-1 (e.g., using computational optimization toolkit 19) before pre-training stage 22. Pre-trained model 23 can undergo computational optimization 29-2 (e.g., using computational optimization toolkit 19) before fine-tuning stage 24. Fine-tuned model 25 can undergo computational optimization 29-3 (e.g., using computational optimization toolkit 19) before refinement with user feedback 26. Refined model 27 can undergo computational optimization 29-4 (e.g., using computational optimization toolkit 19) before output to downstream system(s) 28. Computational optimization(s) 29-1, . . . , 29-4 can all be the same, all be different, or include at least some different optimization techniques.

[0431] Figure 38 is a block diagram of an inference system for operating one or more machine-learned model(s) 1 to perform inference (e.g., for training, for deployment, etc.). A

model host 31 can receive machine-learned model(s) 1. Model host 31 can host one or more model instance(s) 31-1, which can be one or multiple instances of one or multiple models. Model host 31 can host model instance(s) 31-1 using available compute resources 31-2 associated with model host 31.

[0432] Model host 31 can be an implementation of model host system 106.

[0433] Model host 31 can perform inference on behalf of one or more client(s) 32. Client(s) 32 can transmit an input request 33 to model host 31. Using input request 33, model host 31 can obtain input(s) 2 for input to machine-learned model(s) 1. Machine-learned model(s) 1 can process input(s) 2 to generate output(s) 3. Using output(s) 3, model host 31 can return an output payload 34 for responding to input request 33 from client(s) 32. Output payload 34 can include or be based on output(s) 3.

[0434] Model host 31 can leverage various other resources and tools to augment the inference task. For instance, model host 31 can communicate with tool interfaces 35 to facilitate tool use by model instance(s) 31-1. Tool interfaces 35 can include local or remote APIs. Tool interfaces 35 can include integrated scripts or other software functionality. Model host 31 can engage online learning interface(s) 36 to facilitate ongoing improvements to machine-learned model(s) 1. For instance, online learning interface(s) 36 can be used within reinforcement learning loops to retrieve user feedback on inferences served by model host 31. Model host 31 can access runtime data source(s) 37 for augmenting input(s) 2 with additional contextual information. For instance, runtime data source(s) 37 can include a knowledge graph 37-1 that facilitates structured information retrieval for information associated with input request(s) 33 (e.g., a search engine service). Runtime data source(s) 37 can include public or private, external or local database(s) 37-2 that can store information associated with input request(s) 33 for augmenting input(s) 2. Runtime data source(s) 37 can include account data 37-3 which can be retrieved in association with a user account corresponding to a client 32 for customizing the behavior of model host 31 accordingly.

[0435] Model host 31 can be implemented by one or multiple computing devices or systems. Client(s) 2 can be implemented by one or multiple computing devices or systems, which can include computing devices or systems shared with model host 31.

[0436] For example, model host 31 can operate on a server system that provides a machine-learning service to client device(s) that operate client(s) 32 (e.g., over a local or wide-area network). Client device(s) can be end-user devices used by individuals. Client device(s) can be server systems that operate client(s) 32 to provide various functionality as a service to downstream end-user devices.

[0437] In some implementations, model host 31 can operate on a same device or system as client(s) 32. Model host 31 can be a machine-learning service that runs on-device to provide machine-learning functionality to one or multiple applications operating on a client device, which can include an application implementing client(s) 32. Model host 31 can be a part of a same application as client(s) 32. For instance, model host 31 can be a subroutine or method implemented by one part of an application, and client(s) 32 can be another subroutine or method that engages model host 31 to perform inference functions within the application. It is to be understood that model host 31 and client(s) 32 can have various different configurations.

[0438] Model instance(s) 31-1 can include one or more machine-learned models that are available for performing inference. Model instance(s) 31-1 can include weights or other model components that are stored on in persistent storage, temporarily cached, or loaded into high-speed memory. Model instance(s) 31-1 can include multiple instance(s) of the same model (e.g., for parallel execution of more requests on the same model). Model instance(s) 31-1 can include instance(s) of different model(s). Model instance(s) 31-1 can include cached intermediate states of active or inactive model(s) used to accelerate inference of those models. For instance, an inference session with a particular model may generate significant amounts of computational results that can be re-used for future inference runs (e.g., using a KV cache for transformer-based models). These computational results can be saved in association with that inference session so that session can be executed more efficiently when resumed.

[0439] Compute resource(s) 31-2 can include one or more processors (central processing units, graphical processing units, tensor processing units, machine-learning accelerators, etc.) connected to one or more memory devices. Compute resource(s) 31-2 can include a dynamic pool of available resources shared with other processes. Compute resource(s) 31-2 can include memory devices large enough to fit an entire model instance in a single memory instance. Compute resource(s) 31-2 can also shard model instance(s) across multiple memory devices (e.g., using data parallelization or tensor parallelization, etc.). This can be done to increase parallelization or to execute a large model using multiple memory devices which individually might not be able to fit the entire model into memory.

[0440] Input request 33 can include data for input(s) 2. Model host 31 can process input request 33 to obtain input(s) 2. Input(s) 2 can be obtained directly from input request 33 or can be retrieved using input request 33. Input request 33 can be submitted to model host 31 via an API.

[0441] Model host 31 can perform inference over batches of input requests 33 in parallel. For instance, a model instance 31-1 can be configured with an input structure that has a batch dimension. Separate input(s) 2 can be distributed across the batch dimension (e.g., rows of an array). The separate input(s) 2 can include completely different contexts. The separate input(s) 2 can be multiple inference steps of the same task. The separate input(s) 2 can be staggered in an input structure, such that any given inference cycle can be operating on different portions of the respective input(s) 2. In this manner, for instance, model host 31 can perform inference on the batch in parallel, such that output(s) 3 can also contain the batch dimension and return the inference results for the batched input(s) 2 in parallel. In this manner, for instance, batches of input request(s) 33 can be processed in parallel for higher throughput of output payload(s) 34.

[0442] Output payload 34 can include or be based on output(s) 3 from machine-learned model(s) 1. Model host 31 can process output(s) 3 to obtain output payload 34. This can include chaining multiple rounds of inference (e.g., iteratively, recursively, across the same model(s) or different model(s)) to arrive at a final output for a task to be returned in output payload 34. Output payload 34 can be transmitted to client(s) 32 via an API.

[0443] Online learning interface(s) 36 can facilitate reinforcement learning of machine-learned model(s) 1. Online learning interface(s) 36 can facilitate reinforcement learning with human feedback (RLHF). Online learning interface(s) 36 can facilitate federated learning of machine-learned model(s) 1.

[0444] Model host 31 can execute machine-learned model(s) 1 to perform inference for various tasks using various types of data. For example, various different input(s) 2 and output(s) 3 can be used for various different tasks. In some implementations, input(s) 2 can be or otherwise represent image data. Machine-learned model(s) 1 can process the image data to generate an output. As an example, machine-learned model(s) 1 can process the image data to generate an image recognition output (e.g., a recognition of the image data, a latent embedding of the image data, an encoded representation of the image data, a hash of the image data, etc.). As another example, machine-learned model(s) 1 can process the image data to generate an image segmentation output. As another example, machine-learned model(s) 1 can process the image data to generate an image classification output. As another example, machine-learned model(s) 1 can process the image data to generate an image data modification output (e.g., an alteration of the image data, etc.). As another example, machine-learned model(s) 1 can process the image data to generate an encoded image data output (e.g., an encoded and/or compressed representation of the image data, etc.). As another example,

machine-learned model(s) 1 can process the image data to generate an upscaled image data output. As another example, machine-learned model(s) 1 can process the image data to generate a prediction output.

[0445] In some implementations, the task is a computer vision task. In some cases, input(s) 2 includes pixel data for one or more images and the task is an image processing task. For example, the image processing task can be image classification, where the output is a set of scores, each score corresponding to a different object class and representing the likelihood that the one or more images depict an object belonging to the object class. The image processing task may be object detection, where the image processing output identifies one or more regions in the one or more images and, for each region, a likelihood that region depicts an object of interest. As another example, the image processing task can be image segmentation, where the image processing output defines, for each pixel in the one or more images, a respective likelihood for each category in a predetermined set of categories. For example, the set of categories can be foreground and background. As another example, the set of categories can be object classes. As another example, the image processing task can be depth estimation, where the image processing output defines, for each pixel in the one or more images, a respective depth value. As another example, the image processing task can be motion estimation, where the network input includes multiple images, and the image processing output defines, for each pixel of one of the input images, a motion of the scene depicted at the pixel between the images in the network input.

[0446] In some implementations, input(s) 2 can be or otherwise represent natural language data. Machine-learned model(s) 1 can process the natural language data to generate an output. As an example, machine-learned model(s) 1 can process the natural language data to generate a language encoding output. As another example, machine-learned model(s) 1 can process the natural language data to generate a latent text embedding output. As another example, machine-learned model(s) 1 can process the natural language data to generate a translation output. As another example, machine-learned model(s) 1 can process the natural language data to generate a classification output. As another example, machine-learned model(s) 1 can process the natural language data to generate a textual segmentation output. As another example, machine-learned model(s) 1 can process the natural language data to generate a semantic intent output. As another example, machine-learned model(s) 1 can process the natural language data to generate an upscaled text or natural language output (e.g., text or natural language data that is higher quality than the input text or natural language, etc.). As another example, machine-learned model(s) 1 can process the natural

language data to generate a prediction output (e.g., one or more predicted next portions of natural language content).

[0447] In some implementations, input(s) 2 can be or otherwise represent speech data (e.g., data describing spoken natural language, such as audio data, textual data, etc.). Machine-learned model(s) 1 can process the speech data to generate an output. As an example, machine-learned model(s) 1 can process the speech data to generate a speech recognition output. As another example, machine-learned model(s) 1 can process the speech data to generate a speech translation output. As another example, machine-learned model(s) 1 can process the speech data to generate a latent embedding output. As another example, machine-learned model(s) 1 can process the speech data to generate an encoded speech output (e.g., an encoded and/or compressed representation of the speech data, etc.). As another example, machine-learned model(s) 1 can process the speech data to generate an upscaled speech output (e.g., speech data that is higher quality than the input speech data, etc.). As another example, machine-learned model(s) 1 can process the speech data to generate a textual representation output (e.g., a textual representation of the input speech data, etc.). As another example, machine-learned model(s) 1 can process the speech data to generate a prediction output.

[0448] In some implementations, input(s) 2 can be or otherwise represent latent encoding data (e.g., a latent space representation of an input, etc.). Machine-learned model(s) 1 can process the latent encoding data to generate an output. As an example, machine-learned model(s) 1 can process the latent encoding data to generate a recognition output. As another example, machine-learned model(s) 1 can process the latent encoding data to generate a reconstruction output. As another example, machine-learned model(s) 1 can process the latent encoding data to generate a search output. As another example, machine-learned model(s) 1 can process the latent encoding data to generate a reclustering output. As another example, machine-learned model(s) 1 can process the latent encoding data to generate a prediction output.

[0449] In some implementations, input(s) 2 can be or otherwise represent statistical data. Statistical data can be, represent, or otherwise include data computed and/or calculated from some other data source. Machine-learned model(s) 1 can process the statistical data to generate an output. As an example, machine-learned model(s) 1 can process the statistical data to generate a recognition output. As another example, machine-learned model(s) 1 can process the statistical data to generate a prediction output. As another example, machine-learned model(s) 1 can process the statistical data to generate a classification output. As

another example, machine-learned model(s) 1 can process the statistical data to generate a segmentation output. As another example, machine-learned model(s) 1 can process the statistical data to generate a visualization output. As another example, machine-learned model(s) 1 can process the statistical data to generate a diagnostic output.

[0450] In some implementations, input(s) 2 can be or otherwise represent sensor data. Machine-learned model(s) 1 can process the sensor data to generate an output. As an example, machine-learned model(s) 1 can process the sensor data to generate a recognition output. As another example, machine-learned model(s) 1 can process the sensor data to generate a prediction output. As another example, machine-learned model(s) 1 can process the sensor data to generate a classification output. As another example, machine-learned model(s) 1 can process the sensor data to generate a segmentation output. As another example, machine-learned model(s) 1 can process the sensor data to generate a visualization output. As another example, machine-learned model(s) 1 can process the sensor data to generate a diagnostic output. As another example, machine-learned model(s) 1 can process the sensor data to generate a detection output.

[0451] In some implementations, machine-learned model(s) 1 can be configured to perform a task that includes encoding input data for reliable and/or efficient transmission or storage (and/or corresponding decoding). For example, the task may be an audio compression task. The input may include audio data and the output may comprise compressed audio data. In another example, the input includes visual data (e.g. one or more images or videos), the output comprises compressed visual data, and the task is a visual data compression task. In another example, the task may comprise generating an embedding for input data (e.g. input audio or visual data). In some cases, the input includes audio data representing a spoken utterance and the task is a speech recognition task. The output may comprise a text output which is mapped to the spoken utterance. In some cases, the task comprises encrypting or decrypting input data. In some cases, the task comprises a microprocessor performance task, such as branch prediction or memory address translation.

[0452] In some implementations, the task is a generative task, and machine-learned model(s) 1 can be configured to output content generated in view of input(s) 2. For instance, input(s) 2 can be or otherwise represent data of one or more modalities that encodes context for generating additional content.

[0453] In some implementations, the task can be a text completion task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent textual data and to generate output(s) 3 that represent additional textual data that completes a textual sequence

that includes input(s) 2. For instance, machine-learned model(s) 1 can be configured to generate output(s) 3 to complete a sentence, paragraph, or portion of text that follows from a portion of text represented by input(s) 2.

[0454] In some implementations, the task can be an instruction following task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent instructions to perform a function and to generate output(s) 3 that advance a goal of satisfying the instruction function (e.g., at least a step of a multi-step procedure to perform the function). Output(s) 3 can represent data of the same or of a different modality as input(s) 2. For instance, input(s) 2 can represent textual data (e.g., natural language instructions for a task to be performed) and machine-learned model(s) 1 can process input(s) 2 to generate output(s) 3 that represent textual data responsive to the instructions (e.g., natural language responses, programming language responses, machine language responses, etc.). Input(s) 2 can represent image data (e.g., image-based instructions for a task to be performed, optionally accompanied by textual instructions) and machine-learned model(s) 1 can process input(s) 2 to generate output(s) 3 that represent textual data responsive to the instructions (e.g., natural language responses, programming language responses, machine language responses, etc.). One or more output(s) 3 can be iteratively or recursively generated to sequentially process and accomplish steps toward accomplishing the requested functionality. For instance, an initial output can be executed by an external system or be processed by machine-learned model(s) 1 to complete an initial step of performing a function. Multiple steps can be performed, with a final output being obtained that is responsive to the initial instructions.

[0455] In some implementations, the task can be a question answering task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent a question to answer and to generate output(s) 3 that advance a goal of returning an answer to the question (e.g., at least a step of a multi-step procedure to perform the function). Output(s) 3 can represent data of the same or of a different modality as input(s) 2. For instance, input(s) 2 can represent textual data (e.g., natural language instructions for a task to be performed) and machine-learned model(s) 1 can process input(s) 2 to generate output(s) 3 that represent textual data responsive to the question (e.g., natural language responses, programming language responses, machine language responses, etc.). Input(s) 2 can represent image data (e.g., image-based instructions for a task to be performed, optionally accompanied by textual instructions) and machine-learned model(s) 1 can process input(s) 2 to generate output(s) 3 that represent textual data responsive to the question (e.g., natural language responses, programming language responses, machine language responses, etc.). One or more output(s)

3 can be iteratively or recursively generated to sequentially process and accomplish steps toward answering the question. For instance, an initial output can be executed by an external system or be processed by machine-learned model(s) 1 to complete an initial step of obtaining an answer to the question (e.g., querying a database, performing a computation, executing a script, etc.). Multiple steps can be performed, with a final output being obtained that is responsive to the question.

[0456] In some implementations, the task can be an image generation task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent context regarding a desired portion of image content. The context can include text data, image data, audio data, etc. Machine-learned model(s) 1 can be configured to generate output(s) 3 that represent image data that depicts imagery related to the context. For instance, machine-learned model(s) 1 can be configured to generate pixel data of an image. Values for channel(s) associated with the pixels in the pixel data can be selected based on the context (e.g., based on a probability determined based on the context).

[0457] In some implementations, the task can be an audio generation task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent context regarding a desired portion of audio content. The context can include text data, image data, audio data, etc. Machine-learned model(s) 1 can be configured to generate output(s) 3 that represent audio data related to the context. For instance, machine-learned model(s) 1 can be configured to generate waveform data in the form of an image (e.g., a spectrogram). Values for channel(s) associated with pixels of the image can be selected based on the context. Machine-learned model(s) 1 can be configured to generate waveform data in the form of a sequence of discrete samples of a continuous waveform. Values of the sequence can be selected based on the context (e.g., based on a probability determined based on the context).

[0458] In some implementations, the task can be a data generation task. Machine-learned model(s) 1 can be configured to process input(s) 2 that represent context regarding a desired portion of data (e.g., data from various data domains, such as sensor data, image data, multimodal data, statistical data, etc.). The desired data can be, for instance, synthetic data for training other machine-learned models. The context can include arbitrary data type(s). Machine-learned model(s) 1 can be configured to generate output(s) 3 that represent data that aligns with the desired data. For instance, machine-learned model(s) 1 can be configured to generate data values for populating a dataset. Values for the data object(s) can be selected based on the context (e.g., based on a probability determined based on the context).

[0459] Figure 39 is a block diagram of an example networked computing system that can perform aspects of example implementations of the present disclosure. The system can include a number of computing devices and systems that are communicatively coupled over a network 49. An example computing device 50 is described to provide an example of a computing device that can perform any aspect of the present disclosure (e.g., implementing model host 31, client(s) 32, or both). An example server computing system 60 is described as an example of a server computing system that can perform any aspect of the present disclosure (e.g., implementing model host 31, client(s) 32, or both). Computing device 50 and server computing system(s) 60 can cooperatively interact (e.g., over network 49) to perform any aspect of the present disclosure (e.g., implementing model host 31, client(s) 32, or both). Model development platform system 70 is an example system that can host or serve model development platform(s) 12 for development of machine-learned models. Third-party system(s) 80 are example system(s) with which any of computing device 50, server computing system(s) 60, or model development platform system(s) 70 can interact in the performance of various aspects of the present disclosure (e.g., engaging third-party tools, accessing third-party databases or other resources, etc.).

[0460] Network 49 can be any type of communications network, such as a local area network (e.g., intranet), wide area network (e.g., Internet), or some combination thereof and can include any number of wired or wireless links. In general, communication over network 49 can be carried via any type of wired or wireless connection, using a wide variety of communication protocols (e.g., TCP/IP, HTTP, SMTP, FTP), encodings or formats (e.g., HTML, XML), or protection schemes (e.g., VPN, secure HTTP, SSL). Network 49 can also be implemented via a system bus. For instance, one or more devices or systems can be co-located with, contained by, or otherwise integrated into one or more other devices or systems.

[0461] Computing device 50 can be any type of computing device, such as, for example, a personal computing device (e.g., laptop or desktop), a mobile computing device (e.g., smartphone or tablet), a gaming console or controller, a wearable computing device, an embedded computing device, a server computing device, a virtual machine operating on a host device, or any other type of computing device. Computing device 50 can be a client computing device. Computing device 50 can be an end-user computing device. Computing device 50 can be a computing device of a service provided that provides a service to an end user (who may use another computing device to interact with computing device 50).

[0462] Computing device 50 can include one or more processors 51 and a memory 52. Processor(s) 51 can be any suitable processing device (e.g., a processor core, a

microprocessor, an ASIC, an FPGA, a controller, a microcontroller, etc.) and can be one processor or a plurality of processors that are operatively connected. Memory 52 can include one or more non-transitory computer-readable storage media, such as HBM, RAM, ROM, EEPROM, EPROM, flash memory devices, magnetic disks, etc., and combinations thereof. Memory 52 can store data 53 and instructions 54 which can be executed by processor(s) 51 to cause computing device 50 to perform operations. The operations can implement any one or multiple features described herein. The operations can implement example methods and techniques described herein.

[0463] Computing device 50 can also include one or more input components that receive user input. For example, a user input component can be a touch-sensitive component (e.g., a touch-sensitive display screen or a touch pad) that is sensitive to the touch of a user input object (e.g., a finger or a stylus). The touch-sensitive component can serve to implement a virtual keyboard. Other example user input components include a microphone, camera, LIDAR, a physical keyboard or other buttons, or other means by which a user can provide user input.

[0464] Computing device 50 can store or include one or more machine-learned models 55. Machine-learned models 55 can include one or more machine-learned model(s) 1, such as a sequence processing model 4. Machine-learned models 55 can include one or multiple model instance(s) 31-1. Machine-learned model(s) 55 can be received from server computing system(s) 60, model development platform system 70, third party system(s) 80 (e.g., an application distribution platform), or developed locally on computing device 50. Machine-learned model(s) 55 can be loaded into memory 52 and used or otherwise implemented by processor(s) 51. Computing device 50 can implement multiple parallel instances of machine-learned model(s) 55.

[0465] Server computing system(s) 60 can include one or more processors 61 and a memory 62. Processor(s) 61 can be any suitable processing device (e.g., a processor core, a microprocessor, an ASIC, an FPGA, a controller, a microcontroller, etc.) and can be one processor or a plurality of processors that are operatively connected. Memory 62 can include one or more non-transitory computer-readable storage media, such as HBM, RAM, ROM, EEPROM, EPROM, flash memory devices, magnetic disks, etc., and combinations thereof. Memory 62 can store data 63 and instructions 64 which can be executed by processor(s) 61 to cause server computing system(s) 60 to perform operations. The operations can implement any one or multiple features described herein. The operations can implement example methods and techniques described herein.

[0466] In some implementations, server computing system 60 includes or is otherwise implemented by one or multiple server computing devices. In instances in which server computing system 60 includes multiple server computing devices, such server computing devices can operate according to sequential computing architectures, parallel computing architectures, or some combination thereof.

[0467] Server computing system 60 can store or otherwise include one or more machine-learned models 65. Machine-learned model(s) 65 can be the same as or different from machine-learned model(s) 55. Machine-learned models 65 can include one or more machine-learned model(s) 1, such as a sequence processing model 4. Machine-learned models 65 can include one or multiple model instance(s) 31-1. Machine-learned model(s) 65 can be received from computing device 50, model development platform system 70, third party system(s) 80, or developed locally on server computing system(s) 60. Machine-learned model(s) 65 can be loaded into memory 62 and used or otherwise implemented by processor(s) 61. Server computing system(s) 60 can implement multiple parallel instances of machine-learned model(s) 65.

[0468] In an example configuration, machine-learned models 65 can be included in or otherwise stored and implemented by server computing system 60 to establish a client-server relationship with computing device 50 for serving model inferences. For instance, server computing system(s) 60 can implement model host 31 on behalf of client(s) 32 on computing device 50. For instance, machine-learned models 65 can be implemented by server computing system 60 as a portion of a web service (e.g., remote machine-learned model hosting service, such as an online interface for performing machine-learned model operations over a network on server computing system(s) 60). For instance, server computing system(s) 60 can communicate with computing device 50 over a local intranet or internet connection. For instance, computing device 50 can be a workstation or endpoint in communication with server computing system(s) 60, with implementation of machine-learned models 65 being managed by server computing system(s) 60 to remotely perform inference (e.g., for runtime or training operations), with output(s) returned (e.g., cast, streamed, etc.) to computing device 50. Machine-learned models 65 can work cooperatively or interoperatively with machine-learned models 55 on computing device 50 to perform various tasks.

[0469] Model development platform system(s) 70 can include one or more processors 71 and a memory 72. Processor(s) 71 can be any suitable processing device (e.g., a processor core, a microprocessor, an ASIC, an FPGA, a controller, a microcontroller, etc.) and can be one processor or a plurality of processors that are operatively connected. Memory 72 can

include one or more non-transitory computer-readable storage media, such as HBM, RAM, ROM, EEPROM, EPROM, flash memory devices, magnetic disks, etc., and combinations thereof. Memory 72 can store data 73 and instructions 74 which can be executed by processor(s) 71 to cause model development platform system(s) 70 to perform operations. The operations can implement any one or multiple features described herein. The operations can implement example methods and techniques described herein. Example operations include the functionality described herein with respect to model development platform 12. This and other functionality can be implemented by developer tool(s) 75.

[0470] Third-party system(s) 80 can include one or more processors 81 and a memory 82. Processor(s) 81 can be any suitable processing device (e.g., a processor core, a microprocessor, an ASIC, an FPGA, a controller, a microcontroller, etc.) and can be one processor or a plurality of processors that are operatively connected. Memory 82 can include one or more non-transitory computer-readable storage media, such as HBM, RAM, ROM, EEPROM, EPROM, flash memory devices, magnetic disks, etc., and combinations thereof. Memory 82 can store data 83 and instructions 84 which can be executed by processor(s) 81 to cause third-party system(s) 80 to perform operations. The operations can implement any one or multiple features described herein. The operations can implement example methods and techniques described herein. Example operations include the functionality described herein with respect to tools and other external resources called when training or performing inference with machine-learned model(s) 1, 4, 16, 20, 55, 65, etc. (e.g., third-party resource(s) 85).

[0471] Figure 39 illustrates one example arrangement of computing systems that can be used to implement the present disclosure. Other computing system configurations can be used as well. For example, in some implementations, one or both of computing system 50 or server computing system(s) 60 can implement all or a portion of the operations of model development platform system 70. For example, computing system 50 or server computing system(s) 60 can implement developer tool(s) 75 (or extensions thereof) to develop, update/train, or refine machine-learned models 1, 4, 16, 20, 55, 65, etc. using one or more techniques described herein with respect to model alignment toolkit 17. In this manner, for instance, computing system 50 or server computing system(s) 60 can develop, update/train, or refine machine-learned models based on local datasets (e.g., for model personalization/customization, as permitted by user data preference selections).

[0472] Figure 40 is a block diagram of an example computing device 98 that performs according to example embodiments of the present disclosure. Computing device 98

can be a user computing device or a server computing device (e.g., computing device 50, server computing system(s) 60, etc.). Computing device 98 can implement model host 31. For instance, computing device 98 can include a number of applications (e.g., applications 1 through N). Each application can contain its own machine learning library and machine-learned model(s). For example, each application can include a machine-learned model. Example applications include a text messaging application, an email application, a dictation application, a virtual keyboard application, a browser application, etc. As illustrated in Figure 40, each application can communicate with a number of other components of the computing device, such as, for example, one or more sensors, a context manager, a device state component, or additional components. In some implementations, each application can communicate with each device component using an API (e.g., a public API). In some implementations, the API used by each application is specific to that application.

[0473] Figure 41 is a block diagram of an example computing device 99 that performs according to example embodiments of the present disclosure. Computing device 99 can be the same as or different from computing device 98. Computing device 99 can be a user computing device or a server computing device (e.g., computing device 50, server computing system(s) 60, etc.). Computing device 98 can implement model host 31. For instance, computing device 99 can include a number of applications (e.g., applications 1 through N). Each application can be in communication with a central intelligence layer. Example applications include a text messaging application, an email application, a dictation application, a virtual keyboard application, a browser application, etc. In some implementations, each application can communicate with the central intelligence layer (and model(s) stored therein) using an API (e.g., a common API across all applications).

[0474] The central intelligence layer can include a number of machine-learned models. For example, as illustrated in Figure 41, a respective machine-learned model can be provided for each application and managed by the central intelligence layer. In other implementations, two or more applications can share a single machine-learned model. For example, in some implementations, the central intelligence layer can provide a single model for all of the applications. In some implementations, the central intelligence layer is included within or otherwise implemented by an operating system of computing device 99.

[0475] The central intelligence layer can communicate with a central device data layer. The central device data layer can be a centralized repository of data for computing device 99. As illustrated in Figure 41, the central device data layer can communicate with a number of other components of the computing device, such as, for example, one or more

sensors, a context manager, a device state component, or additional components. In some implementations, the central device data layer can communicate with each device component using an API (e.g., a private API).

[0476] The technology discussed herein makes reference to servers, databases, software applications, and other computer-based systems, as well as actions taken and information sent to and from such systems. The inherent flexibility of computer-based systems allows for a great variety of possible configurations, combinations, and divisions of tasks and functionality between and among components. For instance, processes discussed herein can be implemented using a single device or component or multiple devices or components working in combination. Databases and applications can be implemented on a single system or distributed across multiple systems. Distributed components can operate sequentially or in parallel.

[0477] While the present subject matter has been described in detail with respect to various specific example embodiments thereof, each example is provided by way of explanation, not limitation of the disclosure. Those skilled in the art, upon attaining an understanding of the foregoing, can readily produce alterations to, variations of, and equivalents to such embodiments. Accordingly, the subject disclosure does not preclude inclusion of such modifications, variations or additions to the present subject matter as would be readily apparent to one of ordinary skill in the art. For instance, features illustrated or described as part of one embodiment can be used with another embodiment to yield a still further embodiment. Thus, it is intended that the present disclosure cover such alterations, variations, and equivalents.

[0478] Aspects of the disclosure have been described in terms of illustrative embodiments thereof. Any and all features in the following claims can be combined or rearranged in any way possible, including combinations of claims not explicitly enumerated in combination together, as the example claim dependencies listed herein should not be read as limiting the scope of possible combinations of features disclosed herein. Accordingly, the scope of the present disclosure is by way of example rather than by way of limitation, and the subject disclosure does not preclude inclusion of such modifications, variations or additions to the present subject matter as would be readily apparent to one of ordinary skill in the art. Moreover, terms are described herein using lists of example elements joined by conjunctions such as “and,” “or,” “but,” etc. It should be understood that such conjunctions are provided for explanatory purposes only. Clauses and other sequences of items joined by a particular conjunction such as “or,” for example, can refer to “and/or,” “at least one of”, “any

combination of" example elements listed therein, etc. Terms such as "based on" should be understood as "based at least in part on."

[0479] The term "can" should be understood as referring to a possibility of a feature in various implementations and not as prescribing an ability that is necessarily present in every implementation. For example, the phrase "X can perform Y" should be understood as indicating that, in various implementations, X has the potential to be configured to perform Y, and not as indicating that in every instance X must always be able to perform Y. It should be understood that, in various implementations, X might be unable to perform Y and remain within the scope of the present disclosure.

[0480] The term "may" should be understood as referring to a possibility of a feature in various implementations and not as prescribing an ability that is necessarily present in every implementation. For example, the phrase "X may perform Y" should be understood as indicating that, in various implementations, X has the potential to be configured to perform Y, and not as indicating that in every instance X must always be able to perform Y. It should be understood that, in various implementations, X might be unable to perform Y and remain within the scope of the present disclosure.

WHAT IS CLAIMED IS:

1. A computer-implemented method for improved parallelization of a machine-learned sequence processing model, the method comprising:

receiving input requests to process a plurality of input sequences using the machine-learned sequence processing model to generate a plurality of output sequences respectively corresponding to the plurality of input sequences;

generating a plurality of initial attention tensors respectively for the plurality of input sequences, wherein:

one or more respective initial attention tensors are generated for each respective input sequence in parallel over input elements of the respective input sequence; and

the one or more respective initial attention tensors are generated in one or more batches having a first batch size using a prefill system that comprises one or more prefill computing devices and executes one or more layers of the machine-learned sequence processing model; and

autoregressively generating, using the plurality of initial attention tensors, a plurality of output elements for each of the plurality of output sequences in one or more batches having a second batch size, wherein:

the plurality of output elements are generated using a generation system that comprises one or more generation computing devices.

2. The method of claim 1, wherein at least a portion of the machine-learned sequence processing model is partitioned over the one or more generation computing devices along a batch dimension.

3. The method of any one or more of the preceding claims, comprising:

orchestrating the inputs to the prefill system and the outputs from the generation system with a router system that is configured to receive the input requests.

4. The method of any one or more of the preceding claims, wherein at least one of the prefill computing devices is different from at least one of the generation computing devices.

5. The method of any one or more of the preceding claims, wherein the prefill system is configured for a workflow having a compute bottleneck.

6. The method of any one or more of the preceding claims, wherein the prefill system is configured with a lower memory bandwidth as compared to the generation system.

7. The method of any one or more of the preceding claims, wherein the generation system is configured for a workflow having a memory bandwidth bottleneck.

8. The method of any one or more of the preceding claims, wherein the generation system comprises faster or more high-bandwidth memory (HBM) as compared to the prefill system.

9. The method of any one or more of the preceding claims, wherein the generation system comprises larger static random-access memory as compared to the prefill system.

10. The method of any one or more of the preceding claims, wherein the generation computing devices comprise a higher memory bandwidth than the prefill computing devices.

11. The method of any one or more of the preceding claims, wherein the generation system implements more processing cores than the prefill system.

12. The method of any one or more of the preceding claims, wherein the prefill system implements more processing cores than the generation system.

13. The method of any one or more of the preceding claims, wherein the second batch size is larger than the first batch size.

14. The method of any one or more of the preceding claims, wherein the first batch size is one.

15. The method of any one or more of the preceding claims, wherein the one or more respective initial attention tensors describe a relatedness of the respective plurality of input elements to one another.

16. The method of any one or more of the preceding claims, wherein autoregressively generating the plurality of output elements comprises:

processing, using the generation system, the initial attention tensors to determine a likely next output element;

selecting the likely next output element to include in the output sequence; and

updating the initial attention tensors based on the selected output element.

17. The method of any one or more of the preceding claims, comprising:

routing, based on a length of an input sequence, the input sequence to a first group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-gathered partitioning layout.

18. The method of any one or more of the preceding claims, comprising:

routing, based on a length of an input sequence, the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-stationary partitioning layout.

19. The method of any one or more of the preceding claims, comprising:

routing, based on a length of an input sequence, the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout in which weights are stationary along at least one parallelization axis and transferred along at least one other parallelization axis.

20. The method of any one or more of the preceding claims, comprising:

for a length above a threshold, routing the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according

to a hybrid partitioning layout with a first number of axes along which weights are transferred; or

for a length below a threshold, routing the input sequence to a second group of prefill computing devices that implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a second number of axes along which weights are transferred;

wherein the first number is greater than the second number.

21. The method of any one or more of the preceding claims, comprising:

selecting, based on an expected length of the one or more output sequences, a first group of generation computing devices to generate the one or more output sequences, wherein the first group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-gathered partitioning layout.

22. The method of any one or more of the preceding claims, comprising:

selecting, based on an expected length of the one or more output sequences, a second group of generation computing devices to generate the one or more output sequences, wherein the second group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a weight-stationary partitioning layout.

23. The method of any one or more of the preceding claims, comprising:

selecting, based on an expected length of the one or more output sequences, a group of generation computing devices to generate the one or more output sequences, wherein the group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout in which weights are stationary along at least one parallelization axis and transferred along at least one other parallelization axis.

24. The method of any one or more of the preceding claims, comprising:

for an expected length above a threshold, selecting a group of generation computing devices to generate the one or more output sequences, wherein the group of generation

computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a first number of axes along which weights are transferred; or

for an expected length below a threshold, selecting a group of generation computing devices to generate the one or more output sequences, wherein the group of generation computing devices implement the machine-learned sequence processing model with at least a portion of the machine-learned sequence processing model being partitioned according to a hybrid partitioning layout with a second number of axes along which weights are transferred;

wherein the first number is greater than the second number.

25. The method of any one or more of the preceding claims, comprising:

implementing a weight-stationary partitioning layout using a plurality of computing devices arranged in a three-dimensional toroidal configuration, wherein a chip count along a first dimension is one-fourth the chip count of a product of the other two dimensions.

26. The method of any one or more of the preceding claims, comprising:

implementing a weight-gathered partitioning layout using a plurality of computing devices arranged in a three-dimensional toroidal configuration, wherein a chip count along a first dimension is one-fourth the chip count of a product of the other two dimensions.

27. The method of any one or more of the preceding claims, comprising:

implementing a hybrid partitioning layout in which both weights and activations are partially transferred along different axes.

28. The method of any one or more of the preceding claims, comprising:

sharing a weight layout between the weight-stationary layout and the weight-gathered layout.

29. The method of any one or more of the preceding claims, wherein a feed-forward network of the machine-learned sequence processing model is partitioned according to:

a weight-stationary partitioning layout;

a weight-gathered partitioning layout; or

a combination thereof.

30. The method of any one or more of the preceding claims, comprising:

determining an expected ratio of input elements and output elements; and

determining a ratio of the number of prefill computing devices and a number of generation computing devices decode servers based on the expected ratio of input elements and output elements.

31. The method of any one or more of the preceding claims, comprising:

routing a processing task to a group of generation computing devices based on a buffer size of the group and an expected output length.

32. The method of any one or more of the preceding claims, comprising:

shifting a processing task to a different group of generation computing devices with a larger buffer size mid-completion when an actual output length is determined to exceed an expected output length.

33. The method of any one or more of the preceding claims, comprising:

generating an initial attention tensor incrementally by:

receiving a prior attention tensor;

computing, in parallel for an incremental set of input elements, an incremental attention tensor component; and

combining the prior attention tensor with the incremental attention tensor component to obtain the initial attention tensor.

34. The method of any one or more of the preceding claims, wherein the initial attention tensor is obtained incrementally over one or more turns in a multi-turn content generation session.

35. The method of any one or more of the preceding claims, wherein the multi-turn content generation session comprises:

a chat conversation; or

a series of instructions to a tool (e.g., an image generation tool, a text generation tool).

36. One or more non-transitory computer-readable media storing instructions that are executable by one or more processors to cause a computing system to perform operations, wherein the operations comprise:

the method of any one or more of the preceding claims.

37. A computing system, comprising:

one or more processors; and

one or more non-transitory computer-readable media storing instructions that are executable by one or more processors to cause a computing system to perform operations, wherein the operations comprise:

the method of any one or more of the preceding claims.

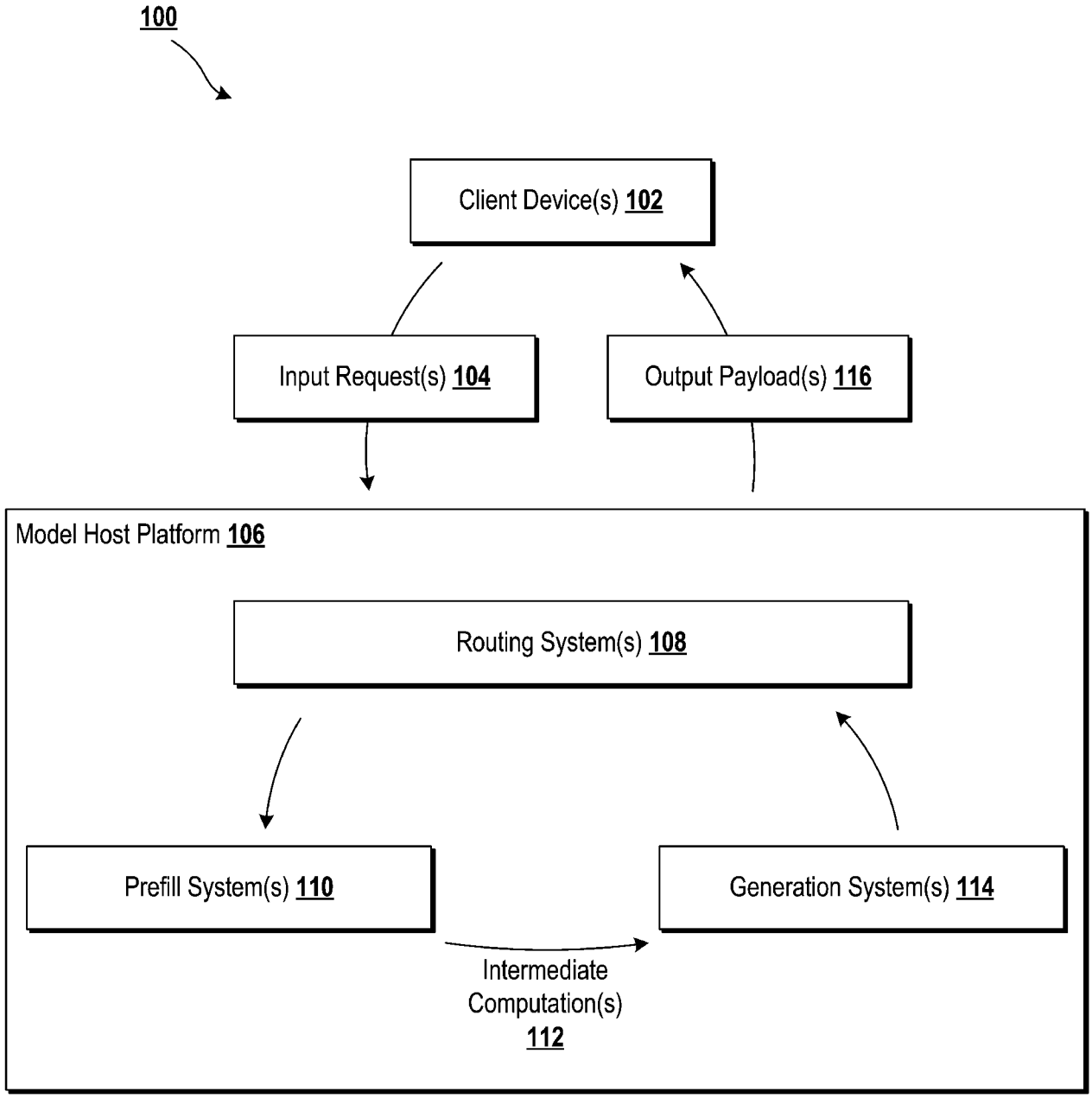

100
Client Device(s) 102
Input Request(s) 104
Output Payload(s) 116
Model Host Platform 106
Routing System(s) 108
Prefill System(s) 110
Generation System(s) 114
Intermediate Computation(s) 112

FIG. 1

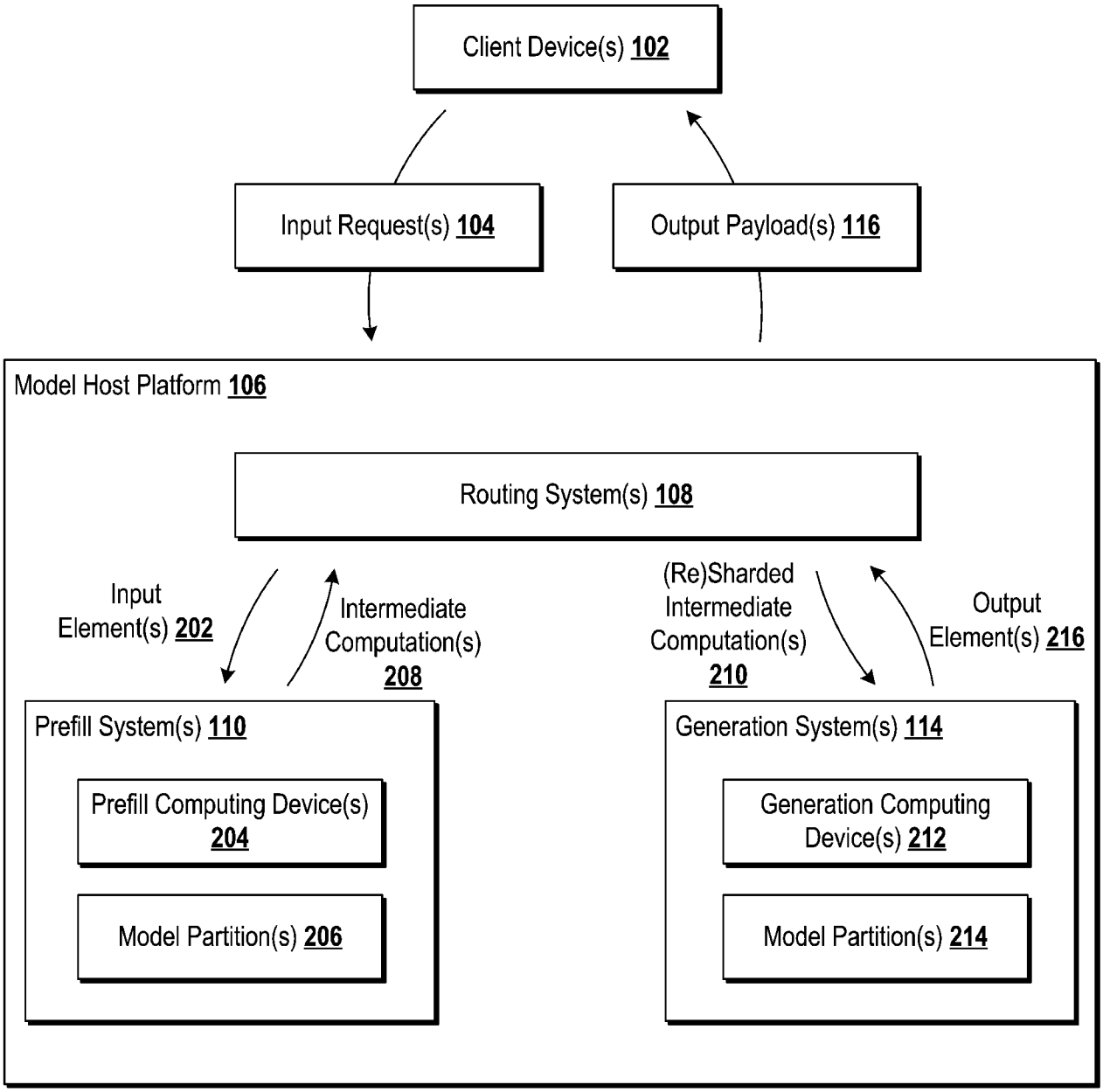

Client Device(s) 102
Input Request(s) 104
Output Payload(s) 116
Model Host Platform 106
Routing System(s) 108
Input Element(s) 202
Intermediate Computation(s) 208
(Re)Sharded Intermediate Computation(s) 210
Output Element(s) 216
Prefill System(s) 110
Prefill Computing Device(s) 204
Model Partition(s) 206
Generation System(s) 114
Generation Computing Device(s) 212
Model Partition(s) 214

FIG. 2

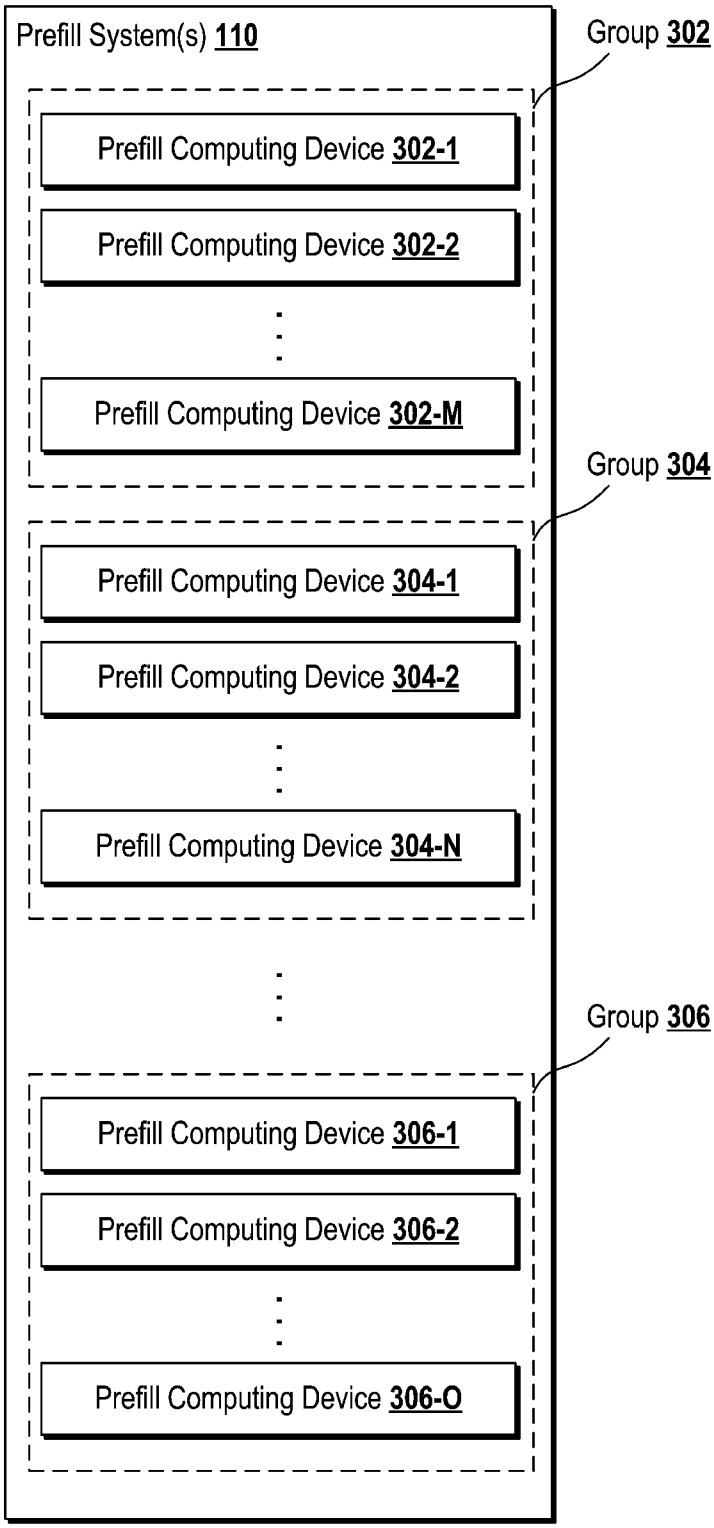

Prefill System(s) 110
Group 302
Prefill Computing Device 302-1
Prefill Computing Device 302-2
Prefill Computing Device 302-M
Group 304
Prefill Computing Device 304-1
Prefill Computing Device 304-2
Prefill Computing Device 304-N
Group 306
Prefill Computing Device 306-1
Prefill Computing Device 306-2
Prefill Computing Device 306-O

FIG. 3

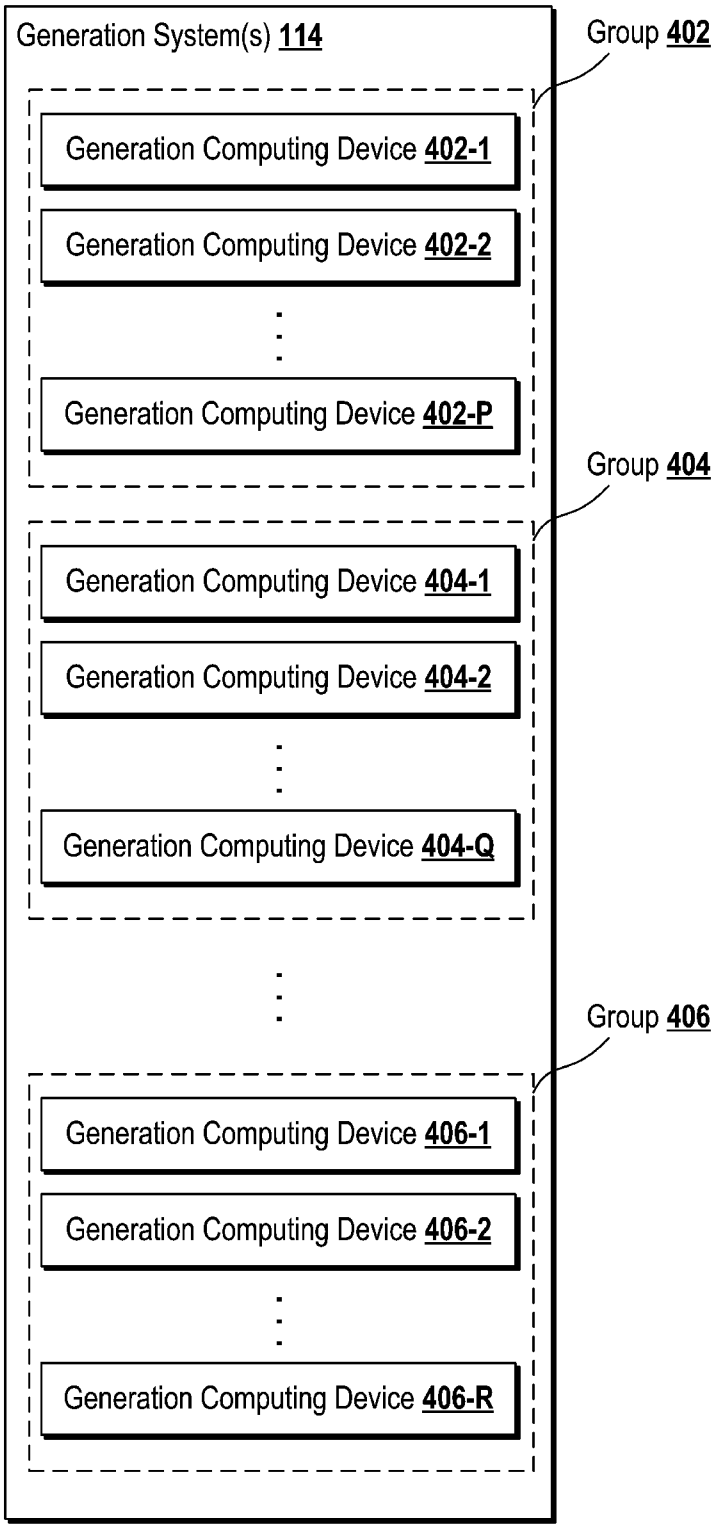

Generation System(s) 114
Group 402
Generation Computing Device 402-1
Generation Computing Device 402-2
Generation Computing Device 402-P
Group 404
Generation Computing Device 404-1
Generation Computing Device 404-2
Generation Computing Device 404-Q
Group 406
Generation Computing Device 406-1
Generation Computing Device 406-2
Generation Computing Device 406-R

FIG. 4

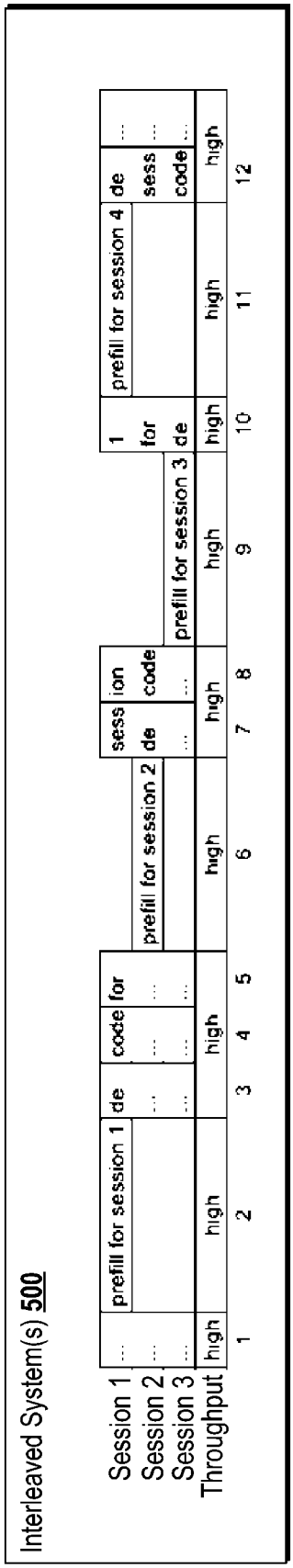

Interleaved System(s) 500
Session 1
Session 2
Session 3
Throughput
prefill for session 1
prefill for session 2
prefill for session 3
prefill for session 4
high
1
2
3
4
5
6
7
8
9
10
11
12

FIG. 5

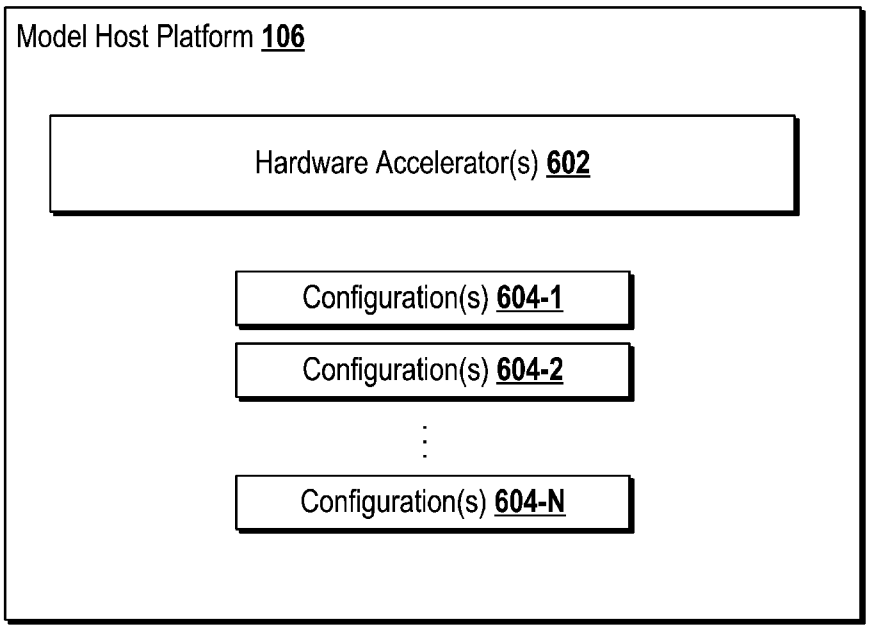

Model Host Platform 106
Hardware Accelerator(s) 602
Configuration(s) 604-1
Configuration(s) 604-2
Configuration(s) 604-N

FIG. 6

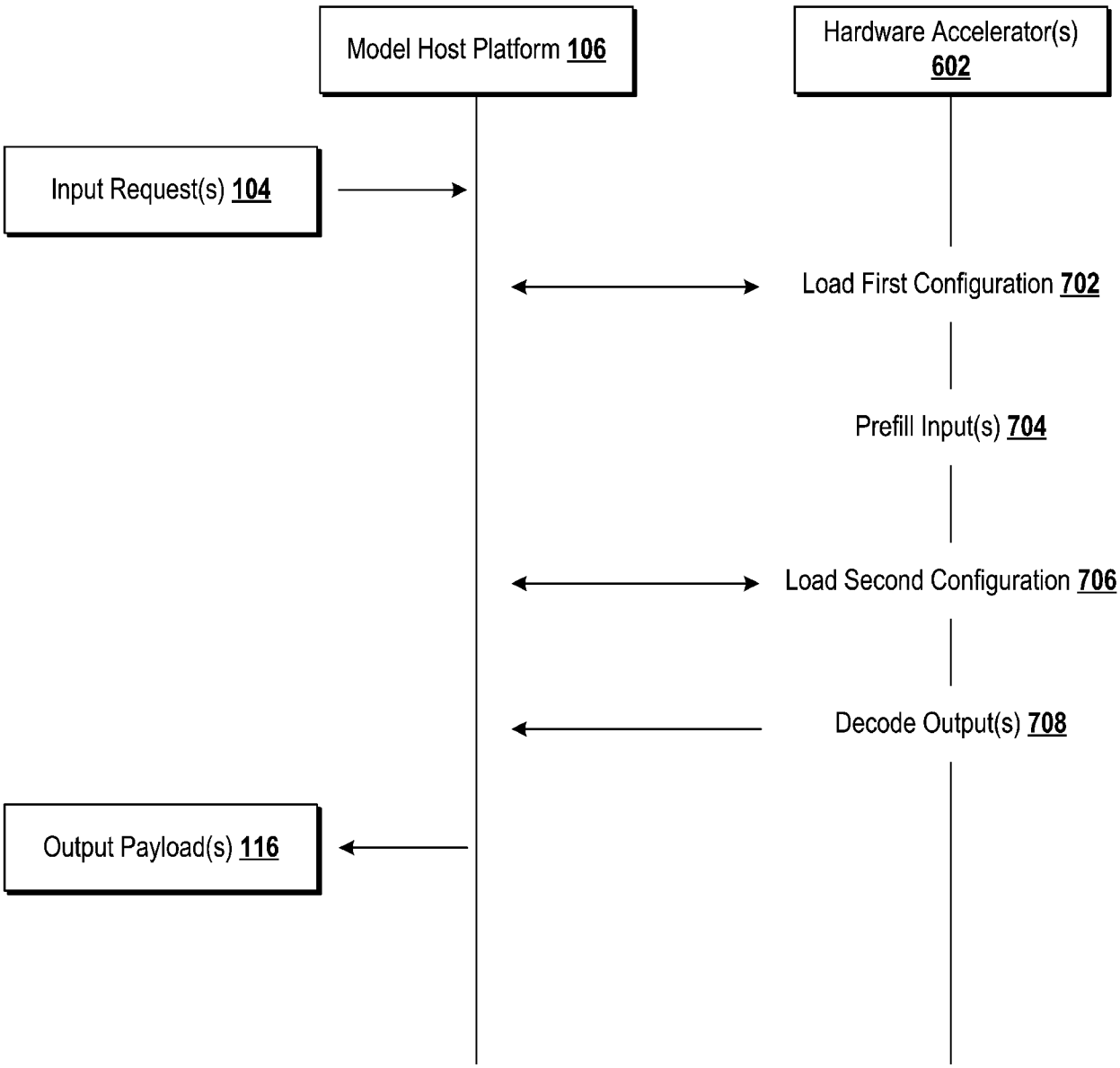

Model Host Platform 106
Hardware Accelerator(s) 602
Input Request(s) 104
Load First Configuration 702
Prefill Input(s) 704
Load Second Configuration 706
Decode Output(s) 708
Output Payload(s) 116

FIG. 7

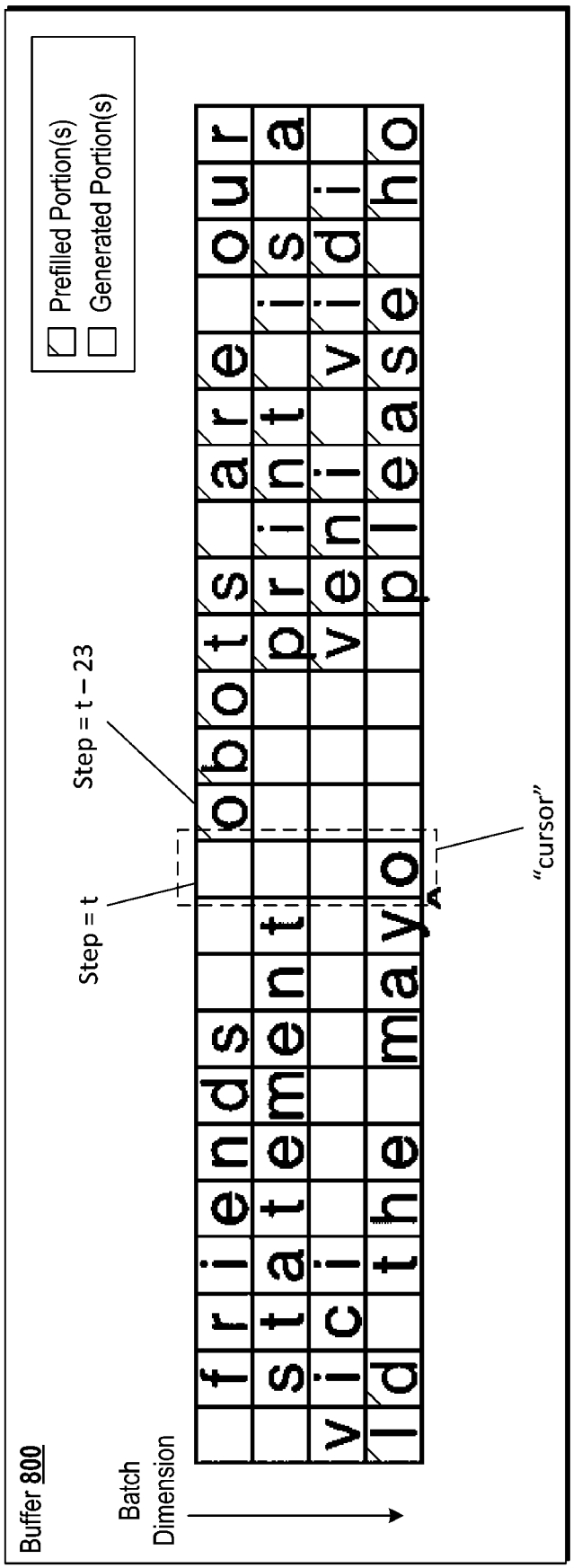

Prefilled Portion(s)
Generated Portion(s)
Buffer 800
Batch Dimension
Step = t – 23
Step = t
"cursor"

FIG. 8

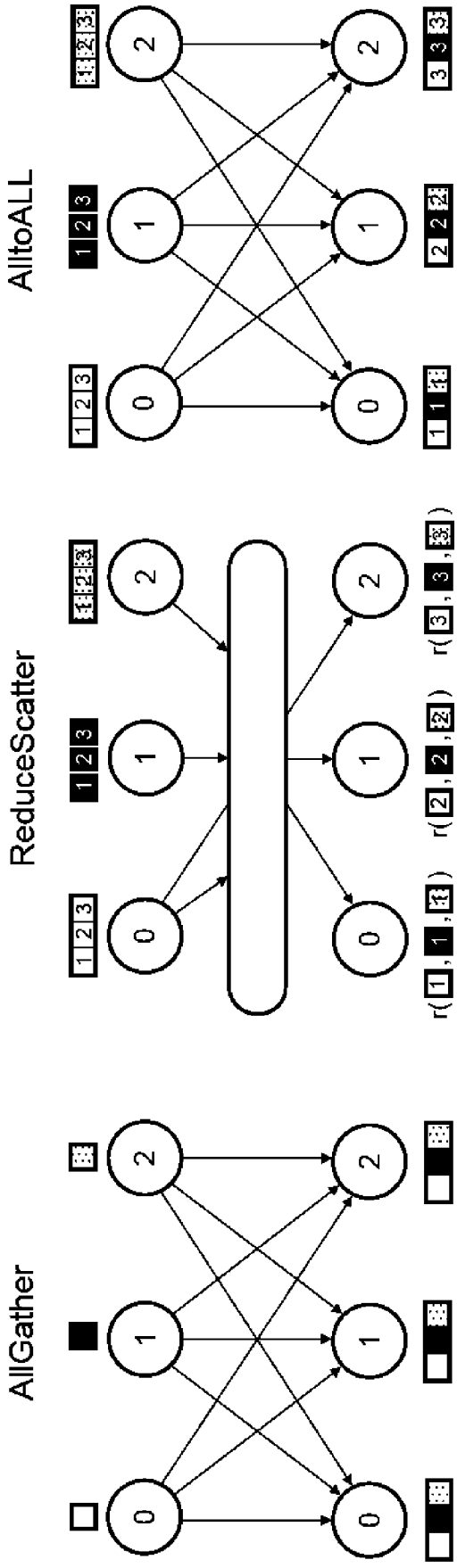


FIG. 9

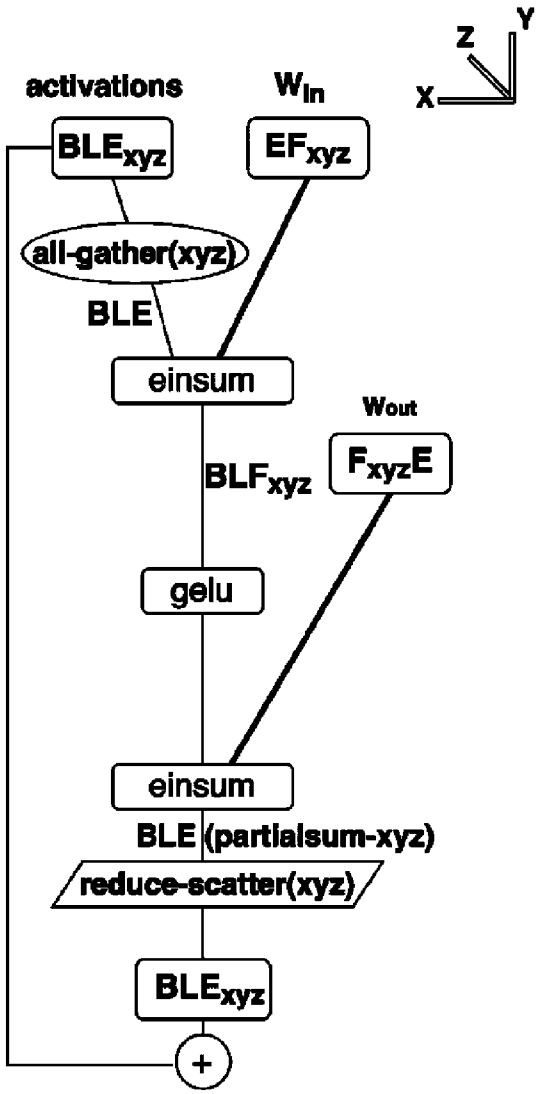

Z
Y
X
activations
W_{In}
BLE_{xyz}
EF_{xyz}
all-gather(xyz)
BLE
einsum
Wout
BLF_{xyz}
$F_{xyz}E$
gelu
einsum
BLE (partialsum-xyz)
reduce-scatter(xyz)
BLE_{xyz}
+

FIG. 10

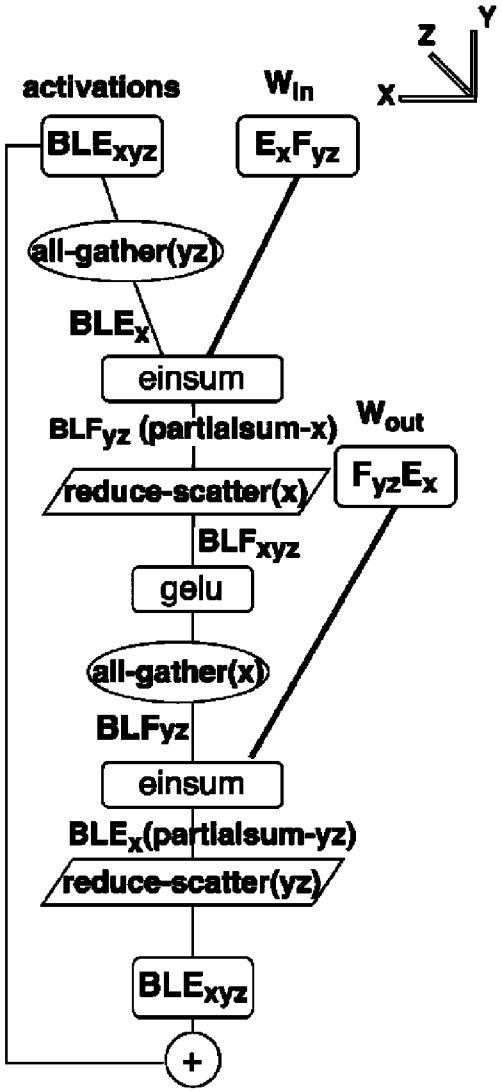

Z
Y
X
activations
W_{In}
BLE_{xyz}
E_xF_{yz}
all-gather(yz)
BLE_x
einsum
BLF_{yz} (partialsum-x)
W_{out}
reduce-scatter(x)
$F_{yz}E_x$
BLF_{xyz}
gelu
all-gather(x)
BLF_{yz}
einsum
BLE_x(partialsum-yz)
reduce-scatter(yz)
BLE_{xyz}
+

FIG. 11

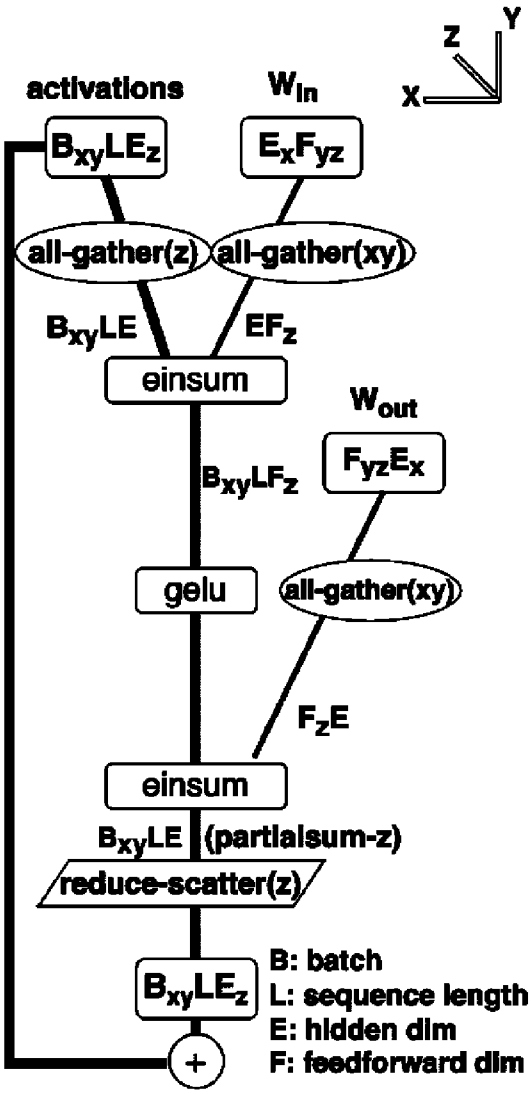

Z
Y
X
activations
W_{in}
$B_{xy}LE_z$
E_xF_{yz}
all-gather(z)
all-gather(xy)
$B_{xy}LE$
EF_z
einsum
W_{out}
$F_{yz}E_x$
$B_{xy}LF_z$
gelu
all-gather(xy)
F_zE
einsum
$B_{xy}LE$
(partialsum-z)
reduce-scatter(z)
$B_{xy}LE_z$
+
B: batch
L: sequence length
E: hidden dim
F: feedforward dim

FIG. 12

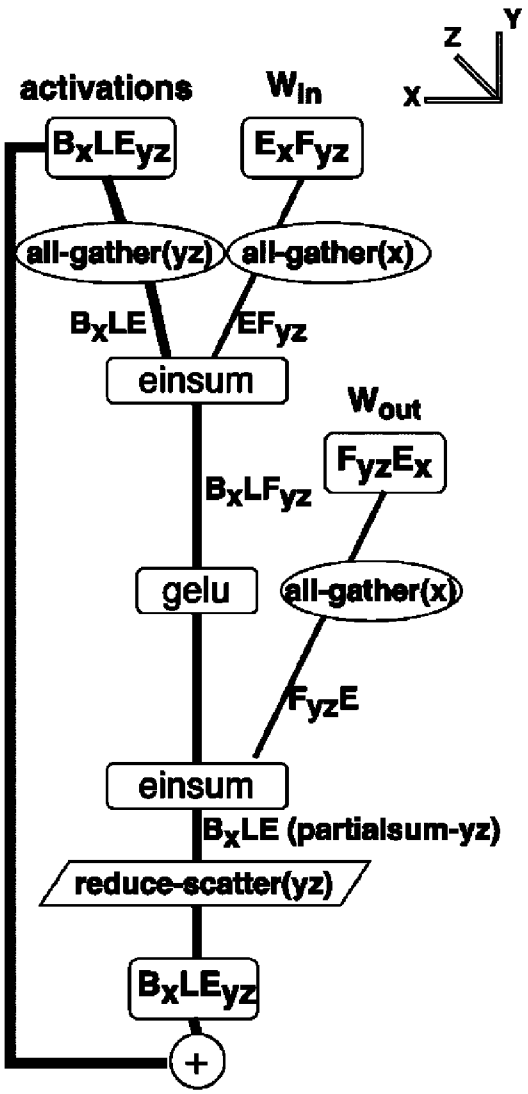

Z
Y
X
activations
W_{in}
B_XLE_{yz}
E_XF_{yz}
all-gather(yz)
all-gather(x)
B_XLE
EF_{yz}
einsum
W_{out}
$F_{yz}E_x$
B_XLF_{yz}
gelu
all-gather(x)
$F_{yz}E$
einsum
B_XLE (partialsum-yz)
reduce-scatter(yz)
B_XLE_{yz}
+

FIG. 13

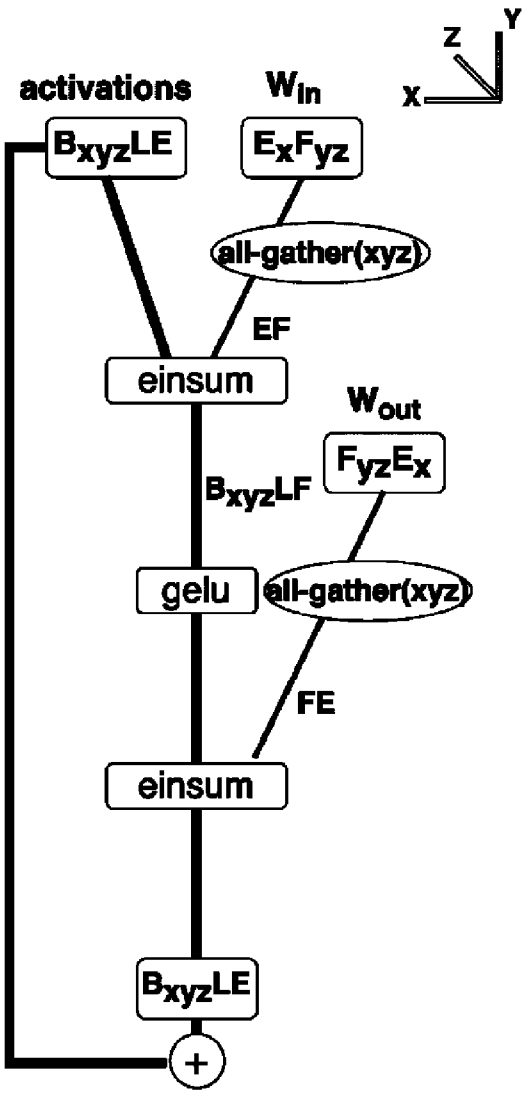

Z
Y
X
activations
W_{in}
$B_{xyz}LE$
E_xF_{yz}
all-gather(xyz)
EF
einsum
W_{out}
$F_{yz}E_x$
$B_{xyz}LF$
gelu
all-gather(xyz)
FE
einsum
$B_{xyz}LE$
+

FIG. 14

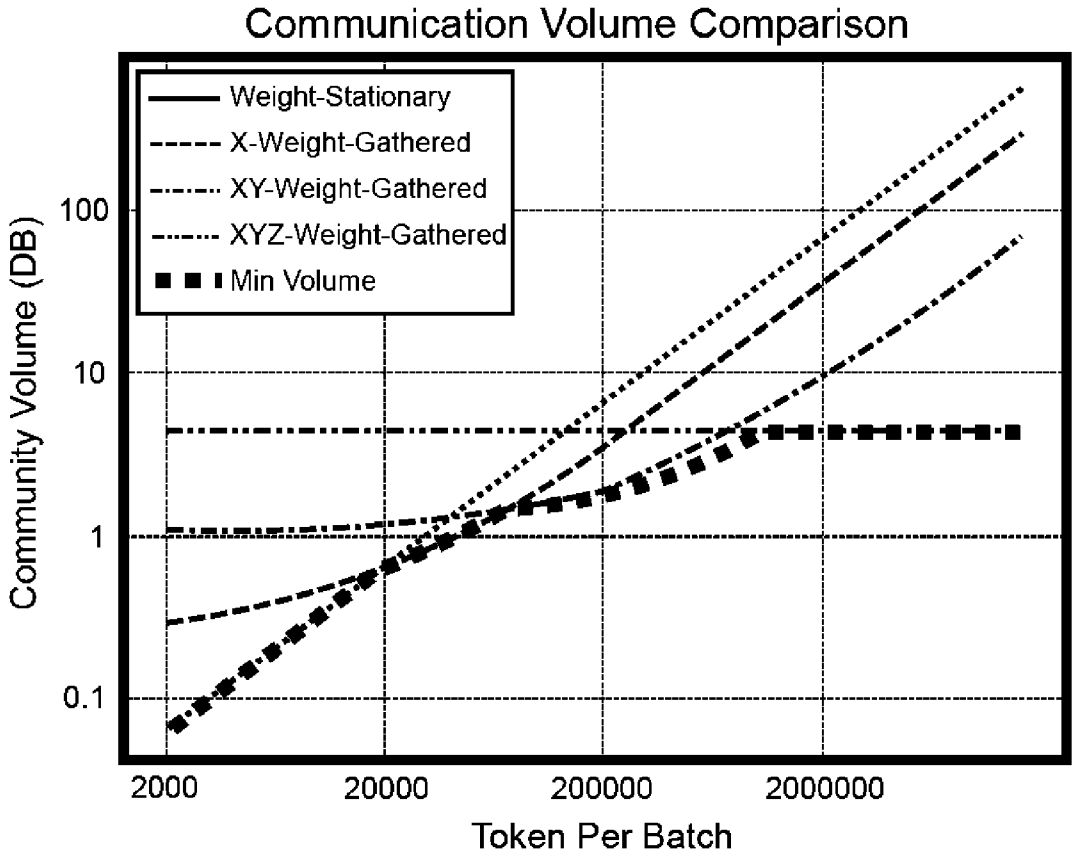
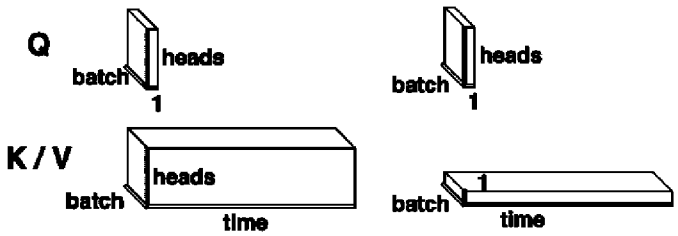

Communication Volume Comparison
Weight-Stationary
X-Weight-Gathered
XY-Weight-Gathered
XYZ-Weight-Gathered
Min Volume
Community Volume (DB)
100
10
1
0.1
2000
20000
200000
2000000
Token Per Batch

FIG. 15



MULTI-HEAD *MULTI-QUERY*

FIG. 16

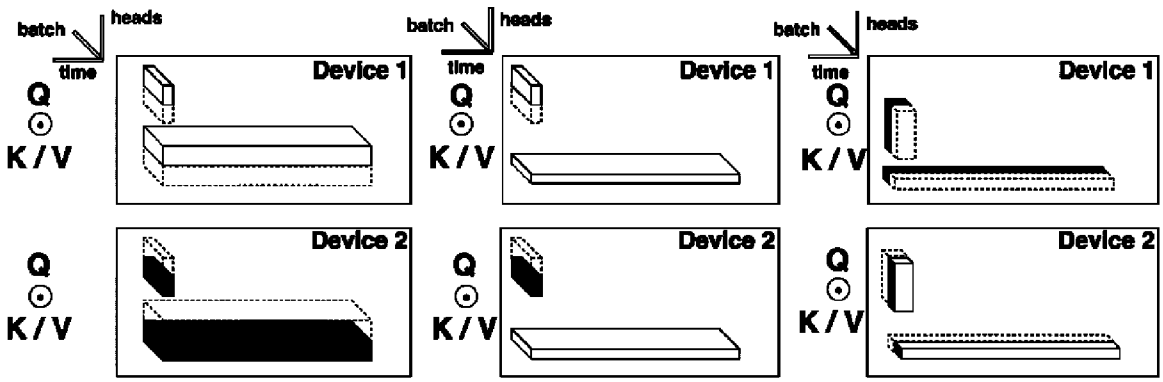


FIG. 17

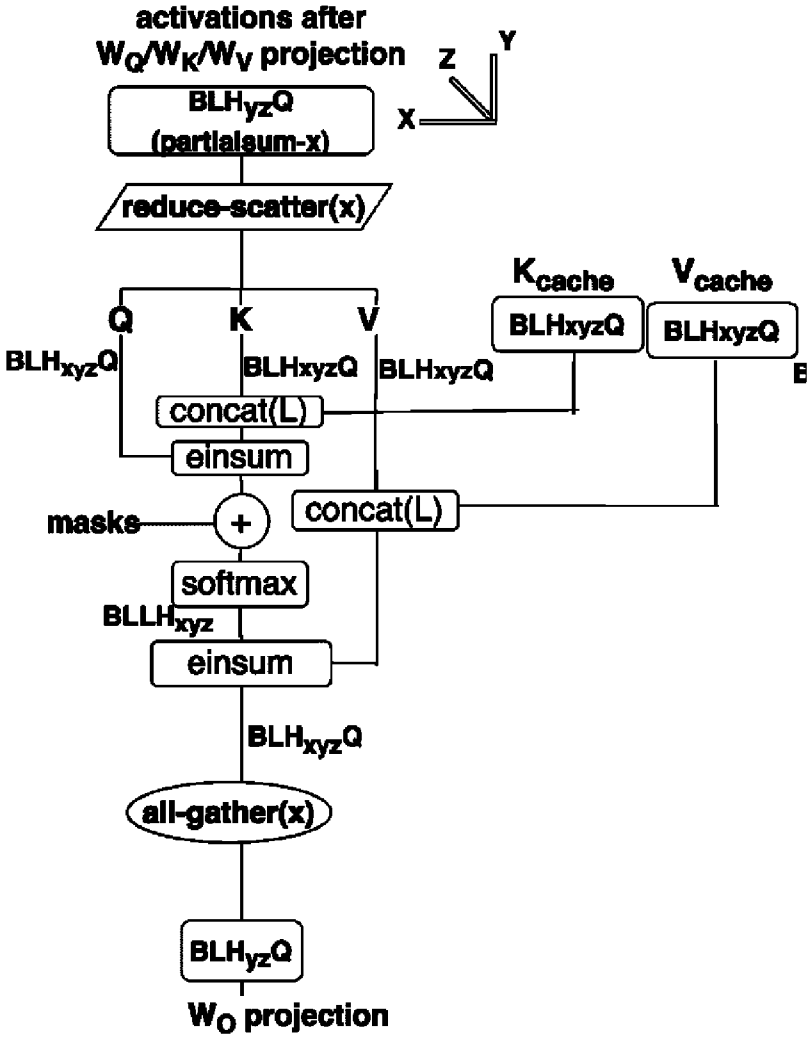

activations after
$W_Q/W_K/W_V$ projection
$BLH_{yz}Q$
(partialsum-x)
Z
Y
X
reduce-scatter(x)
Q
K
V
K_{cache}
V_{cache}
BLHxyzQ
BLHxyzQ
$BLH_{xyz}Q$
BLHxyzQ
BLHxyzQ
concat(L)
einsum
masks
+
concat(L)
softmax
$BLLH_{xyz}$
einsum
$BLH_{xyz}Q$
all-gather(x)
$BLH_{yz}Q$
W_O projection

FIG. 18

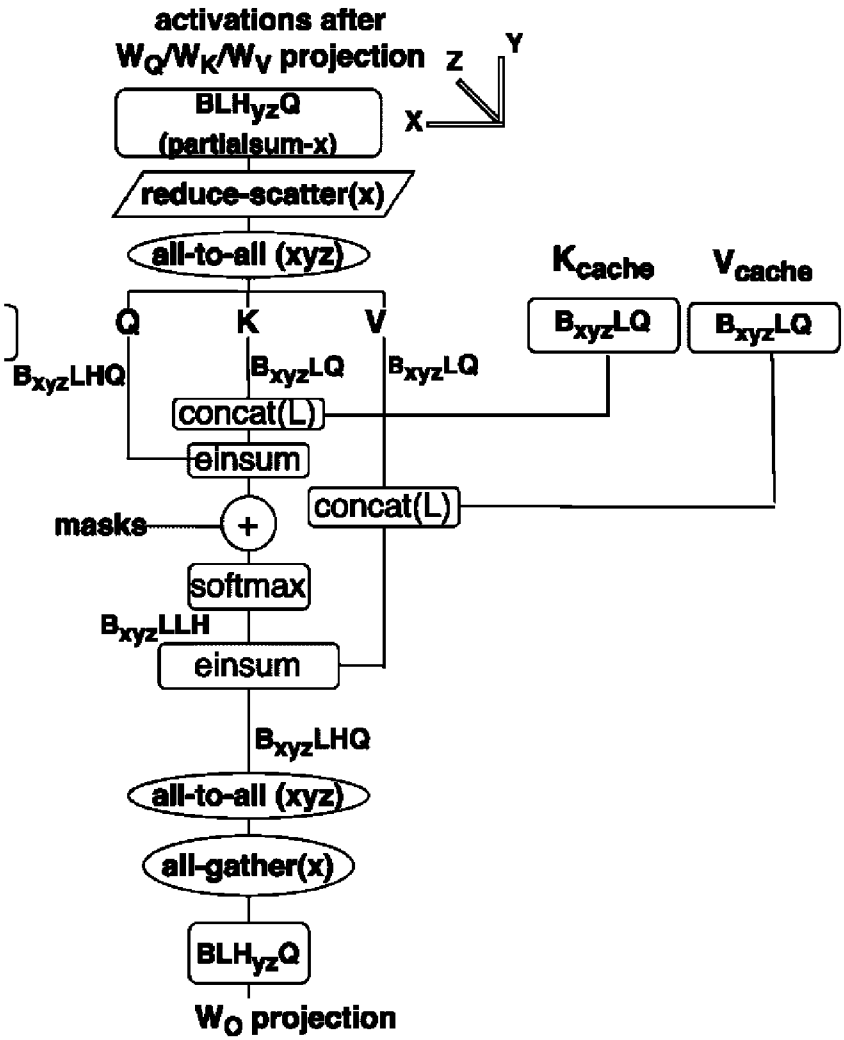

activations after
$W_Q/W_K/W_V$ projection
Z
Y
X
$BLH_{yz}Q$
(partialsum-x)
reduce-scatter(x)
all-to-all (xyz)
K_{cache}
V_{cache}
Q
K
V
$B_{xyz}LHQ$
$B_{xyz}LQ$
$B_{xyz}LQ$
$B_{xyz}LQ$
$B_{xyz}LQ$
concat(L)
einsum
concat(L)
masks
+
softmax
$B_{xyz}LLH$
einsum
$B_{xyz}LHQ$
all-to-all (xyz)
all-gather(x)
$BLH_{yz}Q$
W_O projection

FIG. 19

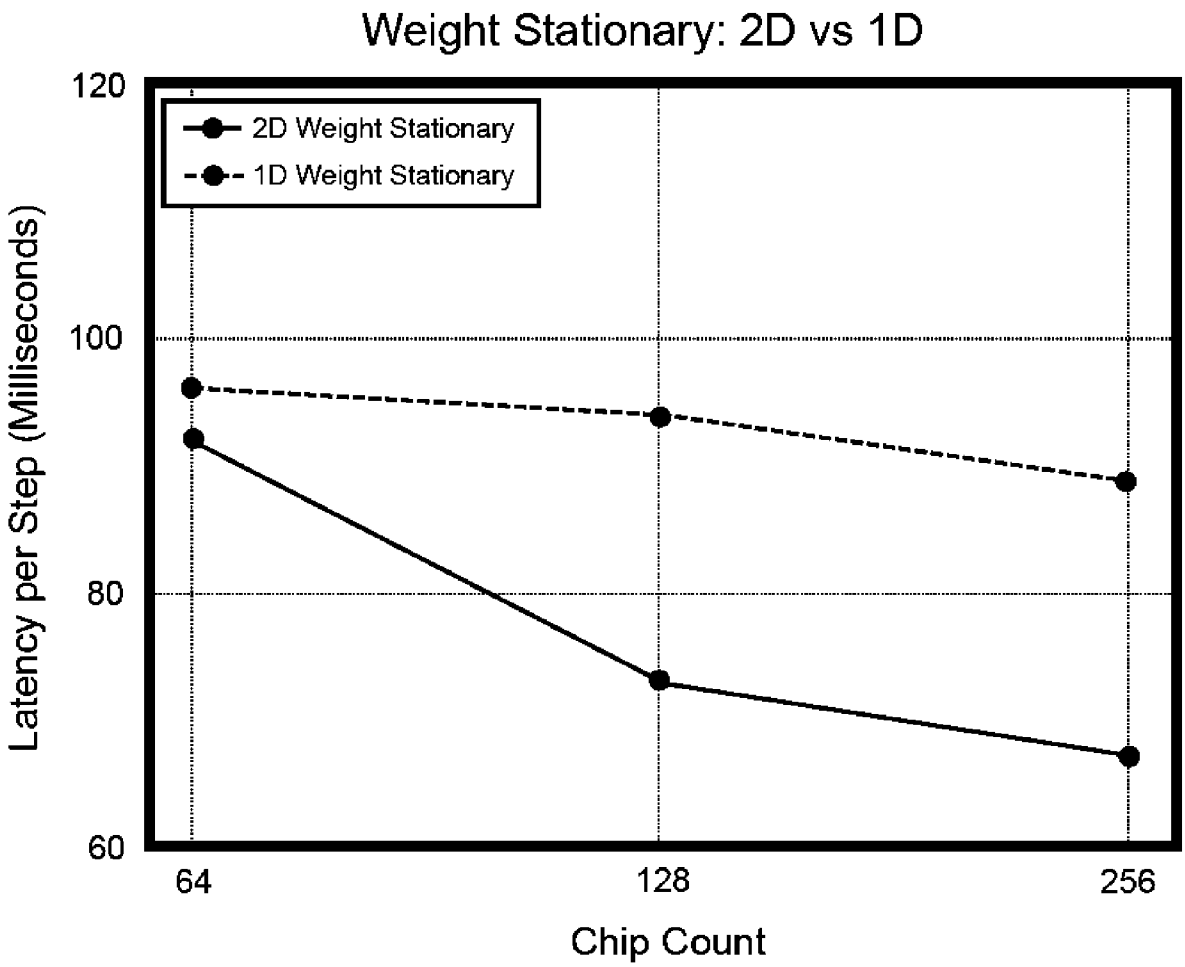

Weight Stationary: 2D vs 1D
2D Weight Stationary
1D Weight Stationary
Latency per Step (Milliseconds)
120
100
80
60
64
128
256
Chip Count

FIG. 20

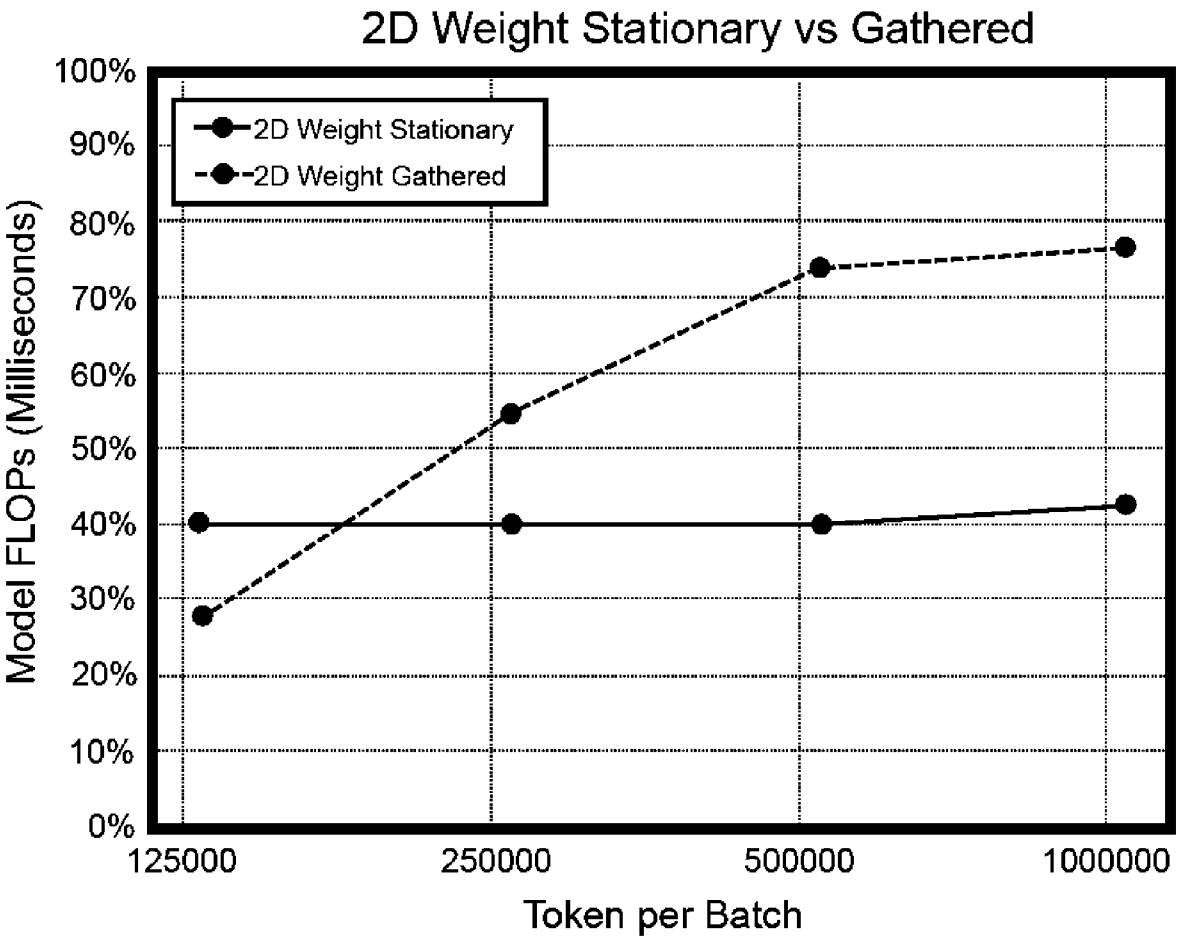

2D Weight Stationary vs Gathered
2D Weight Stationary
2D Weight Gathered
Model FLOPs (Milliseconds)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
125000
250000
500000
1000000
Token per Batch

FIG. 21

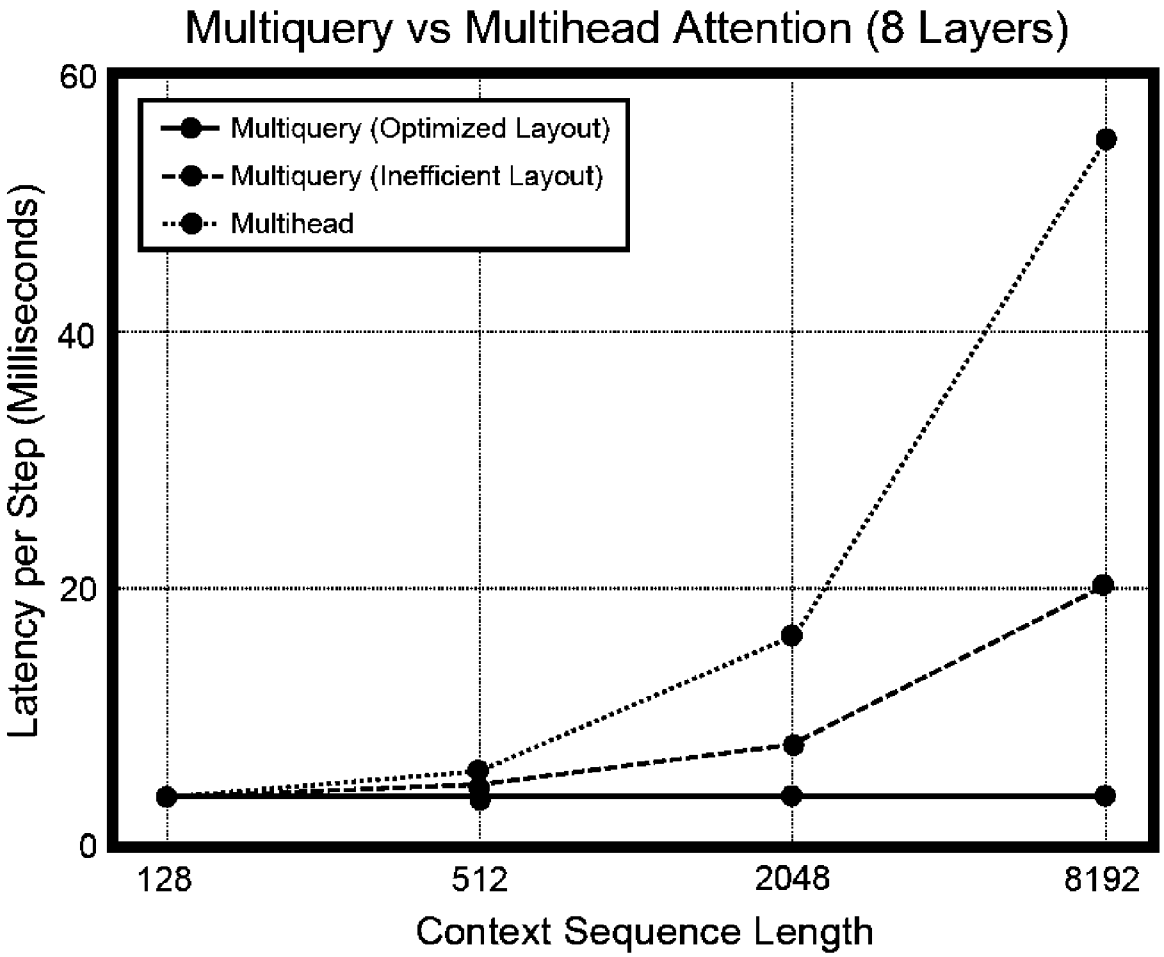

Multiquery vs Multihead Attention (8 Layers)
Multiquery (Optimized Layout)
Multiquery (Inefficient Layout)
Multihead
Latency per Step (Milliseconds)
60
40
20
0
128
512
2048
8192
Context Sequence Length

FIG. 22

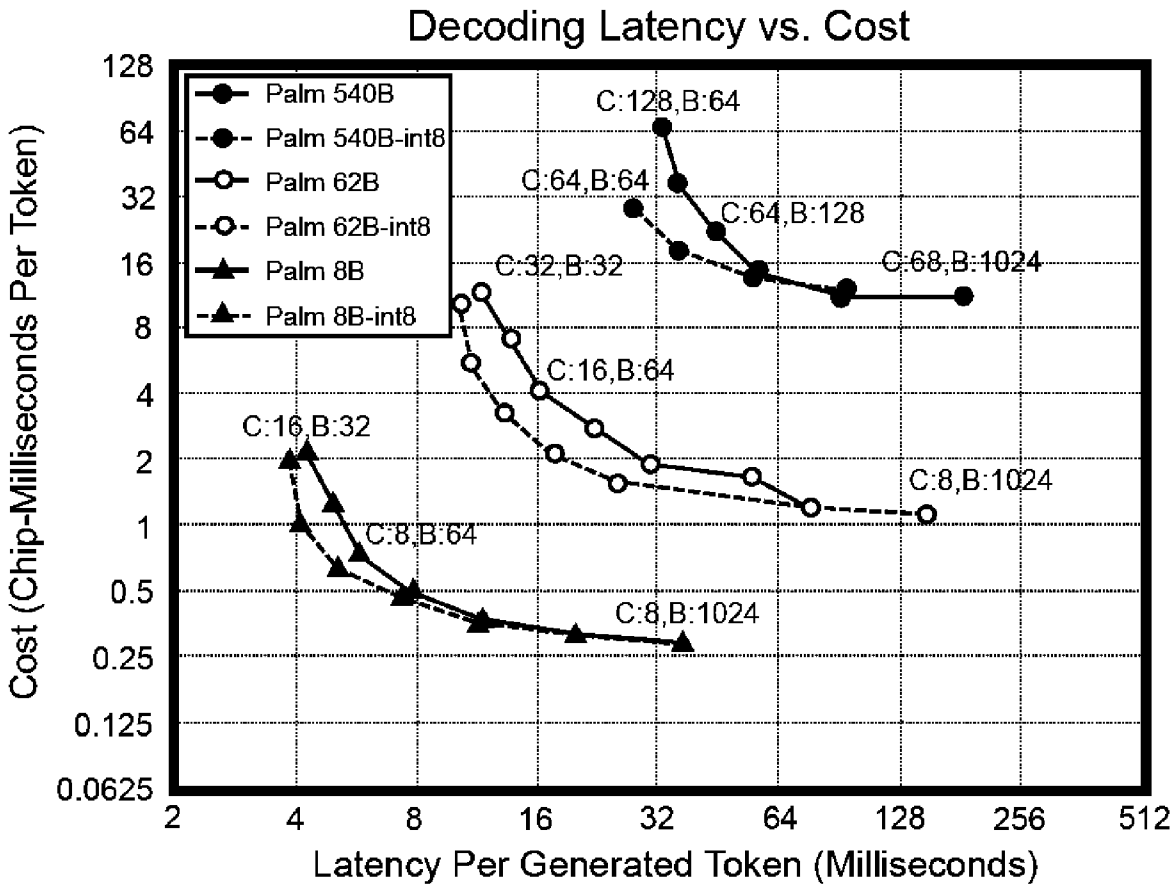

Decoding Latency vs. Cost
Cost (Chip-Milliseconds Per Token)
Latency Per Generated Token (Milliseconds)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0.0625
2
4
8
16
32
64
128
256
512
Palm 540B
Palm 540B-int8
Palm 62B
Palm 62B-int8
Palm 8B
Palm 8B-int8
C:128,B:64
C:64,B:64
C:64,B:128
C:68,B:1024
C:32,B:32
C:16,B:64
C:16,B:32
C:8,B:1024
C:8,B:64
C:8,B:1024

FIG. 23

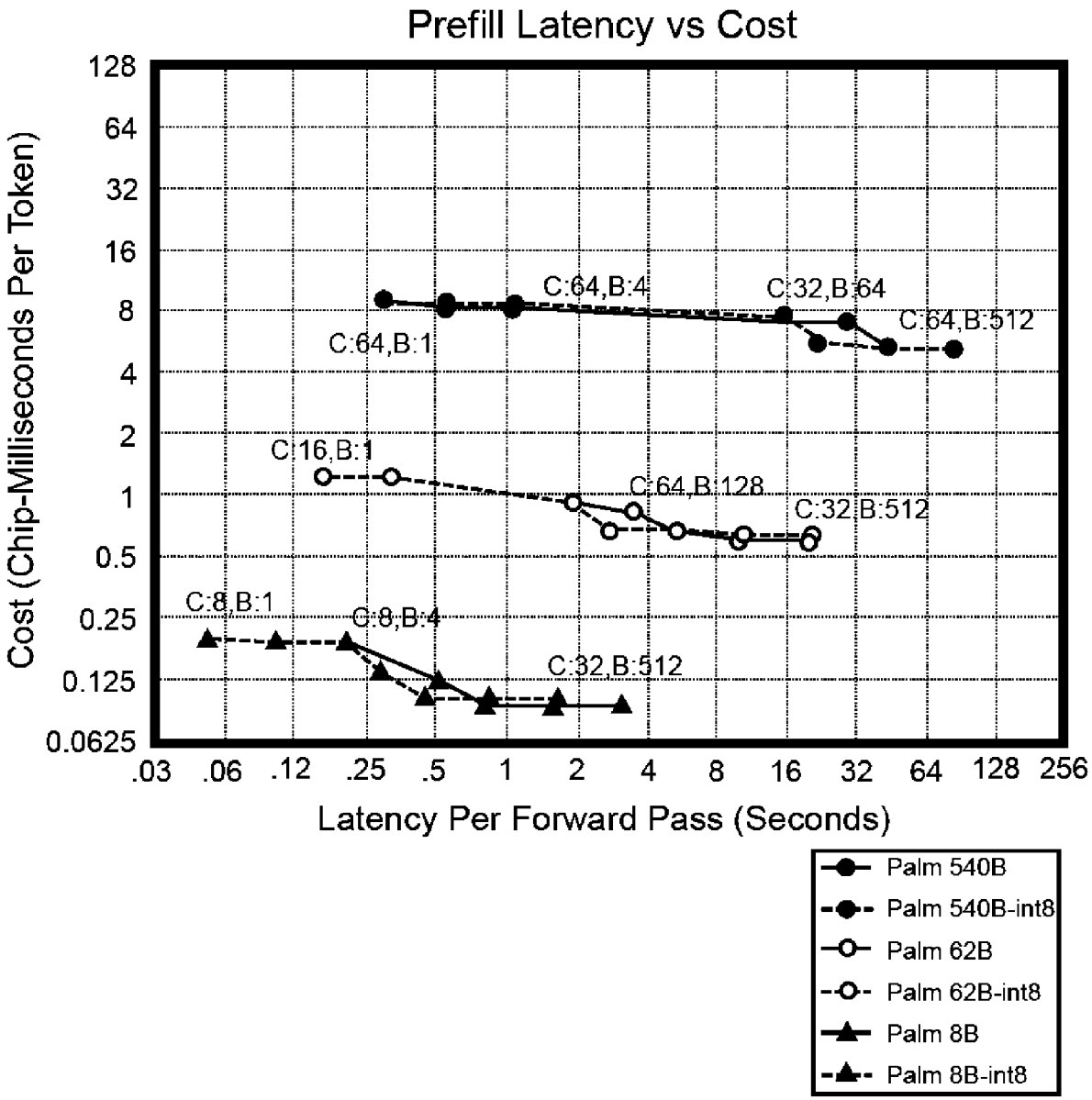

Prefill Latency vs Cost
Cost (Chip-Milliseconds Per Token)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0.0625
.03
.06
.12
.25
.5
1
2
4
8
16
32
64
128
256
Latency Per Forward Pass (Seconds)
C:64,B:4
C:32,B:64
C:64,B:512
C:64,B:1
C:16,B:1
C:64,B:128
C:32,B:512
C:8,B:1
C:8,B:4
C:32,B:512
Palm 540B
Palm 540B-int8
Palm 62B
Palm 62B-int8
Palm 8B
Palm 8B-int8

FIG. 24

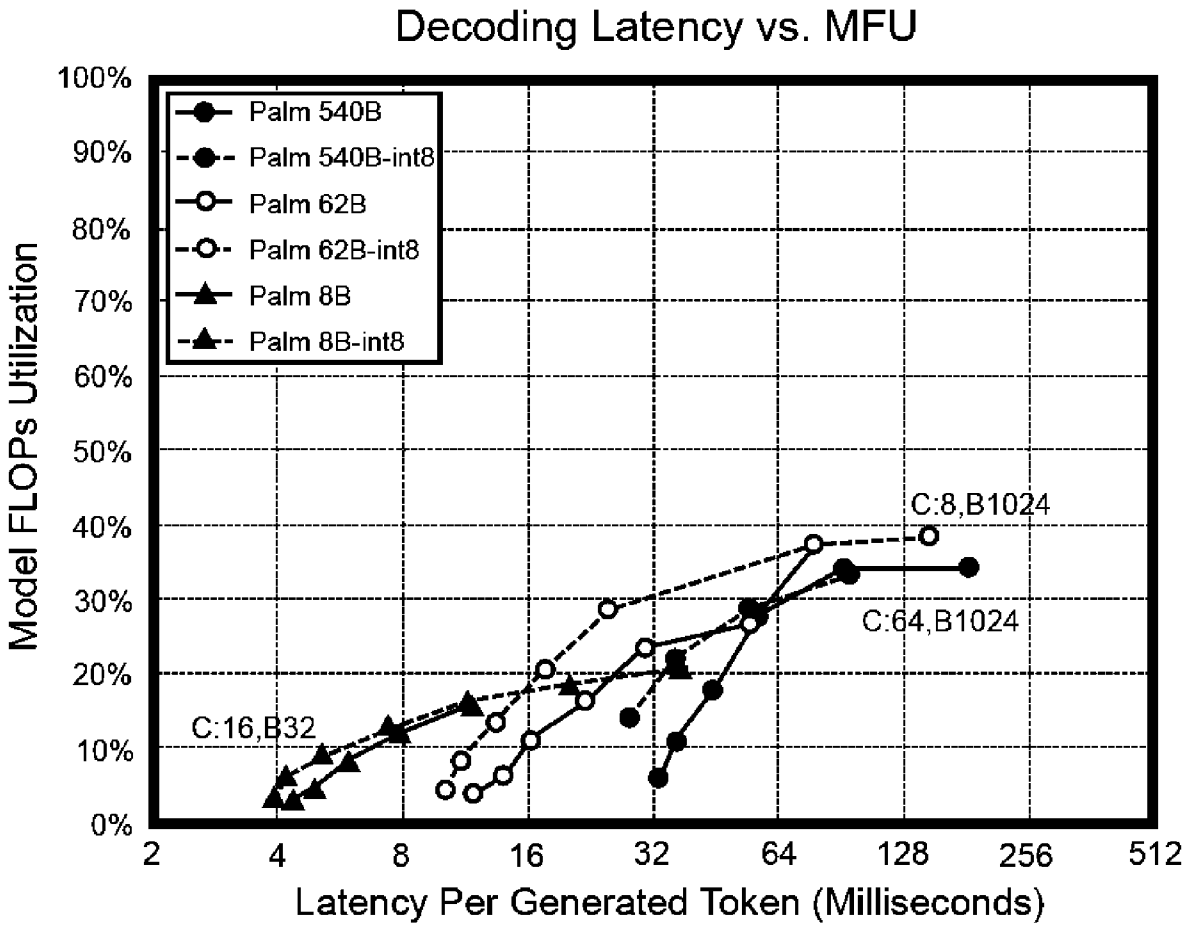

Decoding Latency vs. MFU
Model FLOPs Utilization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Palm 540B
Palm 540B-int8
Palm 62B
Palm 62B-int8
Palm 8B
Palm 8B-int8
C:8,B1024
C:64,B1024
C:16,B32
2
4
8
16
32
64
128
256
512
Latency Per Generated Token (Milliseconds)

FIG. 25

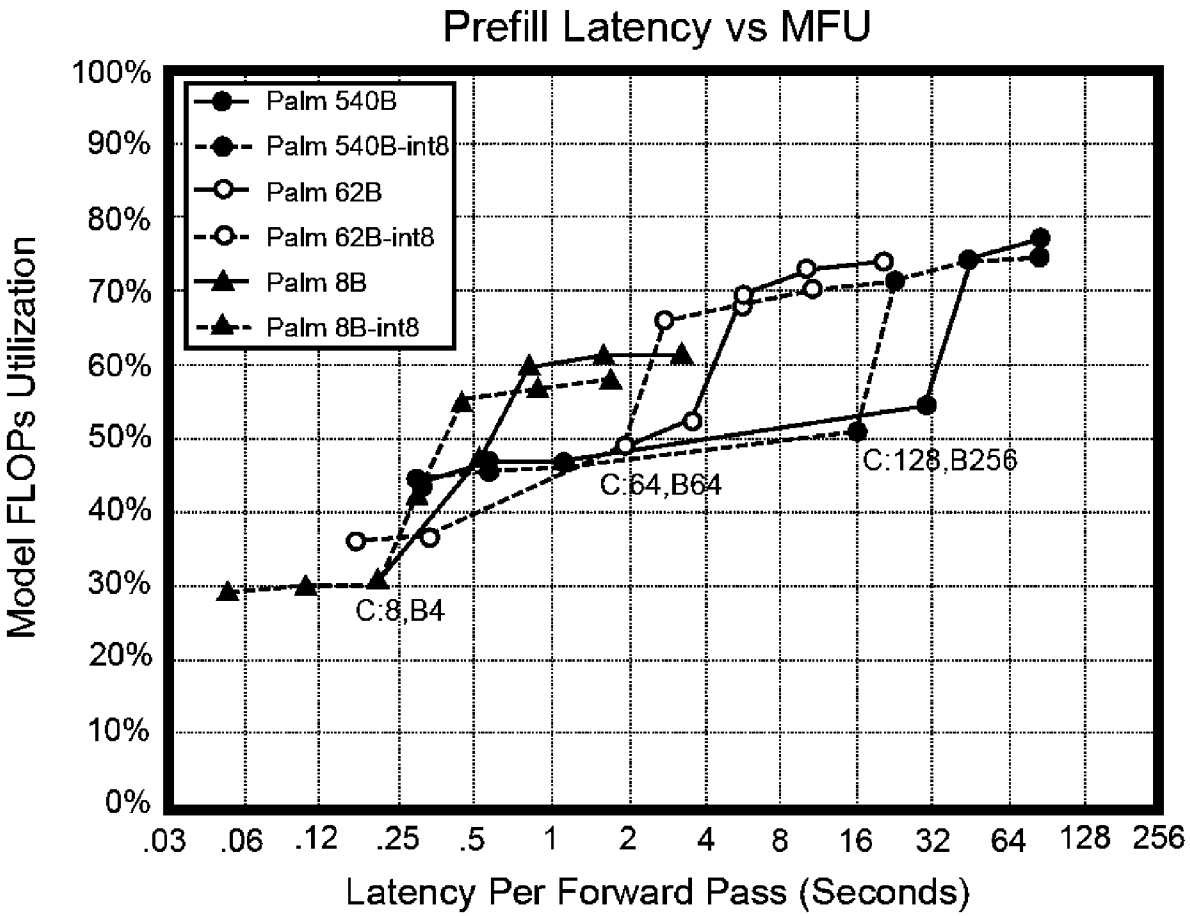

Prefill Latency vs MFU
Model FLOPs Utilization
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Palm 540B
Palm 540B-int8
Palm 62B
Palm 62B-int8
Palm 8B
Palm 8B-int8
C:8,B4
C:64,B64
C:128,B256
.03
.06
.12
.25
.5
1
2
4
8
16
32
64
128
256
Latency Per Forward Pass (Seconds)

FIG. 26

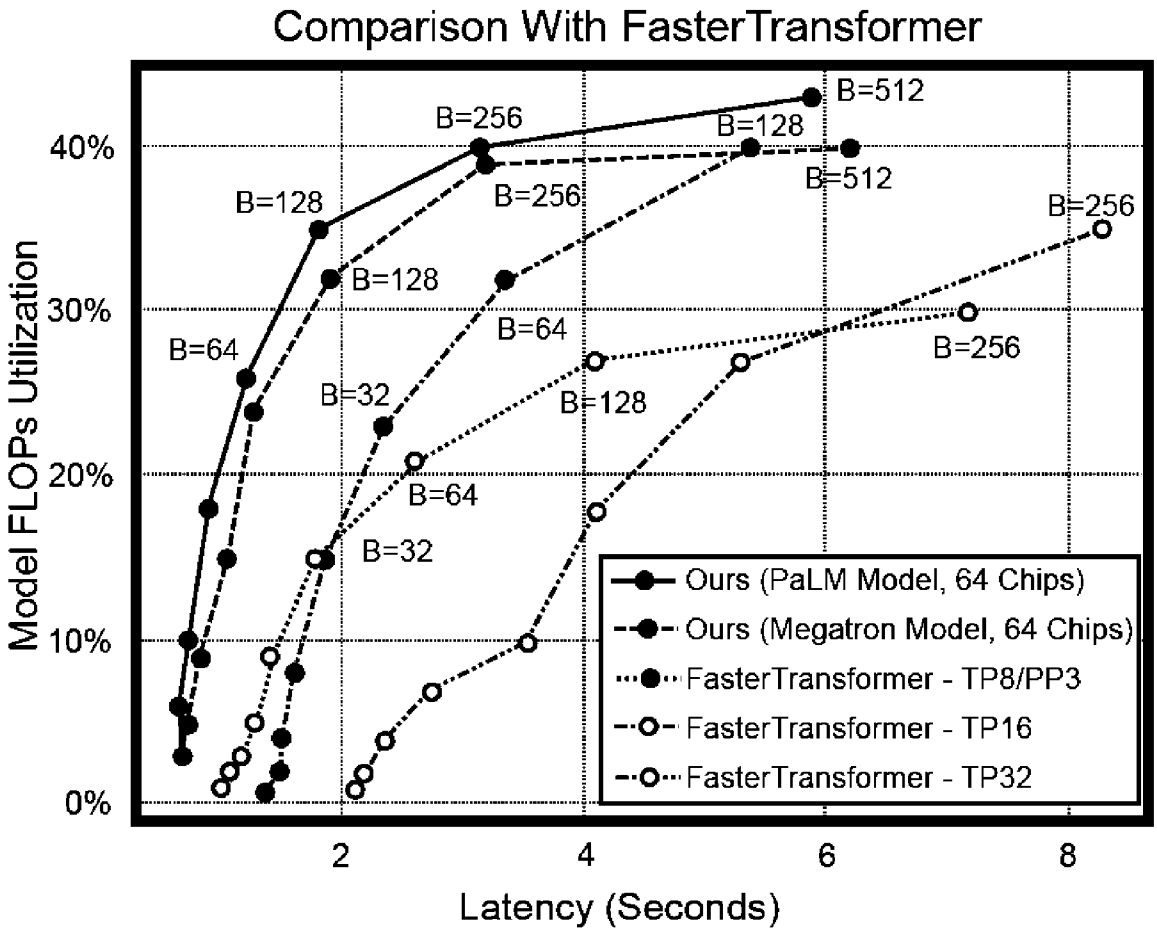

Comparison With FasterTransformer
Model FLOPs Utilization
40%
30%
20%
10%
0%
2
4
6
8
Latency (Seconds)
B=512
B=256
B=128
B=512
B=256
B=128
B=256
B=64
B=128
B=64
B=256
B=32
B=128
B=64
B=32
Ours (PaLM Model, 64 Chips)
Ours (Megatron Model, 64 Chips)
FasterTransformer - TP8/PP3
FasterTransformer - TP16
FasterTransformer - TP32

FIG. 27

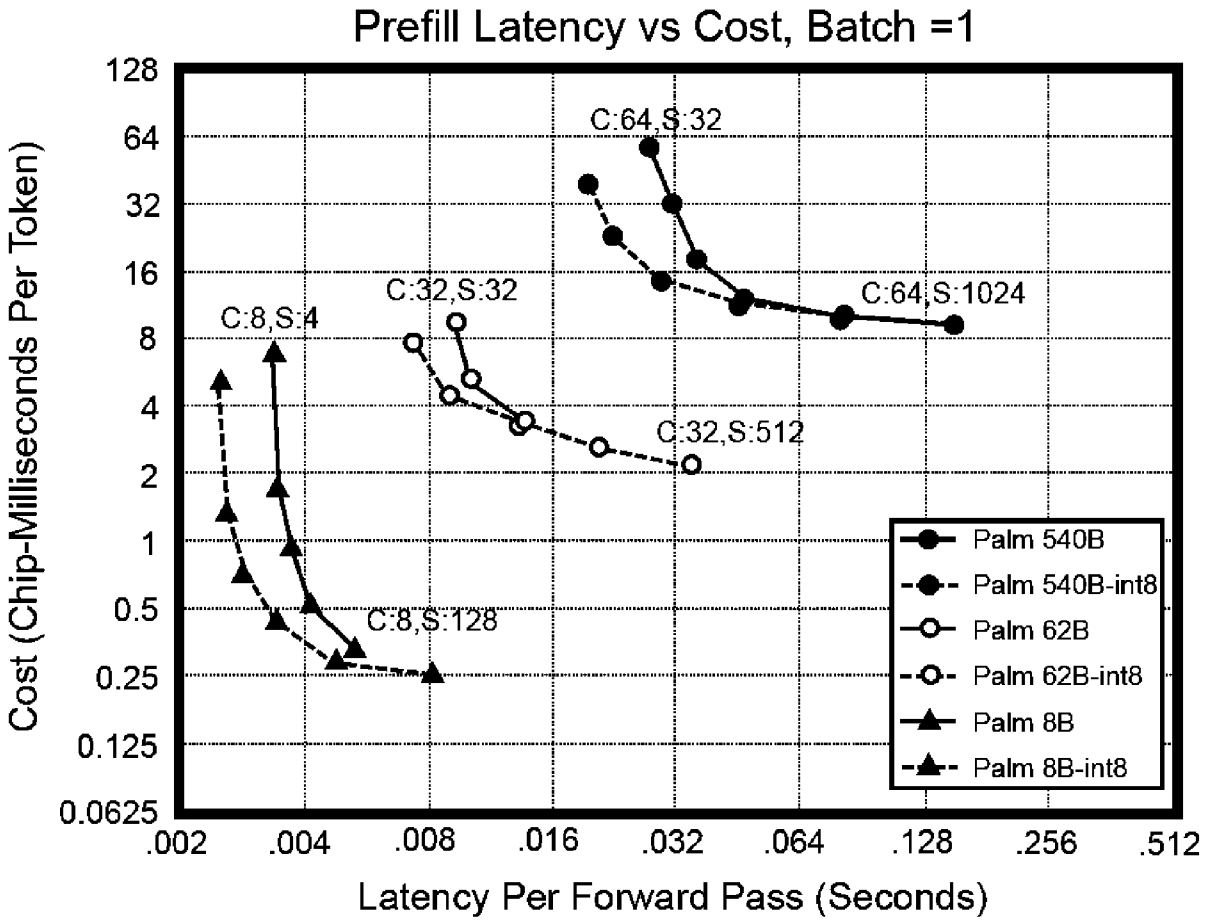

Prefill Latency vs Cost, Batch =1
Cost (Chip-Milliseconds Per Token)
128
64
32
16
8
4
2
1
0.5
0.25
0.125
0.0625
.002
.004
.008
.016
.032
.064
.128
.256
.512
Latency Per Forward Pass (Seconds)
C:64,S:32
C:64,S:1024
C:32,S:32
C:32,S:512
C:8,S:4
C:8,S:128
Palm 540B
Palm 540B-int8
Palm 62B
Palm 62B-int8
Palm 8B
Palm 8B-int8

FIG. 28

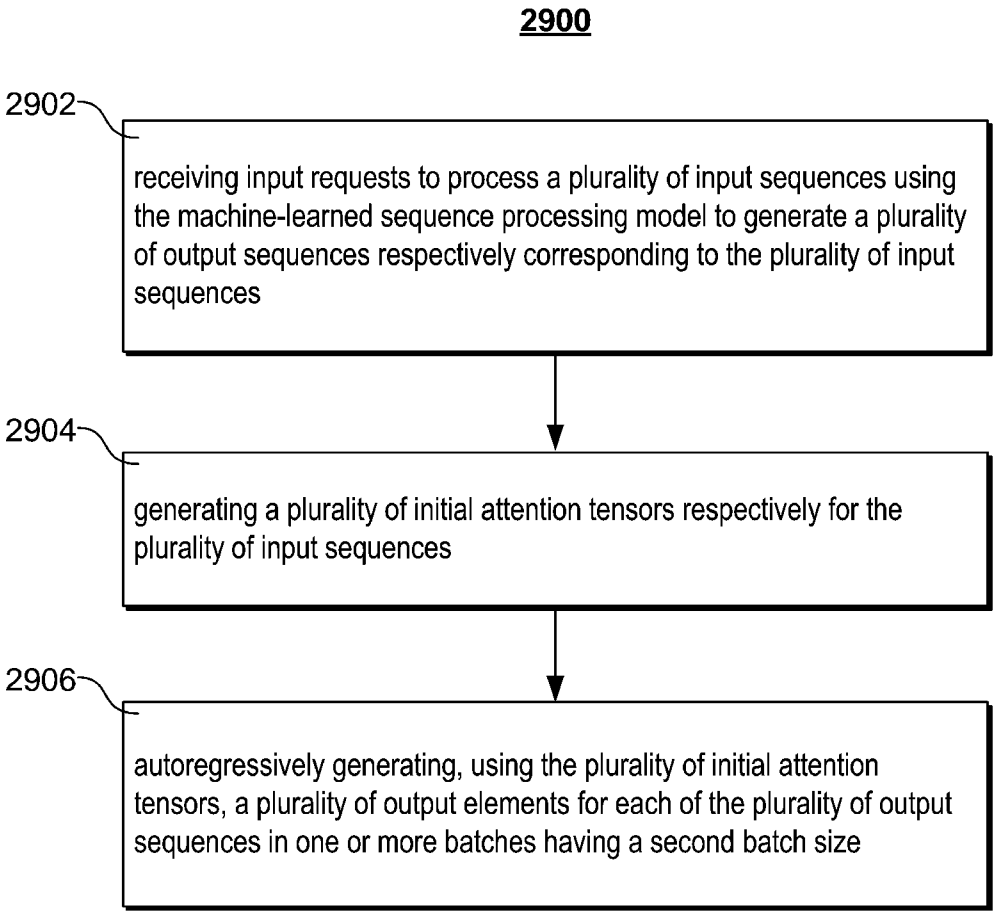

2900
2902
receiving input requests to process a plurality of input sequences using the machine-learned sequence processing model to generate a plurality of output sequences respectively corresponding to the plurality of input sequences
2904
generating a plurality of initial attention tensors respectively for the plurality of input sequences
2906
autoregressively generating, using the plurality of initial attention tensors, a plurality of output elements for each of the plurality of output sequences in one or more batches having a second batch size

FIG. 29

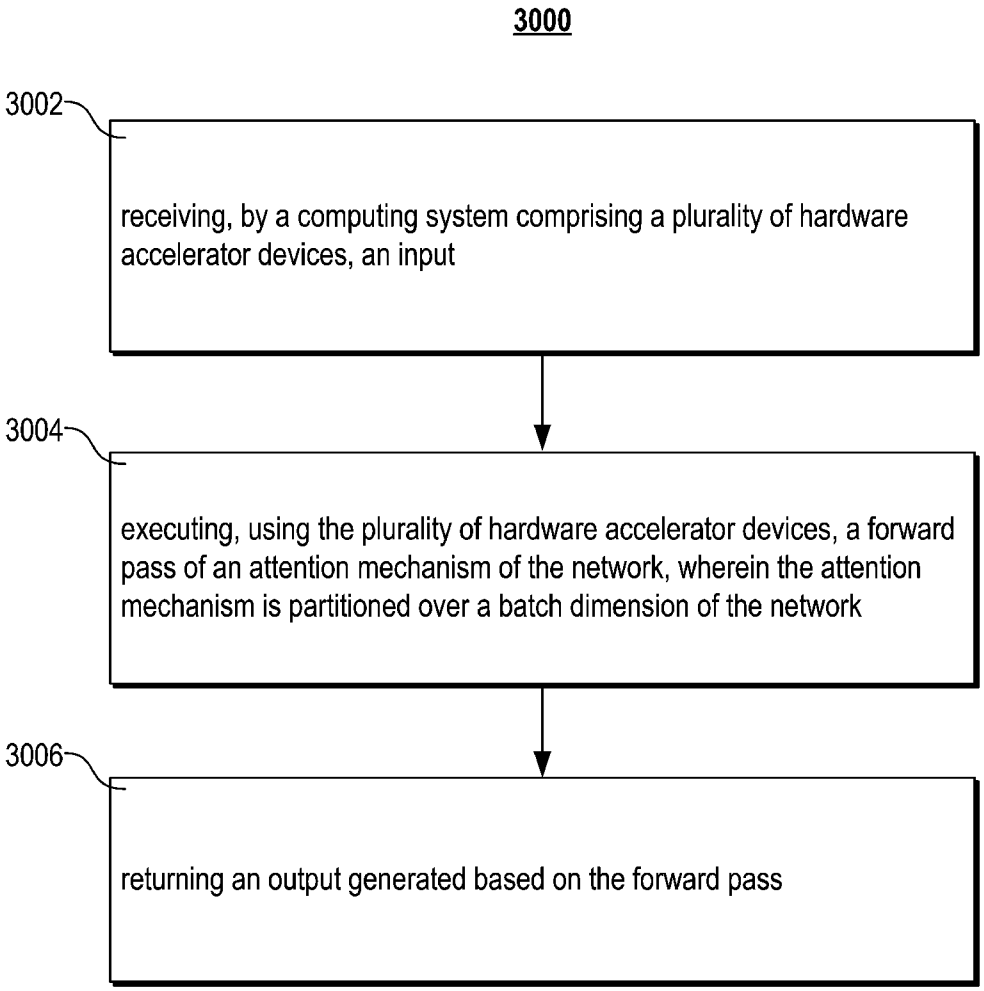

3000
3002
receiving, by a computing system comprising a plurality of hardware accelerator devices, an input
3004
executing, using the plurality of hardware accelerator devices, a forward pass of an attention mechanism of the network, wherein the attention mechanism is partitioned over a batch dimension of the network
3006
returning an output generated based on the forward pass

FIG. 30

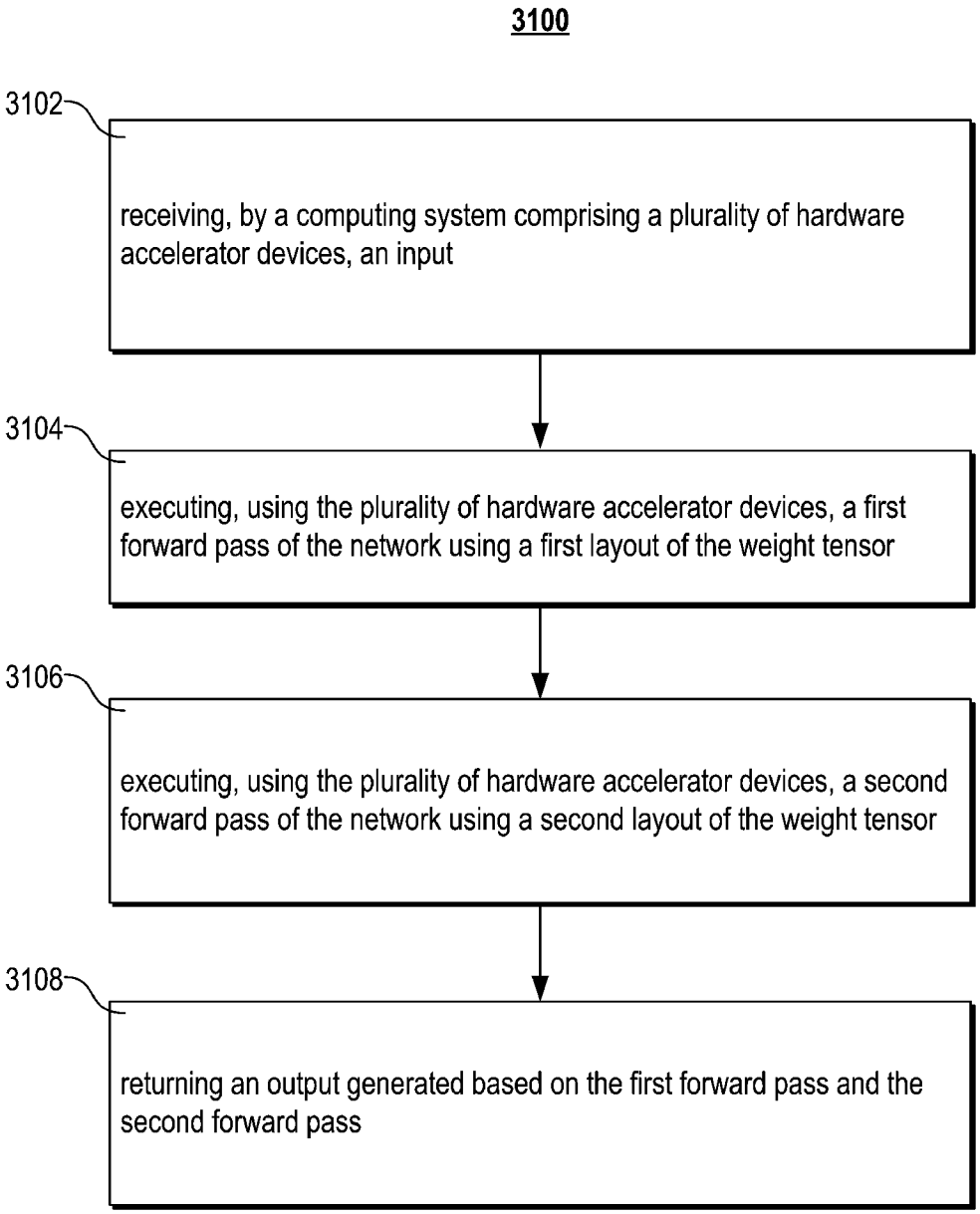

3100
3102
receiving, by a computing system comprising a plurality of hardware accelerator devices, an input
3104
executing, using the plurality of hardware accelerator devices, a first forward pass of the network using a first layout of the weight tensor
3106
executing, using the plurality of hardware accelerator devices, a second forward pass of the network using a second layout of the weight tensor
3108
returning an output generated based on the first forward pass and the second forward pass

FIG. 31

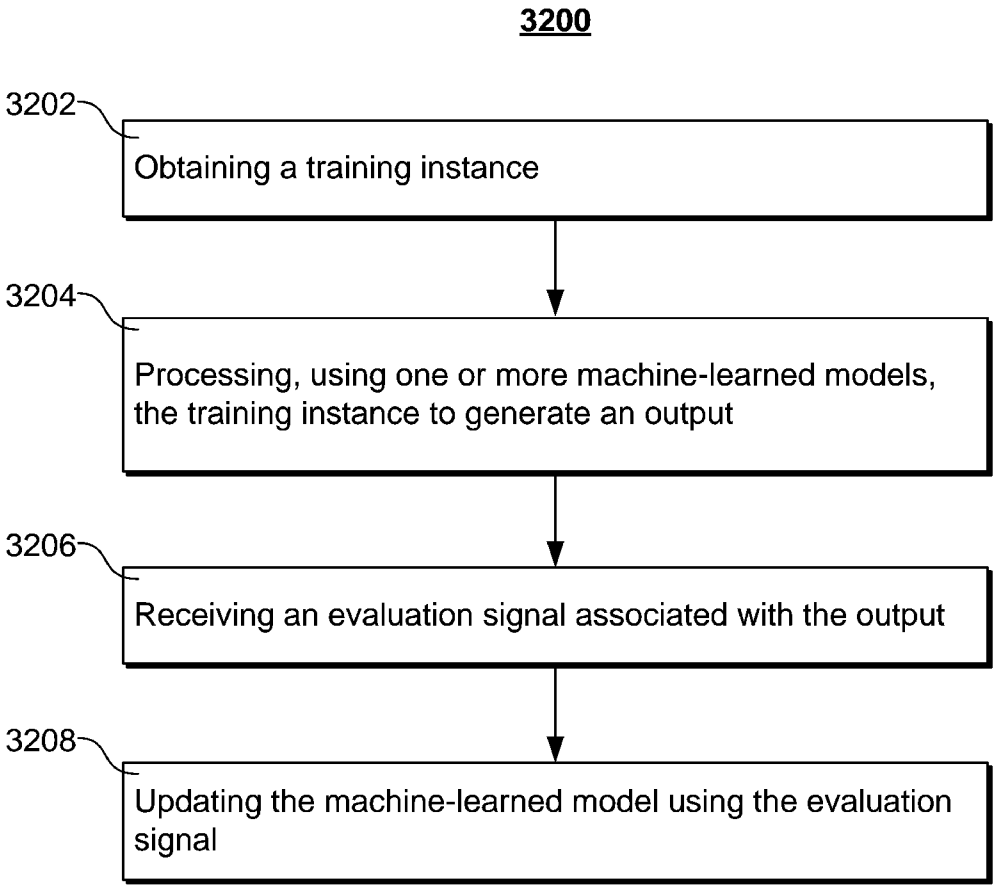

3200
3202
Obtaining a training instance
3204
Processing, using one or more machine-learned models, the training instance to generate an output
3206
Receiving an evaluation signal associated with the output
3208
Updating the machine-learned model using the evaluation signal

FIG. 32

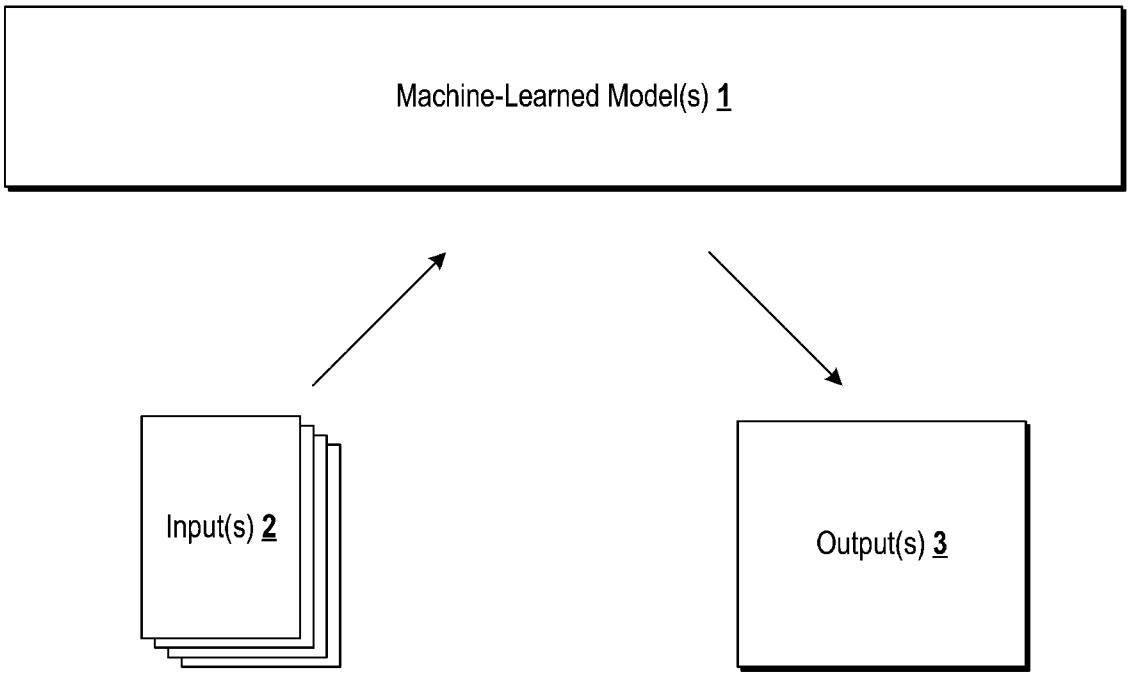


FIG. 33

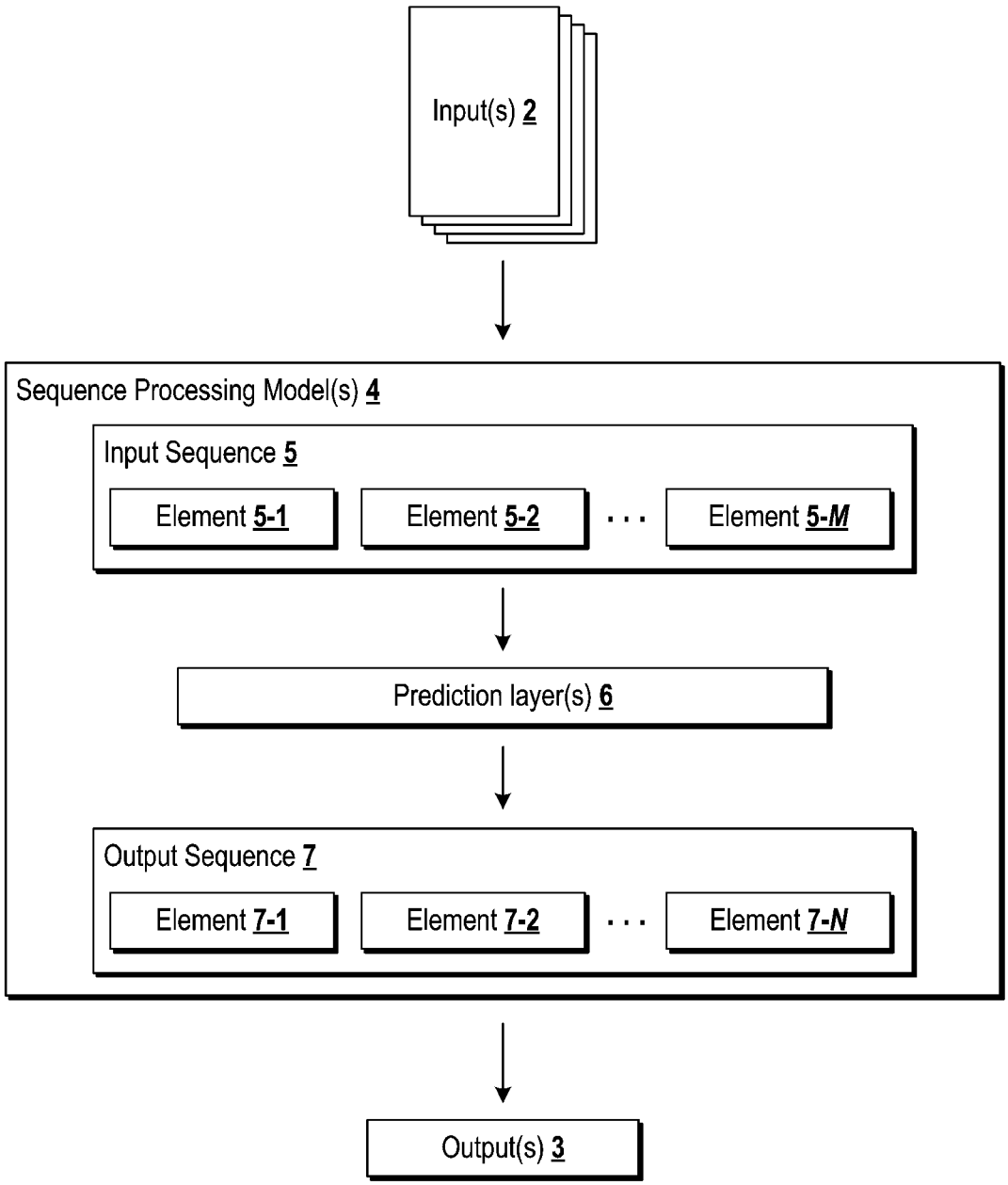

Input(s) 2
Sequence Processing Model(s) 4
Input Sequence 5
Element 5-1
Element 5-2
Element 5-M
Prediction layer(s) 6
Output Sequence 7
Element 7-1
Element 7-2
Element 7-N
Output(s) 3

FIG. 34

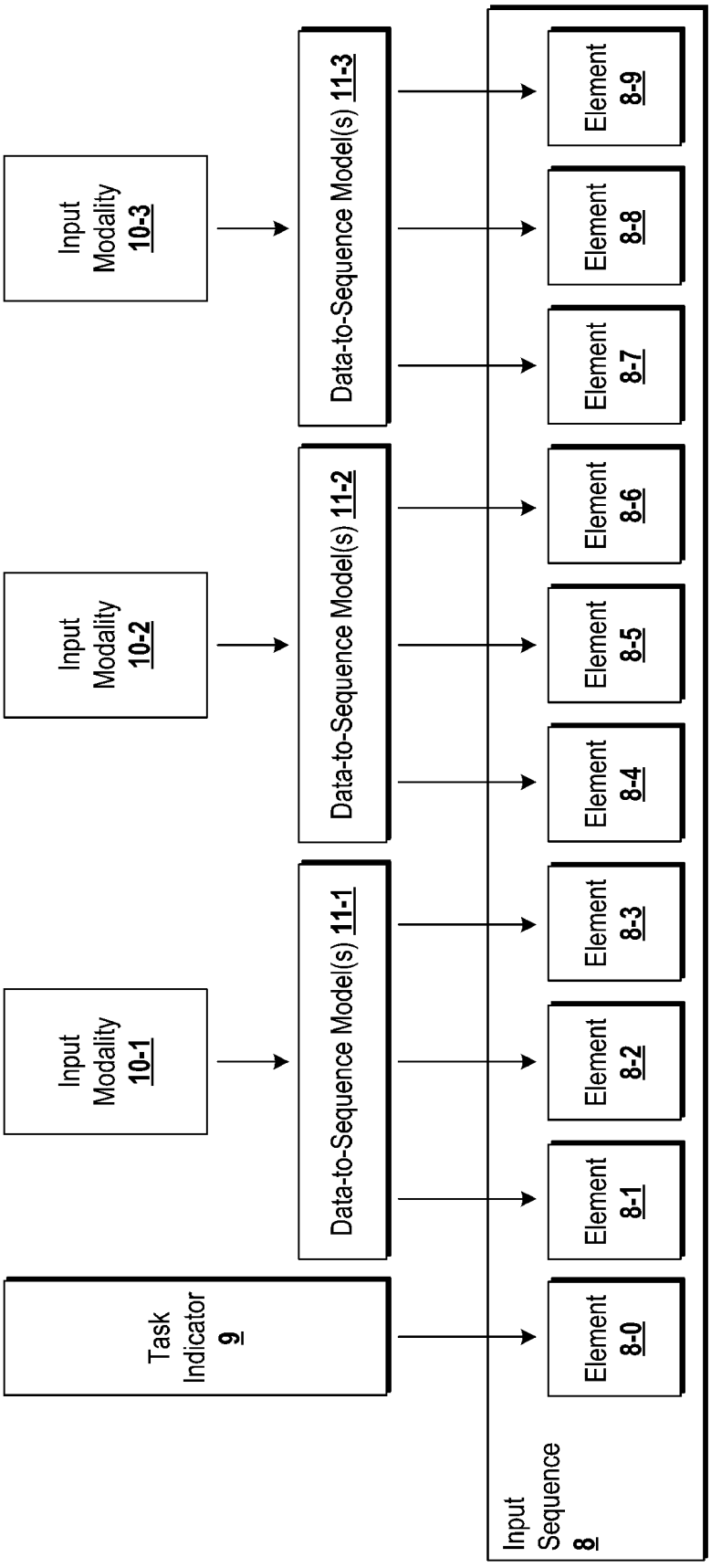

Input Sequence 8
Element 8-0
Element 8-1
Element 8-2
Element 8-3
Element 8-4
Element 8-5
Element 8-6
Element 8-7
Element 8-8
Element 8-9
Task Indicator 9
Input Modality 10-1
Input Modality 10-2
Input Modality 10-3
Data-to-Sequence Model(s) 11-1
Data-to-Sequence Model(s) 11-2
Data-to-Sequence Model(s) 11-3

FIG. 35

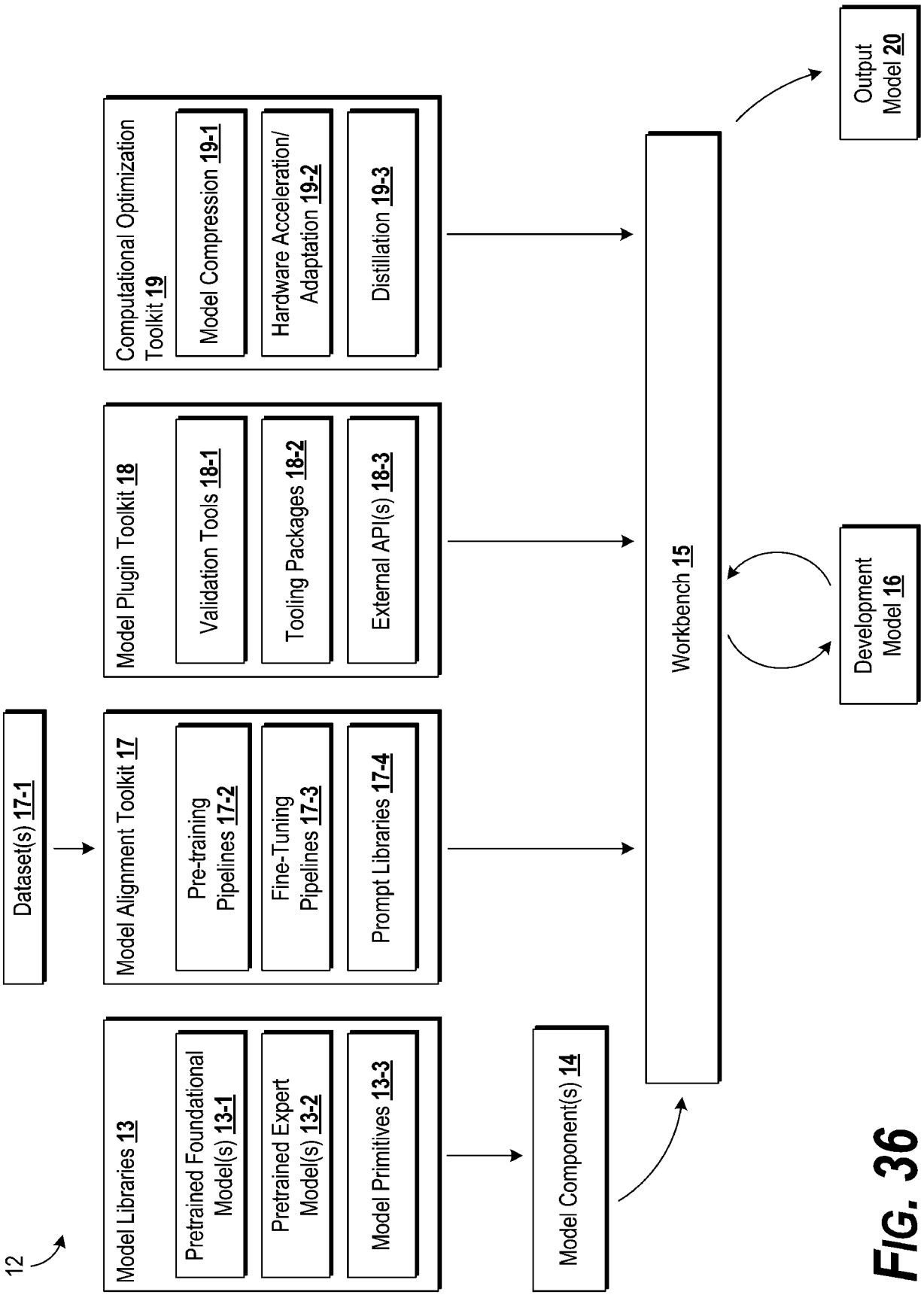

12
Model Libraries 13
Pretrained Foundational Model(s) 13-1
Pretrained Expert Model(s) 13-2
Model Primitives 13-3
Dataset(s) 17-1
Model Alignment Toolkit 17
Pre-training Pipelines 17-2
Fine-Tuning Pipelines 17-3
Prompt Libraries 17-4
Model Plugin Toolkit 18
Validation Tools 18-1
Tooling Packages 18-2
External API(s) 18-3
Computational Optimization Toolkit 19
Model Compression 19-1
Hardware Acceleration/ Adaptation 19-2
Distillation 19-3
Model Component(s) 14
Workbench 15
Development Model 16
Output Model 20

FIG. 36

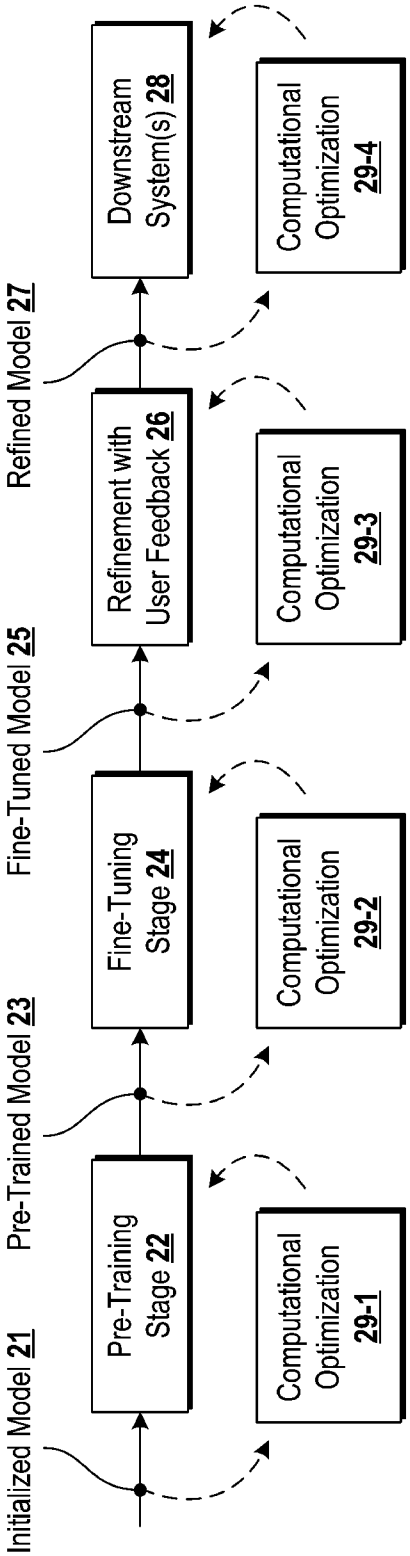

Initialized Model 21
Pre-Training Stage 22
Computational Optimization 29-1
Pre-Trained Model 23
Fine-Tuning Stage 24
Computational Optimization 29-2
Fine-Tuned Model 25
Refinement with User Feedback 26
Computational Optimization 29-3
Refined Model 27
Downstream System(s) 28
Computational Optimization 29-4

FIG. 37

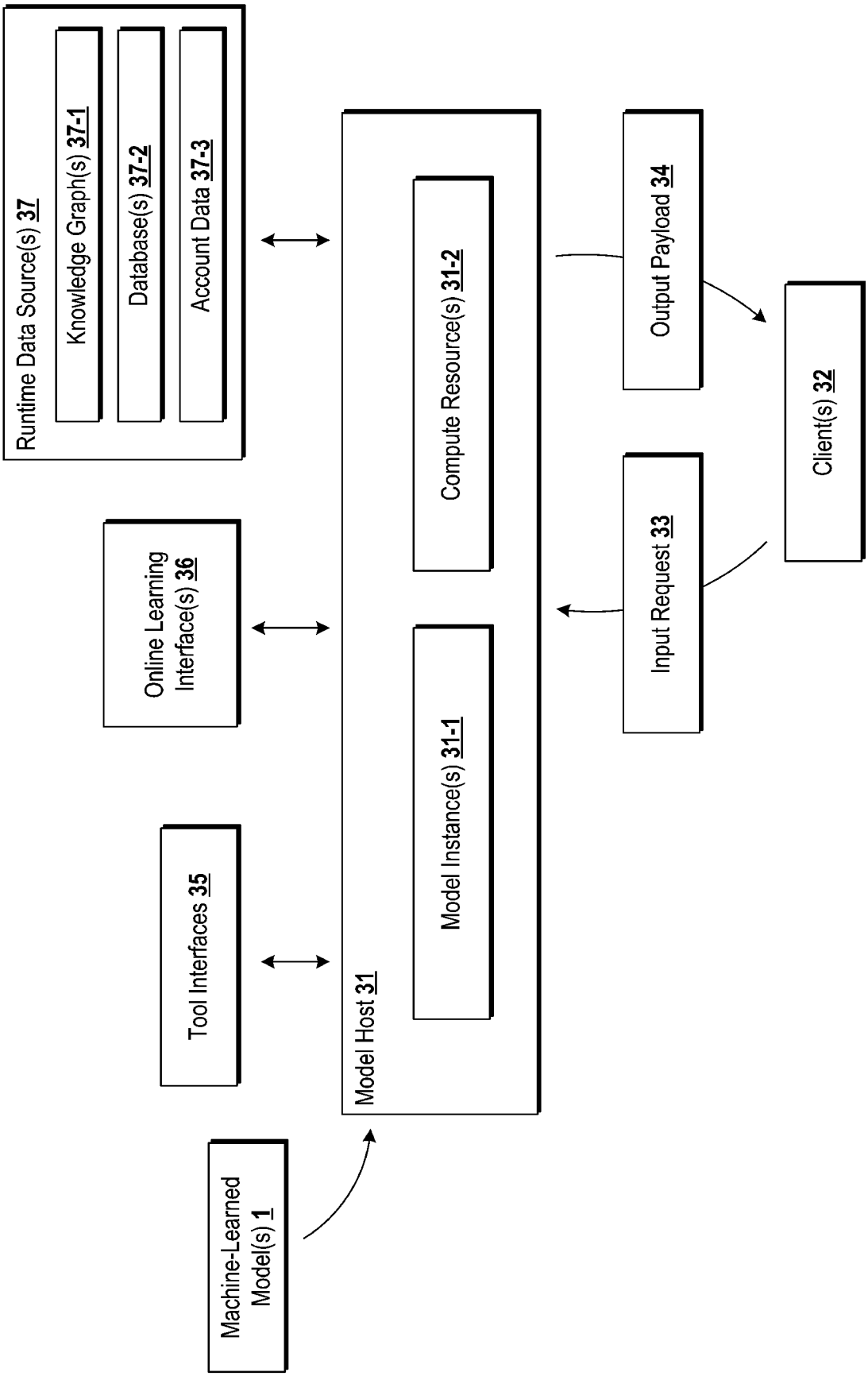

Runtime Data Source(s) 37
Knowledge Graph(s) 37-1
Database(s) 37-2
Account Data 37-3
Online Learning Interface(s) 36
Tool Interfaces 35
Machine-Learned Model(s) 1
Model Host 31
Model Instance(s) 31-1
Compute Resource(s) 31-2
Input Request 33
Output Payload 34
Client(s) 32

FIG. 38

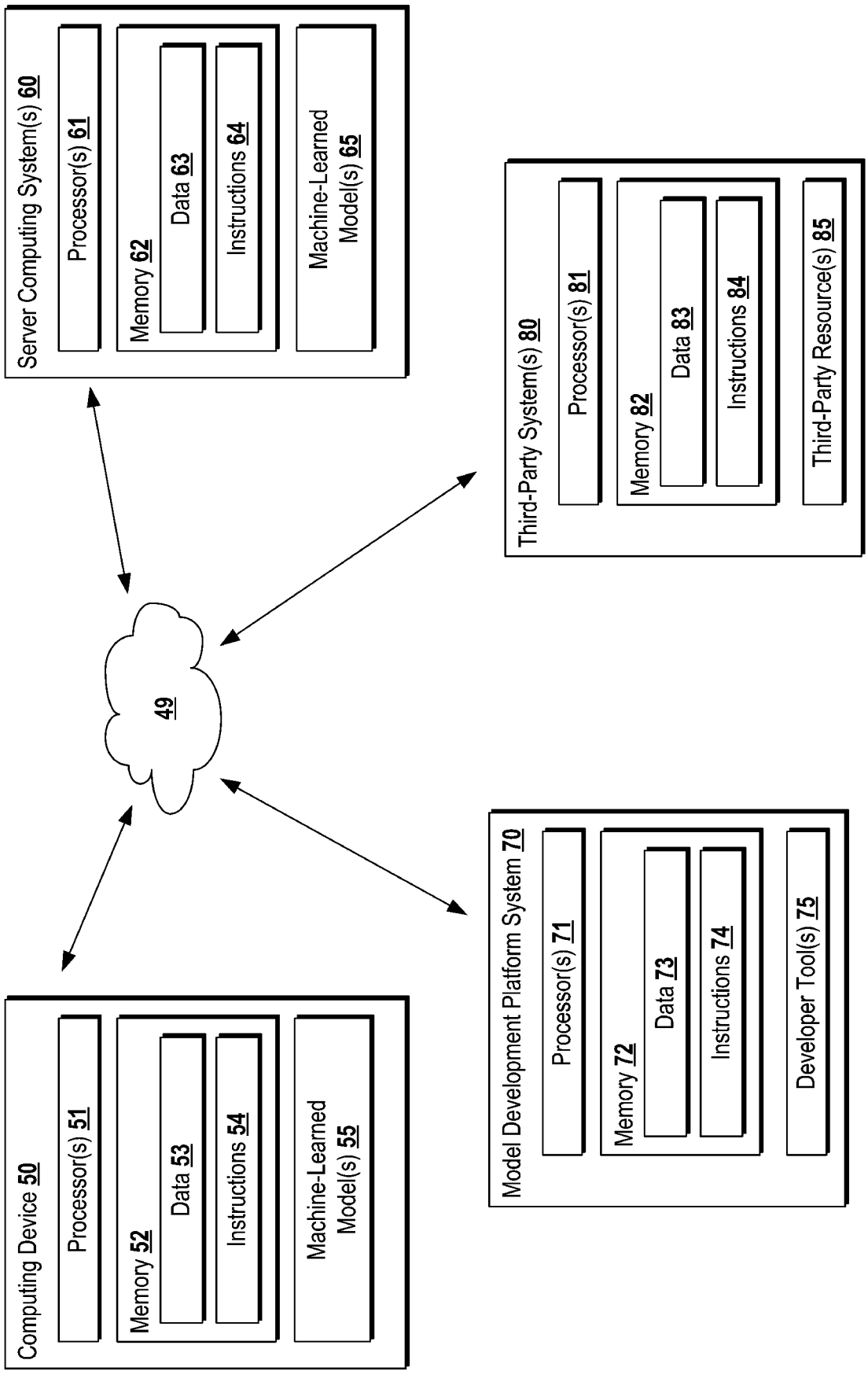

Server Computing System(s) 60
Processor(s) 61
Memory 62
Data 63
Instructions 64
Machine-Learned Model(s) 65
Third-Party System(s) 80
Processor(s) 81
Memory 82
Data 83
Instructions 84
Third-Party Resource(s) 85
49
Computing Device 50
Processor(s) 51
Memory 52
Data 53
Instructions 54
Machine-Learned Model(s) 55
Model Development Platform System 70
Processor(s) 71
Memory 72
Data 73
Instructions 74
Developer Tool(s) 75

FIG. 39

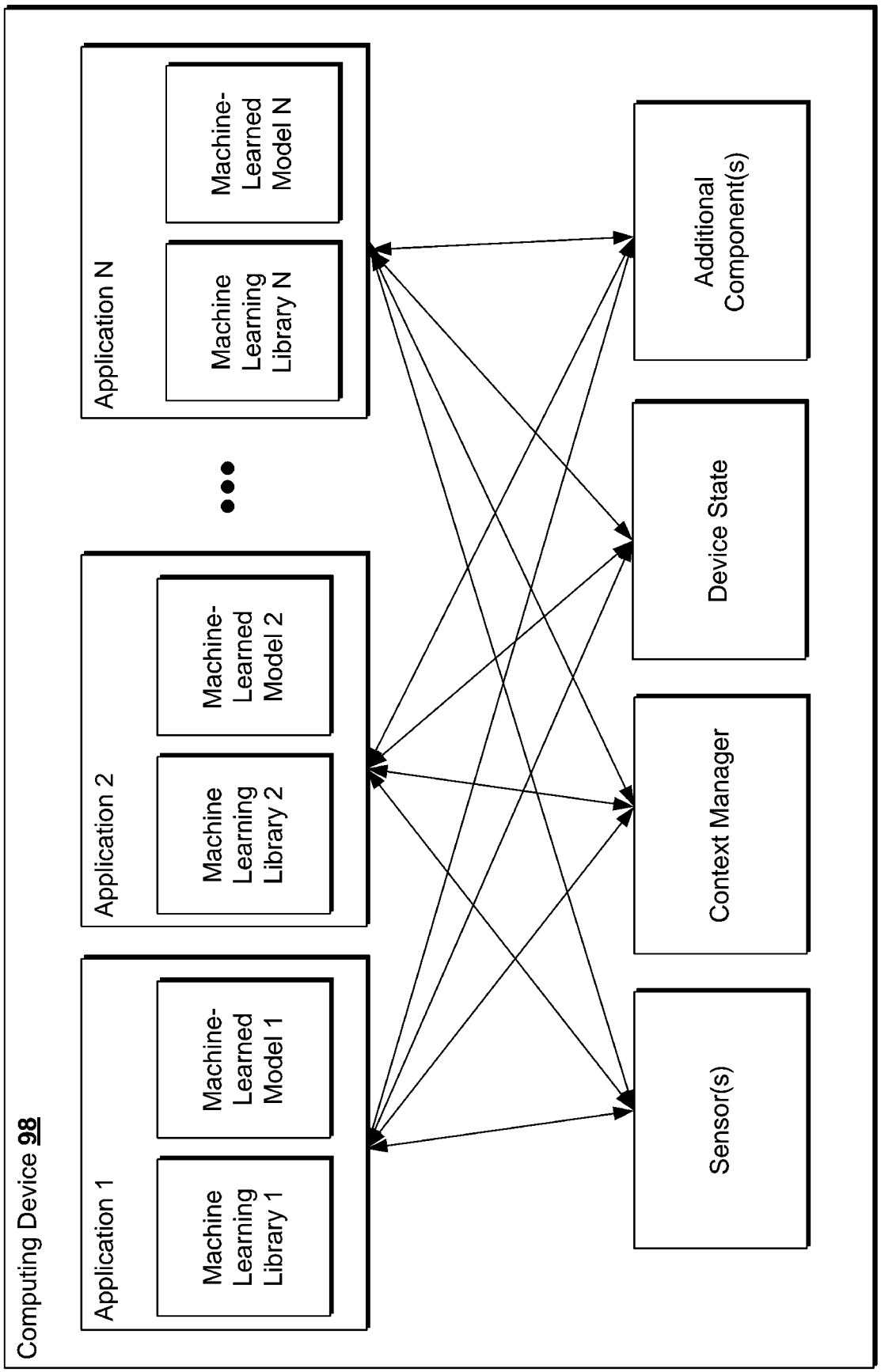

Computing Device 98
Application 1
Machine Learning Library 1
Machine-Learned Model 1
Application 2
Machine Learning Library 2
Machine-Learned Model 2
Application N
Machine Learning Library N
Machine-Learned Model N
Sensor(s)
Context Manager
Device State
Additional Component(s)

FIG. 40

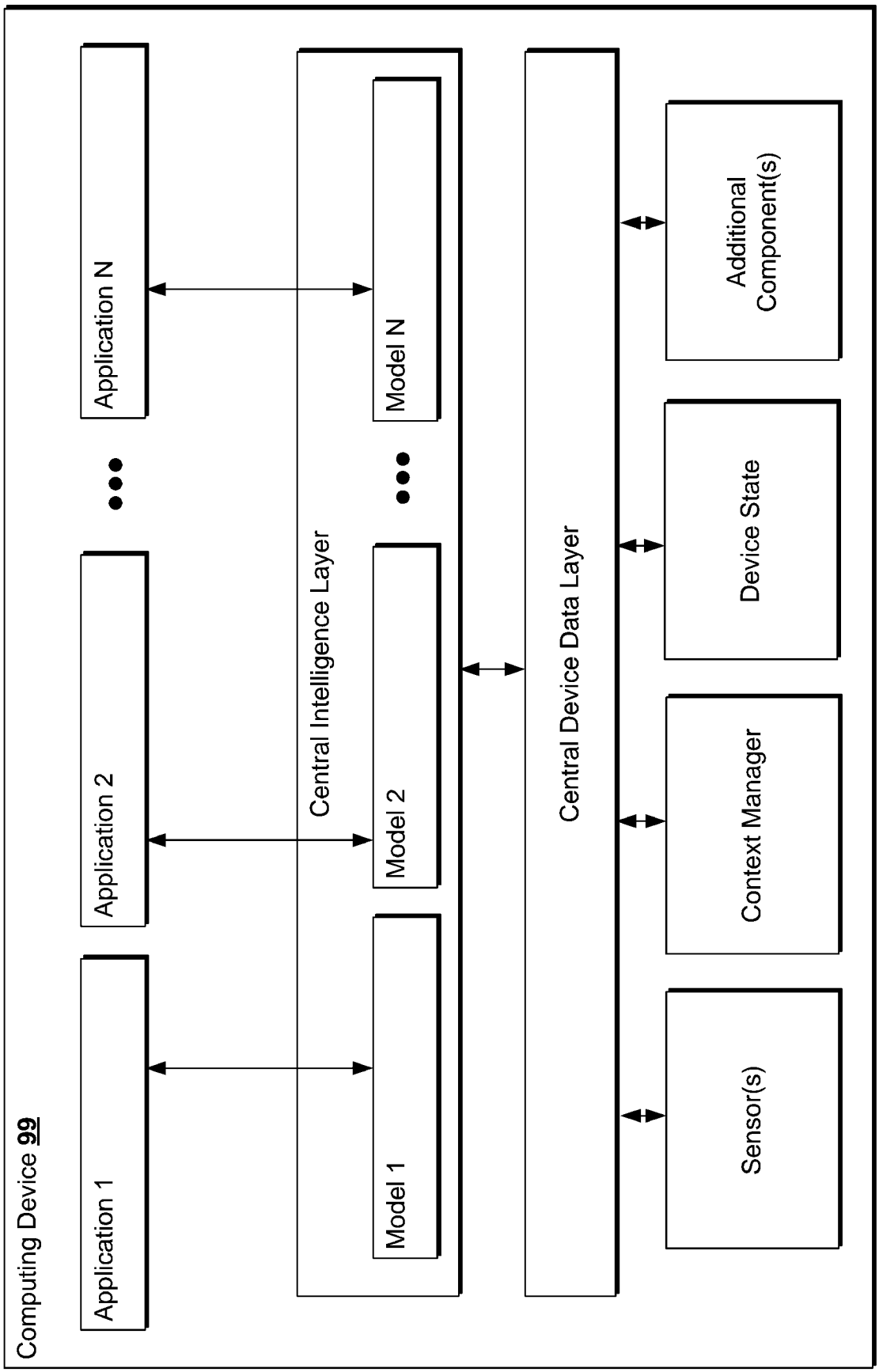

Computing Device 99
Application 1
Application 2
Application N
Central Intelligence Layer
Model 1
Model 2
Model N
Central Device Data Layer
Sensor(s)
Context Manager
Device State
Additional Component(s)

FIG. 41

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2024/053102

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/50 G06N3/063
ADD. G06F3/045

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F G06N

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Pope Reiner ET AL: "Efficiently Scaling Transformer Inference", arXiv.org, 1 November 2022 (2022-11-01), pages 1-18, XP093263089, Retrieved from the Internet: URL:https://arxiv.org/pdf/2211.05102 cited in the application the whole document -----	1-37
X	AMEY AGRAWAL ET AL: "SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills", ARXIV.ORG, CORNELL UNIVERSITY LIBRARY, 201 OLIN LIBRARY CORNELL UNIVERSITY ITHACA, NY 14853, 31 August 2023 (2023-08-31), XP091601783, the whole document -----	1-37

-/--

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance;; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance;; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
4 April 2025	16/04/2025

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Suarez Y Gonzalez, R

2

Form PCT/ISA/210 (second sheet) (April 2005)

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2024/053102

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 2021/177976 A1 (GOOGLE LLC [US]) 10 September 2021 (2021-09-10) page 5, lines 4-32 page 10, line 20 - page 13, line 4 page 18, line 8 - page 19, line 4; figure 3 -----	1-37

2

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2024/053102

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2021177976 A1	10-09-2021	CN 115244511 A	25-10-2022
		EP 4091051 A1	23-11-2022
		EP 4293570 A2	20-12-2023
		US 2023105476 A1	06-04-2023
		WO 2021177976 A1	10-09-2021