NeuStream: Bridging Deep Learning Serving and Stream Processing

Abstract

Modern Deep Neural Network (DNN) exhibits a pattern where multiple sub-models are executed, guided by control flows such as loops and switch/merge operations. This dynamic nature introduces complexities in batching the requests of such DNNs for efficient execution on GPUs. In this paper, we present NeuStream, a programming model and runtime system for serving deep learning workloads using stream processing. NeuStream decomposes the inference workflow into modules and forms them into a streaming processing system where a request flows through. Based on such abstraction, NeuStream is able to batch requests at fine-grained module granularity. To maximize serving goodput, NeuStream exploits a two-level scheduling approach to decide the best batching requests and resource allocation for each module while satisfying service level objectives (SLOs). Our evaluation of NeuStream on a set of modern DNNs like Large Language Models (LLM) and diffusion models, etc., shows that NeuStream significantly improves goodput compared to state-of-the-art DNN serving systems.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Real-time systems**.

Keywords: Deep Learning Serving, dynamic Neural Networks, Streaming Processing System

1 Introduction

The modern DNN applications often exhibit a dynamic execution pattern where each input might follow different execution paths guarded by control flows like loops or branches. For example, an LLM generates tokens iteratively through a decoding model, with different inputs requiring varying numbers of decoding steps to produce sentences of different lengths. Similarly, in a diffusion-based image generation model, a random input tensor may need to undergo a varying number of diffusion steps to generate high-quality images. Their dynamic execution patterns are inherently inefficient when run on GPUs. Typically, to fully utilize a GPU, a DNN model needs to process multiple requests in a batch to reuse the weight data. However, if each input sample follows different execution paths, it becomes difficult to batch the inputs together, necessitating sequential execution. This can easily cause the GPU to become bottlenecked by memory access, thereby reducing overall efficiency.

To address such inefficiency, we observe that the dynamic behavior in these applications is fundamentally different from traditional dynamic neural networks [27]. First, the bodies of these control flows are often much larger in granularity rather than consisting of just a few operators. For example, each decoding step in an LLM involves all the layers of a decoding model. Second, although different inputs follow different execution paths, they often repeatedly go through the same modules at different times. For example, different inputs to a diffusion model need to call the same UNet module multiple times. These patterns motivate us to rethink model serving from the perspective of these repeatedly executed control-flow "bodies": each body model could continuously process inputs and forward its outputs to downstream models based on control flow guidance until the full path is completed. This essentially forms a streaming-based computation flow, where each application is decomposed into modules (i.e., the control flow bodies) and connects these modules in a data-flow manner guided by control flow logic. Such an execution model could effectively address the inefficiency caused by the inability to batch many inputs together. Specifically, each module of the application could batch multiple inputs at a fine-grained granularity for execution and then dispatch their results to downstream modules.

However, executing a DNN model in such a streaming-based manner is non-trivial. First, existing DNN frameworks like PyTorch often represent a DNN model as a single Python application with control flows to manage all the different execution paths. It is challenging to decouple such a model into streaming modules and data-flow connections easily. Second, existing GPUs are designed for batch processing, which cannot efficiently support streaming execution, where multiple heterogeneous modules are continuously and concurrently running. Third, DNN serving often has strict service-level objective (SLO) requirements for latency. Decoupling an application into multiple modules and scheduling them independently makes it more challenging to optimize for the best overall goodput, i.e., the throughput of requests that satisfy SLOs, given that each module could have very different trade-offs between latency and batch size (§2).

In this paper, we present NeuStream, a deep learning serving system that aims to maximize GPU utilization by exploiting the ability of GPUs to batch multiple requests while meeting certain latency SLOs. NeuStream leverages the following key designs to address the aforementioned challenges: First, to program a DNN application in a streaming manner, NeuStream introduces first-class *stream* and *stream-module* interfaces to explicitly express a continuous data flow execution. The *stream-module* concept inherits from the existing DNN module concept in frameworks like

PyTorch, making it natural to adapt existing models. These interfaces implicitly require converting control flows in the native language into data flow connections, i.e., streams, which allows NeuStream to schedule the requests in each stream to optimize GPU utilization. Second, to execute all the heterogeneous stream-modules on GPUs, NeuStream abstracts the GPU resource into fine-grained computation units called *Streaming Processing Units (SPUs)*, where stream-modules can concurrently execute on different subsets of SPUs. Third, given the explicit stream-modules and fine-grained SPUs, NeuStream decomposes the global SLO of each request into partial SLOs for each individual module. NeuStream then decouples the whole scheduling space into two levels: an intra-module level scheduling decision to determine how many requests to execute in a batch for a specific module, and an inter-module level scheduling decision to determine how many resources to allocate across modules. Through such scheduling decoupling, NeuStream can easily scale out to multiple GPUs, i.e., through inter-module resource allocation, and maximize serving throughput, i.e., through intra-module batch execution.

We implemented NeuStream in 5,000 lines of Python code. NeuStream's executors for diffusion models and LLM models are built on top of PyTorch and vLLM [9], respectively. We evaluated NeuStream for serving mainstream DNN applications, such as LLM-based chat service, image generation, and multi-agent tasks, on various NVIDIA GPUs, including the GeForce RTX 4090, Hopper H100, and RTX A6000. Our evaluation demonstrates that NeuStream achieves significantly higher serving goodput compared to state-of-the-art baseline systems, e.g., outperforming Clockwork [7] by up to 5.26× for serving Diffusion models and vLLM by 1.53×- 69.31× for serving OPT models at various request rates, respectively. Looking forward, the rapid advancements in DNN model capabilities could enable more complex execution flows in future applications. We believe NeuStream provides a scalable, efficient, and hardware-friendly system design to accommodate these ever-increasing applications.

2 Motivation

Existing DNN serving systems face the following challenges due to treating the entire model as a monolithic component:

Challenge I: Dynamic execution paths. Most of the existing serving systems consider *static* DNN models whose execution paths are fixed, i.e., independent of the input. *Dynamic* DNN models, in contrast, can adjust their computational path with respect to input, and such models are now becoming prevalent. As illustrated in Figure 1, diffusion [18] models could work at different numbers of denoising steps, depending on preferences on generation quality and inference speed of each input. The LLM [34] generates output tokens autoregressively on a per-input basis, where each request may require a different number of decode iterations.

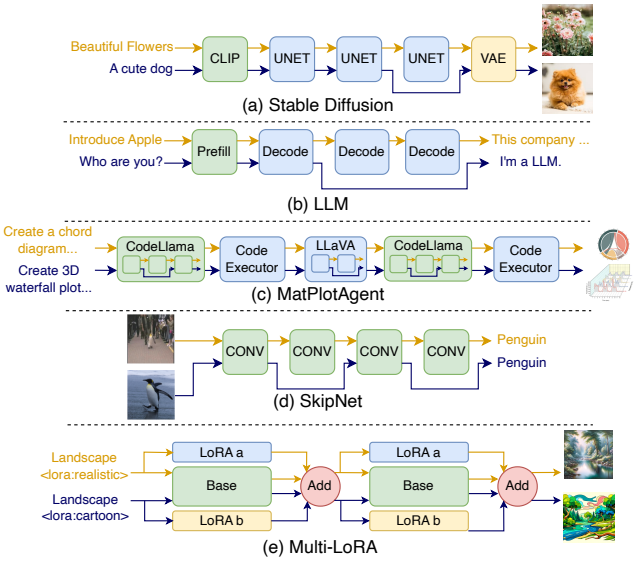

Figure 1. Examples of dynamic neural networks. They dynamically adjust computation paths on per-input basis.

The LLM-based multi-agents [28] further exacerbate this dynamics properties. Skipping and early stopping models such as Skipnet [26] dynamically decide whether to skip certain layers based on the input. Multiple Low-Rank Adaptation (LoRA [8]) requests share the same base model layer but enter into different LoRA adapter layers. Dynamic per-input execution paths in these models pose challenges for the optimizations that have been key to obtaining high GPU efficiency. For example, batching cannot be applied if the execution paths of inputs are not identical. Also, one cannot *proactively* perform scheduling actions (e.g., when or whether to execute a request), because dynamic execution paths have unpredictable latency performance.

Challenge II: Distinct functional phases. There are different main functional phases during a dynamic model inference. For example, stable diffusion inference involves text encoding, diffusion and decoding phases, and LLM inference contains prefilling and decoding phases. We find that different phases have distinct batching characteristics. Figure 2 and Figure 3 show the request execution latency on each phase, which is batch's latency / batch size (i.e., excluding queuing time). On a certain phase, as the batch size increases, the decreasing slope of a curve implies that batching leads to only a marginal gain in improving the execution latency (and thus the throughput). Once the gain becomes insignificant, adding more requests to the batch no longer improves the throughput. Instead, it proportionally extends the total processing time for the batch, inadvertently delaying all included requests. Thus, it is necessary to identify a critical threshold for batch size in each phase, beyond which the marginal gain becomes insignificant. Given that the slopes of different phases are distinct, we should identify the threshold

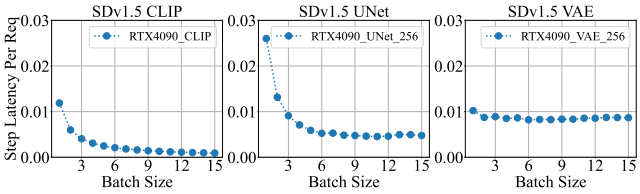

Figure 2. Diffusion phase execution latency on RTX 4090.

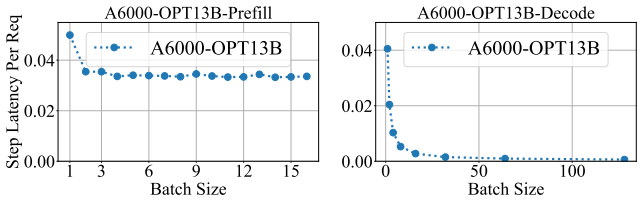

Figure 3. LLM phase execution latency on RTX A6000.

specific to each phase. Batching more requests on a phase should only be considered when the batch size of the scheduled request is below its phase-specific threshold.

Insight I: decompose a DNN model into modules. The above challenges motivate us to *decompose* a dynamic DNN into multiple functional modules (e.g., phases), and schedule executions at the granularity of modules. This enables to batch the arrived request on a specific module, irrespective of their dynamic execution paths. For example, diffusion model requests with different numbers of iterations could batch at the granularity of iteration by decomposing the UNet from the computation. Serving multiple LoRA[8] adapters enables batching in the shared base model by decomposing the computation between the base model and the LoRA adapters. Also, the decomposition enables to use different batching designs and better-provisioned device resources for each phase, considering their distinct batching property and SLOs.

Insight II: Integrating stream programming model. The above decomposed execution can be essentially mapped to the stream programming model, which adopts module-level programming that decomposes applications into a set of modules operating on data streams (i.e., channels). However, existing DNN frameworks adopt tensor-level programming that models a program as a dataflow graph, where the nodes are operators and the edges are the tensors. The misaligned programming model is the major obstacle to bringing the benefits of modularity to deep learning in existing frameworks. We therefore advocate combining both programming models to enable a new DNN framework NeuStream for efficient dynamic model inference.

3 NeuStream Abstraction

In this section, we introduce the stream programming and execution models of NeuStream.

```
class StreamModule:
  def __init__(s_in:List[stream],s_out:List[stream]):
  # gather message from mutliple input stream
  def gather()->List[message]:
  #kernel function
  def compute(List[message])->List[message]:
  #assign batches to out streams
  def scatter(List[message]):
```

Figure 4. The stream module abstraction.

3.1 Programming Model

The programmer writes a module-level program that manipulates named, first-class *streams* and stream *modules*: function units that take messages from and produce messages into streams. The computation logic within each stream module in turn, can be defined with a tensor-level program.

Stream module. A stream module provides the basic functionality for stream processing of deep learning inference requests. As shown in Figure 4, it is a schedulable entity consisting of the following blocks:

`stream`. A stream is a data channel that passes data between stream modules. Logically, a module can be regarded as a process in the concept of operating systems and a stream can be conceived as a pipe that connects different processes. Each `message=[header, data]` in the stream queue includes (1) a `header` containing identifying and routing information, and (2) `data` tensors. The header consists of a set of fields, which are set in the course of producing and delivering a message. Messages are ordered into `streams` according to priorities associated. Each stream module could contain (multiple) input streams `s_in` and (multiple) output streams `s_out`, specified in the `_init_` function.

`gather`. The `gather` function is used to get a *batch* from each input stream, which is a list of messages `List[message]`. If a module contains multiple input streams, the `gather` could `combine` the multiple messages of the same request from different input streams into one merging message using aggregation functions like sum, max, or concatenation.

`compute`. A stream module's computation is specified by a user-defined function `compute`, which takes *gathered* batches as input, performs processing, and generates the output batches. The user could directly define operations on `message.data` tensors in the function with tensor-level programming languages.

`scatter`. Given multiple output `streams`, the dynamism-related message routing operations (i.e., control flow) are decoupled from the `compute` of a module. `scatter` provides the programmer a unified function that supports customized rules to route messages to multiple output streams. The function takes the output batches of `compute`, and routes each message in every batch to one output stream based on routing information in the message head. This allows to express the control-flow of the multiple execution paths of dynamic neural networks in Figure 1. To support the iterative

```
class Stable-Diffusion:
  #stream module definitions
  class CLIP(StreamModule):
    def compute(m:List[message])->List[message]
      ... # processing code
  class UNet(StreamModule):
    def compute(m:List[message])->List[message]
      ... # processing code
    def scatter(m:List[message]):
      for msg in m:
        if msg.header.step > 0:
          self.s_out[0].push(msg)
        else:
          self.s_out[1].push(msg)
  class VAE(StreamModule):
    def compute(m:List[message])->List[message]
      ... # processing code
  #stream initialization
  stream text, latent, output, image;
  #define model pipeline via stream connection
  clip = CLIP([text], [latent])
  unet = UNet([latent], [latent, output])
  vae = VAE([output], [image])
```

Figure 5. Program for stable diffusion model inference.

```
class LLM:
  #stream module definitions
  class Prefill(StreamModule):
    def compute(m:List[message])->List[message]
      ... # processing code
  class Decode(StreamModule):
    def compute(m:List[message])->List[message]
      ... # processing code
    def scatter(m:List[message]):
      for msg in m:
        if msg.header.last_token != "EOS":
          self.s_out[0].push(msg)
        else:
          self.s_out[1].push(msg)
  #stream initialization
  stream prompt, prefix, output;
  #define model pipeline via stream connection
  prefill = Prefill([prompt], [prefix])
  decode = Decode([prefix],[prefix, output])
```

Figure 6. Program for LLM inference.

processes like diffusion model or LLMs, stream modules could use the same stream as both the output and the input stream. The messages routed to this output stream actually pass back to the input stream for the next iteration.

Stream program. Under our abstraction, a DL program is a collection of stream modules connected by streams. Each module consumes messages from its input streams and produces results on its output streams. The program consists of two blocks: the stream modules definitions and stream modules interconnects. Figure 5 illustrates the program of the Stable Diffusion. It first uses stream modules to define three phases: text encoder (CLIP), backbone (U-Net), and image decoder (VAE). Next, the program initializes streams and defines the model inference pipeline via module stream connection. The `CLIP` module transforms the input prompt into a *latent stream* including text embedding and initial noisy latent, which is connected to the U-Net. The `UNet`

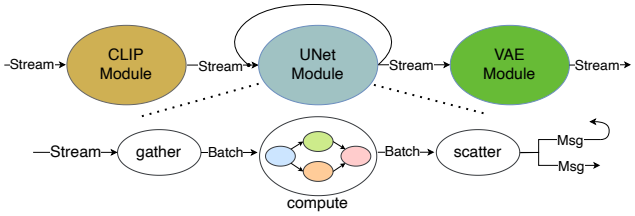

Figure 7. Module graph for Stable Diffusion.

module iteratively denoises the latent image messages in the *latent stream*, and outputs the denoised latent messages. We decrease the iteration number of latent messages for each U-Net execution, and route denoised latent messages with the positive number of iterations back to *latent stream* for UNet to execute the next iteration, and leave others to the *output stream* connecting to `VAE`. The `VAE` decoder transforms the latent messages back into images. Figure 6 illustrates the program of the LLMs, which contains two stream modules. The prefill module processes a user's prompt to generate the first token (including KV cache) of the response into *prefix stream*. Following it, the decoding module sequentially generates subsequent tokens (and updates KV cache) in multiple steps. In particular, after running one decode step, the termination condition will be checked: if a special stop token is met, we route the token to *output stream*, otherwise, each generated token is appended and fed back into *prefix stream* for the decode module to generate the next token.

3.2 Execution Model

The execution of a stream program can be modeled as a *stream graph*: a hierarchical (two-level) computation graph of modules communicating via data streams, in which each module is further represented as a dataflow of lower-level operations on messages. Figure 7 illustrates the stream graph corresponding to Figure 5. Each module performs local computation on input streams from its in-edges and produces output streams as its out-edges. The stream graph abstraction supports the `gather-compute-scatter` operations defined in stream modules. A module can maintain a local state s. Its execution starts with its initial state s_0 and proceeds in steps. In each step, the module consumes the next batch from its input streams by calling `gather`, updates its state and produces an output batch by executing a series of operations of `compute`, and routes output messages into downstream modules by `scatter` according to the stream connection.

Different from the dataflow of existing DL frameworks, the stream graph has two major features. First, instead of mixing control-flow and dataflow together, the stream graph decouples the module communication (control flow) and computation (dataflow) into high-level module graph and low-level operation graph, respectively. This enables efficient batching at the granularity of modules, given that data

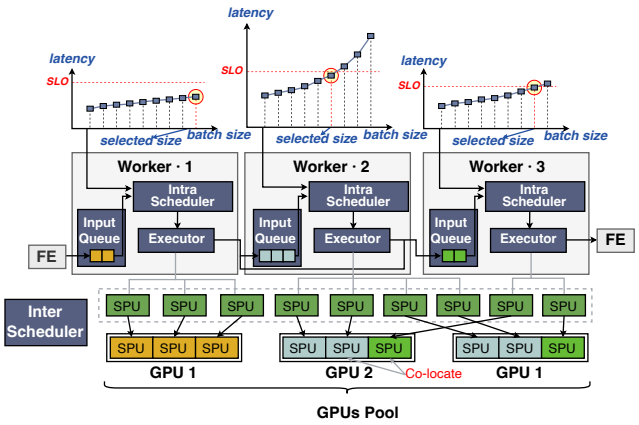

Figure 8. NeuStream's two-level scheduling framework.

in a single stream have the same static operation graph defined in `compute`. Second, modules are connected via streams with the continuous flow of data, instead of static tensors consuming exactly once. This supports a decomposed and continuous stream computation model, extended with the ability to customize execution at each module (e.g.,batching and resource allocation).

4 NeuStream Scheduling

The goal for NeuStream is to maximize the system goodput, i.e., the number of served requests that meet their SLO requirements per unit time. The hierarchical streaming graph naturally leads to a two-level scheduling framework: *intra*-module scheduling on how many requests to include in a batch for a specific module, and *inter*-module scheduling on how much resource to be allocated across modules. Figure 8 illustrates the architecture of NeuStream, which is composed of a frontend and multiple stream workers. Each worker is responsible for executing a single module. The frontend continuously sends incoming requests to the first stream worker. Upon receiving request messages, the executor in a worker performs the `gather-compute-scatter` execution model, sending new messages to downstream workers based on stream connection. The batching decision is made by the intra-module scheduler at each worker, utilizing partial SLO and profiled latency function of each module. The resource allocation decision is made by the inter-module scheduler, which aims to balance the maximum goodput permitted by each module under an appropriate resource allocation.

4.1 Intra-module Scheduling

Partial SLOs. In the presence of control flow, it is unclear how to define end-to-end latency SLOs for individual requests with dynamic and even unpredictable execution paths. Since NeuStream decouples routing control-flow from execution, we take the approach of assigning *Partial* SLOs to individual modules, which is similar to the partial SLO concept in microservices [13, 25]. Specifically, given the module

Algorithm 1: Intra-module scheduler for batching on a module m_i

Input: Latency function $L_i(b, a_i)$ of batch size b and resource allocation a_i, partial SLO_i, input stream in
Output: batch_request

```
1  Function IntraScheduler(L_i, SLO_i, in):
2      batch_request ← [ ];
       // get maximum batch size based on partial SLO
3      batch_limit ← argmax_b L_i(b, a_i) ≤ SLO_i;
4      for request r ∈ in do
5          if r.budget≤ 0 then
               // drop request that fail to meet upstream
                  partial SLOs
6              in.remove(r);
7              continue;
8          end
9          if length(batch_request) < batch_limit then
10             batch_request.append(r);
11         end
12     end
13     return batch_request;
```

execution path $\varphi = \{m_1, \ldots, m_k\}$ of a certain request, we consider its end-to-end latency $SLO = \sum_{i \in \{1,\ldots,|\varphi|\}} SLO_{\varphi(i)}$, where $SLO_{\varphi(i)}$ is the partial SLO for module $\varphi(i)$.

Batch scheduling algorithm. As long as all modules meet their respective partial SLOs, the request SLO is guaranteed. This approach ensures independent batch scheduling across individual modules. Algorithm 1 describes the procedure of intra-module batch scheduling for one execution step on a specific module m_i. The algorithm receives the following inputs for m_i: (1) $L_i(b, a)$: profiled execution latency of batch size b and resource allocation a (detailed in Section 4.2), which includes the duration of executing `compute` and `scatter`, (2) SLO_i: partial SLO (in latency), and (3) in: the input stream. For every step, if the stream in is non-empty, the scheduler determines a batch: we first obtain the maximum permitted batch size $batch\_limit_i = argmax_b L_i(b, a) \leq SLO_i$ without violating the partial-SLO of this module (line 3). Then for each request in the input stream, we add a request into the batch if it satisfies:

C1. The request meets the partial SLOs on the upstream modules. In particular, let $\varphi_r = \{m_1, \ldots, m_k\}$ be the execution path of previous steps of a request r before executing the module m_i at the current step, and t and t_0 be the current time and request issued time, respectively. We require the request budget $B_r = \sum_{j \in \{1,\ldots,|\varphi_r|\}} SLO_{\varphi_r(j)} - (t - t_0) \geq 0$. In line with other SLO-aware scheduling approaches [7, 10, 32] that drop requests unlikely to meet their SLOs, we removes requests with negative budget from the stream to avoid continuously blocking the subsequent incoming requests (line 5-8). However, in more sophisticated systems, these removed

requests may not be dropped entirely; instead, they can be placed in a low-priority queue or escalated to an upper-level scheduler for re-dispatch.

C2. The current batch size is smaller than the maximum permitted batch size *batch_limit* (line 9-11).

The above two conditions together achieve a large goodput at a specific module m_i by ensuring that the requests in the batch can still meet partial SLOs after the execution of this module, and the GPU operates at high efficiency. In the stream of each module, we sort the request messages based on their arrival time at the frontend. One could also prioritize requests with tight SLOs based on the request budget.

4.2 Inter-module Scheduling

Streaming Processing Units (SPUs). To facilitate resource allocation, we introduce an abstraction of SPUs, which can be considered as a uniform partition of physical devices, as illustrated in Figure 8. Given a total of N SPUs and M modules, the inter-scheduler determines the number of SPUs a_i for each module m_i. Let k_i be the average number of execution times of a specific module m_i in the execution path. Given the batch scheduling of Algorithm 2 and resource allocation a_i, we define the normalized goodput $g_i(a_i, SLO_i)$ of m_i as:

$$g_i(a_i, SLO_i) = \frac{batch\_limit_i}{SLO_i \times k_i} = \frac{argmax_b L_i(b, a_i) \le SLO_i}{SLO_i \times k_i} \tag{1}$$

The goal of inter-scheduler is to optimize the overall pipeline by maximizing the minimum module normalized goodput:

$$\begin{aligned} &\max \quad \min_{i \in \{1,\dots,M\}} \; g_i(a_i, SLO_i) \\ &\text{s.t.} \sum_i a_i \le N \;\forall i \in 1, \dots, M \\ &\qquad a_i mem \ge Mem_i \;\forall i \in 1, \dots, M \;\; (Memory\ limit) \end{aligned} \tag{2}$$

where mem is the memory capacity of a SPU and Mem_i is the memory footprint of module m_i.

Allocation algorithm. Algorithm 2 presents pseudocode for our iterative SPU allocation process. The algorithm first initializes all modules by allocating the minimum number of SPUs a_i that satisfies the memory limit condition for each module m_i (line 2). This step ensures that each module can be put on the SPUs under the memory constraint. Then, the algorithm finds the modules that yield the minimum normalized goodput, and allocates $\Delta(\Delta \ge 1)$ SPU to that module (line 12-14). To this end, for each module, the algorithm maps its allocated SPUs onto physical devices (e.g., GPUs) through module placement detailed below (line 4-9), and gets the profiled latency function $L_i(b, a_i)$ of the module (line 10). Based on the latency functions and equation (1), the algorithm computes normalized goodput g_i of each module m_i. The algorithm continues allocating Δ SPUs at a time until all the SPUs are allocated.

Algorithm 2: Inter-moduler scheduler for resource allocation across modules

Input: stream graph G, function partial *SLO* for each module, total number of SPUs N, devices D
Output: [SPU_allot, SPU_map]

```
1  Function InterScheduler(G, SLO, N):
       // assign initial SPUs satisfying memory limit
2      SPU_allot ← Initialize(G, SLO);
3      while N ≥ Δ do
           // map SPU allotment to physical devices
4          sort(SPU_allot);
5          for i ∈ {1, ..., M} do
6              SPU_map[i] = best_fit(SPU_allot[i], D);
7              if SPU_map[i] = ∅ then
8                  SPU_map[i] = partition(SPU_allot[i], D);
9              end
10             L_i(b, a_i) = get_profile_latency( SPU_map[i], D);
11         end
           // assign SPUs to bottleneck module
12         m = argmin_{i∈{1,...,M}} g_i(SPU_allot[i], SLO_i);
13         SPU_allot(m).increase(Δ);
14         N = N − Δ;
15     end
16     return [SPU_allot, SPU_map];
```

Module placement. When mapping SPUs to physical devices (line 4-9 in Algorithm 2), the scheduler considers modules in descending order of SPU allocation, maps SPUs of a module onto a device that has the least available SPUs to accommodate the module (line 5). If no single device has sufficient available resources to accommodate a module, we uniformly map SPUs of a module onto the fewest possible devices to reduce communication costs (line 7-9). In this case, we rely on model-parallel partition tools to partition a module (operation graph) across multiple devices. We can enumerate possible combinations of data parallelism and model parallelism, and use the profiled latency function to identify the configuration that yields the best module goodput, i.e., max $argmax_b L_i(b, a_i) \le SLO_i$.

Colocating modules. The scheduler allows the two modules to be annotated as *co-located*, and maps the SPUs of two co-located modules on the same set of devices to reduce the communication and increase the data locality. Given a workload, NeuStream could automatically search for the modules to be co-located: For a pair of originally non-collocated modules (m_i, m_j), the scheduler re-maps SPUs by annotating them as *co-located*, and treats the annotation valid only if co-locating m_i and m_j could improve their overall normalized goodput, i.e., $\min\{g_i, g_j\}$, based on the profiled latency functions and equation (1).

5 Implementation

NeuStream was implemented with 5K lines of Python code.

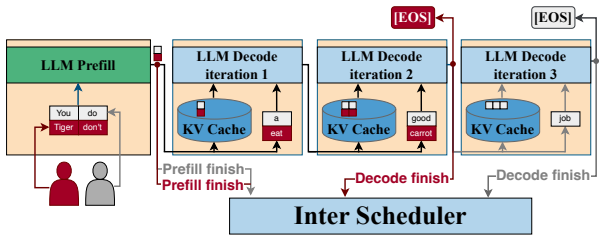

Figure 9. State management for LLM.

Module execution. We built the executor of diffusion models and LLM models on top of PyTorch [2] and vLLM[9], respectively. For diffusion models, we directly use PyTorch for sub-module parallelism across multiple cores on a single GPU. For larger LLMs, we use vLLM to provide sub-module parallelism across multiple GPUs. Each module is executed in its own process by default, however, co-located modules can be executed in the same process to optimize module switching and communication overhead (e.g., only passing pointers to the output data that's already on the device). The module invocation number depends on the workload, which can be determined statically before execution in diffusion or dynamically during execution in LLMs with the EOS token being detected. To compute the normalized goodput, we could use the average invocation number over the requests that have already been served.

State management. Each module can optionally manage the state for each request that can be used across different messages. The state of a module is a set of key/value pairs, which can be stored in a key-value store and operated by a module executor via `get`, `put`, and `delete` operations. NeuStream's inter-module scheduler tracks the *progress* of a specific request r and notifies a module to evict its state once the module completes the processing of r. In cases where a module is repeatedly invoked within a loop in the stream graph, the module is only considered finished when r exits the loop. Figure 9 illustrates how NeuStream handles state in LLMs. The decode module is stateful by treating the KVCache as the state of a request and updating the state (KVCache) at each iteration by inserting the key and the value of the new token. To reduce memory fragmentation, following PagedAttention [9], NeuStream partitions the KVCache into blocks along the sequence length dimension, with each block containing KVCache for multiple tokens. These blocks are represented as keys in the key-value store. At each decoding step, the decode module retrieves the addresses of these blocks, performs attention directly on the distributed blocks, and appends the KVCache of the newly generated token to the last block. Once the inter-module scheduler tracks that the decode module (loop) has completed processing for a certain request (e.g., encountering an EOS token), it notifies the decode module's executor to evict the state of that request.

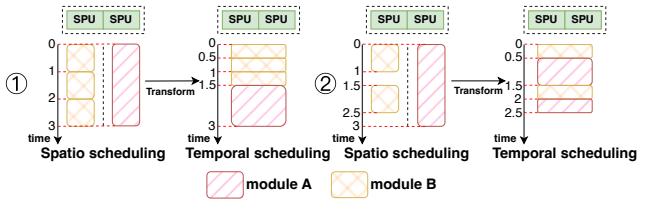

Figure 10. Spatial-to-temporal scheduling transformation with EFTF algorithm in a single device.

Streaming Processing Units. We could leverage spatial partitioning such as Multi-process-service (MPS[15]) to split a GPU resource into multiple SPUs. However, it cannot guarantee performance isolation and often results in unpredictable throughput, and also cannot adapt to workload variations. Thus, we transform the spatial scheduling into an equivalent or even better temporal scheduling by leveraging the predictability in the latency of a single module. Specifically, we adopt an Earliest Finish Time First (EFTF) algorithm, if modules are executed by the SPUs on the same device, we order the batches of different modules in increasing order of finish time in *spatial* scheduling, and schedule them in this order with transformed *temporal* scheduling. Figure 10 gives an illustrative example. In spatial scheduling, two SPUs are located in the same device, and each is responsible for a module. By contrast, as a module takes up the two SPUs in temporal scheduling, the execution time of a single module is halved. In the first case, the batch of module A arrives continuously, and we insert three module A's batches before module B's batch in temporal scheduling based on their finish time in original spatial scheduling. In the second case, the second module A's batch arrives after the scheduling of module B's batch, but with an earlier finish time in original spatial scheduling. So we allow module A's batch to be executed concurrently with module B's using a new CUDA stream with high priority. We see that each batch's finish time is no later than that in spatial scheduling.

Stream. We borrow a similar concept of data streaming from traditional streaming systems [3, 24] and additionally perform specific optimizations for DNNs: the data flit in a DNN typically consists of large tensors, so we pass only data references in the stream to avoid heavy data copying. To this end, only the metadata is put into the stream to facilitate scheduling, while the actual tensors are fetched as needed.

Profiling. Given a total of K SPUs on a device, if a module is executed on n devices with a_i SPUs, we first profile the latency function of $L_i(b)$ by taking the entire n devices, and estimates the latency function $L_i(b, a_i) = \frac{L_i(b)nK}{a_i}$ for the `get_profile_latency` interface.

Name	Type	Task	Year
StableDiffusion [18]	Diffusion	Text to Image	2022
Palette [21]	Diffusion	Image Restoration	2022
DiT_XL/2 [17]	Diffusion	Image Generation	2023
DiT_S/2	Diffusion	Image Generation	2023
OPT-6.7B [34]	LLM	Text Generation	2022
OPT-13B	LLM	Text Generation	2022
OPT-30B	LLM	Text Generation	2022
OPT-66B	LLM	Text Generation	2022
MatPlotAgent [28]	Multi-agent	Multi-agent Assistant	2024

Table 1. Experiments setting.

6 Evaluation

6.1 Experiment Setup

Models. We evaluate NeuStream on three types of modern model families listed in Table 1. For diffusion, we select widely used models for different tasks, which include text-to-image (Stable Diffusion [18]), image restoration (Palette [21]), and class-conditional image generation (DiT [17]). For LLM model, we select the OPT [34]. We choose several model sizes to test performance under different numbers of devices. For multi-agent, we use MatPlotAgent [28] designed to automate scientific data visualization tasks. The model is comprised of three LLMs: a Code Llama [20] to generate preliminary code, LLaVA [12] to generate image-text question/answer, and the second Code Llama to refine the initial code.

Cluster testbed. We evaluate NeuStream using three types of servers equipped with different accelerators: The first one is a local workstation with Intel Xeon Gold 6230R CPUs and NVIDIA GeForce RTX 4090 (24GB) GPUs, running on Ubuntu 20.04 with CUDA 11.8. The second one is a cloud server with Intel Xeon Platinum 8480C CPUs, 8*NVIDIA Hopper H100 (80GB) GPUs with NVLink enabled, running on Ubuntu 22.04 with CUDA 12.0. The third one is a cloud server with AMD EPYC 7513 and 4 RTX A6000 (48GB) GPUs, running on Ubuntu 18.04 with cuda 12.2.

Offered load. Similar to prior work [10], we generate the request arrival times using Gamma Process, which is parameterized by rate and coefficient of variance (CV). By scaling the rate and CV, we can control the workload's rate and burstiness, respectively. For diffusion models, we use two types of image sizes (256×256, 512×512) and use a uniform iteration distribution over the recommended range, e.g., $[30 - 50]$ for Stable Diffusion [1]. For LLM models, we use LMSYS-Chat-1M [35], a large-scale dataset containing real-world conversations with 25 state-of-the-art LLMs. For multi-agent, we use MatPlotBench [28] consisting of complex data visualization problems.

Metrics. We use *goodput* to measure the performance of the systems, which is the number of served requests that meet their SLO requirements per unit of time. We set partial SLO of each module based on its inference time on a specific hardware. Diffusion model serving focuses on request-level SLO which is the sum of partial SLOs, whereas LLM serving focuses on token-level partial SLOs on the time to first token (TTFT) and average time per output token (TPOT) for prefill and decode modules, respectively. Each experiment was repeated 5 times.

Baselines. We choose the most representative baselines with a focus on batching: for Diffusion with predictable execution times, Clockwork serves as a strong baseline using a proactive scheduling approach. Clockwork was originally implemented with a legacy TVM backend that has limited operator coverage. For a fair comparison of the scheduling design, similar to prior work [10, 33], we re-implemented Clockwork using the latest PyTorch backend. For LLMs, we select vLLM [9] as a strong baseline, which supports iteration-level scheduling [29] and PagedAttention [9]. We adopt the latest vLLM version (v0.5.4) released in August 2024.

6.2 Results on Diffusion Models

For Diffusion models that can fit into a GPU, we focus on a single GPU placement.

Goodput under varying offered load. Figure 11 shows the results of serving diffusion models with 256x256 image size on RTX 4090 GPU. In each row, we change one parameter to see how workload variation affects the performance of each serving system. Figure 11's first row shows the goodput with varying rates. NeuStream provides significantly higher goodput due to batching more requests on the module level (e.g., up to 5.26× at 4 requests/s in the DiT-S/2 model). Figure 11's 2nd row varies the CV of the workloads. The traffic becomes more bursty with a higher CV, which increases the possibility of SLO violation, resulting in quick goodput reduction in Clockwork. By contrast, the more bursty increases the batching opportunities in NeuStream, which alleviates the negative effects of high busty requests (outperforming Clockwork by $1.37 \sim 4.04$× across models at a high CV of 4). Figure 11's 3rd row shows the effect of different SLO. SLO scale equals 1 means the latency requirement is the single request inference latency. As SLO scale increases, the latency requirement becomes looser. We see that even SLO is only slightly larger than the model inference time, NeuStream gets a much higher throughput due to module-level batching (up to 3.13× at a small SLO scale of 1.2).

The computational cost of StableDiffusion depends on the input image size. For large 512×512 images, as shown in Figure 12, it almost saturates the RTX GPU even with a single request, limiting the batching gain of NeuStream. However, we observe that NeuStream can effectively scale up to a more powerful H100 GPU (up to 1.90× as compared to Clockwork across offered loads).

Batching efficiency. To fully understand NeuStream's benefits on batching, we plot the running batch size on DiT module for DiT-S/2 along the time in Figure 13(a). On average,

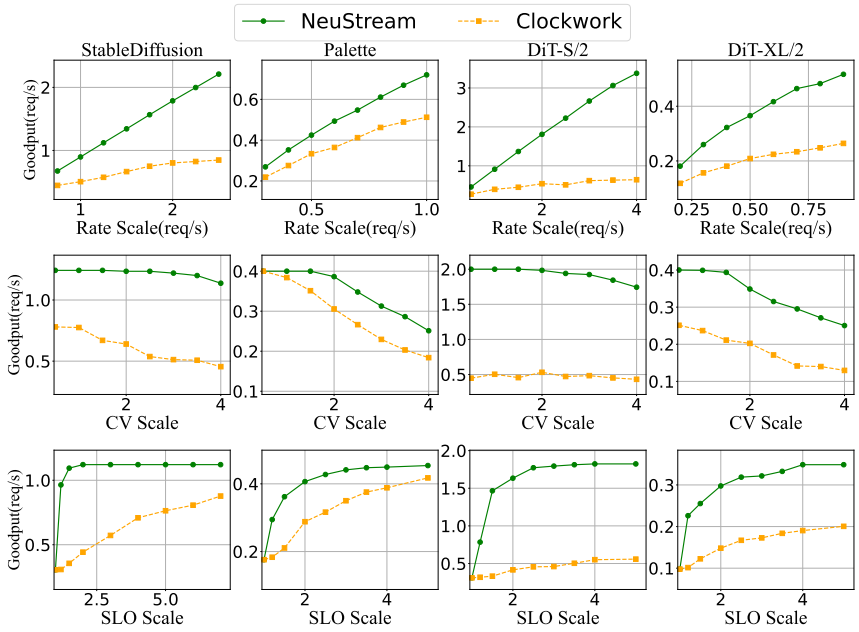

Figure 11. Experiments of generating 256*256 images on NVIDIA RTX4090.

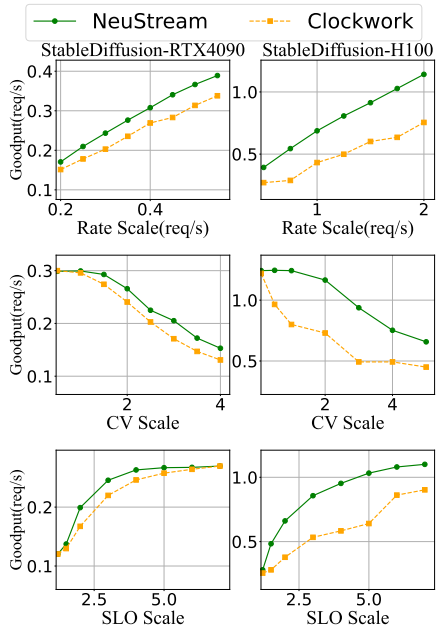

Figure 12. Generating 512*512 images on RTX4090 and H100, respectively.

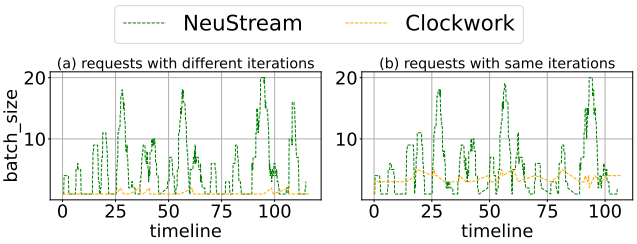

Figure 13. Batch size of DiT module along the time, where requests start with different/same number of iterations.

the batch size of NeuStream is 5.35× larger than Clockwork due to efficiently batching requests that (1) arrive at different times; (2) require different number of iterations to finish; or (3) start with different number of iterations. Figure 13(b) increases the batching opportunity of Clockwork by setting the same number of iterations for each request, i.e., disabling the above case (3). We see that NeuStream still has a larger batch size (1.67× on average larger).

6.3 Results on LLM Models

For LLMs, our evaluation covers multi-resource scenarios (e.g., OPT-30B/-66B across 2/4 GPUs). We focus on the goodput of decode module that has a much larger impact on the end-to-end latency.

Goodput under varying offered load. Figure 14 shows the results of serving OPT models on RTX A6000 GPUs. The 1st row shows that, as the request load increases, vLLM goodput starts to decrease significantly. This is because vLLM greedily prioritizes prefill requests and the use of preemption leads to resource starvation of decode module. In contrast, NeuStream is SLO-aware and balances the resource provision across the modules. For OPT-13B, NeuStream outperforms vLLM by 1.53× at 2 requests/s. Under an even higher rate of 4 requests/s, NeuStream achieves 37.21× more goodput than vLLM, due to decode resource starvation and prefill congestion (caused by insufficient memory for KV cache) in vLLM. As shown in Figure 14's 2nd and 3rd rows, NeuStream performs better than vLLM to handle more burstiness (a higher CV) and stringent SLOs (a smaller SLO scale) that aggravates the resource contention, e.g., 2.25× at the CV of 4 and 7.98× at SLO scale of 1.2 on OPT-13B. Figure 14's 2nd and 3rd columns show NeuStream consistently outperforms vLLM when scaling the serving to accommodate larger models on multiple devices (e.g., NeuStream achieves up to 69.31× at 3 requests/s when serving the largest OPT-66B model on 4 devices).

Figure 15 further shows the result of serving OPT models on more powerful NVIDIA H100 GPUs, where each system is able to handle a relatively higher offered load as compared to that in A6000 GPUs. We see that NeuStream can still achieve significantly more goodput as compared to vLLM on various traffic patterns and SLOs (e.g., up to 11.44× at 8 requests/s when serving the largest OPT-66B model on 4 H100 GPUs).

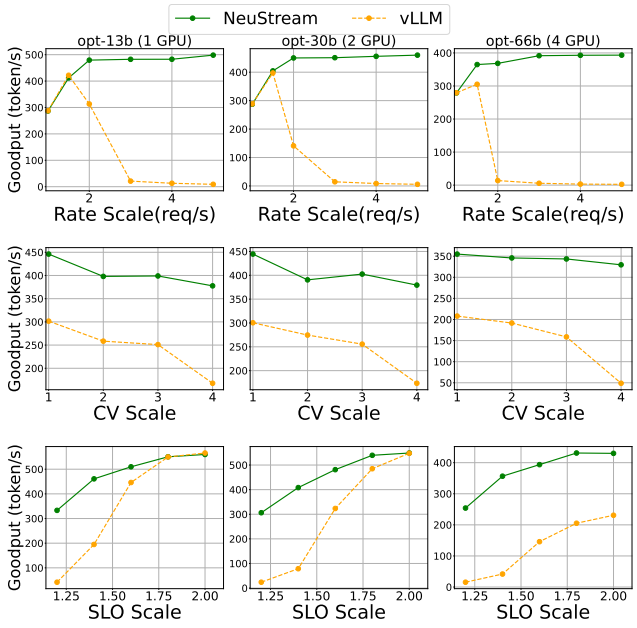

Figure 14. Goodput of LLM decode module on A6000.

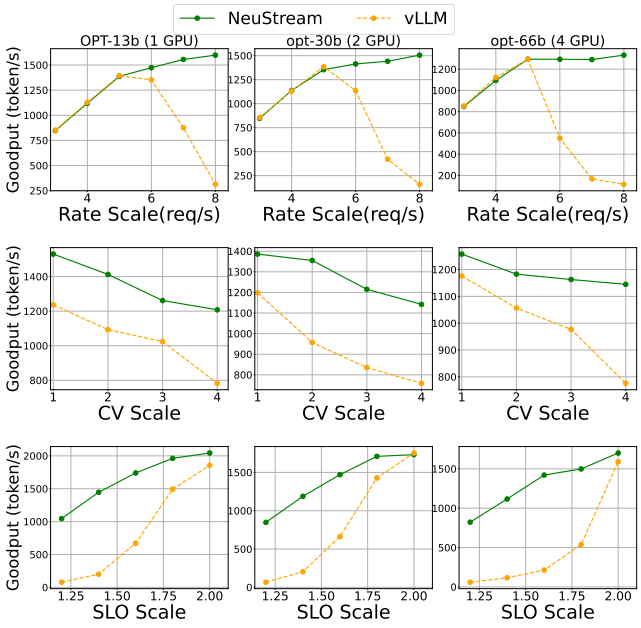

Figure 15. Goodput of LLM decode module on H100.

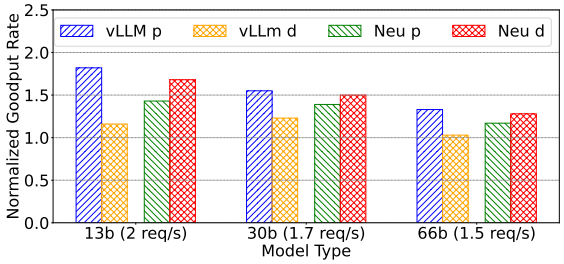

Figure 16. Module's normalized goodputs of each LLM.

Normalized goodput of modules. For different LLMs, Figure 16 shows the normalized goodput of each module on A6000. We see that NeuStream's allocation ensures a balanced processing speed between stages, since our inter-module scheduling explicitly allocates SPUs to maximize the minimum module normalized goodput. By contrast, vLLM exhibits a significant imbalance between the two stages, as it greedily assigns resources to prefill with the use of preemption. Thus, vLLM overall goodput is limited by the slower (or bottleneck) decode module.

6.4 Results on Multi-agent Model

For the multi-agent model, different LLM agents are placed on different GPUs.

Goodput under varying offered load. Figure 17 and Figure 18 show the goodput of decode module for each LLM agent on RTX 4090 and H100, respectively. NeuStream significantly outperforms vLLM across LLM agents (e.g., achieving 13.46× at 1 requests/s, up to 1.54× at CV of 4, and 2.93× at SLO scale of 1.4 on RTX 4090 GPUs, meanwhile, achieving 1.38× at 2.2 requests/s, up to 1.2× at CV of 4, and 2.11× at SLO scale of 1.6 on NVIDIA H100 GPUs).

Cost/Model			SD	OPT	M-agents
Profile			4.32 min	5.2 min	11.37 min
Schedule			1.02%	1.95%	5.31%
Comm.	colocate		<0.1%	<0.1%	/
	non-colocate	PCIe	2.6%	12.10%	0.1%
		NVLink	0.66%	0.98%	/
Switch			0.81%	<0.1%	0.22%
Memory			3.62%	3.39%	4.79%

Table 2. Execution Cost for Stable Diffusion (SD), OPT-30B (OPT), and multi-agents (M-agents).

Normalized goodput of modules. Figure 19 shows the goodput of each LLM agent's module on RTX 4090. We see that NeuStream achieves balanced speeds across stages of LLM agents, alleviating the bottleneck. Both the normalized goodput of prefill/ decode stages of vLLM is less than NeuStream. This is because imbalanced prefill and decode phases in vLLM lead to high prefill congestion due to insufficient memory to hold new KV cache.

6.5 Execution Cost

Table 2 reports the execution cost of NeuStream on different types of models.

Profiling. Profiling the latency function is a one-time *pre-processing* stage (e.g., only a few minutes), which is negligible compared to the subsequent long serving time (e.g., hours and days). For diffusion, profiling only depends on batch

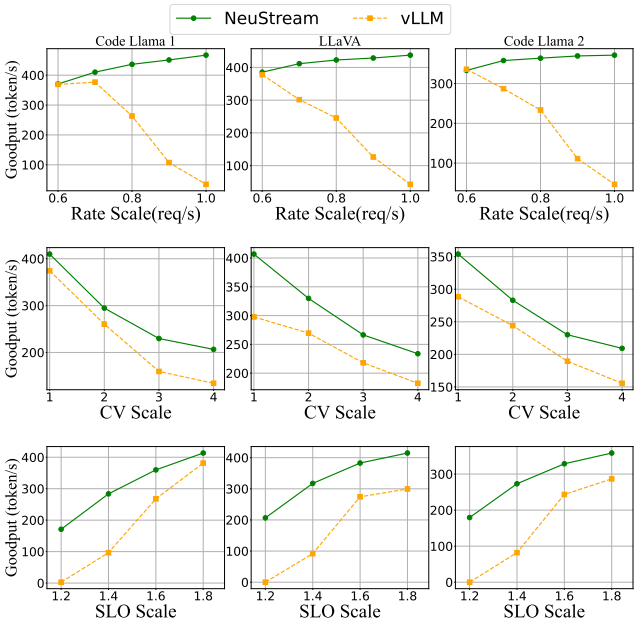

Figure 17. Goodput of agent's decode module on RTX4090.

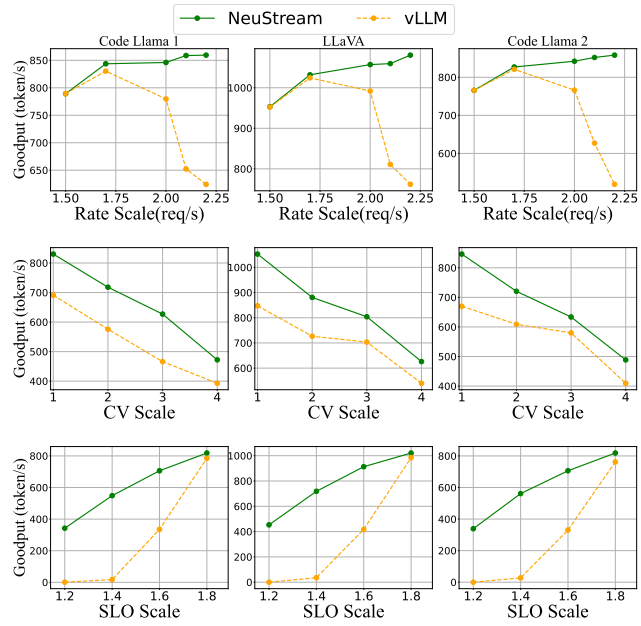

Figure 18. Goodput of agent's decode module on H100.

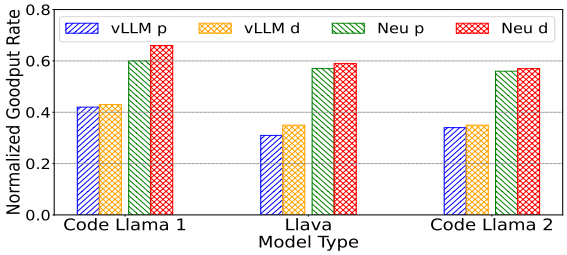

Figure 19. module normalized goodputs in MatPlotAgent

size. For LLMs, the profiling further depends on input token length. To reduce overhead, we use profiling and interpolation to figure out the latency function as DistServe [36].

Scheduling. Running the scheduling algorithm might incur an overhead of batching decisions, resource allocation and state tracking. We see that NeuStream only incurs a slight scheduling cost (< 2%) in LLMs, and still very little overhead (< 5.3%) even more LLMs are involved in multi-agents.

Communication. We test different inter-communication scenarios: For co-located case, the communication overhead is not visible in end-to-end request latency as modules only pass data pointers. For non-colocated case, we place each module (or agent) on a different GPU for SD and LLM (or multi-agents). We still observe low overheads for diffusion and muli-agents due to only transferring small activation messages. For LLM, transferring KV cache via PCIe in A6000 could incur a relatively large overhead, which can be alleviated via either co-locating or transferring via NVLink in H100.

Switching. We see that the overhead of switching between modules is little for all the models (e.g., <1%) as the most context of module resides in the GPU memory when switching between modules (e.g., the model weight used by both prefill and decode modules).

Memory. Although time-sharing of SPUs increases heterogeneous GPU kernels to be concurrently executed, it does not introduce significant memory overhead in our workload (e.g., < 5%), since the workspace mainly consists of small activations. Also, our module granularity (e.g., sub-models) does not split individual or fused kernels, without yielding additional memory traffic.

7 Related work

DNN serving systems. Clockwork [7] leverages the predictable execution times of individual DNN inferences. Shepherd [32] aggregates request streams into moderately sized groups to improve predictability. Systems like Nexus[22], INFaaS [19], gpulet [4], Gslice [6], focus on the serving of multiple static DNN models with spatio-temporal sharing of GPUs. AlpaServe [10] further introduces model parallelism for statistical multiplexing. However, these systems fail to take into account the dynamic property of modern DNNs, resulting in missed opportunities for optimization. Systems for dynamic DNNs have also been studied in recent years. BrainStorm [5] proposes a new data abstraction called Cell to optimize the dynamism of different parts within a single input, e.g., a token inside a request for Mixture-of-Experts (MoE). Nimble [23] and Cocktailer [31] express and execute dynamic neural networks from a compiler perspective.

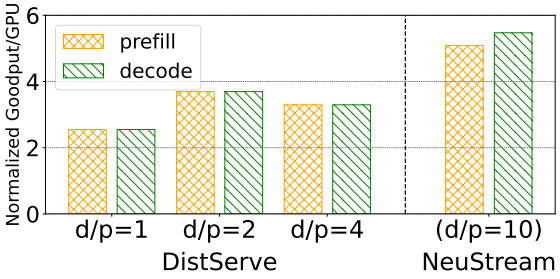

Figure 20. Normalized per-GPU goodput of instances with OPT-13B model on LMSYS-Chat-1M dataset and H100.

NeuStream is orthogonal to these works by exploring the dynamism and batching opportunity across multiple inputs and supports SLO-aware DNN serving.

Specialized systems for LLM serving. LLM inference serving is a fast-developing area. Orca [29] introduces iteration-level scheduling to increase the GPU utilization. vLLM [9] proposes a novel memory management strategy PagedAttention for KVCache to increase the throughput of LLM serving. These systems are optimized and specialized for LLMs. our NeuStreamabstraction not only naturally enables the aforementioned optimizations for LLM models but also generalizes to other DNN models (e.g., as shown in Figure 1).

Streaming systems. Traditional streaming processing systems such as Naiad[14], Spark [30], Flink [3], StreamScope [11], Apache Storm [24], etc., process continuous data flows in real-time. However, these streaming systems primarily handle I/O-intensive data such as application logs, sensor events, etc. In contrast, DNN applications are often compute-intensive, requiring the execution of tensor-based inputs in large batches on GPUs to achieve high throughput. This leads us to different trade-offs and design options for NeuStream.

8 Discussion

Disaggregated LLMs. Although the idea of decomposing a LLM workload has emerged in more recent work (e.g., DistServe [36] and Splitwise [16]), NeuStream presents a generic decomposition abstraction and system that can be used for different workloads, not just specialized for LLMs. Moreover, based on this abstraction, NeuStream offers additional optimization opportunities, such as optimizing resource allocation and SLO-aware scheduling at fine-grained streaming modules and SPUs. For example, given a pair of prefilling and decoding instances, DistServe allows to assign different degrees of model parallelism (e.g., tensor parallelism) to each instance. Figure 20 shows the normalized per-GPU goodput of each instance in DistServe, when serving OPT-13B on LMSYS-Chat-1M dataset used in Sec. 6.3. Here, D/P gives the resource usage ratio of decoding instance to prefill instance. Note that the minimum granularity of resource allocation in DistServe is the entire GPU. Given the relatively low average input-output ratio of requests, we observe that orchestrating prefill and decoding in DistServe is challenging at the coarse-grained GPU allocation model: under tight SLO constraint, insufficient GPU allocation for decoding instance (e.g., D/P=1) at the consumer side would limit the full processing capability and resource utilization of prefill instance at the producer side, however, increasing GPU allocation for decoding instance faces the growing scaling overhead of model parallelism (e.g., the per-GPU goodput even degrades when increasing D/P ratio from 2 to 4). By contrast, NeuStream can minimize the mismatch between the processing capability of P/D instances at fine-grained SPUs abstraction and allocation.

Handle large heterogeneous cluster. NeuStream could scale to a large-scale serving cluster by treating it as multiple independent copies of a small cluster and scheduling each individual small cluster. NeuStream can allocate resources for individual small clusters guided by the profiled normalized goodput, e.g., it finds the modules in a certain small cluster that yields the minimum normalized goodput, and dynamically allocates SPU to that module (and thus the cluster). Also, in a heterogeneous cluster, different types of GPUs can be abstracted into different numbers of SPUs, which can still be handled by NeuStream.

Program re-writing and correctness. Our module inherits the PyTorch module definition, where developers only need to decouple the control-flow and data-flow into `scatter` and `compute` functions, respectively. Since the control-flow code constitutes only a small portion of the overall code, the rewriting effort is not significant, e.g., <7% LOC re-writing in Stable Diffusion. We currently adopt a common numerical testing approach to assess the correctness of the rewritten code by constructing a test set, e.g., in Stable Diffusion, the same prompt/seeds should produce identical outputs within a numerical error bound before/after the code changes.

Suitable scenarios. When employing NeuStream, an interesting question is what are the scenarios one benefits from using it? As illustrated in Sec. 2, NeuStream is useful for serving models with dynamic execution paths or distinct functional phases. However, serving static models with homogeneous phases might not benefit from NeuStream.

9 Conclusion

NeuStream addresses the inefficiencies of serving modern DNN applications with dynamic execution patterns on GPUs. By introducing the concepts of streams and stream-modules, NeuStream enables a continuous data flow execution model that effectively maximizes GPU utilization while meeting stringent latency SLOs. The abstraction of GPU resources into Streaming Processing Units and the decomposition of scheduling into intra-module and inter-module levels further

enhance its scalability and scheduling efficiency. Our evaluation of NeuStream demonstrates its superior performance in serving complex DNN applications, such as LLM-based chat, image generation, and multi-agents. As DNN models continue to evolve and become increasingly complex, NeuStream offers a scalable and efficient solution for future advancements in deep learning applications.

References

[1] HuggingFace. Diffusers doc. https://huggingface.co/docs/diffusers/v0.27.0/en/stable_diffusion.

[2] PyTorch. https://pytorch.org.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. The Bulletin of the Technical Committee on Data Engineering 38, 4 (2015).

[4] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22). 199–216.

[5] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, et al. 2023. Optimizing dynamic neural networks with brainstorm. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 797–815.

[6] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In Proceedings of the 11th ACM Symposium on Cloud Computing. 492–506.

[7] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 443–462.

[8] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021).

[9] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles. 611–626.

[10] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 663–679.

[11] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. {StreamScope}: Continuous Reliable Distributed Processing of Big Data Streams. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). 439–453.

[12] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2023. Improved baselines with visual instruction tuning. arXiv preprint arXiv:2310.03744 (2023).

[13] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. 2021. Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In Proceedings of the ACM Symposium on Cloud Computing. 442–457.

[14] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farminton, Pennsylvania) (SOSP '13). Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[15] NVIDIA Corporation. 2024. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html Accessed: 2024-05-21.

[16] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative llm inference using phase splitting. arXiv preprint arXiv:2311.18677 (2023).

[17] William Peebles and Saining Xie. 2023. Scalable diffusion models with transformers. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 4195–4205.

[18] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 10684–10695.

[19] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In 2021 USENIX Annual Technical Conference (USENIX ATC 21). 397–411.

[20] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023).

[21] Chitwan Saharia, William Chan, Huiwen Chang, Chris Lee, Jonathan Ho, Tim Salimans, David Fleet, and Mohammad Norouzi. 2022. Palette: Image-to-image diffusion models. In ACM SIGGRAPH 2022 conference proceedings. 1–10.

[22] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 322–337.

[23] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. Proceedings of Machine Learning and Systems 3 (2021), 208–222.

[24] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 147–156.

[25] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile dynamic provisioning of multi-tier internet applications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3, 1 (2008), 1–39.

[26] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. 2018. Skipnet: Learning dynamic routing in convolutional networks. In Proceedings of the European conference on computer vision (ECCV). 409–424.

[27] Canwen Xu and Julian McAuley. 2022. A survey on dynamic neural networks for natural language processing. arXiv preprint arXiv:2202.07101 (2022).

[28] Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. 2024. MatPlotAgent: Method and Evaluation for LLM-Based Agentic Scientific Data Visualization. arXiv preprint arXiv:2402.11453 (2024).

[29] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 521–538.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10).

[31] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. 2023. Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 681–699. https://www.usenix.org/conference/osdi23/presentation/zhang-chen

[32] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. {SHEPHERD}: Serving {DNNs} in the wild. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). 787–808.

[33] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[34] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068 (2022).

[35] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. (2024). arXiv:2309.11998 [cs.CL]

[36] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. arXiv preprint arXiv:2401.09670 (2024).