Asynchronous End-to-End Audio and Image Processing for Faster Neural Network Inference

Team ZARN: Alicia Lyu, Ruijia Chen, Zach Potter, Nithin Weerasinghe

1 Introduction

A neural network inference system typically handles a large number of requests from diverse applications. Many such requests are latency-sensitive [4, 1]. The variety of input types, including audio, image, and video data, contributes to this diversity. Processing data directly adds to the latency of inference and increases the complexity of both serving a large number of queries and meeting the Service Level Objectives (SLOs) of different applications. However, the pre-processing and post-processing that surround the actual inference are commonly considered part of the client program and, as such, are rarely considered in the design of the inference system itself. Research on inference schedulers has focused on GPU resource allocation [4, 1] for the actual inference and overlooked the optimization opportunity for large marginal benefit from media data processing.

There are indeed exceptions to this norm. For example, Nexus[5], a video analysis platform, implemented a full-swing resource manager on a GPU cluster, including CPU allocation for data processing. Although it observed the need for CPU management, especially for resource-intensive job such as video processing, we argue that its model on the collaboration between CPUs and GPUs does not fully exploit the nature of this workload. The latter, we argue, constitutes the conceptual innovation of our work.

The nature of the inference workload is that there is no resource contention between data processing and inference, as the former predominantly happens on CPUs with the latter on GPUs. Furthermore, as the stages of processing a request (e.g. data pre-processing, inference, data post-processing) are linearly arranged, the dependency between stages and, in turn, between GPU and CPU allocations, is simple. We abstract the dependency into two archetypes: 1) CPU task–GPU task, 2) GPU task–CPU task. As long as every *local* dependency is satisfied, no *global* coordination between CPU and GPU is needed. With *decentralized* coordination, implementing a CPU scheduler does not require any *centralized* knowledge of the GPU scheduling. In other words, we can have two independent schedulers: one for CPU and one for GPU.

Dividing scheduling for CPU and GPU allows us to focus on designing a CPU scheduler and plug it into any out-of-box GPU scheduler, treating the latter as a black box (GPU scheduling is extensively investigated). In our implementation, the NVIDIA Triton Server[2] is used as this black box inference server. So, another way to say that Nexus is "a fully implemented system"[5, p. 322] is that it is not modularized and extensible, whereas we see **modularization and extensibility** as the strength of our work.

We can summarize our system design of a CPU scheduler with a few points:

- **Data parallelism and pipelining**: Since each request needs to wait on network I/O from the inference server, a natural design is to process multiple batches of data at the same time and pipeline their CPU allocations. This design in also seen in Nexus[5].

- **Scheduling based on "social trust"**: Unlike an OS scheduler, our scheduler has good knowledge of each request, as they follow similar scheme procedurally. The data size is also mostly bounded for a single request, and, eventually, the inference request code needs to be adapted to a `Client` to participate in our system. Therefore, the scheduler relinquishes some control to the requests and *does not preempt processes*[1]. It entrusts them to request and relinquish CPU as they see fit.

- **Local control of stage dependency**: Each request manages all of its local dependencies and only requests the CPU if the previous stage is finished, effectively nullifying the need for a centralized coordinator and simplifying the design of the CPU scheduler.

- **Pluggable priority policy**: We also enable multiple priority policies pluggable to the scheduler.

[1] The use of process and request is mostly interchangeable in this paper, though, more precisely, the process is a running job on the computer that executes the request, whereas the request is only a specification of what to be done.

Right now, our system implements FIFO and SLO-oriented policies, which is discussed in Section 3, which adds to the extensibility of our system.

- **Agnostic to and extensible with respect to model and data types**: Our implementation is not specific to a model or a data type. Any inference request can be adapted to use our system with minimal coding change. We implemented two types of requests, as discussed in Section 4.

We see our work contributing to the topic of inference scheduling system in the following aspects:

- We emphasize on the importance and high marginal benefits of improving data processing for meeting inference SLOs.
- We propose a new conceptual framework of the relationship between the CPU scheduler and GPU scheduler that decentralizes the dependencies therebetween.
- We implement a flexible CPU scheduler, extensible to include customary priority policies and processing of different data types and pluggable to out-of-box inference servers.

2 Background

2.1 Neural Network Inference

Neural network inference refers to a workload where a pre-trained machine learning model is utilized to make predictions based on new data. This process can be dissected into several stages: 1) the data (which may be in various forms such as audio, image, or video) is prepared and pre-processed to be compatible with the model, 2) the actual inference occurs (the neural network takes the data as input and outputs a prediction), 3) post-processing refines the output of the model for use in downstream applications (e.g. converting the output matrix into human-readable texts). SLOs play a pivotal role, defining the expected performance metrics such as response time and throughput that the system aims to meet.

These systems typically employ a combination of CPUs (Central Processing Units) and GPUs (Graphics Processing Units). CPUs, designed to perform a wide range of tasks due to their ability to execute sequential instructions and manage system operations effectively, are primarily employed in the pre-processing and post-processing stages of an inference request. These stages often require decision-making, data handling, and other non-compute intensive tasks that CPUs handle efficiently [5, p. 331]. Conversely, GPUs are specialized for parallel processing, making them ideal for the compute-intensive inference stage.

2.2 Traditional OS Scheduling

Our scheduler does not replace the OS scheduler but rather builds on top of it. Process switching by the OS scheduler still happens in the background, as there are multiple processes running on the computer: the docker container that runs the Triton Server, the python program that sends request, the python program of our scheduler, and any process that handles the request. Our scheduler simply *bounds* the scheduling overhead of the OS scheduler by filtering the requests and deciding which one to process.

Traditionally, an OS schedules processes to do time sharing on CPUs, as only one process can run at a time on a core. Several concepts such as context switching, preemption, yielding, and priority are employed in a traditional OS scheduler and also borrowed by our work.

Context switching, the mechanism associated with scheduling, refers to the overhead incurred when switching from one process to another. Our scheduling filters requests and limits simultaneous processes handled by the OS scheduler, thereby decreasing the overhead from context switching.

As a policy choice, the system can switch between processes either by *preempting* a currently running job (e.g. because it used up its time share) or by waiting for the job to *yield* itself. Our system borrows the latter; we wait for processes to yield but not preempt.

Another policy choice is which process to run next. Some of such policies involve adding a *priority* component. Our scheduler borrows this concept and devises a SLO-oriented policy that favors requests with the nearest deadline. We believe, with more knowledge of the requests and their respective SLOs, our scheduler can outperform the OS scheduler, justifying monopoly on the choice of which request to handle next. Figure 2 shows actual comparison between the OS scheduler (under system name "non_coordinated_batch") and our scheduler (under system name "pipeline", followed by the policy name).

3 Design

Our code repository is open sourced at github.com/alicia-lyu/pipelined-data-processing-for-nn-inference. Our code is written in Python, and its design follows the object-oriented scheme and is typed and well-organized into modules.

3.1 Client

We created an abstract class `Client` as the basis for all clients that perform a single request of any kind. It is initialized with `t0: float`, the arrival time of the request, and a `signal_pipe: Connection`, which is a dedicated connection channel to a `Scheduler`. It comes with two methods for communication:

- `wait_signal(self, signal_awaited: Message) -> None` blocks the client until the awaited signal is received. It also sends a message to the scheduler that it is waiting for a specific signal. The client requests CPU by calling `self.wait_signal(WAITING_FOR_CPU)`.
- `send_signal(self, signal_to_send: Message) -> None` sends a signal to the scheduler without blocking. The client relinquishes CPU by calling `self.send_signal(CPU_AVAILABLE)`.

In addition, any derived client classes must implement two abstract methods `run(self)` and `log(self)`. Take `TextRecognitionClient` for example. Its `run` is divided into 5 stages sequentially:

1. Pre-processing (CPU): Multiple images are converted to blobs and stacked together as a matrix.
2. Text Detection (GPU): Data is sent to the Triton Server for inference using OpenCV's EAST model, outputting the the information of text box(es) within those images.
3. Cropping (CPU): Pre-processed images are cropped into the text box(es), which stack together and form a matrix.
4. Text Recognition (CPU): Matrix representing the cropped text boxes is sent to Triton Server again for inference, with model ResNet.
5. Post-processing (CPU): The output matrix from ResNet is converted to human-readable texts.

Every time before a CPU stage, the client request for CPU by invoking `wait_signal(Message.WAITING_FOR_CPU)` and is blocked until it gets one. After the CPU is granted, the client proceeds, relinquishing the CPU upon finishing the CPU stage. Note that relinquishing the CPU does not block the client but allows it to proceed to the next GPU stages if any, until blocked again by the next CPU task.

During execution, the client records timestamps at key points and stores them in `stats`. These statistics are logged into a file and sent back to the parent process. Specifically, `stats` is managed by a `multiprocessing.Manager`, shared between the client and the parent process which sent the client. In our case, its parent process is `System`, which is responsible for creating client processes that follow a distribution pattern.

3.2 Scheduler

The `Scheduler` class takes a few arguments:

- `parent_pipes: List[Connection]` are used to communicate with all clients
- `policy: Policy` specifies the policy to choose the next task; `FIFO` and `SLO_ORIENTED` are currently built-in
- `cpu_tasks_cap` is the maximum for the number of tasks sent to the OS scheduler (i.e. the number of clients granted CPUs) at the same time
- `deadlines: List[float]` as the list of deadlines of all clients relative to the start of the scheduler
- `lost_cause_threshold: int` specifies how many seconds after the deadline we deem a request a "lost cause" and give up trying to meet its deadline

Algorithm 1: Scheduler Algorithm

Data: $active\_cpus = 0, children\_states = \{\}$
while *true* **do**
  if *A signal from the clients comes* **then**
    $active\_cpus \leftarrow update()$;
    $children\_states \leftarrow update()$;
  if *CPU is available* **then**
    $candidate \leftarrow policy()$;
    Allocate CPU to the $candidate$;
  if *An allocation decision is made* **then**
    $active\_cpus \leftarrow update()$;
    $children\_states \leftarrow update()$;

The scheduling algorithm is summarized in Algorithm 1. The `Scheduler` manages the state of all active clients in `children_states`. Right now, it supports three states: `CPU`, `GPU`, `WAITING_FOR_GPU`. Whenever a signal comes from a pipe (`ready = select.select(self.parent_pipes, [],`

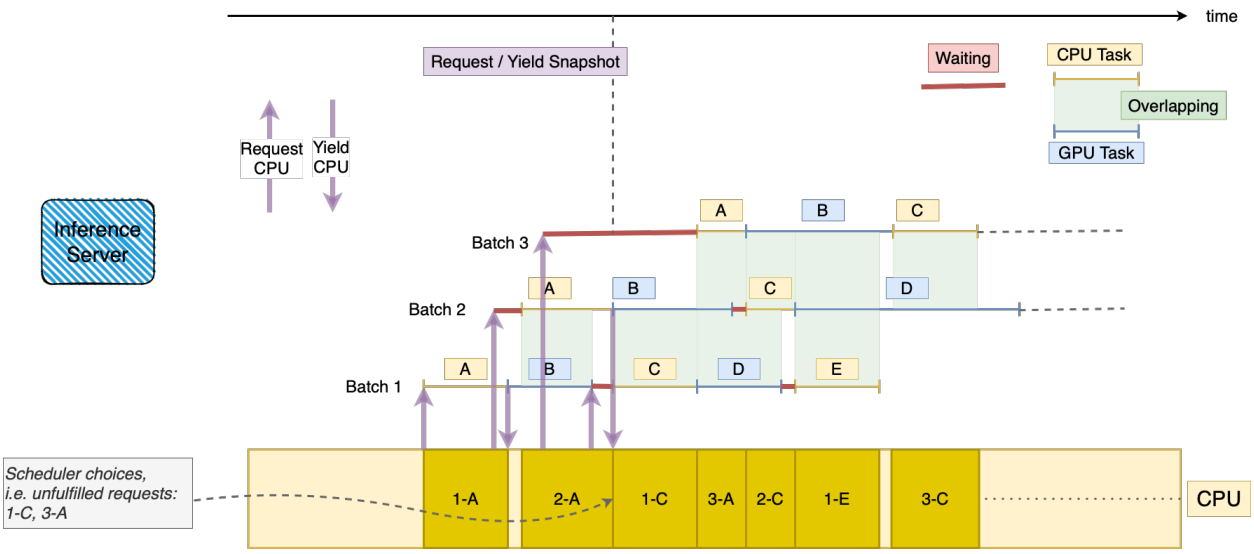

Figure 1: Scheduler Workload Example, policy = FIFO, cpu_tasks_cap = 1

`[], self.timeout))` and whenever an allocation decision is made, the scheduler updates the state of a child. This is mainly to facilitate the decision of the next client to execute, for it must be `WAITING_FOR_CPU`.

Every time a CPU is available, a candidate client is chosen by `self.policy()`, for which we have `fifo(self) -> bool` and `slo_oriented(self) -> bool` built in. FIFO simply chooses the client with the smallest `client_id`, given that it is `WAITING_FOR_CPU`. SLO oriented chooses the client with the nearest deadline; the deadline can be past but no more than `lost_cause_threshold`.

Figure 1 visualizes the CPU allocation process with the FIFO policy and cpu_tasks_cap = 1. This figure highlights a few points discussed in Section 1:

- **Data parallelism and pipelining**: Multiple batches are processed at the same time and their CPU stages are pipelined (yellow blocks). We can keep the CPU busy for another batch when the current batch is waiting on inference. The green shadow represents the overlapping of CPU stages with GPU stages across batches.
- **Separate schedulers for GPU and CPU**: The inference server (blue box on the left) is treated as a black box and has no bearing on CPU scheduling.
- **Local control of stage dependency**: Stage dependency is enforced by each batch/request individually. Each only requests the CPU when the previous GPU stage is done, and each relinquishes CPU once it pivots from a CPU task to a GPU task (the purple arrows).
- **Scheduling based on "social trust"**: The scheduler does not preempt tasks, as the requests are predictable procedurally and each request's data size and duration are both mostly bounded.
- **Dedicated resources and short processing time** Only a few tasks (in this case only one) are sent to the OS scheduler, dramatically decreasing the context switch overhead. With dedicated resources, CPU stages finish more quickly. However, there is *significant delay in response time*. That is the premium to pay for getting dedicated resources.

4 Evaluation

We implemented two additional systems to compare with our scheduler (under the name "pipeline" in the figures and stats):

1. **Naive sequential**: Only after completing the processing of the previous batch does the system start processing the next batch.
2. **Non-coordinated processes**: The system starts a process every time a new batch of data arrives without coordinating CPU usage, which means multiple batches may occupy CPU at the same time, and the OS-level scheduler takes over the job of coordinating them.

To decrease the noise in our experiments, we randomly generate the `intervals` following a specific distribution and `priorities` for each request in a `Comparison` instance before the experiments actually

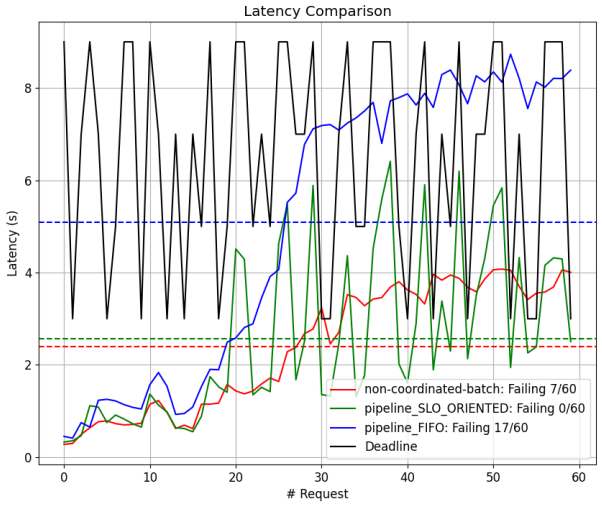

Figure 2: Latency comparison on the non-coordinated system and the pipeline system with FIFO and SLO-oriented policies. Setting A1: Data arrival follows Poisson Distribution with mean = 0.3 s. Requests come in with batch size of 2. Inference task is text detection and recognition of images.

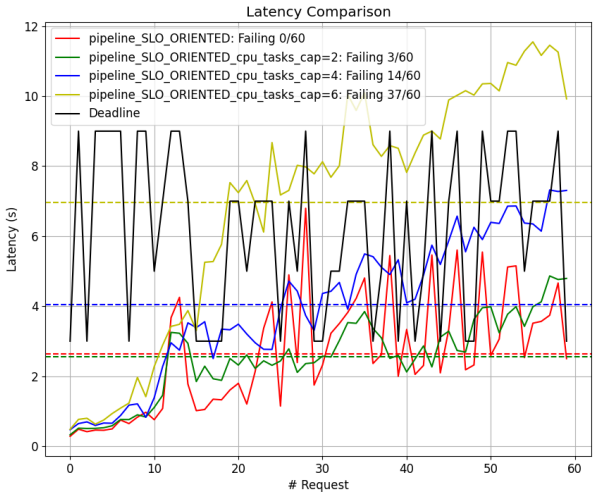

Figure 3: Latency comparison with different cpu_tasks_cap. Setting A2: Data arrival follows Poisson Distribution with mean = 0.25 s. Requests come in with batch size of 2. Inference task is text detection and recognition of images.

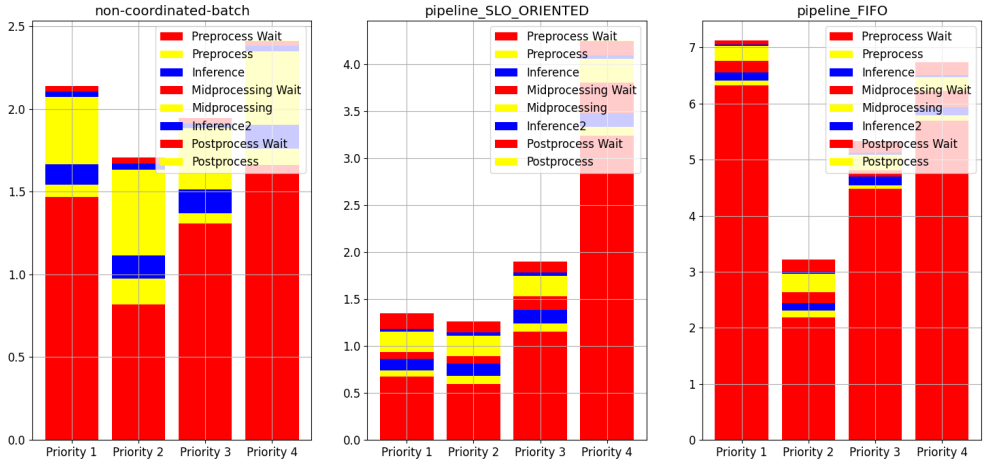

Figure 4: Median time that each stage takes on the non-coordinated system and the pipeline system with FIFO and SLO-oriented policies. Setting A1.

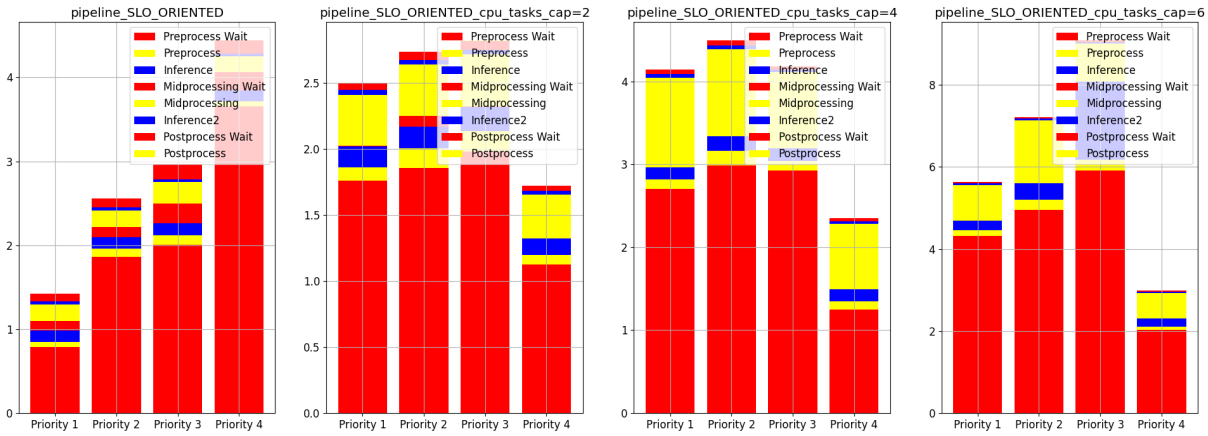

Figure 5: Median time that each stage takes with different cpu_tasks_cap. Setting A2.

begin. Therewith, we ensure the arrival times and latency goals stay the same across different systems.

We run our experiments on a deep learning instance on Google Cloud Platform with the machine type n1-standard-2 (2 vCPUs, 7.5 GB RAM) with 1 NVIDIA T4 GPU; its OS type is Linux/Debian.

The included experiment results can be divided into 4 groups[2]: A1, A2, B1.1, B1.2.

- Setting A1: Data arrival follows Poisson Distribution with mean = 0.3 s. Requests come in with batch size of 2. Inference task is text detection and recognition of images. Experiments are done on one set of systems: non-coordinated, pipelined with FIFO, pipelined with SLO-oriented. Results are shown in Figures 2 and 4.
- Setting A2: Data arrival follows Poisson Distribution with mean = 0.25 s. Requests come in with batch size of 2. Inference task is text detection and recognition of images. Experiments are done on one set of systems: all are pipelied with SLO-oriented policy, but with different cpu_tasks_cap 1, 2, 4, 6. Results are shown in Figures 3 and 5.
- Setting B1: Data arrival follows Poisson Distribution with mean = 5 s. Requests come in with batch size of 2. Inference task is speech recognition of audios. Experiments are done on 2 sets of systems:
    1. Non-coordinated, pipelined with FIFO, pipelined with SLO-oriented. Results are shown in 6 and 8.
    2. All are pipelied with SLO-oriented policy, but with different cpu_tasks_cap 1, 2, 4, 6. Results are shown in 7 and 9.

4.1 Meeting SLOs Better

Meeting SLOs is one of the most important performance metrics of an inference system. Our current experiments only considers the latency goal among all possible forms of SLOs. Other policies focusing on other SLOs can certainly be implemented and plugged into the scheduler, as one possible direction of future work.

From Figure 2 (A1), we can see the pipeline system with SLO-oriented policy meets the SLOs better than the pipeline system with FIFO policy and the non-coordinated system, with 0/60 failing rate compared to 17/60 and 7/60. This corroborates our conjecture that scheduling clients with nearest latency can better meet SLOs. It is also visually obvious that the SLO-oriented policy matches the latency goal very well.

[2]We define a deadline miss as that a request takes 10%+ more time to complete than the latency goal.

Figure 2 (A1) also shows that the pipeline system with FIFO policy has longer latency than the non-coordinated system. This could come from a few reasons, including higher execution overhead of Python. Also, only letting on request be processed at a time under-utilizes the CPU, especially because the CPU on our machine has 2 core. The OS level scheduler allows multiple tasks to run simultaneously and better utilizes the CPU. This assumption is corroborated by the fact that when allocating 2 tasks simultaneously on the CPU compared to 1 task in Figure 3 (A2), the latency is slightly reduced due to higher CPU utilization.

However, as cpu_tasks_cap further increases in Figure 3 (A2) (which means more tasks are allocated simultaneously), the order in which those tasks are processed is determined by the OS-level scheduler, resulting in us having *less control* over which process is prioritized. Thereby, we observe an increase in the rate of failure to meet SLOs. The system is more prone to missing deadlines with higher cpu_tasks_cap.

4.2 Tradeoffs and Limitations

From Figure 9 (B1.2), the trade-off with respect to **dedicated resources** is observed. As cpu_tasks_cap increases from 1 to 6, time waiting for CPU decreases, while the execution time on CPU decreases as well. Allowing more tasks to be sent to OS scheduler together makes the CPU resources more accessible, thereby reducing the waiting time to start each CPU stage. After a CPU stage is granted, the wait is up, but each request has to share the CPU with other requests. The OS scheduler switches between them, creating context switch overhead. The final effect—whether that increases total latency or not—is different across workloads. Figure 3 (A2) shows increased latency for image processing, and Figure 7 (B1.2) shows decreased latency for audio processing, as cpu_tasks_cap increases from 1 to 6.

One explanation of the difference is different natures of workloads: for audio processing in Figure 8 and 9 (Setting B1), we could hardly see the inference time (the blue area) compared to image processing in Figure 4 and 4 (Setting A1). The inference time is minimal for the audio client. Now that the CPU is the bottleneck for audio data, CPU efficiency represents most of the system's efficiency. The other parts of the audio workload is similar to common CPU-intensive workloads. The OS scheduler may be better optimized in this regard, leading to lower latency for scheduling multiple audio processing tasks simultaneously compared to image processing.

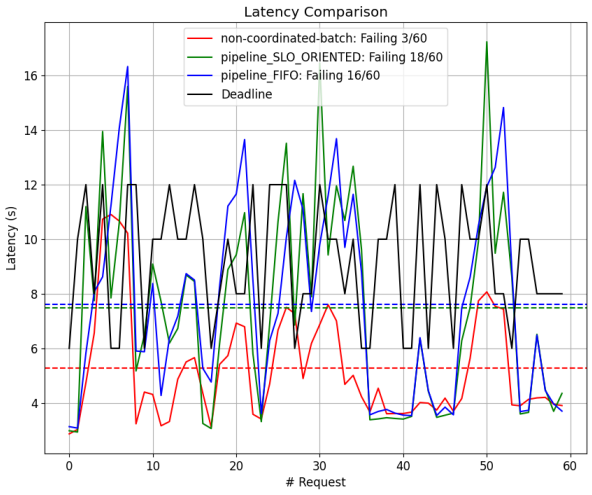

Figure 6: Latency comparison on the non-coordinated system and the pipeline system with FIFO and SLO-oriented policies. Setting B1: Data arrival follows Poisson Distribution with mean = 5 s. Requests come in with batch size of 2. Inference task is speech recognition of audios.

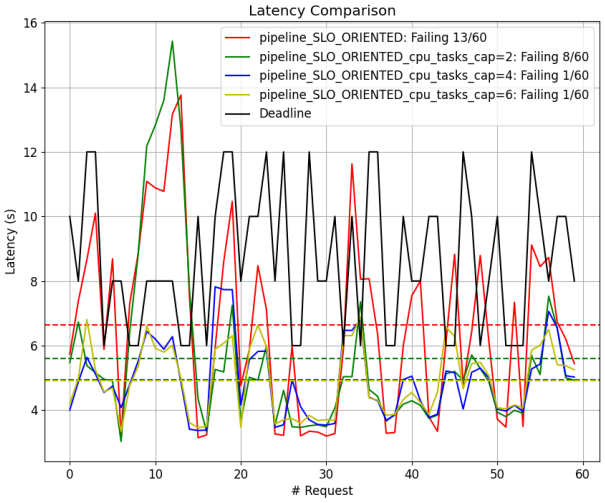

Figure 7: Latency comparison with different cpu_tasks_cap. Setting B1: Data arrival follows Poisson Distribution with mean = 5 s. Requests come in with batch size of 2. Inference task is speech recognition of audios.

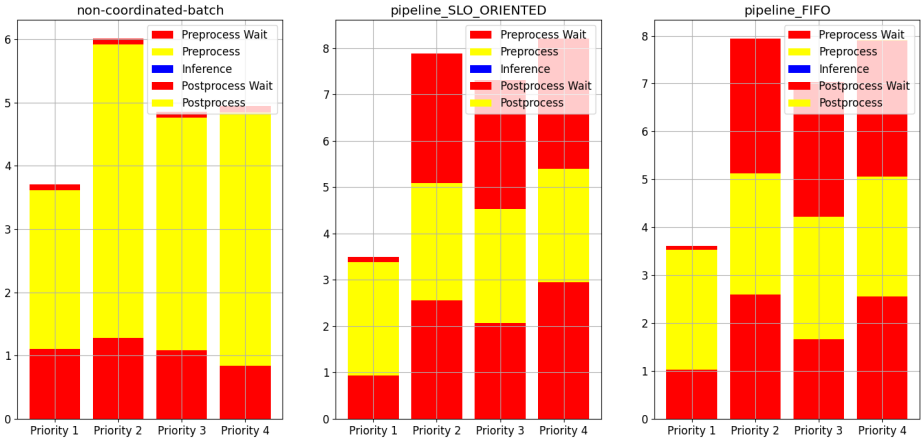

Figure 8: Median time that each stage takes on the non-coordinated system and the pipeline system with FIFO and SLO-oriented policies. Setting B1.

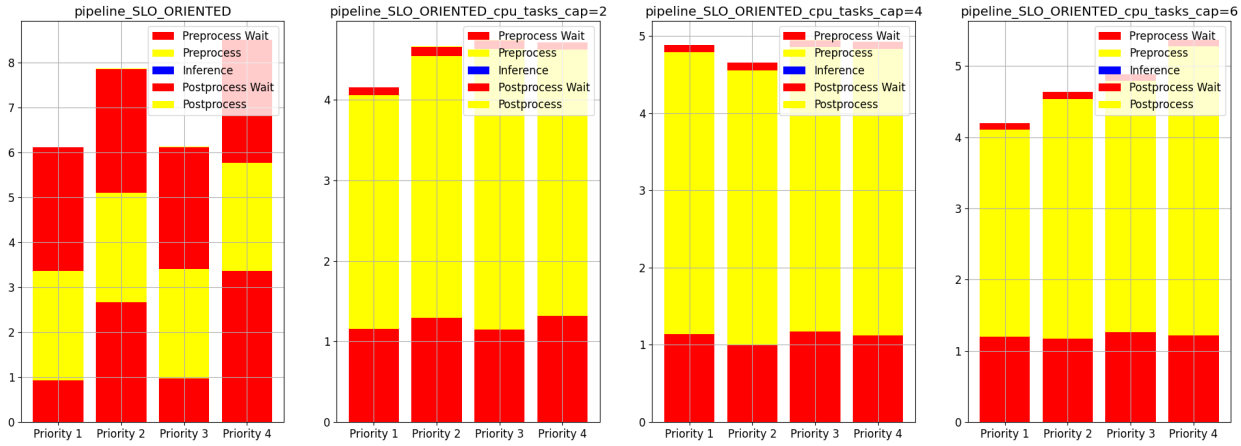

Figure 9: Median time that each stage takes with different cpu_tasks_cap. Setting B1.

5 Related Work

5.1 Benefits of Overlapping CPUs and GPUs

Nexus recognizes the need to optimize data processing on CPUs for video analysis by overlapping it with GPUs. Their experiments reveal that this technique is critical for tight-SLO/small model regime. In the example of game analysis, disabling overlapping results in 7.4x throughput reduction. With more relaxed SLOs and larger models, the importance of overlapped preprocessing (OL) is diminished.[5]

Their overlapping strategy is fairly simple, and the CPU scheduler is only an ephemeral part of their work: they use a thread pool of workers to do pre-processing and post-processing, while another thread is dedicated to launching batched executions on the GPU.

As discussed in Section 1, they did not observe the separable nature of CPU and GPU scheduling. They did not exploit this to build an extensible and pluggable CPU scheduler, but only added a case-specific implementation in their full swing video analysis system.

5.2 Overview of the Triton Server and Client

The Triton Inference Server is an open-source application designed by NVIDIA for inference tasks, capable of running on both CPU and GPU hardware across various environments[2]. Triton has a built-in scheduler for inference tasks, employing multiple techniques, including concurrent execution of multiple models, dynamic batching, particularly for diverse types of queries.

However, input goes through three sequential stages, not just one inference stage, to output the results: preprocessing, inference, and post-processing. However, Triton scheduler does not encompass the data processing part of the job, leaving that to the client program, although some middle processing stages between 2 inference jobs may be cooperated into the model repository and executed on the server side with Model Ensembles[3].

6 Future work

Our system design and experiments do not fully explore the potential of the conceptual framework of separating of CPU and GPU schedulers. There are more implications on system design, but also necessitates some justifications to be made or problems to be solved before that.

Although the dependency between CPU and GPU is decentralized and only requires local enforcement, different workloads do have different bottlenecks, which may require a change in behavior in the other scheduler. For example, for our B2 experiments, the CPU is over-utilized, while the GPU is under-utilized. Naturally one would think to have GPU share some of the workload. Whether this inter-dependent behavior change breaks the current conceptual framework, or just necessitates some patches on it requires future investigation.

Also, we built the system to be pluggable to other inference server and extensible to different request types, data types, and policies, but there have not been sufficient development and experiments in this respect. Exploration of our current system using different settings may also yield new observation of this design.

If this conceptual framework can survive all the challenges and prove to be robust in diverse experiments, there are multiple pathways forward therefrom. One possibility is to have CPUs and GPUs scale independently, without the concern for the compatibility of old schedulers.

7 Work Distribution

- **Alicia Lyu**: Design the system, refactor `Client`, `Scheduler` and `System`, develop `Comparison`, design and run experiments, make poster, write report and introduction
- **Ruijia Chen**: Develop `TextRecognitionClient`, `Scheduler`, and `System`, run and analyze experiments, help write report and introduction
- **Zach Potter**: Develop `AudioRecognitionClient`, help write report and introduction
- **Nithin Weerasinghe**: Develop `AudioRecognitionClient`

References

[1] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *USENIX*, 2020.

[2] NVIDIA. Nvidia triton inference server. `https://developer.nvidia.com/triton-inference-server`. Accessed: 2024-03-08.

[3] NVIDIA. Nvidia triton inference server tutorial part 5. https://github.com/triton-inference-server/tutorials/tree/main/Conceptual_Guide/Part_5-Model_Ensembles. Accessed: 2024-05-07.

[4] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. Infaas: Automated model-less inference serving. *USENIX*, 2021.

[5] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.