Astraea: Towards QoS-Aware and Resource-Efficient Multi-stage GPU Services

Wei Zhang
Shanghai Jiao Tong University
China
zhang-w@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
China
chen-quan@cs.sjtu.edu.cn

Kaihua Fu
Shanghai Jiao Tong University
China
midway@sjtu.edu.cn

Ningxin Zheng
Microsoft Research Asia
China
Ningxin.Zheng@microsoft.com

Zhiyi Huang
University of Otago
New Zealand
hzy@cs.otago.ac.nz

Jingwen Leng
Shanghai Jiao Tong University
China
leng-jw@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Multi-stage user-facing applications on GPUs are widely-used nowadays, and are often implemented to be microservices. Prior research works are not applicable to ensuring QoS of GPU-based microservices due to the different communication patterns and shared resource contentions. We propose Astraea to manage GPU microservices considering the above factors. In Astraea, a microservice deployment policy is used to maximize the supported peak service load while ensuring the required QoS. To adaptively switch the communication methods between microservices according to different deployments, we propose an auto-scaling GPU communication framework. The framework automatically scales based on the currently used hardware topology and microservice location, and adopts global memory-based techniques to reduce intra-GPU communication. Astraea increases the supported peak load by up to 82.3% while achieving the desired 99%-ile latency target compared with state-of-the-art solutions.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Neural networks**; • **Networks** → **Cloud computing**.

KEYWORDS

Microservice, GPU, Resource management, QoS

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507721

ACM Reference Format:
Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: Towards QoS-Aware and Resource-Efficient Multi-stage GPU Services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507721

1 INTRODUCTION

User-facing services have strict quality-of-service (QoS) requirements, and many of them (e.g., Intelligent personal assistant [25], Graph processing [56], Conversational AI and Cyber Security [10]) rely on GPUs to provide high computational ability [7]. Due to the increasing complexity of real application scenarios, the services are becoming large and complex. For instance, many in-production image services [3, 4, 6, 66] on GPUs are implemented with "big models" including multiple neural networks or machine learning modules. Image caption [3, 4, 6] has two modules, an encoder using a deep convolutional neural network and a decoder based on a stack of LSTM layers [73]. Many AI-related services, e.g., text-to-text [53, 65], text-to-image [39, 66, 77, 84], text-to-speech [26, 61], text-to-video [31], image-to-image [17, 33, 74], image-to-text [48, 73], speech-to-text [27, 46, 54], the neural personalized recommendation [44, 45], are also implemented by connecting multiple AI modules. Besides, another type of multi-stage GPU applications is cloud gaming. GPUs in the cloud perform connected stages like game logic, remote rendering, and hardware encoding [42, 52, 62]. In addition, there are some large models (such as GPT-3 [21]), which cannot be placed in a single GPU. In this case, the model has to be split into several subgraphs (kernels).

For the multiple-stage user-facing services, stages are often implemented to be loosely coupled microservices [37]. A single microservice stage often under-utilizes a powerful GPU, hence some of the microservices will be deployed on a GPU especially when the load is low. Figure 1 shows an example where an application with a four-stage pipeline is deployed on three GPUs from two nodes.

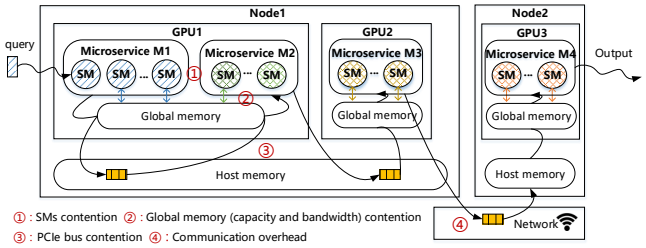

Figure 1: An example of deploying GPU microservices.

The tail latency required for each stage is strict due to the dependencies between microservices and their back pressure effects [37]. QoS violations occur when one or more critical microservice instances experience load spikes (diurnal or unpredictable workload patterns [19]) or shared-resource contention.

There are some research works on managing the latency of independent GPU applications [25, 28, 29, 43, 80, 81]. Clipper [28] adaptively batched the requests to improve the GPU throughput while maintaining tail latency. ClockWork [43] presented a runtime time-sharing scheduler for DL requests. Baymax [25] and Laius [80] ensured the QoS of a user-facing service when it co-runs with best-effort applications on time-sharing GPUs and spatial-sharing GPUs, respectively. However, ensuring the short latency of a single-stage may increase the latency of other stages due to the limited shared resources. They are not applicable for GPU microservices that have complex dependency relationships, ignoring the communication overhead between stages and the efficiency of the stage graph.

Prior works on scheduling CPU microservices [18, 36, 38, 51, 71] are also not adequate. Firstly, the communication overhead for CPU microservices occupies about 5% of the end-to-end latency (smaller than 2ms) for CPU microservices in DeathStarBench [37], but the counterpart between stages in a GPU service takes up to 45% (240ms) according to our measurement. In this case, CPU microservices use a unified heavy communication framework (e.g., RPC) through Network Interface Cards (NICs), and "randomly" schedule the microservices based on resource constraints. Secondly, the very limited GPU global memory constraints the deployment of GPU microservices, but the machine resources are often considered unlimited when deploying CPU microservices [63]. Moveover, GPU microservices contend for SMs (shown at ①), global memory capacity and bandwidth (shown at ②), and PCI-e bandwidth (shown at ③) that are not considered before. Thirdly, they relied on sophisticated resource isolation mechanisms (e.g., Cache Allocation Technology), but there is no such support on off-the-shelf GPUs.

This work aims to maximize the throughput of a multi-stage user-facing service with QoS guarantee. As the load of service varies and the contention situations are only known at runtime, an offline method is not applicable for effectively deploying GPU microservices. This work resolves this problem based on two main insights. 1) The communication overhead should be the first-class factor while deploying GPU microservices. Deploying the microservice graph based on the interconnect topology of GPUs matters for GPU microservices. 2) The optimal deployment changes dynamically due to factors such as real-time user load, performance/resource trade-offs, resource contention, and communication overhead.

We propose a runtime system named **Astraea** to achieve the above purpose. Astraea is composed of an *online performance predictive model*, a *microservice deployment policy*, and an *auto-scaling communication framework*. Without assuming resource isolation on GPU, Astraea uses the online predictive model to predict the shared global bandwidth usage, duration, and throughput of each GPU microservice under various resource configurations. Based on the prediction, Astraea dynamically tunes the microservice deployment. In Astraea, microservices communicate through the auto-scaling communication framework. When the deployment changes, the communication framework automatically scales based on the currently used hardware topology and microservice location. It also eliminates the expensive data transfer between main memory and global memory for the microservices on the same GPU, by proposing global memory-based communication.

This paper makes four main contributions.

- **Comprehensive characterization of GPU microservices.** The characterization helps address the challenges in managing GPU microservices.
- **An online microservice performance predictor.** The ML-based predictor can precisely predict global bandwidth usage, duration, and throughput of each GPU microservice under various resource configurations.
- **A lightweight deployment policy for GPU microservices.** The policy considers communication overhead, global memory capacity, shared resource contention, and pipeline stall when managing the GPU resources.
- **An auto-scaling GPU communication framework.** Similar to the unified communication frameworks, Thrift [2] and gRPC [11] for CPUs, the proposed framework enables auto-scaling without modifying the microservice source codes, no matter if the microservices are on the same GPU, different GPUs, or different nodes.

We implement Astraea and evaluate it on a GPU server with two Nvidia 2080Ti GPUs, and three DGX-2 machines with Nvidia V100 GPUs. Astraea effectively increases the supported peak load by up to 82.3% compared with Laius [80] and 45.1% compared with FIRM [63], while ensuring the required QoS.

2 RELATED WORK

There have been some efforts on related topics: GPU resource management and microservice architecture.

There have been some resource allocation or scheduling work for GPU co-location. TimeGraph [49] and EffiSha [23] used priority-based scheduling to guarantee the performance of real-time kernels on GPUs. Prophet [24] identified "safe" co-locations where the performance interference would not result in QoS violations. DART [76] employed a pipeline-based scheduling architecture with data parallelism, where heterogeneous CPUs and GPUs are arranged into nodes with different parallelism levels. Baymax and GrandSLAm [25, 47] reordered GPU kernels for ensuring QoS of a user-facing task at co-location on GPUs. Salus [78] and G-NET [79] designed effective task sharing models on GPUs. These queuing-based methods are not applicable to the modern spatial multitasking GPUs where kernels share SMs spatially.

There have been some prior works on eliminating QoS violations for CPU microservices. Bao et al. [18] analyzed the performance degradation of microservices and developed a workflow-based scheduler to minimize latency and improve utilization. Based on the workloads' characteristics, HyScale and ATOM [41, 50] designed resource controllers that combine horizontal and vertical scaling with dynamic resource division to improve the processing time of microservices. Considering the complexity of performance prediction, Seer [38] proposed an online performance prediction system. FIRM [63] and Sinan [82] presented ML-based frameworks for resource sharing across microservices based on isolation techniques for shared resources. They are not applicable to GPU microservices lacking isolation support on GPUs.

Aiming at a single RNN inference model instead of general microservices, BatchMaker [40] improved both the latency and throughput of RNN inference using cellular batching. But the cellular batching is not applicable to DNNs with fixed inputs such as CNNs and MLPs applications, while Astraea can handle various microservices. Nexus [69] provides a mechanism to breakdown end-to-end SLO into different stages in the context of DNN models. These works do not consider the dependency relationship between microservices as Astraea does. They ignored the characteristics of the microservice architecture, thus resulted in the inefficient pipeline and low resource utilization compared with Astraea.

3 PERFORMANCE ISSUES OF GPU MICROSERVICES

In this section, we characterize GPU microservices, show performance issues, analyze the shortcomings of existing work, and discuss the deployment challenges.

3.1 Benchmarks and Experimental Platforms

Table 1 lists the benchmarks that cover a wide spectrum of real multi-stage GPU applications. The benchmarks include six AI-based GPU microservices from AIbench [1], and a cloud gaming benchmark. The benchmarks reflect the real productions: *natural language processing* (text-to-text), *image processing* (img-to-img), *image generation* (text-to-img), *image caption* (img-to-text), *speech synthesis* (text-to-speech), *speech recognition* (speech-to-text), and *cloud gaming*. Since cloud gaming mainly involves rendering and image compression [42] and there is no standard benchmark, we build the cloud gaming benchmark using open-sourced implementations of rendering and compression [5, 9]. Nvidia Data Loading Library [14] is used for data preprocessing between microservices.

We use two Nvidia RTX 2080Ti GPUs as the experimental platform and choose four workloads from Table 1 to investigate the performance issues. Our study does not rely on any specific feature of 2080Ti, it can be applied to other GPUs.

3.2 Characterizing GPU Microservices

The latency of a multi-stage user-facing query is determined by the processing time of each stage and the communication time between the stages, the throughput is determined by the throughput of each stage. In order to maximize the supported throughput of service and ensure the required latency target, a microservice system should resolve four problems.

Table 1: Seven representative GPU microservices

Workload	Microservices	Algorithm	Language
Img-to-img [17]	Face recognition Image enhancement	FR-API [8] FSRCNN [33]	PYTHON& CUDA
Img-to-text [73]	Image feature extraction Image caption	VGG [70] LSTM	C++ & CUDA
Text-to-image[66]	Semantic understanding Image generation	LSTM [67] DC-GAN [64]	C++& CUDA
Text-to-text [65]	Text summarization Text translation	BERT [32] Opennmt [16]	PYTHON& CUDA
Text-to-speech [61]	Text-to-wave Waveform synthesis	Deep Voice 3 [32] WaveNet [16]	PYTHON& CUDA
Speech-to-text [46]	ASR Machine Translation	LSTM [67] LSTM [67]	PYTHON& CUDA
Cloud Gaming [12]	3D Rendering Image compression	RayTracing [5] FFmpeg [9]	C++& CUDA

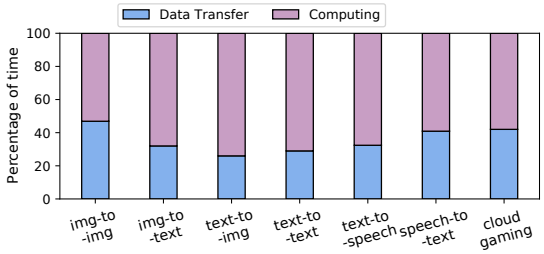

Figure 2: Breaking down the end-to-end latency.

1) Large communication overhead. GPU microservices communicate through the host main memory inside the node but through Network Interface Card (NIC) cross nodes. Taking *image-to-image* as an example, when the batchsize is 32, the microservice "face recognition" transfers 253.125MB data to the microservice "image enhancement". The latency increases by 17% when shifting inter-node deployment to cross-node deployment. Even if the microservices are on the same GPU, the communication is still expensive. In Figure 1, when m_1 sends the result to the next microservice m_2, its data is first transferred from the global memory to the host memory, and then transferred back to the global memory used by m_2. Figure 2 shows the breakdown of the end-to-end latencies of the benchmarks when the microservices are on the same node. The communication takes 26.4% to 46.9% of the latency. It is crucial to minimize the communication overhead when deploying multi-stage GPU microservices.

2) Require adaptive unified communication. The appropriate deployment of multi-stage service changes with different hardware. The appropriate deployment also changes with the load of a service in the scenario of minimizing resource usage. In this case, the upstream/downstream communication target and method(e.g., from main memory-based communication to NIC-based communication, and vice versa) vary. We need to take different communication patterns as the first-class factor during resource scheduling and deploying. A unified low overhead communication framework that scales according to the deployment without modifying the source code of microservices is required.

3) Limitation on global memory space. Figure 3 shows the global memory usage and the corresponding GPU utilization when the first microservice of *img-to-img* and the second microservice of *cloud gaming* use different batchsizes. As shown in the figure, the global memory of GPU is only able to host the microservice with a batchsize smaller than 256, where the GPU utilization is lower than

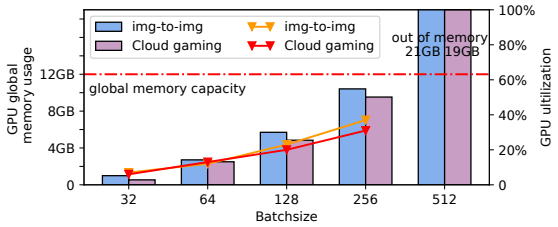

Figure 3: Global memory usage of the microservices (FR-API and FFmpeg in Table 1) with different batch sizes.

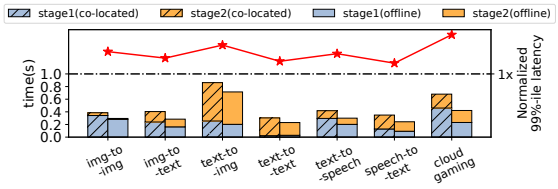

Figure 4: QoS violation with intuitive SM allocation policies.

37% and 35%. In this situation, we are not able to deploy multiple instances of a global memory-consuming microservice on a GPU.

4) Shared-resource contention. A service achieves the highest throughput when its stages have identical throughputs due to the pipeline effect. An intuitive policy is carefully allocating SMs based on the offline profiles of microservices so that the throughputs of the stages are identical, and the aggregated processing time is shorter than the QoS target.

However, the offline policy often results in the QoS violation. Figure 4 illustrates the QoS violation of benchmarks using this policy. In the figure, the stars represent the normalized 99%-ile latencies of the benchmarks (the right y-axis). The bars "stage1/2 (offline)" and "stage1/2 (co-located)" represent the offline-profiled and actual processing time of microservice stages respectively (the left y-axis). As observed, the actual processing time of the stages is longer than their offline time, and all the benchmarks suffer from QoS violations. This is mainly because the microservices on the same GPU contend for PCI-e bandwidth and global memory bandwidth, although the SMs are explicitly allocated. It is crucial to handle the unstable runtime contention due to the dynamic microservice deployment.

3.3 Deficiencies of Prior Work

Several prior works target similar problems. Laius [80] manages the SM allocation to ensure the QoS of a single application, while the performance of the co-located low priority applications can be sacrificed. It is not applicable for GPU microservices through simple adaption for three reasons. 1) It ignores the pipeline interaction between microservices. There is no support to breakdown the QoS of the entire service into the latency requirements of each microservice stage. 2) Ensuring the QoS of a microservice stage may result in the long processing time of other co-running microservices. 3) It does not consider the deployment of microservices across multiple GPUs even multiple nodes.

FIRM [63] and SINAN [82] minimize the resource usage of traditional CPU microservices while ensuring the required QoS. Since the communication overhead takes a minor part of the latency for CPU microservices (e.g. 5% in DeathStarBench [37]), all the microservices communicate through sockets based on NIC. They use Kubernetes to randomly deploy microservices after the resources are allocated without considering different communication patterns. On the contrary, optimizing the communication mechanism is crucial for the GPU microservices. In addition, the main memory can be easily scaled to meet the requirements for CPU microservices. FIRM [63] and SINAN [82] are more focused on fast resources (e.g., cores, LLC, memory bandwidth, main memory) reallocation between microservices. There are not such isolation techniques available on GPUs. And the global memory in the GPU is limited, which should be considered in the microservice deployment.

We also compare Astraea with Laius and FIRM through extensive experiments in Section 8.

3.4 Design Principles of Astraea

Based on the characterization study in Section 3.2, we design Astraea following four principles.

(1) The microservice deployment on GPUs should consider the different communication overheads. The communication overhead across nodes, across GPUs on the same node, and on a GPU show great gaps. Astraea should consider such differences when deploying GPU microservices, due to the relatively large data among stages. It is better to use the fewest nodes to host a service for the low cross-node overhead.

(2) The communication overhead between microservices on the same GPU should be reduced. The CPU-GPU data transfer between microservices results in the long end-to-end latency. The PCI-e bandwidth contention also leads to increased communication overhead and long latency.

(3) Microservices have to be scheduled across GPUs/nodes considering the limited global memory space. Since global memory space is one of the resource bottlenecks for a GPU, Astraea should be able to use multiple GPUs to host an entire multi-stage user-facing service.

(4) The microservice pipeline efficiency should be maximized while achieving the required QoS online. Since the pipeline efficiency is affected by both the percentage of SM resources allocated to each microservice and the runtime contention behaviors, Astraea considers runtime contention of the shared resources (e.g., global memory bandwidth).

4 THE ASTRAEA METHODOLOGY

Figure 5 shows an overview of the Astraea system. Astraea maximizes the supported peak load of a multi-stage GPU service while ensuring the desired 99%-ile latency target. To achieve this purpose, Astraea comprises an *online performance predictive model*, a *microservice deployment policy*, and an *auto-scaling communication framework*.

The performance model predicts the duration, and shared resource usage of a microservice stage if it is allocated a given number of SMs in a GPU (Section 5). Different from other performance models, our model also predicts the shared resource usage (e.g., the

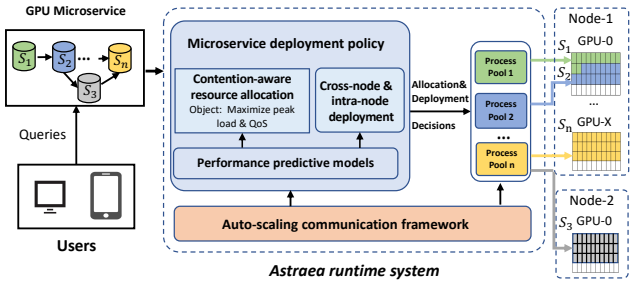

Figure 5: Design overview of Astraea.

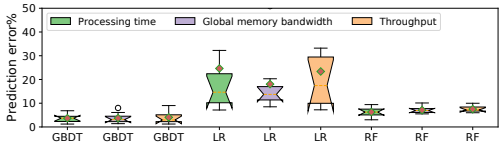

Figure 6: Errors of predicting duration, global memory bandwidth and throughput with GBDT, LR, and RF.

consumed global memory bandwidth). The shared resource usage is used to quantify the runtime contention between microservices on a GPU since only the SMs can be explicitly allocated. We also use the process pool technique [80] to enable dynamic SM allocation.

The microservice deployment policy distributes the microservices to the available GPUs, based on the predicted performance, the requirement on minimizing communication overhead, and the requirement on making the throughputs of the stages identical (Section 6). One challenging part here is that some microservices may not scale well with the number allocated to SMs or some GPUs only have small numbers of free SMs. We therefore propose to use both multi-instance and SM allocation techniques to balance the throughput of the stages. Another challenge is restricting the performance degradation caused by the runtime contentions, by finding an appropriate trade-off between the communication overhead and the resource contention on each GPU.

As the communication patterns between microservices vary with GPU and node interconnect topologies, the communication framework adapts the communication no matter the microservices are on the same GPU, on the same node, or on different nodes (Section 7). The communication framework also proposes global memory-based communication to eliminate unnecessary global memory-main memory data transfer.

Astraea first collects the performance data of each microservice online, and trains the performance predictor until its accuracy meets requirements. When the load/resource changes, Astraea predicts the performance and shared resource usage of each microservice under different resource configurations. According to the prediction, the deployment policy determines the percentages of SMs and the number of instances allocated to each microservice. At the same time, Astraea uses a communication-aware deployment strategy to deploy all instance while ensuring QoS target. Once the microservice deployment is determined, the auto-scaling communication framework will automatically select the appropriate upstream and downstream microservices to communicate based on the resource topology and load balancing, and choose the best communication method(through global memory, Nvlink, etc).

5 PREDICTING PERFORMANCE AND RESOURCE USAGE

The inefficient microservice pipeline is due to the contention of shared resources and the different throughputs of the microservice stages. Astraea predicts the *processing duration*, the *global memory bandwidth usage*, and the *throughput* of each microservice to support the efficient resource allocation policy. The task duration denotes the period time from when a user query is received to the time point when the microservice gets its result. The task throughput represents the number of queries that can be processed per second at each microservice. Besides, Astraea also predicts the FLOPs (floating point operations) and the required global memory space of the microservices with different workloads.

For each microservice, we use the microservice's *input parameters*, *input data size*, *batchsize* and *percentage of computational resources* as input features, since they dominate a microservice's performance. All the input features can be collected with Nsight systems [15] online on the backup server (an independent server).

More precisely, to build a performance model for a microservice, we submit queries with random inputs, execute them with different computational resource quotas and collect the corresponding input features online. In order to guarantee the accuracy of the performance model, Astraea collects online data continuously, incrementally updates the training set, and train the model until the accuracy of the model reaches a pre-defined threshold (e.g., 90%). We randomly select 80% of the samples to trains the model and use the rest to evaluate the accuracy of the trained model.

When a batch of new samples are obtained (e.g., 100 samples), we first predict its performance with the current ready-trained model. If the accuracy is higher than the threshold, the model training stops. Otherwise, the new batch of samples are used to further update the model incrementally. During online training, queries are executed in solo-run mode to avoid resource contention. Based on the prediction results, at most 200×10=2000 samples (200 different inputs and 10 different percentages of computational resources increasing from 10% to 100% with step 10%) are sufficient for each microservice, as the modeling method is not sophisticated.

Since the QoS target of a user query is within hundreds of milliseconds to support smooth user interaction [30], it is crucial to choose the modeling technique that shows both high accuracy and low complexity for online prediction. We evaluate a spectrum of low latency algorithms for the performance prediction: Linear Regression (LR) [68], Gradient Boosting Decision Tree (GBDT) [75], and Random Forest [20].

Figure 6 presents the prediction errors with LR, GBDT and RF after 60 minutes training. GBDT and RF show high accuracy (95.75% and 92.08%, respectively) for predicting the microservice performance. We also measured the execution time of different prediction models. The time of predicting with *GBDT* is shorter than 1*ms*, while the RF model runs longer than 5*ms*. We therefore choose GBDT as our performance modeling technique.

6 DEPLOYING MICROSERVICES

In this section, we discuss the resource allocation and microservice deployment in Astraea. We first determine the resource allocation for each microservice instance, and then identify an appropriate microservice deployment across nodes/GPUs.

6.1 Contention-Aware Resource Allocation

Based on our accurate features and precise prediction, we design a contention-aware resource allocation policy. Specifically, we first estimate the minimum number of GPUs based on the load pressure level to deploy a multi-stage service. Then, the SMs of the GPUs are allocated to each microservice.

Determine the minimum number of GPUs. When the load of a service is high, deploying its microservices on a single GPU leads to serious resource contention. Occupying multiple GPUs requires careful selection of the number of GPUs to avoid resource waste and minimize the expensive cross-node communication. In this case, Astraea uses the predicted number of floating-point operations and the global memory footprint of microservices to determine the minimum number of GPUs required as shown in Equation 1. The parameters are defined in Table 2. In the equation, the computational ability and the global memory space are the two factors to determine the number of required GPUs.

$$n = MAX(\frac{\sum_{i=1}^{n} C(i,s)}{G}, \frac{\sum_{i=1}^{n} M(i,s)}{F}) \quad (1)$$

Fined-grained resources allocation. Once the number of GPUs is determined, Astraea allocates resources to microservices.Directly allocating resources based on the performance prediction in Section 5 cannot ensure the QoS of GPU microservices due to the contention on shared resources.

We therefore formalize the resource allocation problem to be a single-objective optimization problem with multiple constraints on shared resource usage in Equation 2. The designed object function is to maximize the smallest throughput of microservices in a multi-stage service, while ensuring the end-to-end latency shorter than the QoS target. Without loss of generality, we consider user-facing applications with various microservice dependency graphs. Our design does not depend on any specific microservice dependency.

There are five constraints in the optimization problem. 1) The accumulated global memory bandwidth required by all the microservices on a GPU should be less than its available global memory bandwidth. 2) The accumulated SM quotas allocated to concurrent instances should not exceed the overall available SMs. 3) The number of microservice instances on a GPU should not exceed 48 (MPS allows at most 48 client-server connections per-device). 4) The global memory capacity should be smaller than the GPU global memory limit. 5) The latency required for the entire service should be smaller than the QoS target.

In a complex service, the latency of a user request is determined by the Critical Path (CP, the path of the maximal duration) of the execution graph. One problem here is that the CP of a service changes based on the performance of microservice instances, the allocated resources, and underlying shared-resource contention. It is not applicable to statically identify the CP of service through offline analysis. We therefore identify the CP of service online using the weighted longest path algorithm [55]. In the algorithm,

Table 2: The variables used in the optimization problem

Variable	Varible description	Provided by
A_i	The ith part of a multi-stage service A	Benchmarks
p_i	The resource quotas of the ith microservice	Section 6.1
s	The batchsize of Microservice A_i	scheduler
n	The total number of GPUs	Section 6.1
BW	The available global memory bandwidth	Nvprof
I	The maximal client CUDA contexts per-device	Volta MPS
R	The overall computational resources respectively	Nvprof
N_i	The number of the i-th microservice's processes	Scheduler
$f(p_i,s)$	The predicted throughput of A_i	Section 5
$b(p_i,s)$	The predicted global mem. BW usage of A_i	Section 5
$g(p_i,s)$	The predicted latency of A_i	Section 5
$M(i,s)$	The global mem. footprint of A_i with batch size s	Section 5
$C(i,s)$	The float point operations of A_i with batch size s	Section 5
G	The GFLOPS of the used GPU	Nsight compute
$Trans$	The communication overhead	Section 7
CP	End-to-end latency with CP	Section 6.1
F	The global memory capacity of the used GPU	Nvprof

we take into account the communication overhead ($Trans$) and computation patterns in microservice architectures.

$$\begin{aligned}
&Objective = && MAX(\min_{i=1}^{n} N_i \times f(p_i,s)) \\
&\text{Constraint-1:} && \textstyle\sum_{i=1}^{n} N_i \times p_i \le n \times R, \quad 0 \le p_i \le R \\
&\text{Constraint-2:} && \textstyle\sum_{i=1}^{n} N_i \le n \times I, \quad 0 \le N_i \le I \\
&\text{Constraint-3:} && \textstyle\sum_{i=1}^{n} N_i \times b(p_i,s) \le n \times BW \\
&\text{Constraint-4:} && \textstyle\sum_{i=1}^{n} N_i \times M(i,s)) \le n \times F \\
&\text{Constraint-5:} && CP(\textstyle\sum_{i=1}^{n} g(p_i,s), Trans) \le QoS
\end{aligned} \quad (2)$$

In Equation 2, we assume a service has n microservice stages. Table 2 lists the variables in Equation 2. $f(p_i,s)$, $b(p_i,s)$ and $g(p_i,s)$ are the predicted throughput, global memory bandwidth usage and latency of the i-th microservice when it is allocated p_i SM quotas respectively. The optimal number of instances for each microservice stage and the resource quotas can be obtained by solving Equation 2.

Resource allocation space exploration. Given the large resource allocation space in microservices, it is essential to quickly identify the boundaries of that space that allow the service to meet its QoS. Assuming we are deploying M microservice stages on n GPUs, the SM allocation granularity is P (in % of a GPU's SMs), and the maximum number of instances in each microservice stage is N. In this case, there are $C_{\frac{n}{P}-1}^{M-1} \times N^M$ possible allocation combinations. When $n = 10$, $M = 20$, $P = 10\%$, $N = 10$, there are 1.07×10^{40} possible resource allocation combinations. It is impossible to enumerate all the combinations with reasonable time at runtime.

Many algorithms can be used to solve optimization problems, such as heuristics approaches [34, 72] and reinforcement learning (RL). RL is recently widely used in CPU resource scheduling problems [58, 63]. We do not choose such a black-box method because of its data starvation, in which a large number of real samples are needed to achieve good results.

We use a heuristics approach that effectively avoids local optima to solve the problem. To enable quick convergence, we employ the Epsilon-Greedy algorithm, which is one of the widely adopted randomized greedy algorithms in this domain [57]. It has low overhead (10ms) according to our experiments. Even if the prediction was slightly inaccurate, a common compensation mechanism [83] can be adopted to solve it. However, there is no QoS violation caused by inaccurate prediction in our experiments.

6.2 Identifying Appropriate Deployment

Randomly deploying microservices like FIRM results in QoS violation due to the large communication overhead, although the above step finds the number of instances for each stage and the computing resource quotas needed for each instance.

As the communication across node show much higher overhead than intra-node, Astraea first minimizes the cross-node communication. Specifically, the cross-node deployment first allocates microservice stages to multiple servers if a server is not capable to host an excessive number of microservices. On each node, cross-GPU deployment deploys the instances of the allocated microservice stages to multiple GPUs.

Algorithm 1: Clustering of Microservice Stages

Input :The computing requirements of the n stages $C_1, \cdots C_n$
Input :Balanced distance matrix for the n stages X (nxn)
Output :The k clusters $A_1, \cdots A_k$
Parameter :The remaining computing power of the k nodes $R_1, \cdots R_k$

Initialize $R_1, \cdots R_k$ with the full computing power of the k nodes;
Select k inital stages which are far from each other;
Assign the inital stages to the corresponding k clusters $A_1, \cdots A_k$;
Update $R_1, \cdots, R_k$ according to the initial k clusters;
Put the rest $n-k$ stages into set $Rest$
while $Rest$ *is not* $\emptyset$ **do**
    Randomly select a stage i in $Rest$
    Select cluster A_j with $min(Dist_{i,A_j})$ and $R_j > C_i$;
    Add stage i into cluster A_j;
    Update $R_j \leftarrow R_j - C_i$;
    Remove stage i from $Rest$;
Return $A_1, \cdots A_k$;

6.2.1 Cross-Node Deployment. The principle here is to minimize the inter-node communication overhead. We formalize the cross-node deployment as a graph min-cut problem. Each microservice is a node in the graph and the communication between microserives corresponds to the edge between two nodes. The communication data sizes are used as the weights of the edges. Our policy tries to divide the graph into N (the number of nodes in the cluster) subgraphs while minimizing the aggregated weight of the broken edges. We therefore propose a Balanced-Kmeans++ algorithm optimized for the min-cut problem. This algorithm generates k clusters of microservice stages to be mapped to the k nodes. Algorithm 1 shows the details of the clustering algorithm for microservice stages. The algorithm adapts the Balanced-Kmeans++ with a constraint on the size of a cluster due to the limited computing power of a node. Compared with traditional Kmeans++, it also avoids assigning too many stages to one single node.

$$Dist_{i,A_j} = \begin{cases} \infty & otherwise \\ \Sigma_{m \in A_j} X_{i,m} & if\ \{m | X_{i,m} \neq \infty\} \neq \emptyset \end{cases} \quad (3)$$

To effectively locate the high communication overhead, we also carefully design the distance used in algorithm 1. First, the distance between stage i and j is defined as $X_{i,j} = 1/Com_{i,j}$ where $Com_{i,j}$ is the transferred data volume. Besides, the distance between stage i and cluster A is defined in Equation 3.

6.2.2 Intra-Node Cross-GPU Deployment. There are often many microservices to be deployed on each node. Exhaustively searching through the entire space for the optimal deployment brings long search time that reveals in the end-to-end latency of queries.

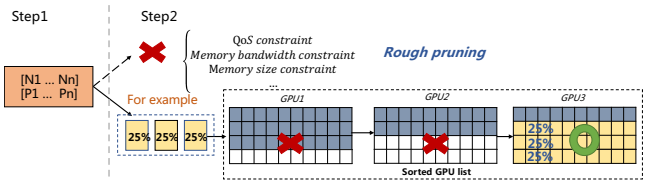

Figure 7: The microservice deployment scheme of Astraea.

We therefore propose a search strategy to quickly find out a reasonable deployment scheme through adaptive pruning as shown in Figure 7. The idea of pruning is finding the appropriate trade-off between inter-GPU communication and shared resource contention.

When deploying the instances of a microservice stage, if the data transmitted between two adjacent stages is particularly large, Astraea deploys the adjacent stage on the same GPU or topo-close GPUs (with smaller communication overhead). For the resource contention, we sort the remaining GPUs according to their available resources. The partial ordering of GPU resources is related to the characteristics of microservices. As the global memory capacity is a major resource bottleneck for GPU microservices, Astraea sets it as the highest priority in the deployment scheme.

Astraea prefers to deploy microservices on GPUs with fewer free resources. In this way, Astraea avoids excessive resource fragmentation available in the resource pool. Also deploying instances of the same stage on the same GPU as much as possible enables resource sharing among the instances from the same application, reducing the global memory consumption.

7 AUTO-SCALING COMMUNICATION FRAMEWORK

When the load of multi-stage service changes, the number of GPUs/nodes changes in consequence and the microservices are also dynamically deployed. For intra-node communications between microservice instances, servers (such as the DGX-1) have both inter-GPU point-to-point (P2P) interconnects such as NVLink (20-25GB/s) [35] and shared interconnects such as PCIe (8-12GB/s). For cross-node communications, however, Remote Procedure Call (RPC) is often used. An auto-scaling communication framework is required so that the microservices are able to freely communicate with each other without modifying the source code.

7.1 Unified Communication API

We propose a unified communication API for the programmers. With our API (Listing 1), developers only need to set a unique identifier for each stage with *SetStageIdentifier* and then specify the following stages by *AppendSuccessorStage*. The actual data is transferred through function *publish* and *receive*. Specifically, The auto-scaling communication framework includes a central controller. After the deployment, each instance will communicate with the central controller and send its location(which Node, which GPU) to the controller. The central controller will return the upstream and downstream services that each microservice should communicate with based on the location of all microservices and load balancing. The auto-scaling communication framework will bind the

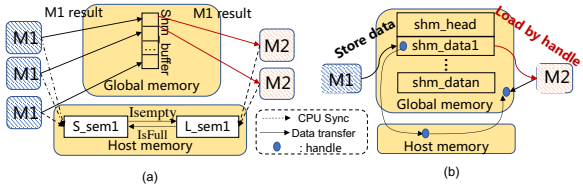

Figure 8: The global memory based communication.

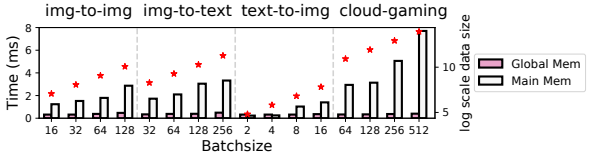

Figure 9: Communication overhead with the main memory-based and global memory-based mechanisms.

fastest communication channel (global memory-based or NVlink) to each instance. Once the deployment changes, the communication framework updates and rebinds the new communication channel automatically without any code change.

Listing 1: Interface of Unified Communication Mechanism

```
void SetStageIdentifier(string unique_id);
void AppendSuccessorStage(string unique_id);
void publish(void * data, size_t size);
void receive(void** pdata, size_t* size);
```

7.2 Optimizing Intra-GPU Communication

The microservice instances run in different processes. In this case, adjacent microservices (e.g., M_1 and M_2 in Figure 1) communicate by copying data back and forth between GPU global memory and the CPU main memory. However, the data passed from M_1 to M_2 is already in the global memory space of M_1, although M_2 is not allowed to access the data directly.

If M_1 is able to share the data with M_2, the expensive memcpy (device to host, host to device) can be eliminated, and the latency of a query can be further reduced. The PCI-e overhead caused by the memcpy is also eliminated. We propose and implement a global memory-based intra-GPU communication in our communication framework.

In the mechanism, the results of $M1$'s instances is temporarily stored in the global memory, only the data handler is passed to $M2$. The instances of $M2$ are able to access results from the global memory directly through the data handler. Figure 8 illustrates the design of the global memory-based communication. The mechanism can also be used for cross-GPU communication, if gpuDirect based on NVLink is enabled.

As shown in Figure 8(a), it is possible that there are multiple instances for both $M1$ and $M2$. To this end, a producer-consumer protocol is adopted to ensure the correctness of the communication. As shown, Astraea creates a shared buffer for each microservice pair and creates a store lock *S_Sem1* and a load lock *L_Sem1* to record the occupancy of the buffer. The locks are achieved through the main memory.

In more detail, when $M1$ passes its result to $M2$ on the same GPU, its process on the host passes a global memory handle (8 bytes) to the process of $M2$ on the CPU side (Figure 8(b)). Once $M2$ gets the data handle, it is able to directly access the data from the global memory. The transfer of data handle is implemented using the CUDA IPC [13]. The sender process gets the IPC handle for a given global memory pointer using cudaIpcGetMemHandle(), passes it to the receiver process using standard IPC mechanisms on the host side, and the receiver process uses cudaIpcOpenMemHandle() to retrieve the device pointer from the IPC handle.

Figure 9 shows the communication time of four benchmarks with the default and the global memory-based mechanisms. The two microservices do not contend for the PCI-e bandwidth. As observed, global memory-based communication greatly reduces the overhead when the data is larger than 0.02MB. The larger the data is, the more performance gain is achieved. If the data size is small (e.g., only 2 bytes), using memcpy takes a shorter time. This is because CUDA IPC incurs a small, fixed overhead when probing, transferring, and decoding the IPC handle in the global memory-based mechanism. For services that adopt large batchsizes, the transmitted data is usually larger than 0.02MB. A hybrid communication mechanism based on the size of the data is achievable if needed.

8 EVALUATION OF ASTRAEA

In this section, we evaluate Astraea in maximizing the supported peak load, while ensuring the required QoS.

8.1 Experimental Setup

We evaluate Astraea with the benchmarks in Table 1 on a machine equipped with two Nvidia RTX 2080Ti GPUs and three DGX-2 machines [60] equipped with V100 GPUs. Table 3 summarizes the software and hardware experimental configurations. Astraea does not rely on any special hardware features of 2080Ti or V100, and is easy to be set up on later GPUs (e.g., A100). The peak global memory bandwidths of 2080Ti and V100 are 616 GB/s and 897 GB/s [59]. Except the large scale evaluation in Section 8.5 and Section 8.6, the experimental results are from the machine with 2080Ti.

Load Generation. To represent a production environment, we provide an open-loop asynchronous workload generator to simulate users' requests. The arrival time of user requests follows an exponential distribution as in CPU microservices [37]. In Astraea, users' requests are executed in batches. If a request in the batch is about to be timed out, Astraea patches some empty requests into the batch, and starts pipeline inference in order to satisfy the QoS requirement.

Comparison Baselines. Since the multi-stage GPU service is a timely topic, we find few prior work. We therefore compare Astraea with state-of-the-art resource management system for CPU microservice *FIRM* [63] and the resource management work for GPU co-locations *Laius* [80]. When adapting FIRM for GPUs, we implement its idea to GPU microservices, and find the optimal resource allocation offline (i.e., the tuneable configurations, SMs and the number of instances) without QoS violation for each microservice. The optimal allocation is identified in a brute force way to eliminate the runtime overhead of reinforcement learning in FIRM.

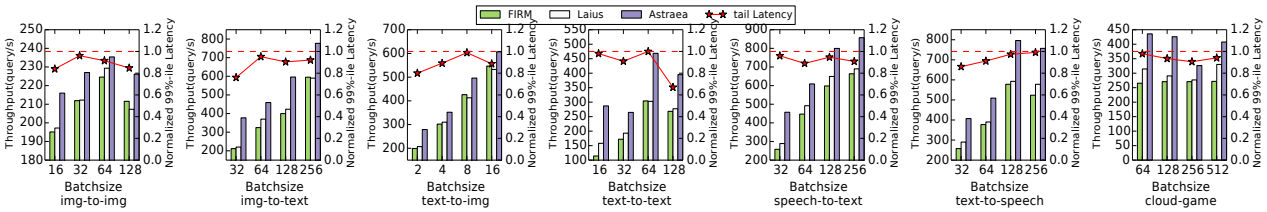

Figure 10: The throughput of the benchmarks with FIRM, Laius and Astraea. The stars show the normalized 99%-ile latencies of the benchmarks with Astraea (corresponding to the right y-axis).

Table 3: Hardware and software specifications.

	Specification
Hardware	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz Two Nvidia GeForce RTX 2080Ti
	Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz NVIDIA DGX-2 with 16 Tesla V100s-SXM3
Software	Ubuntu 16.04.5 LTS with kernel 4.15.0-43-generic CUDA Driver 410.78 CUDA SDK 10.0 CUDNN 7.4.2

We adapt Laius to microservices by taking the stage with the lowest throughput as the non-QoS task to maximize its throughput while ensuring the total latency of all stages. Since Laius is designed for a single GPU, for fair comparison, we schedule two microservices of a benchmark on one GPU using Laius. The total throughput of the benchmark for Laius is calculated as the minimum throughputs of all the GPUs. The QoS target of a user query is ranging from 50ms to 200ms according to 1.5X solo-run duration of each application and is defined as the end-to-end 99%-ile latency.

8.2 Maximizing Throughput and Guaranteeing QoS

Since a server is capable to host a benchmark, we first conduct experiments on a single server. Figure 10 shows the throughput of benchmarks with FIRM, Laius and Astraea, while ensuring the 99%-ile latency target. In this figure, x-axis shows the batchsize. The GPU throughput is defined as the supported peak load pressure for multi-stage applications while guaranteeing 99%-ile tail latency. Astraea increases the throughput of benchmarks by 15.5% to 45.1% compared with FIRM, and by 10.9% to 82.3% compared with Laius.

FIRM shows low throughput because it ignores the impact of communication overhead and resource contention when deploying the microservices. As FIRM randomly deploys the microservices on the GPUs of a single node, it incurs large communication overhead and unnecessary shared resource contention. In this case, to amortize the overhead, it allocates more resources to each microservice for speeding up the execution. The large communication overhead results in low throughput while ensuring the QoS. Meanwhile, Laius suffers from high PCI-e contention without global memory-based communication and ignores the pipeline efficiency. It achieves a slightly lower throughput compared with FIRM, because FIRM considers multiple instances (scale-out) while Laius only adjusts the resources of each microservice.

Table 4 shows the number of instances in each microservice stage, and the percentage of SMs allocated to each microservice instance with Astraea. For the stage that has a long processing time

Table 4: The detailed resource allocation with Astraea.

	img-to img	img-to text	text-to img	text-to text	speech to-text	text-to text	cloud gaming
batchsize1	(9, 10%) (1, 10%)	(2, 10%) (8, 10%)	(2, 10%) (8, 10%)	(1, 30%) (1, 20%)	(4, 20%) (2, 10%)	(8, 10%) (1, 20%)	(2, 30%) (2, 20%)
batchsize2	(9, 10%) (1, 10%)	(2, 10%) (8, 10%)	(1, 10%) (4, 20%)	(1, 30%) (1, 20%)	(4, 20%) (2, 10%)	(8, 10%) (1, 20%)	(2, 30%) (2, 20%)
batchsize3	(9, 10%) (1, 10%)	(1, 10%) (9, 10%)	(1, 70%) (1, 10%)	(1, 10%) (1, 70%)	(6, 10%) (3, 10%)	(9, 10%) (1, 10%)	(3, 20%) (10, 40%)
batchsize4	(4, 20%) (1, 10%)	(1, 10%) (9, 10%)	(1, 90%) (1, 10%)	(1, 10%) (1, 70%)	(6, 10%) (3, 10%)	(8, 10%) (1, 10%)	(3, 20%) (2, 20%)

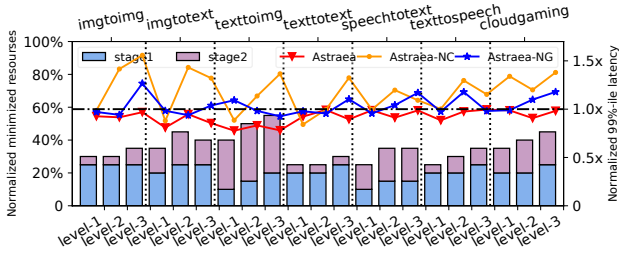

Figure 11: The resource usages of benchmarks under different load levels with Astraea, and the 99%-ile latencies of the benchmarks with Astraea, Astraea-NC and Astraea-NG.

(e.g., stage 1 for *img-to-img*), Astraea automatically creates more instances for it to increase the total throughput of the pipeline.

8.3 Considering Bandwidth Contention and Effect of Global Memory-based Communication

Astraea makes sure that the accumulated bandwidth usage of microservices is smaller than the peak global memory bandwidth of GPU. In this subsection, we verify the need of this constraint in deploying microservices (the constraints in Equation 2). We compare Astraea with Astraea-NC that disables the constraint in Astraea.

Figure 11 shows the 99%-ile latency of the benchmarks with Astraea-NC. According to this figure, Astraea-NC suffers from QoS violations in 14 out of the 21 test cases of the user-facing services. For instance, img-to-img suffers from up to 2X QoS violation at high load level (level-3). The QoS violations are due to the global memory bandwidth contention not being contained in Astraea-NC.

We also verify the need for the global memory-based communication. The blue line in Figure 11 presents the 99%-ile latency of

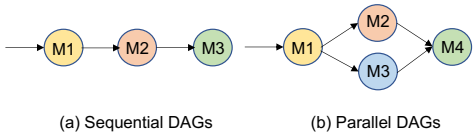

Figure 12: Structures of artificial multi-stage GPU services.

benchmarks with Astraea-NG, an Astraea variation that disables the global memory communication. Observed from Figure 11, 9 out of the 21 test cases suffering from QoS violations with Astraea-NG. The QoS violation is due to the large communication overhead between host memory and global memory.

In addition, Figure 11 reports Astraea's resource usages and the corresponding 99%-ile latencies under three loads, where the load level i is higher than j, if $i > j$. According to this figure, Astraea reduces more resource usage when the load is lower, but always guarantee the required QoS. Therefore, Astraea can finely tune the resource allocation and deployment based on different loads and the contention between the microservices on the same GPU.

8.4 Generalizing for Complex Microservices

To show the generalization of Astraea, we design several artificial benchmarks including the artificial three-stage benchmarks and the benchmarks with a complex interconnect graph. Figure 12 shows the structures of the graphs in this section. The first set of benchmark is based on three PCI-e intensive, compute-intensive and memory-intensive scientific workloads in Rodinia [22]. The intensities of compute intensive microservices and memory intensive microservices is configurable.

We create 3×3×3 = 27 workloads using the artificial benchmarks (3 microservices with different compute intensities, 3 microservices with different memory access intensities, and 3 microservices with different PCI-e intensities) to evaluate Astraea for complex microservices (Figure 12(a)). They are denoted by c_1, c_2, c_3, m_1, m_2, m_3, p_1, p_2, and p_3. $c_i/m_i/p_i$ is more PCIe/compute/memory intensive than $c_j/m_j/p_j$, if $i > j$.

Figure 13 shows the throughput of the 27 artificial benchmarks. Astraea improves the throughput of the benchmarks by 37.8% compared to FIRM, and by 39.7% compared with Laius. Figure 14 shows the resource allocation of Astraea, where it launches different numbers of instances of microservices, and allocates different percentages of the SMs to them. For example, Astraea launches 1 instance of *p1*, 2 instances of *c1*, and 5 instances of *m1* for the 1st benchmark. Astraea also allocates different percentages of the SMs to the same microservice in different benchmarks. It reveals Astraea can automatically adjust the resource allocation and deployment.

The second set of artificial benchmarks with complex graphs in Figure 12(b) is based on one PCI-e intensive, two compute-intensive and one memory-intensive workloads in Rodinia. We create $3 \times 2 \times 1 \times 3 = 18$ workloads and Figure 15 shows the throughput of benchmarks, where Astraea significantly improves the throughput by 38.1% and 32.1% compared to FIRM and Laius without QoS violations.

Astraea is applicable to complex microservices because it does not rely on specific features of the applications.

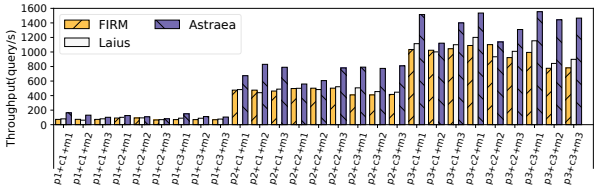

Figure 13: The throughputs of the artificial benchmarks with FIRM, Laius, and Astraea.

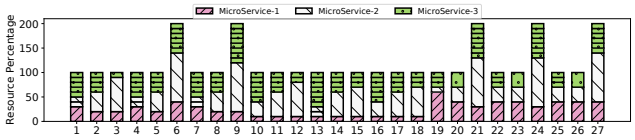

Figure 14: Resource allocation for maximizing the throughput of the benchmarks with Astraea.

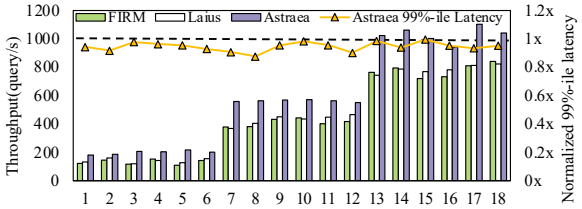

Figure 15: The throughputs of the benchmark with the complex graph with FIRM, Laius and Astraea and the corresponding 99%-ile latencies in Astraea.

8.5 Large Scale Evaluation on DGX-2

Astraea is evaluated on a DGX-2 machine for maximizing the throughput. Figure 16 shows the throughput of benchmarks, with the ensured 99%-ile latency target, where the x-axis shows the batch sizes. Astraea increases the throughput by 31.2% and 40.2% compared with FIRM and Laius. It shows Astraea is scalable on large-scale GPU machines.

High-end GPU servers, such as DGX-2, support GPU-to-GPU direct communication based on Nvidia NVLink [35], without using PCI-e bus. We also evaluate Astraea on DGX-2 with GPU-to-GPU direct communication enabled. Figure 17 shows the throughput of Astraea at maximum batchsize with GPU-to-GPU communication enabled. Comparing Figure 16 with Figure 17, Astraea's throughput is increased by 26.4% on average benefiting from the GPU-to-GPU communication. However, FIRM and Laius cannot benefit from the GPU-to-GPU communication as they do not manage the deployment of microservices across multiple GPUs.

8.6 Scheduling Across Multiple Servers

To evaluate the effectiveness of Astraea's cross-node deployment, we evaluate Astraea with three DGX2 servers. For each node, we use two V100 GPUs. We stretch the artificial benchmark above into

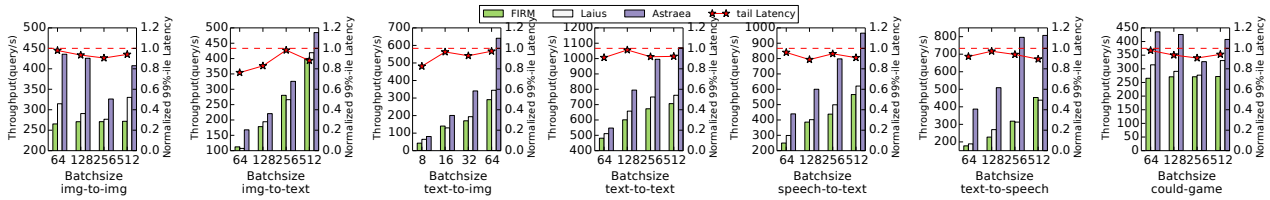

Figure 16: The throughput of the benchmarks on DGX-2 with FIRM, Laius and Astraea. The stars show the normalized 99%-ile latencies of the benchmarks with Astraea (corresponding to the right y-axis).

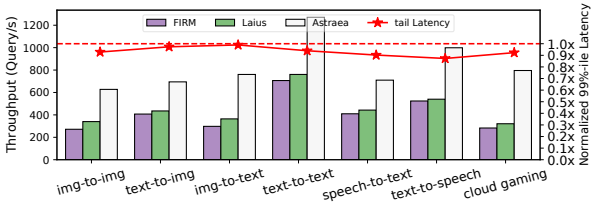

Figure 17: Maximizing throughput with NVlink.

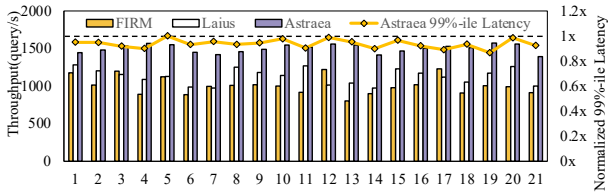

Figure 18: The latency and throughput results of the benchmarks on multiple servers with FIRM, Laius and Astraea.

the application with 21 microservices, including multiple copies of PCIe/memory/compute-intensive tasks. And we randomly arrange these microservices into 21 combinations. For communication between servers, we copy the shared data from the GPU to the CPU memory and then use gRPC technology [11] to establish a TCP connection between microservices.

As to the setting of baselines, for each combination, FIRM deploys microservices on different nodes using the default policy in Kubenetes. Laius first manually finds the segmentation method with the least communication overhead, and then each server still uses the previous Laius for resource management.

As shown in Figure 18, Astraea achieves higher throughput than FIRM and Laius by 47.2% and 32.9%. On average, Astraea improves the throughput by 30.2% compared with Laius while maintaining the QoS. Observed from Figure 15 and 18, FIRM performs worse when the microservice has a complex topology or more pipeline stages. That is because FIRM ignores the impact of communication overhead on the microservice deployment. Random deployment causes resource contention and much communication overhead.

8.7 Overhead of Astraea

Training overhead. The overhead of training models for predicting microservice performance is acceptable. Collecting the training samples of all microservices and training process finished within 60 minutes using a single GPU. As for the online predicting, each prediction completes in 1 ms, which is much shorter than the QoS target of a service. **Resource allocation overhead**. As stated in Section 6, Astraea needs to solve the optimization problem using the simulated annealing algorithm to identify the appropriate resource allocation and solve the deployment scheme using balanced-Kmeans++. Our measurement shows that this operation in our experiments completes in 10ms on a single CPU. We also measured the overhead on a large-scale GPU microservice-based application (50 microservices). Our scheduler can still complete within 30ms. **Communication overhead**. Astraea needs to set up global memory-based communication for microservices that require data transfer. The setup operation for a pair of microservices using CUDA IPC is only one-off when the end-to-end service is launched and completes in 1ms. To conclude, the overhead of Astraea is acceptable for real-system deployment.

8.8 Architectural Implications

We discuss three architectural implications in this subsection.

(1) If we build a datacenter for lightweight multi-stage services, it is more efficient to use servers that equip with a single or a small number of GPUs. For lightweight services, Astraea tends to run all the microservice stages on a single GPU and use multiple GPUs for independently responding to user requests (scale-out operation). The global memory-based communication effectively works in this situation. On the contrary, servers with multiple GPUs are more suitable for the services that cannot be held by a single GPU, as cross-node communication is time-consuming. In this case, Astraea prefers to deploy adjacent stages with large communication data to the same GPU/node.

(2) Weight-sharing technology can be used to reduce the global memory footprint when multiple instances of microservice stages are on the same GPU. However, it is not always the best to deploy the instances of a microservice stage on the same GPU. This is because these instances require the same types of shared resources, and may bring serious resource contention. There is a trade-off between global memory consumption and other shared resource contention.

(3) If a single model is extremely huge (e.g. GPT3 [21]), we can split the model where the operators show different characterizations, take some adjacent layers/ops as one microservice stage, and apply Astraea to deploy the entire model.

9 CONCLUSION AND FUTURE WORK

We propose Astraea to manage resources online for GPU microservices. The resource allocation and deployment policy consider the pipeline efficiency, the shared resource contention, and the communication overhead. An auto-scaling communication framework with the global memory-based communication mechanism is also proposed to enable effective GPU microservice communication. Experimental results show that Astraea increases the peak supported load by up to 82.3% while achieving the desired 99%-ile latency target compared with FIRM and Laius. Managing applications that have both CPU and GPU components (though lacking in benchmarks) would be interesting future work.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2019YFF0302600, and National Natural Science Foundation of China (NSFC) grant (62022057, 61832006, 61632017, 61872240, 62072297). Quan Chen and Minyi Guo are the corresponding authors.

REFERENCES

[1] [n. d.]. AIBench: A Datacenter AI Benchmark Suite. https://www.benchcouncil.org/AIBench/.
[2] [n. d.]. Apache Thrift. https://thrift.apache.org..
[3] [n. d.]. Automatic Alternative Text. https://wordpress.org/plugins/automatic-alternative-text/.
[4] [n. d.]. Best Caption For Facebook and Instagram. https://play.google.com/store/apps/details?id=com.caption.facebook.instagram&hl=en_US.
[5] [n. d.]. bgfx - Cross-platform rendering library. https://github.com/bkaradzic/bgfx.
[6] [n. d.]. Caption AI :Captions and Hashtags for Instagram/FB. https://play.google.com/store/apps/details?id=caption.ai&hl=en_US.
[7] [n. d.]. The datacenter has an appetite for GPU compute. https://www.nextplatform.com/2020/02/15/the-datacenter-has-an-appetite-for-gpu-compute/.
[8] [n. d.]. Facial recognition api for Python. github.com/ageitgey/face_recognition.
[9] [n. d.]. FFmpeg:A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/.
[10] [n. d.]. GPU in AI & Machine Learning Use Cases. https://www.weka.io/blog/gpu-for-ai-ml-deep-learning/.
[11] [n. d.]. gRPC. https://www.grpc.io/.
[12] [n. d.]. Moonlight. https://moonlight-stream.org/.
[13] [n. d.]. NVIDIA CUDAAPI. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html.
[14] [n. d.]. NVIDIA DALI. https://github.com/NVIDIA/DALI.
[15] [n. d.]. Nvidia Night Compute. docs.nvidia.com/nsight-compute/NsightCompute/index.html.
[16] [n. d.]. OpenNMT: An open source neural machine translation system. opennmt.net/.
[17] Yancheng Bai, Yongqiang Zhang, Mingli Ding, and Bernard Ghanem. 2018. Finding tiny faces in the wild with generative adversarial network. In *the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 21–30. https://doi.org/10.1109/CVPR.2018.00010
[18] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. 2019. Performance Modeling and Workflow Scheduling of Microservice-based Applications in Clouds. *IEEE Transactions on Parallel and Distributed Systems* (2019). https://doi.org/10.1109/TPDS.2019.2901467
[19] Luiz André Barroso and Urs Hölzle. 2009. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 4, 1 (2009), 1–108. https://doi.org/10.2200/S00516ED2V01Y201306CAC024
[20] Gérard Biau and Erwan Scornet. 2016. A random forest guided tour. *Test* 25, 2 (2016), 197–227.
[21] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54. https://doi.org/10.1109/IISWC.2009.5306797
[23] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–16. https://doi.org/10.1145/3018743.3018748
[24] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 17–32. https://doi.org/10.1145/3037697.3037700
[25] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices* 51, 4 (2016), 681–696. https://doi.org/10.1145/2872362.2872368
[26] Yutian Chen, Yannis Assael, Brendan Shillingford, David Budden, Scott Reed, Heiga Zen, Quan Wang, Luis C Cobo, Andrew Trask, Ben Laurie, et al. 2018. Sample efficient adaptive text-to-speech. *arXiv preprint arXiv:1809.10460* (2018).
[27] Yu-An Chung, Wei-Hung Weng, Schrasing Tong, and James Glass. 2019. Towards unsupervised speech-to-text translation. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 7170–7174. https://doi.org/10.1109/ICASSP.2019.8683550
[28] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.
[29] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1145/3458817.3476143
[30] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794
[31] Kangle Deng, Tianyi Fei, Xin Huang, and Yuxin Peng. 2019. IRC-GAN: introspective recurrent convolutional GAN for text-to-video generation. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 2216–2222. https://doi.org/10.24963/ijcai.2019/307
[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://doi.org/10.18653/v1/n19-1423
[33] Chao Dong, Chen Change Loy, and Xiaoou Tang. 2016. Accelerating the super-resolution convolutional neural network. In *European conference on computer vision (ECCV)*. Springer, 391–407. https://doi.org/10.1007/978-3-319-46475-6_25
[34] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE computational intelligence magazine* 1, 4 (2006), 28–39.
[35] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37
[36] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. 2021. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 932–941. https://doi.org/10.1109/IPDPS49936.2021.00102
[37] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 3–18. https://doi.org/10.1145/3297858.3304013
[38] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 19–33. https://doi.org/10.1145/3297858.3304004
[39] Lianli Gao, Daiyuan Chen, Jingkuan Song, Xing Xu, Dongxiang Zhang, and Heng Tao Shen. 2019. Perceptual Pyramid Adversarial Networks for Text-to-Image Synthesis. (2019). https://doi.org/10.1609/aaai.v33i01.33018312
[40] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15. https://doi.org/10.1145/3190508.3190541
[41] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *the 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004. https://doi.org/10.1109/ICDCS.2019.00197
[42] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. 2015. dJay: enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit GPU load balancing. In *Proceedings of the sixth ACM symposium on cloud computing*. 58–70. https://doi.org/10.1145/2806777.2806942

[43] 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.

[44] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. 2020. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 982–995. https://doi.org/10.1109/ISCA45697.2020.00084

[45] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[46] Ye Jia, Melvin Johnson, Wolfgang Macherey, Ron J Weiss, Yuan Cao, Chung-Cheng Chiu, Naveen Ari, Stella Laurenzo, and Yonghui Wu. 2019. Leveraging weakly supervised data to improve end-to-end speech-to-text translation. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 7180–7184. https://doi.org/10.1109/ICASSP.2019.8683343

[47] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16. https://doi.org/10.1145/3302424.3303958

[48] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*. 3128–3137. https://doi.org/10.1109/CVPR.2015.7298932

[49] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*. 17–30.

[50] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. 2019. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *the 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 80–90. https://doi.org/10.1109/ICDCS.2019.00017

[51] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. 2019. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software* 157 (2019), 110380. https://doi.org/10.1016/j.jss.2019.07.008

[52] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. 2019. GAugur: Quantifying performance interference of colocated games for improving resource utilization in cloud gaming. In *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*. 231–242. https://doi.org/10.1145/3307681.3325409

[53] Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345* (2019). https://doi.org/10.18653/v1/D19-1387

[54] Yuchen Liu, Jiajun Zhang, Hao Xiong, Long Zhou, Zhongjun He, Hua Wu, Haifeng Wang, and Chengqing Zong. 2019. Synchronous Speech Recognition and Speech-to-Text Translation with Interactive Decoding. *arXiv preprint arXiv:1912.07240* (2019).

[55] Ming Lu and Heng Li. 2003. Resource-activity critical-path method for construction planning. *Journal of construction engineering and management* 129, 4 (2003), 412–420. https://doi.org/10.1061/(ASCE)0733-9364(2003)129:4(412)

[56] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *the ACM International Conference on Management of data (SIGMOD)*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[57] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid task manager for latency-critical cloud workloads. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 409–420. https://doi.org/10.1109/HPCA.2017.13

[58] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. 2020. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 167–179. https://doi.org/10.1109/HPCA47549.2020.00023

[59] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://www.nvidia.com/en-us/data-center/nvidia-ampere-gpu-architecture/.

[60] NVIDIA. 2019. NVIDIA DGX-2 System User Guide. docs.nvidia.com/dgx/dgx2-user-guide/index.html.

[61] Wei Ping, Kainan Peng, and Jitong Chen. 2018. Clarinet: Parallel wave generation in end-to-end text-to-speech. *arXiv preprint arXiv:1807.07281* (2018).

[62] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. 2014. VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 2 (2014), 1–25. https://doi.org/10.1145/2632216

[63] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 805–825.

[64] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[65] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[66] Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. 2016. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396* (2016).

[67] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *the Fifteenth annual conference of the international speech communication association*.

[68] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 329. John Wiley & Sons.

[69] 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. https://doi.org/10.1145/3341301.3359658

[70] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[71] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *the 46th International Symposium on Computer Architecture (ISCA)*. 513–526. https://doi.org/10.1145/3307650.3322227

[72] Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.

[73] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. In *the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*. 3156–3164. https://doi.org/10.1109/CVPR.2015.7298935

[74] Fei Wang, Liren Chen, Cheng Li, Shiyao Huang, Yanjie Chen, Chen Qian, and Chen Change Loy. 2018. The devil of face recognition is in the noise. In *the European Conference on Computer Vision (ECCV)*. 765–780. https://doi.org/10.1007/978-3-030-01240-3_47

[75] Zeyi Wen, Jiashuai Shi, Bingsheng He, Jian Chen, Kotagiri Ramamohanarao, and Qinbin Li. 2019. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2706–2717. https://doi.org/10.1109/TPDS.2019.2920131

[76] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. In *Real-Time Systems Symposium (RTSS)*. IEEE, 392–405. https://doi.org/10.1109/RTSS46320.2019.00042

[77] Guojun Yin, Bin Liu, Lu Sheng, Nenghai Yu, Xiaogang Wang, and Jing Shao. 2019. Semantics Disentangling for Text-to-Image Generation. In *the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2327–2336. https://doi.org/10.1109/CVPR.2019.00243

[78] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610* (2019).

[79] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-net: Effective {GPU} sharing in {NFV} systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 187–200.

[80] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 58–68. https://doi.org/10.1145/3330345.3330351

[81] Wei Zhang, Kaihua Fu, Ningxin Zheng, Quan Chen, Chao Li, Wenli Zheng, and Minyi Guo. 2021. CHARM: Collaborative Host and Accelerator Resource Management for GPU Datacenters. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 307–315. https://doi.org/10.1109/ICCD53106.2021.00056

[82] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. (2021). https://doi.org/10.1145/3445814.3446693

[83] Liang Zhou, Laxmi N Bhuyan, and KK Ramakrishnan. 2020. Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 637–349. https://doi.org/10.1109/MICRO50266.2020.00059

[84] Minfeng Zhu, Pingbo Pan, Wei Chen, and Yi Yang. 2019. Dm-gan: Dynamic memory generative adversarial networks for text-to-image synthesis. In *the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5802–5810. https://doi.org/10.1109/CVPR.2019.00595