Making the Most of Serverless Accelerators

Aditya Ramkumar

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-88
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-88.html

May 13, 2022

Making the Most of Serverless Accelerators

by Aditya Ramkumar

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Joseph Gonzalez

Professor Joseph Gonzalez
Research Advisor

5/13/2022

(Date)

* * * * * * *

Natacha Crooks

Professor Natacha Crooks
Second Reader

5/13/2022

(Date)

Making the Most of Serverless Accelerators

by

Aditya Ramkumar

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Gonzalez, Chair
Professor Natacha Crooks

Spring 2022

Abstract

Making the Most of Serverless Accelerators

by

Aditya Ramkumar

Master of Science in Computer Science

University of California, Berkeley

Professor Joseph Gonzalez, Chair

Serverless computing has the potential to make application deployment over cloud resources more seamless and scalable. However, most work from major cloud providers has been focused around easily divisible resources like CPUs and local DRAM. Since many applications like ML inference benefit from specialized hardware like GPUs, incorporating them into the serverless setting will unlock new use cases. However, serverless GPUs come with a unique set of challenges: price, cold starts, dealing with memory, etc. In this thesis, I address some theoretical and practical issues that prevent serverless GPUs from becoming a reality.

Due to their utility, GPUs in a serverless setting are often highly in demand and limited in supply leading to overwhelmed systems. Users of these GPUs often have tight cost or latency constraints (for example, inference for self-driving cars). In this thesis, I explore scheduling policies to efficiently allocate GPU resources to requests based on user-provided Service Level Objectives. I further consider a heterogenous set of resources (both CPUs and GPUs), and explore how policies rooted in admission control can prevent the system from being overwhelmed.

While exploring GPU scheduling in a FaaS setting, there are some inherent system limitations. Most existing solutions require that applications carefully design their tasks to manually share resources and clean up properly when they finish, providing overhead for application developers and leaving scope for inefficient resource utilization. Another challenge is that when initializing a new accelerator resource, there is significant startup latency due to container and language runtime initialization. Recently, Nathan Pemberton proposed a new Kernel as a Service (KaaS) paradigm, where the system is responsible for managing GPU memory and schedules user kernels across the entire pool of available GPUs rather than relying on static allocations. Since resources are managed at the system-level with KaaS, it opens up a new set of challenges around request scheduling. I explore various policies, and evaluate them for metrics like cold start minimization, average wait time, and fairness among other metrics.

Contents

List of Figures

List of Tables

Acknowledgments

Thank you Nathan Pemberton for being an fantastic research mentor during my master's program. I deeply appreciate the guidance on technical and writing matters, and would not have been able to complete this work without your support. Thank you Professor Joseph Gonzalez and Professor Natacha Crooks for being incredible advisors/readers for my research and thesis. I admire the way both of you think about open questions and problems, and hope to develop some of those skills in myself. Thank you Vikram Sreekanti for opening the door for research to me. You got me excited about this work and exploring research. Finally, thanks to my peers, friends, and family for the constant support and motivation.

Chapter 1

Introduction

The increased performance of specialized computer hardware such as GPUs and TPUs over the past few years has led to orders of magnitude more powerful machine learning applications from self-driving cars, to better recommendation systems. The machine learning lifecycle can be roughly broken down into two phases: model training and model inference. Model training is the computationally expensive process of finding the right parameters to fit a certain neural architecture given training data, and is throughput-oriented. Model inference is the process of using a trained model to make predictions given input data. As compared to training, this process has tighter latency and cost constraints. The same model is potentially evaluated many times with different data; however, it takes roughly the same amount of time for each evaluation.

While most of the focus of machine learning systems research has been around model training, there are a number of challenges around the process of inference, particularly around latency, throughput, and resource utilization. Clipper, a prediction serving framework developed at UC Berkeley's RISE Lab, uses user-friendly abstractions and optimizations like caching and adaptive batching to better support model serving [2]. However, it relies on static allocations of resources (a "serverful" model) and has drawbacks due to inefficient resource utilization, limited scalability, and poor statistical multiplexing. In particular, a Clipper server always needs to be up and running with the models that could be queried; if Clipper decides to allocate specialized hardware to a particular model, the user will need to pay the cloud provider for this as well. Given its server-based architecture, Clipper needs to estimate the peak load for every model and allocate resources accordingly. If a particular model is not used, the user is paying for server resources that they don't need. And if a particular model's request frequency increases, a fixed size server or set of resources would not be able to handle the increase in load.

An execution model that promises to solve these problems is serverless computing. Serverless computing has made it easier than ever to deploy applications without having to manage the physical infrastructure, driving increased revenue for cloud providers. Over the past

decade, we have witnessed an increased reliance on public cloud infrastructure with the rise of AWS, Microsoft Azure and Google Cloud Platform [15]. Application developers typically reserve a few dedicated servers (EC2 instances in the case of AWS) to service their workloads. While there are tools that help automate the process, developers need to manage their cluster by accounting for node failures, ensuring correct software dependencies, setting up auto scaling, worrying about network/port configurations, and databases for external storage, etc. Moreover, servers need to be provisioned for the peak workload, leaving them underutilized for a majority of the time. Serverless systems, like AWS Lambda, Azure Functions and Cloudburst, attempt to solve this problem by having users specify a DAG of functions that they want executed instead of worrying about resource management - and the system leverages the principles of statistical multiplexing to increase resource utilization.

Clipper stands to benefit from the serverless architecture due to model serving's potentially bursty workload and short execution time. I generalize some of the ideas proposed in Clipper to a serverless setting, roughly modeling the Cloudburst FaaS architecture. However, there still is an issue surrounding system capacity; we don't want to accept more requests than we can handle, since that would lead to every request having a high time to service/latency. So, for the FaaS setting, we aim to address a core issue of multiresource serverless scheduling with different resource types (like CPU/GPU): how can we hit user defined service level objectives (SLOs), minimize cold starts and maximize throughput? Essentially, we want to ensure the system is only taking on requests it can handle by maintaining a constrained request queue to avoid cascading SLO misses.

Our serverless approaches to model serving improved our ability to adapt to variable resource demands and gracefully react to excessive load. However, the underlying serverless systems struggle to effectively utilize GPU resources. For example, the Ray and Cloudburst serverless/distributed computing frameworks support GPU-enabled remote functions (called tasks in Ray), but require that applications carefully design their tasks to manually share resources and clean up properly when they finish. In practice, this is difficult to achieve, suggesting that there would be utility to a system-managed solution. Another challenge is cold starts, which happen because of container and language runtime initialization. Traditional strategies used with CPU instances to handle them don't transfer well to GPU instances, because it's too expensive to keep models warm on a GPU when not being used.

These challenges are addressed by the recently proposed Kernel-as-a-Service (KaaS) model [9]. KaaS explicitly decouples host functions from GPU kernels at the system level. Following the principles of serverless computing, our system takes user descriptions of GPU functionality rather than providing GPU allocations to them directly, putting the responsibility of GPU memory management and scheduling on the system (so it can make optimal use of these expensive resources). In this work, I explore various scheduling algorithms and policies for how the system can allocate requests onto a set of heterogeneous resources. In particular, I look into scheduling policies like Round Robin, Balance, Shortest Time Remain-

ing First (SRTF), and Multi-Level Feedback Queue Scheduling (MLFQS). I also look into an exclusive policy where the system assigns resources directly to certain clients, and rebalances those resources if necessary (similar to a managed version of FaaS). I discuss the merits and drawbacks of each of these policies, particularly in the context of completion time, number of cold starts and fairness.

Chapter 2

Background

2.1 Serverless Computing

In traditional application development and cloud computing models, users typically allocate a certain number of machines and resources to service their tasks. In this model, the application developer estimates their peak resource usage, and allocates accordingly. One downside of the traditional serverful model is that the application developer may misjudge their load; too many resources allocated would mean additional unnecessary cloud spending, and too few resources would mean they wouldn't be able to service all requests. Furthermore, request frequency and load varies over time, meaning that even if resources are provisioned perfectly for peak usage, there would still be unnecessary usage and spending during non-peak times. This problem can be addressed with autoscaling, but this comes with its unique set of challenges. Autoscaling is slow since it involves booting a whole VM, and involves configuration from application developers. Additionally, it still doesn't address multi-resource waste since it's hard to right-size the individual servers.

A solution that promises to solve these problems is serverless computing. In this model, users provide a particular function or set of functions, which are executed by the cloud provider on (typically) lazily allocated resources. Serverless computing avoids explicit provisioning of resources in favor of time-bounded invocations of functionality. Users ask for some function or service to be invoked without considering how or where it will run. These functions are typically narrow in purpose and use few resources, favoring multiple invocations rather than a single large function. When the function has completed, the resources are freed. Often, functions are user defined, called Function-as-a-Service (FaaS), though functionality may also take the form of scalable services like a database.

Serverless computing doesn't come without its drawbacks. Infrequently-used serverless code may suffer from greater response latency than code that is continuously running on a dedicated server, virtual machine, or container. This is because, unlike with autoscaling, the

cloud provider typically "spins down" the serverless code when not in use and reallocates resources to other clients. When the serverless function is invoked again, the container and runtime need to be re-initialized often through VM or container forking, leading to significant additional latency [6]. This is known as the cold start problem. Since serverless request patterns often vary over time, the process of spinning down and re-initializing containers may happen many times. To minimize the impact on users, cloud providers often keep function executors allocated or "warm" in anticipation of a new request [11].

2.2 Serverless and Distributed Computing Frameworks

Cloudburst

Cloudburst is a serverless computing platform developed at UC Berkeley's RISE Lab [13]. In my thesis, I explore integrating GPU support into this platform. Additionally, I model system assumptions around the graph of functions execution model, container cold start times, memory/KVS interfacing, etc. based on Cloudburst for developing algorithms for admission-control based scheduling.

Cloudburst aims to resolve fundamental challenges in the serverless field today like efficiency, latency and access to state. To use Cloudburst, users provide a graph of functions which are scheduled and executed by the system on executor nodes. Unlike existing solutions like AWS Lambda, Cloudburst enables state sharing though functional composition, direct communication and access to shared, mutable storage. The architecture is based on the principle of logical disaggregation with physical colocation (LDPC), which relies on the idea of separating compute and storage but enabling low-latency state through caches that live near the compute.

Cloudburst itself is built on Anna [16], a low-latency, autoscaling key-value store. Anna avoids expensive locking and lock-free atomic instructions, and instead employs a wait-free, shared-nothing architecture, where each thread in the system is given a private memory buffer. To resolve conflicting updates, Anna encapsulates user data in lattice data structures, which have associative, commutative, and idempotent merge functions.

Ray

I additionally explore the Ray framework in my thesis, which is what the Kernel as a Service (KaaS) codebase was developed on [7]. In my work, I treat Ray the same way I do Cloudburst: as a serverless computing platform. I model scheduling policies for KaaS based on how the Ray system is modeled.

Ray is a distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. It achieves scalability and fault tolerance by abstracting the control state of the system in a global control store and keeping all other components stateless. It uses a shared-memory distributed object store to efficiently handle large data, and it uses a bottom-up hierarchical scheduling architecture to achieve low-latency and high-throughput scheduling. It uses a lightweight API based on dynamic task graphs and actors to express a wide range of applications in a flexible manner.

Ray provides users with a Python API to run stateless functions (called tasks) or stateful objects (called actors) across a cluster. Data is communicated using references to objects in an immutable object store (called Plasma, the equivalent of Anna for Cloudburst). References can be created before their associated object is available (i.e. futures); Ray takes advantage of this to create lazily-executed graphs of tasks/actors that are scheduled only when their input references are available. Ray maintains a number of processes on each node (called "workers") that execute tasks and actors. A single worker may execute many tasks, but actors are always run by a dedicated worker. To control resource usage, users may annotate tasks or actors with resource requirements such as the number of GPUs required. Ray then ensures they are run on a worker that has been allocated those resources. However, Ray does not enforce isolation of resources (instead relying on applications to ensure they do not exceed their limits). Furthermore, tasks running on the same worker share resources at a fine granularity and must ensure all resources are freed before exiting. Since actors are persistent and run on dedicated workers, they are less sensitive to resource management concerns but there cannot be more GPU-enabled actors than available GPUs in the system.

Chapter 3

Serverless Model Serving with GPUs

In this section, I explore various approaches for how we can design and build model serving systems. I then focus particularly on the serverless approach and discuss how we practically integrate GPUs into the Cloudburst serverless computing platform. I talk about challenges and the process around implementation, and discuss how optimizations like batching can improve performance.

3.1 Model Serving Approaches

With the rise of machine learning systems in production systems, there is a need for infrastructure to support model serving. There have been several different approaches, but most can be categorized as one of the following approaches: (1) Materialize queries (2) "Model in a container" and (3) serverless.

These three approaches share many motivations. First, they allow for separate development of ML models, which is useful as most development is done by data scientists who don't necessarily work on the systems that interface with the users directly. Keeping the model development separate allows for seamless updating, A/B testing, and retraining models on newer datasets. In addition, optimizations can be made to improve throughput of the models by batching requests that it sends to the model during times when the number of requests is high, as the Clipper system does. Finally, with all these approaches, it's possible to independently scale serving capacity and share models between multiple applications.

The "materialize queries" approach attempts to take advantage of the batching abilities of most machine learning models. Given a distribution of incoming requests, this system evaluates and materializes results offline by storing them in a database. The real-time queries then just perform a simple database lookup, which is significantly faster than running the model per request. However, this approach has the drawback of having a limited range of inputs, and doesn't support evaluating models on new data.

The "model in a container" system is a popular approach taken by several different systems such as Clipper [2], TensorflowServing [8], and Amazon Sagemaker [5]. These systems package a model and its dependencies into a Docker container. These models are deployed on hardware of choice, and the system provides an API endpoint for clients to query. While this approach has a higher latencency than materializing queries, it allows for a broader and dynamic set of request inputs.

Finally, the "serverless" approach applies the containerized model approach to the serveless setting, which fits in well with this model serving need. It scales efficiently and saves money by only allocating resources when a model is in demand. There are several challenges with doing model serving on traditional, stateless serverless systems such as Amazon Lambda [15]. AWS Lambda has high latency, low throughput and is stateless so it can't hold model hyperparameters between calls. Cloudburst, a FaaS platform developed at Berkeley allows for caching of prior requests and model hyperparameters in its object store Anna. Additionally, it provides support for a graph of functions, which allows for code reuse between models and more independent scalability. This graph of functions approach also provides the ability to use accelerators like GPUs for only the most useful parts of the model evaluation.

3.2 GPUs and Containers

As discussed, the serverless approach is a great fit for the model serving. However, there is limited support for GPUs in serverless systems [4], which provide tremendous utility for this particular use case as well as others. In this subsection, we discuss the steps we took to add GPUs to Cloudburst.

GPUs are a scarce and expensive resource, making proper usage of them important. When clients want to use a model, they need to initialize a container and runtime for it, which can cause significant overheads due to cold starts. A bigger issue though is that GPUs are harder to share amongst multiple models, since multiplexing GPU memory and threads isn't as easy as for CPUs. In many cases, it doesn't necessarily make sense to use a GPU due to these issues, particularly when a request wants to be executed more cheaply or has loose time constraints.

In order to make GPU-supported serverless functions a reality, we package their dependencies and containerize them. Containers, as mentioned previously in the context of Clipper, allow for models' code and dependencies to be abstracted to work on any system.

We made several modifications so that the Cloudburst system could support serving models. In particular, we needed to add support for GPU-enabled instance types in addition to the existing CPU-only instances. We also had to modify the docker images that run the

functions by adding the appropriate NVIDIA plugins and drivers. We highlight some of the throughput wins in our GPU-enabled version of Cloudburst in the next section.

3.3 System Setup

In our setup, clients submitting requests for various models were connected to a Clipper backend. We modified the client to make API calls to Cloudburst as well, and benchmarked the results.

3.4 Pipeline and Use Cases

There are two pipelines that we ran that are both common use cases. These pipelines can be represented as graphs of functions as shown in the figure below. The first is a pipeline of ensemble models that has a preprocessing step. This is a common technique used to increase robustness of result, improve accuracy, and reduce overfitting of the models. Cloudburst takes advantage of running these models in parallel and combining results quickly.

The second pipeline involves a cascade of two vision models. This happens when the first model has a confidence lower than a preset threshold, after which a second model is run and the result determined by taking the result of the more confident model.

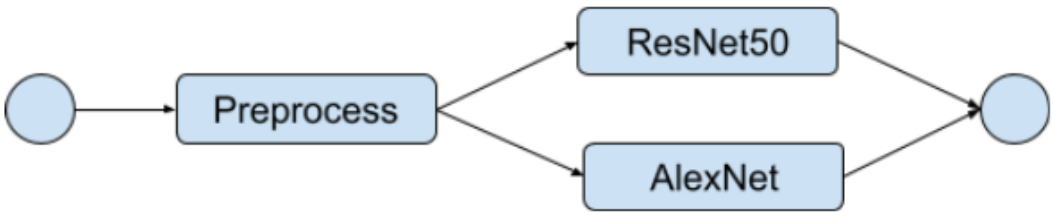


Figure 3.1: Pipeline of ensemble models that has a preprocessing step. This is a common technique used to increase robustness of result, improve accuracy, and reduce overfitting of the models.

3.5 Evaluation of Systems

As the table shows, implementing the pipelined model on Cloudburst on a single node setup even without GPU support already shows a roughly 33% latency speedup over Clipper. And

Figure 3.2: Pipeline involving a cascade of two vision models. This happens when the first model has a confidence lower than a preset threshold, after which a second model is run and the result determined by taking the result of the more confident model.

Pipeline	Clipper	Cloudburst
Ensemble	0.34251	0.215425
Cascade	3.07523	2.179395

Table 3.1: Average latency (ms) for pipeline between Clipper and Cloudburst show the improved latency of the Cloudburst serving system.

furthermore as the figures show, adding the ability for Cloudburst to use GPUs greatly increased the system's throughput. This allows for 10x higher throughput at larger batch sizes.

Our system currently has a fixed global batch size for processing requests. Taking inspiration from Clipper, we want to pursue adaptive batching based on incoming request rate, even possibly to the granularity of individual GPU executors. Another improvement to improve GPU utilization is to allow multiple containers to share the same GPU, perhaps through a multiplexing process. Future cost analysis of the overhead of context switching between function threads could be used to improve scheduler decisions.

3.6 Summary

A serverless environment is a great fit for a modelserving use case due to its ability to handle varying client request patterns scalably. Since model serving and other applications benefit from GPU support, we added them to the Cloudburst serverless platform and got the most out of them through optimizations like batching. In future sections, we'll look at how we can allocate GPUs to requests more optimally though better scheduling.

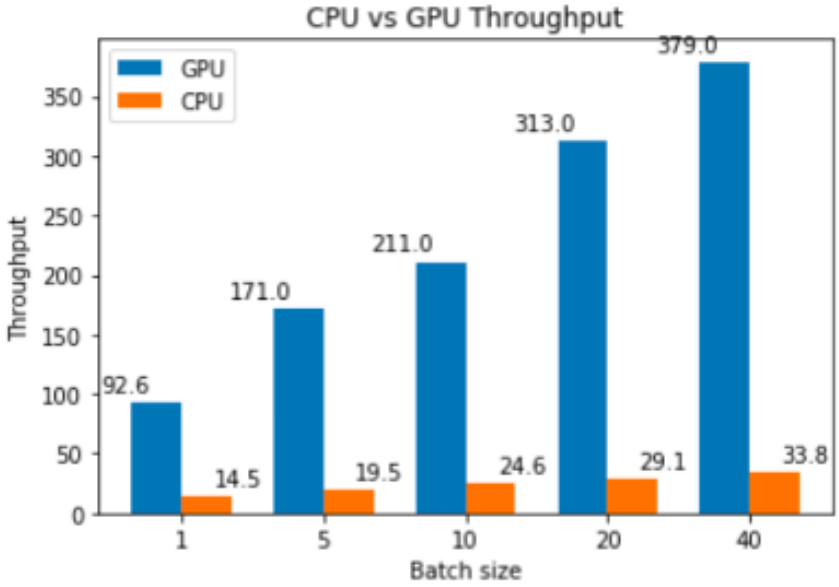


Figure 3.3: With large batch sizes, the GPU supports 10x higher throughput compared CPUs in terms of images classified.

Chapter 4

Scheduling Under Constraints with FaaS

Now that we have a working serverless computing system with GPU support, we can think about how to best make use of these limited, expensive resources.

In this section, we explore solving the problem of real-time model inference in the context of serverless computing. We aim to build a scheduler to make the most efficient use of expensive hardware resources and meet user provided service level objectives (SLOs) given function execution estimates. Namely, we consider cost and latency SLOs, which are particularly important in certain applications (for example, self-driving cars may need to make split-second decisions on whether to stop for pedestrians). In such cases, it may be better to reject a request outright in order to meet other requests. To explore the problem further and prototype solutions, we looked at real function execution times and built a simulator for scheduling based on these estimates.This is based on the Cloudburst DAG architecture and models similar performance bottlenecks, such as network or IO latency [13]. Since specialized hardware greatly boosts machine learning application performance, we also added GPU support and optimizations, including multiplexing, to Cloudburst that we then folded into our simulator scheduling algorithm. We explored various DAG scheduling algorithms over multiple resource types, which is particularly useful in the context of inference since certain operations are significantly sped up with GPUs and others aren't worth the cost.

At a high level, our scheduler algorithm is built around admission control: we determine at request time which queries can meet SLOs, pre-plan a schedule, and reserve necessary resources over individual CPUs and GPUs.This ensures the query can run to completion in the allotted time. If a query SLO can't be met, we reject it immediately to ensure the system is constrained; the user can either update their SLOs or try again. In the process, we consider keeping popular models warm, preselecting an execution plan, using (and batching) existing DAGs if they are already on the system, etc. We also keep in mind subtleties like SLO accuracy and ensure that a single bad SLO doesn't lead to cascading misses. Ultimately, we

address the question: how do we place the DAGs of models on a heterogeneous cluster of CPUs and GPUs to minimize invocation rejection based on SLOs?

4.1 Historical Approaches

Our work from the previous section regarding adding GPU support to Cloudburst was later incorporated into a larger serving framework called Cloudflow. Cloudflow is a system developed at Berkeley by Vikram Sreekanti to address model serving [12]. It provides a simple API like Clipper and realizes it on an autoscaling serverless back-end. Cloudflow also implements performance-critical optimizations including operator fusion and competitive execution. While this addresses many of the concerns from Clipper, it doesn't take into account SLOs, which can lead to an unconstrained system which misses many deadlines. We also take into consideration DAG and function placement, which can increase system capacity and performance.

Inference as a Service (INFaaS) by Romero et al. has similar ideas as what we propose: it tackles issues of profiling "model-variants", the space of modifications to a model including architecture, input batching, underlying hardware and uses these estimates in function placement [10]. Our system architecture and assumptions are slightly different in the sense that we look at DAGs rather than individual functions and optimize for SLOs. We differentiate by exploiting different system properties, such as data locality. For example, if the model is already loaded onto CPU memory, to what extent does that speed up an invocation versus loading it onto GPU memory and then invoking it. We also address operator fusion and competitive execution, similar to Cloudflow.

Clockwork Comparison

There are a few other similar papers in the space such as Sequoia [14], Wukong [1] and Clockwork [3]. Clockwork, most notably, uses SLOs for admission control in FaaS inference. Clockwork enables user-provided SLOs for DNN inference by leveraging the following observations: (1) DNN inference is fundamentally a deterministic sequence of mathematical operations that has a predictable execution on a GPU and (2) in order to satisfy SLOs with high fidelity, the system must have predictable execution times, which can be realized by consolidating all resource consumption and scheduling decisions. The core logic of clockwork lies in its centralized controller which makes a variety of explicit decisions on model placement and scheduling for all components of the system, including GPUs and worker threads. In doing so, Clockwork is able to support thousands of models while simultaneously meeting 100ms latency targets for requests. While we adopt the same principles of predictable execution via consolidation of choice, our system differentiates itself from Clockwork by allowing richer DAG specifications, in which execution can be segmented over various resource types.

(Note that Clockwork only allows for execution on a GPU.) In doing so, our scheduler is given more flexibility, while still satisfying SLOs.

4.2 DAG Admission Control Algorithm

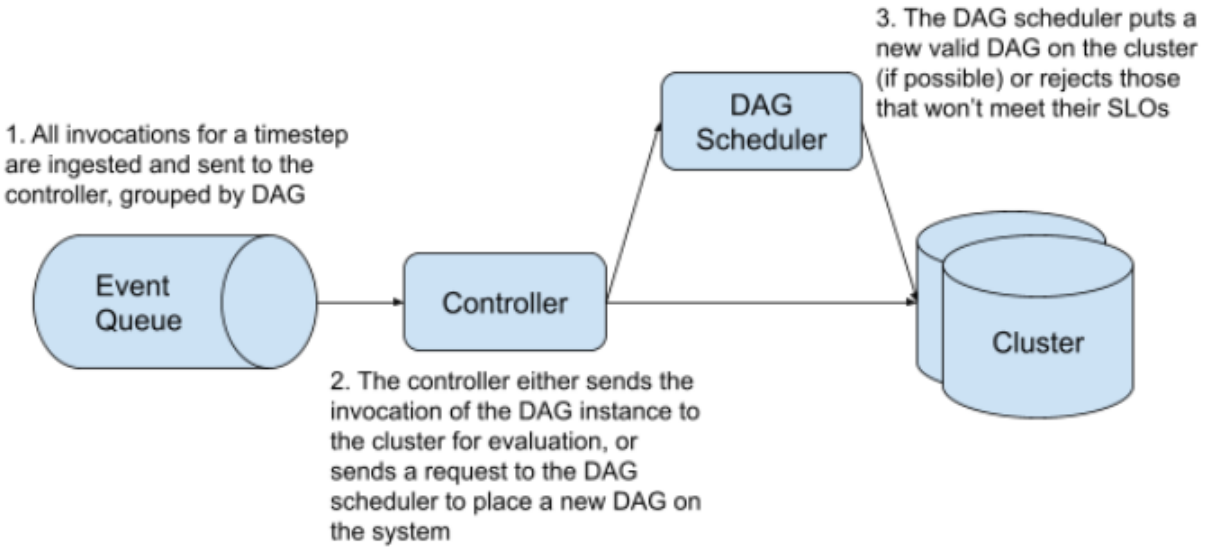


Figure 4.1: SLO-based DAG scheduler

In this section, we present our approach to optimally placing and running functions of DAGs invoked by users. The principles of our FaaS system are used to determine the design of our simulator, where we will compare our algorithms with other comparable algorithms.

System Background

Our vision of a general serverless FaaS system has a few main components. Firstly, there is the cluster of heterogeneous resources. These "nodes" could be any hardware, such as GPUs, TPUs, and CPUs, as well as combinations of hardware, such as containers with attached GPUs to run CUDA code. For our simulator, we only concern ourselves with CPU and GPU nodes, but the simulator could be extended to include many other resource types. Also, while our simulator considers a fixed cluster, the resources allocated to this cluster could be dynamically allocated and deallocated. More detailed information about this can be found in our future work section.

This cluster of resources is managed by a serverless FaaS framework. The functionality

of our framework is loosely based on Cloudburst, specifically because Cloudburst, like our system, allows for the registering of both individual functions and DAGs to connect said functions. While other offerings such as AWS Lambda do not support this functionality, many bolt-on systems can replicate DAGs on top of Lambda's functions [1]. These DAGs are stored in the framework, and can be invoked at any time, with arguments to the root function of the DAG passed as necessary.

Algorithm Overview

At a high level, any algorithm running on our system is passed all DAGs that will be invoked before timestep 0. The algorithm is allowed to generate any data it needs to optimize placement on the DAGs before time starts. Then, once time starts, the framework begins receiving DAG invocations with latency and cost SLOs for the invocations, and the scheduling algorithm must decide where to place functions of the DAGs in order to meet these invocations. As a simple example, a naive version of Clockwork, which we compare our algorithm against, would conduct no DAG data preprocessing, and would put entire DAGs either entirely on a CPU node or entirely on a GPU node, and would route invocations to existing placements and add new placements as it deems fit given the fixed cluster to meet as many invocations as possible.

Phase 1: Preprocessing

Our algorithm operates in two phases. First, it generates data about all the DAGs registered with the serverless framework before timestep 0. The algorithm goes through each DAG, and generates all possible placements of the DAG on each resource type available at the granularity of an individual function. It does not consider SLOs or the current cluster at this point, as the SLOs are determined for every invocation, and the amount of available space on the cluster will change as execution continues. It will generate "rn" total "DAG placements", or matchings for each dag function to a resource type, with r being the number of unique resource types available, and n being the number of functions in the DAG.

It will then evaluate these DAG placements on the possible SLOs that arrive with invocations, in our case latency and cost, although this could be expanded further in future work. It will optimistically assume that the resources that the functions have been assigned have available memory to load the function and will immediately begin execution of the function. For example, for latency, the algorithm computes the latency of executing each individual function on the selected resource, and, given the dependencies of each function, determines the end-to-end latency of the invocation. With these statistics per DAG, it will then delete all of the DAGs that are not pareto efficient relative to all other DAGs generated, using the SLO statistics to determine this. We are then left with a set of DAG placements that lie on the Pareto Curve. Subsequently, for each SLO, the "controller", or the independent node that runs the algorithm, we store a sorted list of all the DAG placements, sorted on the

SLO in question. For our simulated algorithm, this means after the first phase is complete, the controller will have two sorted lists of all possible pareto efficient DAG placements, one sorted on latency and one on cost.

Phase 2: Real Time Scheduling

The second phase of the algorithm occurs while the framework is managing the cluster and handling requests. During this phase, when an invocation or set of invocations is sent to the request queue, the controller routes those requests to that DAG's invocation scheduler, doing this for every request sent at any given timestep. If an invocation can be resolved through a DAG instance already placed on the system, then the invocation is set to run on that DAG instance in the queue of requests to that DAG. If the invocation can be batched with other invocations to that DAG that arrived with it or have arrived previously and are awaiting execution, then it is batched to maximize throughput without violating any SLO's.

However, it is possible neither of these cases is valid for the current DAG instances, and a new DAG instance has to be loaded onto the system. Then, that DAG's invocation scheduler will binary search through all of the per SLO sorted lists, and will find the set of DAGs in that list that meet that the invocation's SLO for that list's SLO type. It will then take all of these sets of DAG placements from each sorted list and union them to determine which DAG placements will (optimistically) meet all SLOs.

After generating this valid set, the algorithm will generate a sample of this set through random sampling, creating a set of a fixed size "s", which is a parameter set by the DAG creator when the DAG is registered. This set of DAG placements will then be checked against the current cluster to determine whether or not it can be placed, and if so how long that will take to load and execute, as that could make it overshoot its latency SLO. This check is done with a greedy algorithm, which takes a list of the functions with the lowest max memory required to be allocated, and compares it to a list of nodes sorted by how much memory is left on them. If the current function can be placed, it is set to be placed on that node, and the algorithm iterates to the next function. If there is not enough memory left, it iterates to the next node. This is done for every resource type, matching the functions set to the resource type to the nodes of that type. The check is run for every DAG placement in the sampled set. If there are multiple valid placements on nodes, one is picked randomly. This placement is then loaded onto the determined nodes, and the invocation is queued to be run. In future work we hope to have more intelligent picking strategies to minimize request rejection rate.

Algorithm Considerations

For every function in a DAG, when determining the latency of the DAG, there is a "pre", "exec", and "post" time. The "pre" time is the time it takes to load the function onto the

resource it's assigned to, and may change with resource type. This "pre" time is considered by the aforementioned DAG placement algorithm when considering whether or not placing a new DAG on the system will meet the SLOs of invocations that have arrived, and is not considered part of execution time when routing an invocation to an already instantiated DAG. The "exec" time is the time it takes to complete the function on the resource type it is assigned to. Because our use case is in prediction serving, we assume that the "exec" time is highly predictable, which is useful for our calculations of predicted latency and cost. For more information on why this is true, please refer to the Clockwork paper cited below. The "post" time is the time it takes to remove the function from the resource. In practice, this simply means that when a DAG instance on the cluster has no more requests left in its queue, all the memory used for the functions of that DAG are freed to be used by other functions that could overwrite that memory.

4.3 Simulated Evaluation

Simulator

Our experimental design is built on top of a simulation infrastructure in which users are able to model various systems. We opt for a simulation-based approach since our research question is primarily focused on evaluating scheduling policies. This approach makes the following assumptions:

- Estimated execution times are faithful. This assumption is valid in our context since we are considering inference workloads which are essentially deterministic. In reality, there may exist some differences in timing, but Clockwork has shown that by consolidating resource consumption and scheduling decisions, an inference system can have very predictable runtimes.
- Network latency is minimal and delivery is reliable. Our simulation infrastructure steps at a 1ms interval and always delivers messages to its components. While it is possible to add unreliability and variable latency to our framework, we do not do so for the scope of this paper. This is a fair assumption in the context of the evaluation since we are comparing scheduling policies relative to each other, in the same context. Since we are primarily focused on throughput behavior, this assumption does not invalidate the observed patterns.

The simulation framework is organized in the following way, loosely around the Actor framework:

- A component has an inbox and outbox.

- A component must implement the following actions: (1) input(): Routes incoming messages. (2) exec(): Computes on input messages and stages output messages. (3) output(): Sends outgoing messages.
- A component may implement its own private state or share state with another component.

Evaluation

Under this model, we have built simulators for a variety of FaaS systems including a heterogenous version of AWS Lambda, Clockwork, and our own scheduling approach. With these simulators defined, we evaluate our scheduling policies on a set of microbenchmarks, shown below.

In our microbenchmark evaluation, we simulate 1200ms with 1200 requests and measure goodput (SLOs achieved/s). Below are reported goodputs (GP) for FIFO scheduling policy, Clockwork, and our own algorithm, for both latency and cost SLOs.

Lat SLO (ms)	Cost SLO (units)	FIFO Lat GP	FIFO Cost GP	CW Lat GP	CW Cost GP	Custom Lat GP	Custom Cost GP
20	60	0.49	199.9	0.069	83.3	0.59	177.35
40	60	1.17	199.9	0.208	83.3	1.37	205.8
60	60	1.99	199.9	0.34	83.3	1.77	177.3
80	60	2.66	199.9	0.42	83.3	2.82	178.7

Table 4.1: Goodputs (GP) for FIFO scheduling policy, Clockwork, and our own algorithm, for both latency and cost SLOs

4.4 Future Work

Our results for this FaaS admission control work seem promising, so there are a few avenues that may be worth pursuing:

- Learning internal cost estimates using online reinforcement learning algorithms: Currently, our system requires users to provide an upper bound cost for individual function execution on CPUs and GPUs. We hope to provide a better user interface by abstracting these values from users. We plan to learn these estimates through an online Q-learning based approach, which updates internal state every function invocation.
- Support for adaptive batching: the Clipper paper makes use of batching by delaying certain requests if it can combine it with another, leading to better hardware utilization. Our preliminary results from the Cloudburst system show a significant increase in throughput when batching requests with GPUs. We hope to incorporate this improvement into our simulator and examine how it affects scheduling policy.

- Workload aware vs workload blind scheduling: Currently, we don't consider future requests in scheduling. If we're expecting a certain request or can predict it, we can allocate other requests with weaker latency SLOs to CPU resources. Alternatively, this may also mean predictively scaling our cluster and keeping models warm.
- Scalability vs static allocation of resources: Currently our policies are simplified for a static cluster, although it should generalize to an autoscaling one in theory. Can we see how this would affect policy through experimental evaluations?
- Scaling at the function level: If we receive many requests for the same model, we currently scale at the DAG level. It's possible that a function that executes quickly and has low demand is replicated if the rest of the DAG needs to be - we can save on this resource usage. It's also possible that multiple DAGs use the same functions, so it may be better to scale those too at the function level to get better statistical multiplexing.
- Better selections of DAGs given SLOs: Given a set of candidate DAGs at admission time, we are currently choosing a random schedule that meets all SLOs (if it exists). Perhaps we want to provide some control of which configuration we chose to the user - e.g. if users care more about costs, we pick the lowest cost that fits the latency SLO.
- Thinking about abstractions to the user: How should the user write functions - should they write a CPU and GPU optimized version of the function, or should we use some common language for both at the expense of performance?

4.5 Summary

In this section we looked at current serverless FaaS and model serving offerings, and saw how they fall short when it comes to the admission and scheduling DAGs over heterogeneous resources. Then, we presented our algorithm that tackles this problem, which uses preprocessing steps and search and sampling methods to find DAGs that will outperform other placement algorithms, such as FIFO scheduling and the current state of the art, Clockwork. Finally, we show how the work for this paper is only the first step in a series of improvements to admission control on DAGs that can be achieved.

Chapter 5

The Issue with GPU+FaaS

We've explored how to schedule in a FaaS setting under constraints to get the most out of our limited GPU resources. However, there are some fundamental limitations with FaaS that we address in this section.

Some systems, including the ones we've described and worked with so far, offer a serverless FaaS+GPU approach. In these systems, users upload a CPU-oriented function and the provider ensures that the function runs in a container that includes a GPU. While these approaches present a familiar interface to accelerators, they give away many of these features that make serverless appealing.

As an example, we added support to the Cloudburst distributed computing framework to support GPU-enabled remote functions, but required that applications carefully design their tasks to manually share resources and clean up properly when they finish. In practice, this is difficult to achieve. Indeed, users are advised in the documentation to force Cloudburst to restart workers on each task invocation to ensure resources are properly freed [8]. Rather than using serverless functions, users are often advised to fall back to non-serverless stateful actors to manually manage GPU resources.

Another challenge that arises with this approach comes from cold start mitigation strategies. CPU functions incur a significant startup latency due to container and language runtime initialization. A Python script that simply imports tensorflow and immediately exits takes 1.9 seconds when the OS buffer cache is warm, a true cold start that must read from disk takes even longer at 6.8 seconds. To minimize the impact on users, cloud providers often keep function executors allocated in anticipation of a new request [13]. This policy is reasonable because SMT threads are cheap and processes are easily idled. The same policy applied to a GPU would cost at least 60x more since the provider would need to keep the much more expensive GPU idle. While the increased cost of a GPU is justified when fully utilized, both resources provide the same utility when idle: zero.

5.1 Physical and Logical Disaggregation

In the serverless model, rather than requiring the system to fit jobs into fixed-sized servers, we make any resource in the system available to any job (physical disaggregation). Rather than requiring jobs to allocate all their resources up-front, we allow them to allocate resources only when they actually need them (logical disaggregation).

It is challenging to deploy accelerators like GPUs in a physically and logically disaggregated FaaS system because of the techniques that make FaaS practical to implement. FaaS functions typically are small and limited in scope, they consume few resources when not executing, and use easily shared and subdivided resources. Providers are free to kill FaaS containers as needed to free resources, or aggressively cache them to reduce cold starts. Furthermore, providers can mitigate the performance costs of explicit state by maintaining shared caches of their data layer. GPUs upend these assumptions. Unlike CPUs, GPUs are expensive, difficult to share on a fine granularity, and have their own memory that must be managed by the user. This makes it hard to share GPU resources since certain models may already be pre-loaded onto it (violating physical disaggregation), and would benefit from us keeping models warm on GPUs to avoid cold starts (violating logical disaggregation).

We saw these challenges arise when working with GPU-enabled Cloudburst. When we wanted to run a new model on a GPU, we had to remove the old model container and load the new one, taking the hit of a full cold start. We could have many warm CPU functions per executor but only a single warm GPU model per executor, even though most models didn't use much of the GPU. Our algorithm was great, but how much better could it be if GPUs could be allocated more flexibly? In the following section, we discuss how to do that.

5.2 KaaS

Fundamentally, these existing approaches struggle because they cede control of GPUs to applications by tightly coupling host and GPU code.

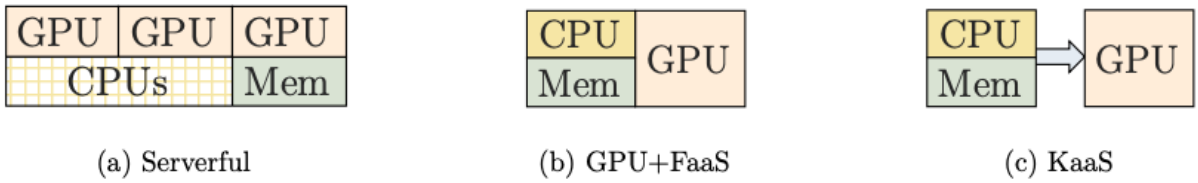
A separate approach called Kernel-as-a-Service (KaaS) was proposed by Nathan Pemberton. He suggested explicitly decoupling host functions from GPU kernels at the system level. Following the principles of serverless computing, our system takes user descriptions of GPU functionality rather than providing GPU allocations. This puts the responsibilities of GPU memory management and scheduling on the system rather than relying on users to make optimal use of these expensive resources.

This strategy elevates GPU functionality to a first-class entity, along with traditional FaaS functions and other serverless services. Rather than providing a Python source file or Linux

container, KaaS functions take the form of a graph of CUDA kernels to be executed. Graph inputs and outputs are provided as objects in the system data layer (e.g. keys in a key-value store). Other buffers like intermediate outputs and temporary buffers are described simply by their size. The system is then responsible for ensuring that all required buffers are available in GPU memory before beginning execution of the kernel graph. KaaS functions are not permitted to dynamically allocate memory or access the data layer, leading to highly predictable resource requirements. Furthermore, KaaS functions do not include any host code whatsoever which simplifies the software environment on KaaS executors and prevents external side effects.

For compute-heavy tasks, KaaS experiences virtually no decrease in aggregate performance, even when there are far more users than available GPUs. For memory-heavy workloads, KaaS performance degrades gracefully when GPU memory requirements exceed capacity.

The KaaS approach only explicitly uses GPUs in its programming and billing model. This means that users are charged only for the time their code actually ran on the GPU rather than paying for long lived allocations. Users also do not need to allocate an entire server or VM just to use its GPUs. In serverful and actor approaches, the user must pay for idle CPU, memory, and disk resources while the GPU is running. Furthermore, KaaS users do not manage the GPU themselves and do not need to (and in fact cannot) maintain any state that might interfere with sharing. Instead, the system is free to manage device memory and allocation at a fine grain. Like serverless more generally, these properties free users from complex deployment decisions and allow for transparent autoscaling.



(a) Serverful (b) GPU+FaaS (c) KaaS

Figure 5.1: Three possible GPU deployment strategies. In a serverful deployment, users get allocated a large collection of CPUs and GPUs for an unbounded period of time. In GPU+FaaS, users run a short stateless task that uses a small number of resources only for the duration of the task. In KaaS, users submit CPU and GPU tasks separately that each run in a resource-specific environment.

Chapter 6

KaaS: Locality Aware Scheduling for Uninterruptible Tasks

As we just saw, GPUs complicate FaaS because users are assigned entire GPUs, which they must manage themselves. A client would load the libraries for and execute requests on GPU executor nodes, which would always have the model loaded and bill the client even if it was not being used. If they needed more accelerator resources than a single machine/serverless node, we'd allocate them another executor with an attached GPU which they could use to service requests. This process violates the principles of the serverless model where we aim to provide physical and logical disaggregation.

In the Kernel-as-a-Service model, the system is entrusted with scheduling requests from multiple clients onto a fixed set of resources. There's a lot more "serverlessness" here, since the client is no longer responsible for managing physical accelerator resources. However, that brings up a challenge: how do we map requests to resources? The system needs to decide how to best schedule, and needs to think about what properties it would like to maintain: fairness, total completion time, resource utilization, cold start minimization, etc.

Before diving into algorithms, there are a few main challenges and cases for concern worth pointing out. One is that there are cold starts whenever a new model is loaded onto a resource. In KaaS, we can keep multiple models cached, so when a model is "switched to" after already being loaded before, there's significantly lower latency than a traditional cold start. The other challenge is that accelerators like GPUs are harder to preempt. This leads to the phenomenon of head of line blocking, particularly when there are certain clients submitting longer-latency requests and others submitting shorter ones. The longer requests sit on GPU resources and "block" the shorter ones from executing; this makes the shorter ones have to wait before they can be scheduled. We'll explore some ways we can tackle this challenge.

6.1 Scheduling Algorithms

In this section, we will explore a set of scheduling algorithms for assigning KaaS requests to GPU executor nodes. We keep in mind metrics such as the time requests spend on the queue before being serviced (and how this varies across clients for fairness), the number of cold starts incurred, and resource utilization.

Our scheduling algorithms are based on a system setup where there are a set of clients, each submitting requests to a queue or set of queues. The scheduling policy decides which requests to pop off the queue(s), and which resources to assign them to. For most policies, all clients' requests feed into a single queue, and use either a FIFO mechanism or priority mechanism to select the next schedulable request.

For the first set of policies, all clients write their request to a single queue, which is processed in a FIFO fashion.

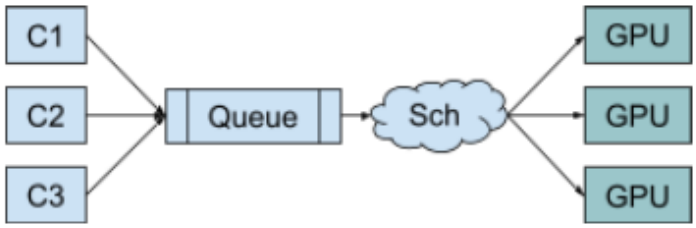


Figure 6.1: KaaS Scheduler: Single Queue Architecture

In order to validate the KaaS model and experiment with various scheduling policies, we built a simulator. The simulator models cold start times, evaluation times, multiple queues (and various queue depths), multiple clients, different model types, etc. The ratios for these times are all based on a real KaaS implementation in Ray. All code for the simulator may be found here: `https://github.com/adityaramkumar/kaas_simulator`

In addition to the algorithm itself, let's also look at a few cases individually to understand each of the scheduling algorithms' strengths and weaknesses before diving into aggregate results, and doing parameter sweeps. To start off with, let's consider a workload with 2 clients: one which submits ResNet50 model requests, and another which submits BERT requests. For the purposes of our simulator, let's assume that ResNet50 takes 5ms to evaluate and has a cold start time of 5ms; and let's say BERT has a cold start time of 10ms and takes 20ms to evaluate. We have ResNet50 requests arriving at times 0, 2 and 3, and BERT requests arriving at time 1. How do our various scheduling policies perform?

Round Robin

This policy routes requests to GPUs in a round-robin fashion with no regard to isolation or current load. The scheduler cycles through resources one at a time and blocks until the next one is available. That is, if there are three GPU resources, requests must be routed to the first GPU, then second, then third, then first again, and so on in that order. This means that GPU resource utilization may not be at 100% even if there are many outstanding requests. Consider the case where a long running request is scheduled on the first GPU, but short requests are scheduled on the second and third GPUs. When a new request arrives, it must wait till the first GPU is free, even though other resources may be free.

This policy doesn't discriminate between different clients and models, so if a client submits a lot of requests it will get a higher share of the resources than others. This is inherently unfair, and gameable if a client wants faster response times (i.e. submit a lot of requests, and cancel outstanding ones after receiving a response). Additionally, minimizing cold starts aren't taken into account with this policy.

Here are the results from running our simple workload with assumptions outlined above with our simulator:

```
Request(arrival_time=0, model_type=ModelType.RESNET) is being scheduled at
timestep 0 on resource GPU1
Request(arrival_time=1, model_type=ModelType.BERT) is being scheduled at
timestep 1 on resource GPU2
Request(arrival_time=2, model_type=ModelType.RESNET) is being scheduled at
timestep 6 on resource GPU1
Request(arrival_time=3, model_type=ModelType.RESNET) is being scheduled at
timestep 19 on resource GPU2
Total time taken is:  25
Total cold starts is:  3
Total cache hits is:  1
Cache hit percentage is:  0.25
Average time waiting is:  5.0
Client(name=Client1, model_type=ModelType.RESNET):
Average time on queue is 6.666666666666667
Client(name=Client2, model_type=ModelType.BERT):
Average time on queue is 0.0
```

In the round robin policy, we can see that our average time waiting is fairly high, primarily due to inefficient resource utilization. If there is a resource with a long request (in this case, GPU2 with BERT), we would need to wait even if there are other free resources. This is because every request is being scheduled on subsequent resources. This leads to the

phenomenon of head of line blocking, since we must wait for the longer BERT model to complete before we can continue servicing the shorter ResNet50 requests.

Balance

The balance policy routes requests to the next *available* GPU regardless of locality or isolation. Unlike the above round robin policy, the balance policy no longer requires that GPUs are allocated to in a particular order. Rather, whenever a resource is free, it can service the next outstanding request. When the queue is not empty, GPU resource utilization will remain at 100%.

Much like the round robin policy, this policy still doesn't discriminate between different clients and models, and is inherently unfair for the same reasons. Additionally, there isn't consideration to locality when deciding which resource to schedule a model on, so there is no consideration for minimizing cold starts.

Here are the results from running our simple workload with assumptions outlined above with our simulator:

```
Request(arrival_time=0, model_type=ModelType.RESNET) is being scheduled at
timestep 0 on resource GPU1
Request(arrival_time=1, model_type=ModelType.BERT) is being scheduled at
timestep 1 on resource GPU2
Request(arrival_time=2, model_type=ModelType.RESNET) is being scheduled at
timestep 6 on resource GPU1
Request(arrival_time=3, model_type=ModelType.RESNET) is being scheduled at
timestep 8 on resource GPU1
Total time taken is:  19
Total cold starts is:  2
Total cache hits is:  2
Cache hit percentage is:  0.5
Average time waiting is:  2.25
Client(name=Client1, model_type=ModelType.RESNET):
Average time on queue is 3.0
Client(name=Client2, model_type=ModelType.BERT):
Average time on queue is 0.0
```

The balance policy is a step up from the round robin policy. In the results here, we can see that total time taken is lower than the round robin policy, because now any resource can be used to service a request rather than just the "next" one. This helps us avoid head of line blocking (to some extent, but if we have a low number of resources and a large number of BERT requests, we'd still see this phenomenon.

Shortest Time Remaining First

In the next set of policies, the above queue is treated like a priority queue, rather than first come first serve.

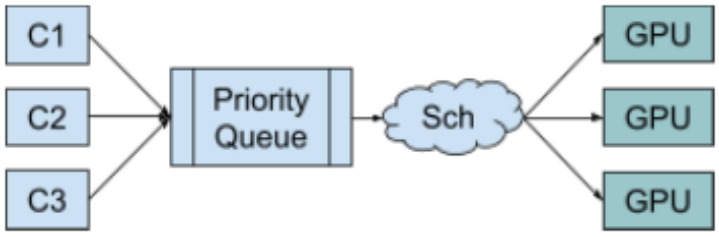


Figure 6.2: KaaS Scheduler: Priority Queue Architecture

The shortest time remaining first policy considers all outstanding requests, and schedules them in ascending order based on how long it takes. This allows us to clear the queue as fast as possible. Much like the balance policy, the SRTF policy waits until any resource is available, before popping a request off the queue.

This policy doesn't take into account model locality as well, so there isn't consideration to cold starts. And unlike the previous policies, this policy is perhaps more unfair. If there are outstanding requests on the request queue, a longer-running model may face starvation and won't be scheduled till all shorter requests are completed.

Here are the results from running our simple workload with assumptions outlined above with our simulator:

```
Request(arrival_time=0, model_type=ModelType.RESNET) is being scheduled at
timestep 0 on resource GPU1
Request(arrival_time=2, model_type=ModelType.RESNET) is being scheduled at
timestep 2 on resource GPU2
Request(arrival_time=3, model_type=ModelType.RESNET) is being scheduled at
timestep 6 on resource GPU1
Request(arrival_time=1, model_type=ModelType.BERT) is being scheduled at
timestep 8 on resource GPU2
Total time taken is:  26
Total cold starts is:  3
Total cache hits is:  1
Cache hit percentage is:  0.25
Average time waiting is:  2.5
Client(name=Client1, model_type=ModelType.RESNET):
```

```
Average time on queue is 1.0
Client(name=Client2, model_type=ModelType.BERT):
Average time on queue is 7.0
```

The SRTF policy aims to tackle challenges around head of line blocking and tries to minimize the overall average time on queue for requests. We can see now that ResNet50 requests are serviced significantly faster than before, however at the cost of BERT wait time. This policy is inherently unfair since it prioritizes a particular model over another always. This can lead to starvation. If there were more ResNet50 requests, it would take even longer for the BERT model to be scheduled.

Multi-Level Feedback Queue Scheduling

Multi-Level Feedback Queue Scheduling is a technique from operating systems thread scheduling that we have adapted to our KaaS use case. The main idea is that requests that take a longer time to run are scheduled on resources less frequently, and requests that take a shorter time to run are scheduled more frequently. To this end, we assign each request a "bucket" corresponding to how long we expect it to take. And we sample requests randomly from buckets such that each bucket will be roughly equally serviced.

An edge case worth considering is if we have a lot of clients that feed into one bucket, and a single client that feeds into a different bucket. The client with its own bucket will be serviced significantly faster, leading to unfairness in the system.

Here are the results from running our simple workload with assumptions outlined above with our simulator:

```
Request(arrival_time=0, model_type=ModelType.RESNET) is being scheduled at
timestep 0 on resource GPU1
Request(arrival_time=2, model_type=ModelType.RESNET) is being scheduled at
timestep 2 on resource GPU2
Request(arrival_time=3, model_type=ModelType.RESNET) is being scheduled at
timestep 6 on resource GPU1
Request(arrival_time=1, model_type=ModelType.BERT) is being scheduled at
timestep 8 on resource GPU2
Total time taken is:  26
Total cold starts is:  3
Total cache hits is:  1
Cache hit percentage is:  0.25
Average time waiting is:  2.5
Client(name=Client1, model_type=ModelType.RESNET):
Average time on queue is 1.0
```

```
Client(name=Client2, model_type=ModelType.BERT):
Average time on queue is 7.0
```

The MLFQS model in theory is an answer to the starvation problem that SRTF brings up; we can't see it in this example, but the BERT model would eventually be scheduled even if there were many more ResNet requests. However, the MLFQS model isn't perfect - if there are multiple clients feeding into the same bucket and a single client feeding into a different bucket, buckets will be equally prioritized. So, the client with its own bucket will be serviced significantly faster, leading to unfairness in the system.

Exclusive

Finally, in the last scheduler policy, we model multiple queues, each of which contain a dedicated set of resources.

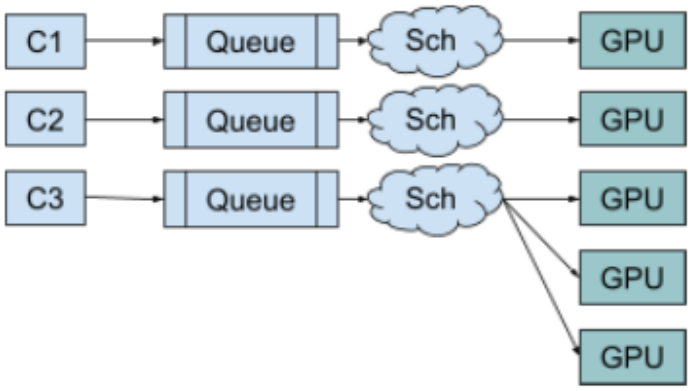


Figure 6.3: KaaS Scheduler: Exclusive Architecture

The exclusive policy ensures that no two tasks run on the same executor, while repeated invocations of the same task are always routed to the same set of executors. To maintain this property, the exclusive policy must kill existing executors to make room for new requests if the number of unique tasks exceeds the number of GPUs. It's worth noting that this policy is what the admission control algorithm and Cloudflow highlighted above do.

This policy directly addresses minimizing cold starts, since a resource only needs to load a single model assuming it doesn't need to switch to a different client/bucket. In the FaaS model, this is an important consideration since cold starts are expensive in the steady state; that is, if a model is loaded onto a resource, and the resource is repurposed for another one, loading the initial model again will still take a significant amount of time. In the KaaS model, models are cached so loading a model that's already been loaded before doesn't incur

as significant an overhead.

It's worth noting that the exclusive policy violates stateless serverlessness since a model may be kept warm on a GPU even when there are no requests for it. A challenge with the exclusive policy is deciding how many GPUs are allocated to each client, and rebalancing accordingly. We propose a policy around this: this is done adaptively through looking at a particular client's request rate * time per request and the total number of GPUs. When a certain queue is a % factor more full than another queue (we use 150% as a heuristic), then "steal" a resource from that queue. There's a tradeoff here - stealing a resource leads to a cold start (at least in non-kaas settings), so we don't want a resource to ping between multiple queues/our heuristic to be too small). If our heuristic percentage is too large, then some requests have a very quick service time compared to others leading to unfairness. Or perhaps more detrimentally, some resources may remain idle.

Here are the results from running our simple workload with assumptions outlined above with our simulator:

```
Request(arrival_time=0, model_type=ModelType.RESNET) is being scheduled at
timestep 0 on resource GPU1
Request(arrival_time=1, model_type=ModelType.BERT) is being scheduled at
timestep 1 on resource GPU2
Request(arrival_time=2, model_type=ModelType.RESNET) is being scheduled at
timestep 6 on resource GPU1
Request(arrival_time=3, model_type=ModelType.RESNET) is being scheduled at
timestep 8 on resource GPU1
Total time taken is:  19
Total cold starts is:  2
Total cache hits is:  2
Cache hit percentage is:  0.5
Average time waiting is:  2.25
Client(name=Client1, model_type=ModelType.RESNET):
Average time on queue is 3.0
Client(name=Client2, model_type=ModelType.BERT):
Average time on queue is 0.0
```

The exclusive policy, which is much like the FaaS systems we looked at before, was designed to minimize the cold starts we see in the other policies. For this policy, we use two queues, each associated with one GPU. We see that once the ResNet model was loaded, every subsequent request was a cache hit. In this setting we don't see an example of resource rebalancing, but different rebalancing policies can help with fairness/resource utilization/etc.

6.2 Results Submitting Requests Under a Poisson Process

In order to simulate a realistic arrival process, we submitted model requests under a Poisson Process. In probability and statistics, a Poisson Process is a model for a series of discrete events where the average time between events is known, but the exact timing of events is random. Essentially, the arrival of an event is independent of the event before (waiting time between events is memoryless). We modeled a queue occupancy of roughly 80%, and submitted with 2 models: BERT and ResNet50. We had 4 GPU resources to service requests, and submitted requests for 1000 timestamps (at roughly 80% queue capacity, assuming requests could be evaluated immediately).

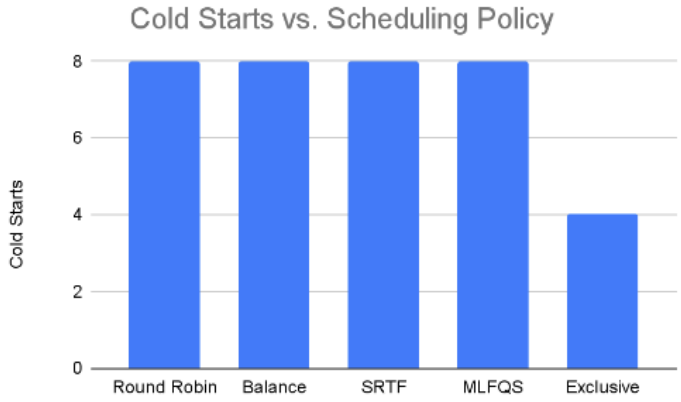


Figure 6.4: Number of cold starts for different scheduling policies.

In our results, we can see that for the RR, Balance, SRTF and MLFQS policies, the number of cold starts is 8. This is the maximum possible number of cold starts, since we have 2 models, each of which can be allocated to 4 distinct resources. These numbers make sense, since all four of these policies don't consider model locality in scheduling decisions, so with a sufficient number of requests all models will be scheduled on all resources eventually. However, with the exclusive policy, since every client is assigned a corresponding pool of resources, there are only 4 cold starts. In this case, the BERT model was assigned 2 GPUs and the ResNet50 model was assigned the 2 other GPUs, leading to a single cold start on each of those resources.

In terms of the time on queue, we can see that the round robin policy performs fairly poorly for both models, since resources aren't utilized well. The head of line blocking phenomenon, as well as queue growth during this time may explain why latencies are significantly higher than the other policies. The other notable policy in terms of latency results is SRTF.

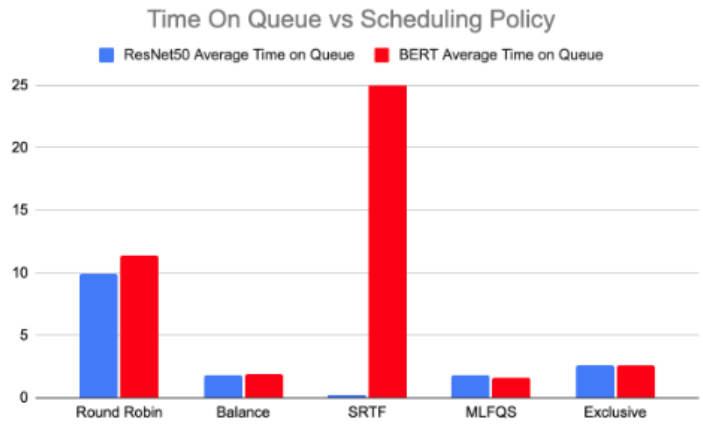


Figure 6.5: Average time spent on queue for different scheduling policies. The SRTF Average Time on Queue for BERT is 80.65 timesteps.

ResNet50 requests are serviced as soon as there are available resources, but this leads to starvation of BERT models. We can see that the average time on the queue for BERT is very high for this reason.

6.3 Hyperparamter Sweeps

In order to see how the various schedulers would perform, we performed sweeps of the various model and system parameters. In particular, we looked at how varying cold start time, model execution time, queue depth (i.e arrival rate in our poisson process), number of GPU resources, and number of clients would affect our results.

Varying Arrival Rate

We measured the average time spent on the queue when varying the poisson arrival rate to our queue. We submitted requests from 2 models (ResNet50 and BERT, with the same cold start/eval time assumptions as above), and had 3 GPU resources available to service requests. We run our simulation for 1000 timesteps.

As we increase the queue depth, the average time to service requests increases unboundedly. This makes sense, since the resources cannot service requests at the same rate (instability) at which they arrive leading to later-arriving requests taking a longer time to be serviced. Unsurprisingly, the Round Robin policy again performs the worst at all queue depths, primarily due to poor resource utilization.

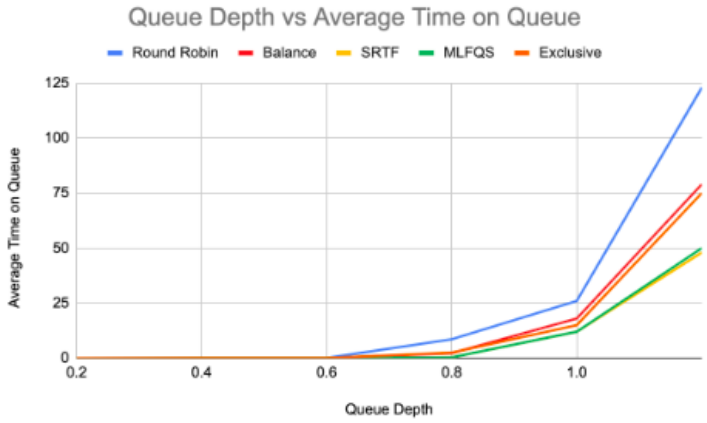


Figure 6.6: Queue Depth vs. Average Time on Queue

Varying Number of Resources

We also measured the average time spent on the queue as we increased the number of GPU resources. Much like before, we submitted requests from 2 models (ResNet50 and BERT, with the same cold start/eval time assumptions as above). This time, we fixed the average queue depth at 80%.

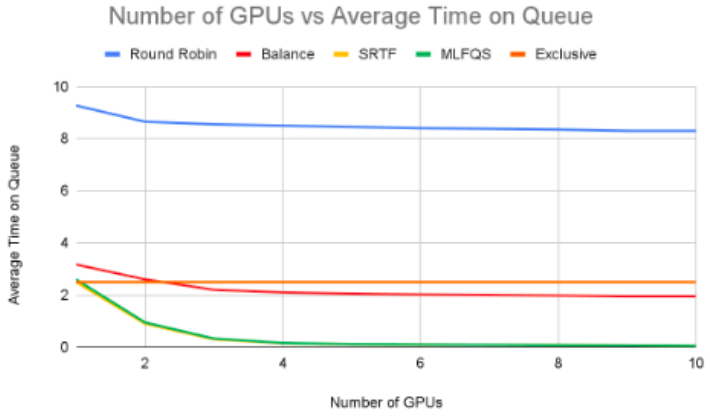


Figure 6.7: Num GPUs vs. Average Time on Queue

As we increased the number of resources, the average time it took to service requests generally tended downwards, since there was a higher chance a free resource would be available and head of line blocking wouldn't be an issue. The Round Robin policy again performed poorly even as the number of GPUs increased because it was likely that the policy would block on a particular resource before continuing.

Varying Cold Start / Execution Time / Number of Clients

Varying the cold start time had no discernible effect on the average time on queue in simulated results, since there are only a constant number of cold starts. As the overall simulation time increased, any changes were overshadowed by the significantly larger number of model invocations.

Varying execution time and the number of clients submitting requests also had no effects on the average time requests spent on the queue, regardless of the scheduling policy. This is because as execution time or number of clients increase, the absolute number of those requests on the queue reduces in order to maintain an 80% queue fill factor.

Chapter 7

Conclusion

In this thesis, I explored the theoretical and practical challenges of integrating GPUs into serverless platforms. Certain applications like model serving have bursty workload patterns that are conducive to a serverless solution and benefit from accelerators. I describe the engineering challenges around adding GPU support for FaaS, and highlight some of the latency and throughput performance wins with optimizations like batching.

I discuss better utilization of these high in demand and limited in supply GPU resources. In order to prevent systems from being over utilized, I developed an admission-control based algorithm that looks at existing resource utilization and decides whether it's acceptable to allow a new request. In particular, I consider user provided service level objectives (SLOs) around cost and latency, and explore how we'd work with these constraints to schedule across a heterogeneous set of resources (CPUs and GPUs).

There are inherent system limitations in the FaaS setting, particularly around the cold start problem, and inefficient resource utilization. I explored the recently proposed Kernel as a Service (KaaS) paradigm, where the system is responsible for managing GPU memory and scheduling requests across the entire pool of available GPUs rather than relying on static allocations. In particular, I looked at policies around request scheduling to improve the performance of KaaS in terms of fairness, minimizing cold starts and minimizing the average waiting time for requests until they're serviced.

The rise of serverless computing has coincided with the increased demand of accelerator resources like GPUs. Ultimately, this thesis explores an efficient, easily-usable abstraction of putting the two together, which can bolster applications like model serving.

Bibliography

[1] Benjamin Carver et al. "Wukong: A scalable and locality-enhanced framework for serverless parallel computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 1–15.

[2] Daniel Crankshaw et al. "Clipper: A {Low-Latency} Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[3] Arpan Gujarati et al. "Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 443–462.

[4] Paras Jain et al. "Dynamic space-time scheduling for gpu inference". In: *arXiv preprint arXiv:1901.00041* (2018).

[5] Ameet V Joshi. "Amazon's machine learning toolkit: Sagemaker". In: *Machine Learning and Artificial Intelligence*. Springer, 2020, pp. 233–243.

[6] Zhen Lin et al. "FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes". In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 2021, pp. 38–45.

[7] Philipp Moritz et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[8] Christopher Olston et al. "Tensorflow-serving: Flexible, high-performance ml serving". In: *arXiv preprint arXiv:1712.06139* (2017).

[9] Nathan Pemberton. "Kernel as a Service (KaaS)". In: 2022.

[10] Francisco Romero et al. "{INFaaS}: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 397–411.

[11] Mohammad Shahrad et al. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 205–218.

[12] Vikram Sreekanti and Harikaran Subbaraj. "Cloudflow". In: *arXiv preprint arXiv:2001.04592* (2020).

[13] Vikram Sreekanti et al. "Cloudburst: Stateful functions-as-a-service". In: *arXiv preprint arXiv:2001.04592* (2020).

[14] Ali Tariq et al. "Sequoia: Enabling quality-of-service in serverless computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 311–327.

[15] Mario Villamizar et al. "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 179–182.

[16] Chenggang Wu et al. "Anna: A kvs for any scale". In: *IEEE Transactions on Knowledge and Data Engineering* 33.2 (2019), pp. 344–358.