Disaggregated GPU Acceleration for Serverless Applications

Henrique Fingler
University of Texas at Austin

Zhiting Zhu
University of Texas at Austin

Esther Yoon
University of Texas at Austin

Zhipeng Jia
University of Texas at Austin

Emmett Witchel
University of Texas at Austin

Christopher J. Rossbach
University of Texas at Austin

Abstract

Serverless platforms have been attracting applications from traditional platforms because infrastructure management responsibilities are shifted from users to providers. Many applications well-suited to serverless environments could leverage GPU acceleration to enhance their performance. Unfortunately, current serverless platforms do not expose GPUs to serverless applications.

We present DGSF, a platform that enables serverless applications to access virtualized GPUs by disaggregating resources. DGSF facilitates provisioning and addresses utilization challenges by allowing a small pool of remote physical GPUs to serve potentially many serverless applications concurrently. With DGSF, the cloud provider decouples GPU resources from others, facilitating resource consolidation.

In this article, we describe how DGSF tackles GPU disaggregation challenges using API remoting virtualization, and optimizations, which include hiding communication latency and pooling resources. Our evaluation shows that these API remoting optimizations can lower the runtime of an application by up to 50% relative to an unoptimized API remoting scheme. Because these optimizations aggressively remove the latency of GPU runtime and object management from the application's critical path, they can enable applications executing on DGSF to have lower end-to-end time than when running on a GPU natively. Through consolidation, DGSF can lower queueing delays of application that use GPUs by up to 53%. We also demonstrate DGSF's flexibility by augmenting applications on AWS Lambda with GPU support.

 This work is based on an earlier work: DGSF: Disaggregated GPUs for Serverless Functions, in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 739-750, doi: 10.1109/IPDPS53621.2022.00077.

1 Introduction

The continuous migration of event-driven applications from conventional deployment infrastructure, such as infrastructure as a service (IaaS), towards serverless platforms [33, 49, 58] is driven by factors like rapid access to scalable resources and the offloading of operational concerns such as infrastructure management. Many applications that are well-suited for serverless environments could leverage GPU acceleration to significantly enhance their performance. Unfortunately, general access to GPUs to serverless platforms is in its infancy [2, 3]. Some providers support GPU acceleration indirectly, by specializing and modifying library APIs (e.g. AWS Elastic Inference exposes ML APIs like TensorFlow) to use GPUs. These services are not accessible to serverless applications that do not use the library supplied by the provider.

Naively supporting GPUs for serverless platforms is trivial: provision a subset of the machines in a datacenter with GPUs, use existing virtualization techniques (e.g., by deploying CUDA-enabled containers [35, 39]) and exclusively schedule applications that use GPUs to those machines. However, this immediately leads to provisioning challenges for the provider. Installing too many GPUs is prohibitively expensive and will lead to under-utilization of GPUs. Provisioning fewer GPUs can lower that cost, but leads to a difficult scheduling problem: matching functions that need CPUs, host memory and GPUs with machines that actually have the required resources requires complex high-latency scheduling algorithms, and remains an active area of research [11, 46, 52, 53]. We believe these problems are the main reason current serverless providers either do not support accelerators or provide minimal applicable support; there is no cost- and complexity-effective practical solution.

Installing GPUs in just some machines can lower that cost, but leads to a difficult problem: matching functions that need CPUs, host memory and GPUs with machines that actually have the required resources requires complex high-latency scheduling algorithms, and remains an active area of research [11, 46, 52, 53]. We believe current serverless

providers do not support accelerators because they lack a practical solution: it is difficult to provision the infrastructure in a cost- and complexity-effective way.

A compelling technique that can assist the design of a GPU-enabled serverless platform is disaggregation of the physical GPUs. Disaggregation allows the provider to independently manage and scale CPU and GPU resources to minimize cost and maximize utilization. Disaggregation simplifies the scheduling problem by separating resources: without disaggregation, both CPU and GPU requirements must be satisfied by a single host while, with disaggregation, CPU and GPU requirements are decoupled.

However, realizing a disaggregated system to support GPUs for serverless functions requires solutions to a number of challenges, which we address in this paper:

C1 Preserving the serverless programming model: the GPU should appear local to the application. Requesting and utilizing a GPU should not require any infrastructure management.

C2 Preserving the expected performance of GPU acceleration in the face of overheads introduced by disaggregation.

C3 Improving and load balancing GPU utilization.

DGSF is a platform for enabling general disaggregated GPUs for serverless functions. DGSF makes use of API remoting [21, 60] specialized for serverless, allowing a virtual GPU to be backed by part of a physical GPU on a remote server. DGSF uses API remoting-based GPU virtualization to share GPUs across potentially many functions, consolidating GPUs to increase utilization. DGSF solves C1 by transparently exposing the GPU runtime API (our prototype uses CUDA) in such a way that the GPU appears to be local from the perspective of the function. DGSF solves C2 by optimizing the API remoting system for the serverless environment by specially handling some interposed API calls. For example, the GPU runtime context and common handles are precreated to reduce initialization overhead. DGSF solves C3 through transparent workload migration across GPUs. Using statistics collected at runtime, DGSF's GPU server can load balance utilization by moving the execution of an application from one GPU to another.

Our DGSF prototype provides functions with CUDA runtime version 10.1. We study the performance of the prototype with six benchmark applications (§6) that use the CUDA API directly or through GPU-enabled libraries, like CuPy, OpenCV, TensorFlow and ONNX Runtime. This paper makes the following contributions.

- DGSF uses novel techniques to specialize API remoting for the serverless environment. These optimizations can improve the runtime of a function by up to 50% relative to unoptimized DGSF.
- We describe new techniques for live migration that use low-level GPU memory management to preserve address space mappings across GPUs.
- During a heavy load of GPU functions, DGSF with GPU sharing can complete all requests in 20% less time relative to DGSF without GPU sharing.

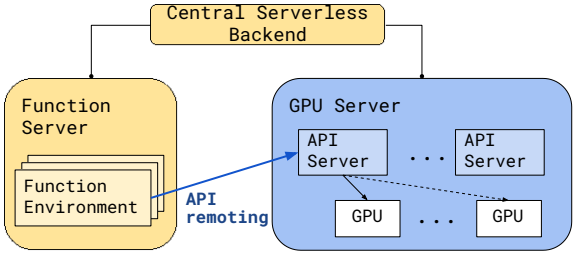


Figure 1: Architecture of a serverless deployments using DGSF. Components in blue are the scope of this work. Components in yellow already exist in serverless deployments.

2 Background

DGSF's goal is to enable the use of GPUs by serverless functions, while not adding more limitations for the user and not making management more difficult for the provider. GPUs for serverless platforms would ideally be as fast as a local GPU (for the client) and easy to consolidate onto a limited number of physical GPUs (for the provider). GPU consolidation is notoriously difficult [16, 32, 34, 37, 43, 50]. DGSF schedules applications optimistically, without needing complex scheduling algorithms. In case a scheduling choice was not optimal, DGSF can live migrate the execution between GPUs.

Virtualization through remote execution removes the need for GPUs to be physically in the same machine that will execute the application, and allows late binding of physical GPUs to functions. There are many flavors of remote execution for accelerators, such as forwarding the PCIe bus traffic [32] and remoting driver and/or API calls [4, 23, 44, 56, 60]. DGSF disaggregates GPUs for serverless functions by virtualizing GPUs at the runtime API layer (CUDA), which allows many serverless functions to use few remote GPU-provisioned servers, potentially increasing utilization. For the provider, this approach retains the "schedule anywhere" benefits of serverless because serverless functions that need GPUs can be scheduled on machines without requiring those machines be provisioned with physical GPUs. DGSF optimizes access to GPUs using API remoting using techniques that range from generic batching to pre-initialization of remote GPUs to hide startup latency.

Although existing work can be used to provide GPUs to serverless functions, none rely on specializations that enable an efficient deployment; DGSF is the first system to meet all requirements. We refer the reader to the original paper for a more detailed comparison to previous work [22].

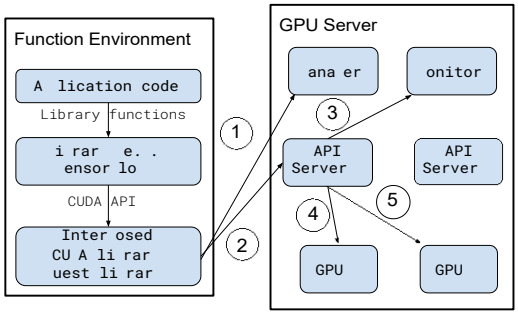


Figure 2: Internal architecture of DGSF.

3 Scope

This work (Figure 1) explores the disaggregation of physical GPUs, which reside within GPU servers, from serverless applications.

A GPU server is a disaggregated GPU machine: it contains GPUs and a few CPUs, and exclusively handles incoming APIs from applications. Scaling up GPU servers in DGSF is simple: a GPU server is provisioned and signals its availability to the main serverless coordinator.

Outside the scope of this work is general serverless function management, such as application scheduling and execution environment management (e.g., optimizing creation [20, 38] and destroying execution environments).

4 Design

This section details DGSF's system architecture. DGSF is agnostic to the serverless platform the applications execute under. In this work we focus on serverless functions. We describe the implementation details in Section 5.

4.1 Serverless GPUs

DGSF disaggregates GPUs from serverless functions using API remoting. On a traditional server with physically attached GPUs, applications access GPUs through vendors' runtime libraries, such as CUDA libraries for NVIDIA GPUs. With API remoting, a shim, which we call guest library, is inserted between the application and the original accelerator library. The guest library intercepts all API calls and executes them at a remote server (API server). API servers are processes on a GPU server that execute the intercepted API calls on behalf of the guest application, using the vendor's runtime library. By interposing the runtime library, DGSF supports transparent GPU acceleration for applications that use the runtime library directly or indirectly, for example, through libraries like TensorFlow for machine learning or CuPy for scientific and array computation that have support for CUDA.

Figure 2 summarizes the architecture of DGSF. A GPU server comprises a set of physical GPUs, a centralized manager, a monitor and API servers. The manager is responsible for setting up the environment, checking the available GPUs and creating the monitor and the initial idle API servers. The monitor is the main component of the GPU server, maintaining GPU and API server statistics and scheduling applications to API servers by using scheduling policies to choose an appropriate API server. The monitor also tracks how much memory is allocated by each API server and the memory and processor utilization of each GPU. Using such data, the monitor can observe each application's behavior and decide whether to rearrange the API server-GPU assignment.

An API server is a process that exclusively handles one serverless function at a time and executes the intercepted APIs on a physical GPU. It is initially assigned to one of the available GPUs, but this assignment can change through live execution migration. API servers are processes, thus multiple can share a physical GPU. API calls intercepted from the applications are forwarded to an API server through TCP. APIs can be handled by executing it on a physical GPU or, if the API is restricted, simulating the result of the call. This is necessary because information internal to the GPU server should not be available to the function. For example, if the function asks how many GPUs there are through the cudaGetDeviceCount function, the API server should always reply with 1 to maintain isolation and provide only what the user requested. In subsection 4.2 we describe the other API calls that require special handling.

Before APIs called by an application start being remoted to an API server, the guest library must first 1 ask the monitor of a GPU server (which was chosen by the serverless backend) the address of an API server. With the address of an API server, the guest library 2 sends information about its kernels and API remoting starts. Statistics are frequently sent by API servers to the monitor and captured independently by the monitor to track utilization of each GPU. An API server is initially assigned to a GPU 4 and will execute all remoted APIs in that GPU. During API execution, the monitor can decide to move the API server to another GPU . 5

4.2 Specializations for Serverless

DGSF classifies APIs into two categories: remotable and localizable. Localizable APIs are not forwarded to the API server because they can be immediately responded by the guest library using internal information or they can be safely ignored. Remotable APIs require the guest library to communicate with the API server and request execution. Some remotable API calls require special attention: ones that do memory management, kernel launching and device management functions.

Memory management. APIs such as cudaMalloc and cudaFree are handled in a special way because DGSF does not use general device memory allocation functions. Instead, DGSF manually allocates physical memory on the GPU, re-

serves virtual address ranges and maps the allocation to the reserved virtual address using CUDA's universal virtual addresses and low-level memory management. This is required to support API server migration from one GPU to another, since the same virtual addresses must be kept to ensure that all memory accesses done by the application are correct. By keeping information about all memory management functions, DGSF knows exactly how much memory an application is using and ensures that it is not violating its limits. Virtual address conflicts cannot occur because there is one virtual address space per CUDA context, and each API server in DGSF has, by construction, one CUDA context per GPU.

Kernel launches. To launch a kernel, a device function pointer must be passed as argument. These function pointers are unique to each CUDA context, thus, different for each GPU. For migration, DGSF makes sure it is using the correct function pointer by keeping a map of the function pointer in the original GPUs to the function pointers on other GPUs. DGSF does not support applications that use multiple contexts (e.g. through using cuCtxCreate) and each API server has only one context for each device. All applications and libraries in our workloads follow this requirement without modifications.

Device management functions. Applications must not observe the entire hardware of GPU servers for isolation. When an application starts execution, it is assigned to an API server, and the API server is assigned to a GPU: the mapping of API server to GPU is one-to-one, meaning that DGSF allocates a single GPU per application, and this GPU can be shared with other API servers. TensorFlow for example, first asks the runtime how many GPUs there are, gets their properties uses the best fitting GPU found. For this reason, the API server must always respond there is only one GPU (index 0), notwithstanding the fact that the API server could be assigned to any GPU (index from 0 to number of GPUs) and that the GPU server probably has more than one GPU. For GPU property queries, the information returned is from the currently active GPU. The application trying to utilize any GPU other than the GPU at index 0 is invalid and will cause an error. Our prototype of DGSF does not support applications that use multiple GPUs because we do not know of multi-GPU applications that are a good fit for serverless. However, there is no fundamental issue preventing DGSF from being extended for multiple GPUs. Such extensions would be straightforward.

4.3 Optimizations for Serverless

Startup optimizations. Each API server initializes the CUDA runtime before accepting API requests because this takes the initialization cost off of the execution path for the serverless function. APIs that would create a context, e.g. cuInit, become a no-op. In our experiments (§7.5) the CUDA runtime initialization takes on average 3:2 seconds. This number can vary, according to our observations, from 2:8 to 3:6, depending on the GPU model, driver version and other hardware parameters. The CUDA initialization time is consistent within a machine, varying by less then 200 milliseconds. Each API server also pre-creates cuDNN and cuBLAS handles, which can be immediately returned when the corresponding API is called (e.g. cudnnCreate). A cuDNN handle takes on average 1:2 seconds to be created on the machines used in our evaluation. A cuBLAS handles takes 0:2 seconds to be initialized. In total, an idle DGSF API worker with its precreated CUDA runtime, cuDNN and cuBLAS handle occupies 755 MB of device memory on its assigned GPU and takes approximately 4:6 seconds to fully initialize.

These optimizations significantly improve the initializion of machine learning models. The impact of such optimizations is presented in §7.3. Native GPU applications cannot pre-initialize their own runtime, since its creation is tied to the process' virtual address space.

Guest library. DGSF precreates cuDNN-specific descriptor structures (e.g. cudnnConvolutionDescriptor_t) on the guest library for immediate return. APIs that create these descriptors are called often and simply allocate a small amount of memory on the host, without changing GPU state, to hold the opaque structure. The pooling of such descriptors avoids the remoting of the corresponding APIs, speeding up most serverless functions that use cuDNN. APIs that only change host state, such as cudaMallocHost, are fully emulated on the client side and are not remoted to the API server.

Optimizing GPU API remoting. The vendor provided GPU libraries are designed for local use, not for use over a network, so there is no limitation in frequency of API calls, which makes designing an efficient API remoting system difficult. DGSF optimizes frequently called GPU APIs in a few ways. First, DGSF's runtime directly emulates some GPU APIs. The semantics of such API functions are preserved through other mechanisms. The attributes of a pointer, for example, can be responded by the guest library without remoting, since it tracks the addresses returned by device memory allocation functions. APIs that don't cause an immediate change to GPU state are accumulated locally and sent in batches to the API server.

DGSF is able to reduce the number of forwarded CUDA APIs when doing inference by up to 48% for ONNX runtime and up to 96% for TensorFlow. Figure 4 shows that these optimizations can reduce inference time by up to 59%. The optimizations presented could be applied to most API remoting systems, which includes non-disaggregated systems.

4.4 Migration

Scheduling applications to API servers is difficult: poor visibility of application properties makes scheduling vulnerable to poor decisions, and such decisions can affect the performance of the applications. For example, some scheduling

decisions may cause load imbalance in GPU utilization. We explore a scenario with such imbalance in §7.5.

To avoid GPU load imbalance, DGSF monitors GPU utilization and, if the monitor notices imbalance, it requests an API server to move execution to another GPU. In order for the application to correctly run on the new assigned GPU, the GPU's virtual address space must remain the same. Translating pointers passed as arguments to API calls is not enough since indirect pointers, like device pointers stored in an application's data structure would not be translated. DGSF maintains device virtual address space by leveraging CUDA's low-level memory management functions to manually manage memory. For example, the cuMemAddressReserve API reserves a virtual address range that will be mapped to a physical memory allocation, created by cuMemCreate. This virtual address range can be remaped to physical memory of a different GPU. On migration, the API server must switch CUDA context since it changes GPUs. This requires all context-dependent data (e.g. CUDA streams) to be moved and translated to the new context. After all data and context are copied between GPUs, the API server can resume function execution.

5 Implementation

Our prototype provides applications with the CUDA runtime version 10:1 and uses OpenFaaS [6] v0.21.1 as serverless platform. To demonstrate DGSF's flexibility, we also deployed our workloads and DGSF's guest library on AWS Lambda. DGSF is agnostic to the serverless platform, implementation and execution environment. DGSF only requires that its shared interposition libraries are correctly loaded to replace the original GPU libraries. This is accomplished with LD_PRELOAD or library path manipulation. We refer the reader to the original DGSF paper [22] for more details of our prototype.

6 Workloads

We evaluate our DGSF prototype on six machine learning-based workloads (Table 1): K-means [28], CovidCTNet [7, 31], face detection [14, 59], face identification [1, 13, 29], a natural language processing-based question answering application [15, 42, 45] and image classification [12, 27, 45]. These are general enough to represent general applications. A more detailed description of each workload, including libraries and models used, application's inputs and batch size is presented in the original paper [22].

7 Evaluation

DGSF's evaluation aims to answer the following questions:

What is the cost of API remoting and what is the impact of DGSF's optimizations?

What is the utilization increase and performance gains when functions are consolidated?

What is the overhead of migration and how can it improve GPUs for serverless functions?

7.1 Testbed

Experiments were performed on AWS EC2 using two p3.8xlarge machines, each with 4 NVIDIA V100 GPUs with 16GB of memory, 32 vCPUs of an Intel Xeon E5-2686, 244 GB of memory and a network interface of up to 10Gbps. We run the function server and the GPU server on identical virtual hardware to avoid performance variability.

7.2 API Remoting

We measure our workloads when executed natively (the baseline) and under DGSF's API remoting mechanism (Table 1). Comparison between GPU and CPU execution is presented to show scale and to demonstrate that DGSF preserves GPU acceleration benefits [30]. For CPU measurements each application uses 6 threads (6 vCPUS is the maximum cores per function in AWS Lambda). Workloads can be faster when running over DGSF's API-remoting than when executed natively because our optimizations aggressively hide runtime latencies (e.g. CUDA initialization) that cannot be hidden in the native environment.

To characterize DGSF's API remoting performance, we break down the execution time of the workloads: CUDA context initialization, input and ML model download, model loading and processing time. Results are shown in Figure 3. For a simple workload like K-means, which uses few CUDA APIs and no cuDNN or cuBLAS, the benefit comes exclusively from pre-creating the CUDA context. Other workloads, such as face detection, also benefit from the optimizations described in §4.3.

On AWS Lambda using DGSF's API remoting, workloads that require more network transfers, such as NLP and image classification, there is a spike in total execution time. This is due to lower, unguaranteed and variable network bandwidth. Other workloads behave similar to our deployment of OpenFaaS.

7.3 Ablation Study

To understand the benefits of each optimization, we perform an ablation study, breaking down execution time as we incrementally add the optimizations described in Section 4.3, and compare against native execution.

We do not show input download from remote storage (S3) since these are not optimized by DGSF and are the same for all comparison points. Results are shown in Figure 4. Benefits are most pronounced for the face identification and image classification workloads.

	K-means	CovidCTNet	Face Detection	Face Identification	Question answering (NLP)	Image classification (ResNet)
Peak GPU Memory Usage	323 MB	7802 MB	13194 MB	3514 MB	4028 MB	7650 MB
Average Runtime (Native)	14.0s	25.1s	18.5s	13.4s	34.3s	26.7s
Average Runtime (DGSF)	9.9s (29%)	22.4s (10%)	16.4s (11%)	10.5s (22%)	32.4s (5%)	24.8s (7%)
Average Runtime (AWS Lambda)	9.9s (29%)	24.6s (2%)	17.9s (3%)	18.0s (-34%)	60.4s (-76%)	47.1s (-76%)
Average Runtime (CPU)	429.1s (-29.6)	99.2s (-2.9)	71.0s (-2.8)	42.1s (-2.4)	347.0s (-9.1)	66.7s (-1.5)
Aprox. Migration Time	12 ms	805 ms	1064 ms	711 ms	555 ms	798 ms

Table 1: DGSF workloads. Times are averaged over three runs after one warmup. Numbers in parentheses are speedup (slowdown, if negative) relative to native.

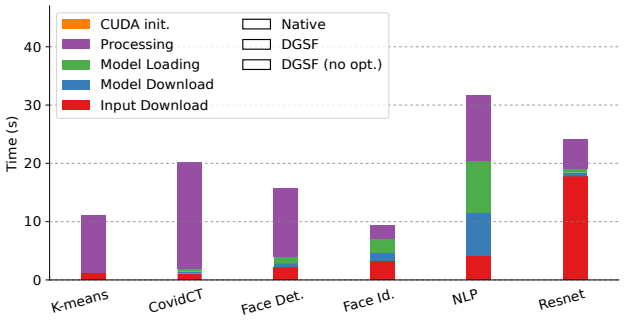


Figure 3: Breakdown of each step of our workloads when running natively, remoted through DGSF with and API remoting without optimizations.

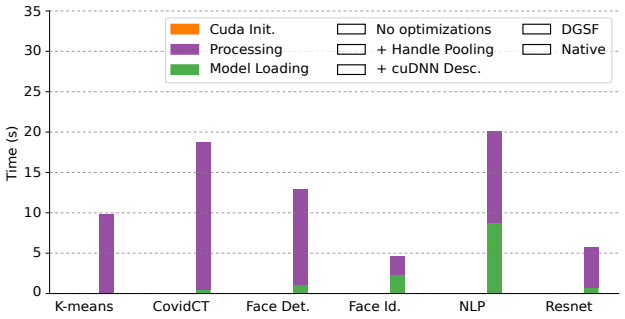


Figure 4: Ablation study of DGSF's optimizations compared to using a GPU natively.

For face identification, the total processing time is 14:5 seconds with DGSF using no optimizations. Context precreation reduces total processing time to 9:6 seconds, removing 4:9 seconds, which is roughly the time taken to initialize the CUDA, cuDNN and cuBLAS libraries (3:2, 1:2 and 0:2 seconds respectively). Avoiding the remoting of cuDNN descriptor creation APIs reduces inference time from 7:2 to 5:7 seconds. Batching APIs and avoiding unnecessary APIs further reduces inference time to 2:3 seconds. In total, DGSF's optimizations reduce inference time of the face identification workload by 67%: from 14:5 to 4:7 seconds. Face detection and NLP have a borderline improvement with DGSF's optimizations because fewer optimized APIs are called.

7.4 Mixed workloads

For the experiments in this section, we mix all six workloads while varying function invocation interval. Scheduling at the GPU server enforces a first-come first-serve policy per serverless function, which means that a serverless function requiring a large portion of the GPU (e.g. face detection), can force other serverless functions to wait in queue. We leave exploration of policies like shortest-function-first, which could improve throughput at some loss of fairness, for future work.

We use a poisson distribution to emulate a real sequence of function invocations. Ten instances of each workload are launched in a random (but consistent) order. On average our workloads utilize 12 seconds of GPU.

To emulate a GPU server under heavy load we use intervals drawn from an exponential distribution with rate equal to 2. This models a scenario where a function is launched on average every two seconds ($\lambda = 0.5$).

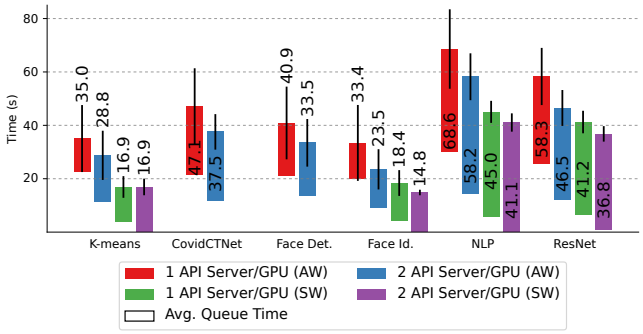


Figure 5: Per workload queueing and execution delay when the GPU server is under a high load, running two different subset of workloads: all workloads (AW) and the four workloads with smaller memory footprints (SW).

If there is no queueing on the GPU server, the end-to-end time for each workloads should not have a large variance and will be close to the uncontended runtime (see Table 1). Sharing reduces the average queue time of each function invocation and, consequently, the average time from launch to finish, as seen in Figure 5. The image classification finishes, on average, 20% faster when sharing is enabled and all workloads are used, due to a reduction of the queue time by

half.

We also emulate a GPU server under light load, which shows that, with DGSF and GPU sharing, the provider can reduce the number of active GPUs on a GPU server to reduce cost, without causing significant performance changes on the workloads. A deeper evaluation of this scenario is presented in the original paper [22].

7.5 Migration

The primary benefit of live migration across GPUs is to recover from scheduling decisions that (unpredictably) harm performance by creating contention or load imbalance. A best-fit scheduling policy tries to condense as many functions as it can into GPUs, while worst-fit tries to spread the load across GPUs, possibly causing fragmentation and higher queueing latency. Cost could be reduced through maximizing function packing, but it can leave some GPUs idle while others are oversubscribed. Migration can help mitigate possible performance issues by moving API servers between GPUs.

We explore a scenario using the NLP and image classification workloads, using only two GPUs, each with 15GB of free memory (1GB is used by the API servers' contexts), and four API servers. We launch two NLP workloads and two image classification workloads. Because the image classification workloads require more data to be downloaded, the NLP workloads will start using the GPUs first. The baseline comparison does not GPU sharing: an NLP workload is assigned to each GPU. Then, when the image classification functions want to use GPUs, they must wait in queue until a GPU is available. The total time to completion is 43:6.

With GPU sharing enabled, more scheduling options become available. A worst-fit scheduler performs the best: one image classification and one NLP workload share each GPU. The total time is 38:9 seconds, an improvement of about 11% over the baseline.

A best-fit approach yields the worst scenario: the two NLP workloads share a GPU and the end to end execution takes 50:6 seconds. Because the NLP workloads are computation-heavy, they don't share the GPU well. The two image classification workloads run serially on the other GPU and finish before the NLP ones, causing one of the GPUs to be idle while the other is contended. This effect can be seen in Figure 6b, where the utilization for GPU 2 falls to zero while GPU 1 stays at 100% for over 24 seconds.

Figure 6b shows that we can improve utilization and, consequently the total runtime, by moving execution of one of the applications running on GPU 1 to GPU 2. The utilization for best fit policy with DGSF's migration mechanism enabled is shown in Figure 6c. When the second image classification workload finishes, an imbalance is observed and the migration of one of the NLP workloads is triggered. DGSF improves the end-to-end runtime to 42:6 seconds, a 16% improvement over best fit with no migration.

8 Related work

GPU virtualization. Cloud providers can expose GPUs to virtual machines using PCIe passthrough which dedicates the hardware interface directly to a virtualized environment, prohibiting sharing and causing underutilization [8]. Full-virtualization [48, 51], mediated pass-through (MPT) [40, 51, 55], para-virtualization [19] and SR-IOV [5, 17, 18] techniques have limitations that have hampered adoption in production cloud environments [61].

Accelerator virtualization specialized for serverless functions is a relatively new research space. Existing literature use CUDA-enabled containers [35] and API-remoting [39] to simply expose GPUs to serverless functions; unlike DGSF, no serverless platform specific optimizations are done, neither is live migration supported.

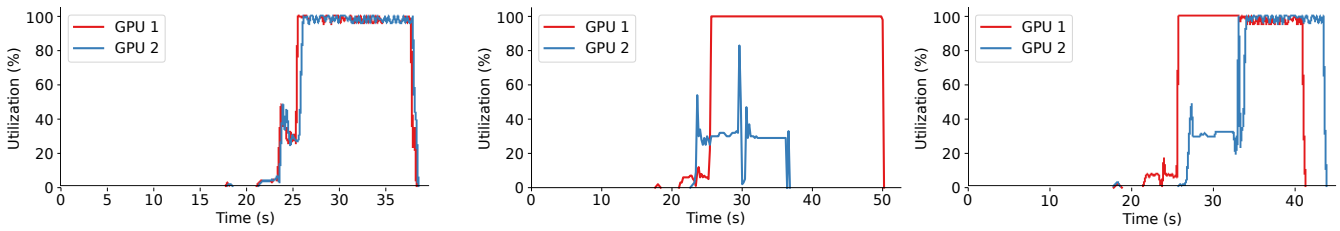
API remoting [4, 26, 56, 60] is a virtualization technique that interposes a user-mode API, forwarding calls to a user-level framework [47] on an appliance VM [54] or remote server [44]. API remoting is attractive for serverless platforms because it Disaggregates accelerator resources from other resources. Scheduling in such scenarios is easier and allows for several optimizations for heterogeneous workloads [9, 24, 52, 53].

GPU consolidation. Although plenty of literature exists, sharing GPUs is difficult [16, 36, 43, 50]. NVIDIA introduced Hyper-Q and MPS to increase utilization and improve sharing. While Hyper-Q is general and used by DGSF, MPS is aimed towards cooperative workloads and is not applicable for serverless. Since GPUs are ubiquitous in machine learning, many papers have focused on sharing for ML workloads [62]. For example, PipeSwitch [10] manually switches context of applications in GPU to ensure high utilization, while Gandiva [57] implements time-slicing.

GPU migration. Execution migration across GPUs is another heavily studied area of research [25, 41, 57] and is tightly coupled with consolidation. NVIDIA's GRID supports live migration of VMs between servers, which is not the case when API remoting is in place. DCUDA [25] uses peer-to-peer GPU memory accesses to migrate kernel executions without manually moving data. We experimented this approach for DGSF but found that it can incur large overheads and memory duplication, likely from the CUDA runtime ensuring safety and memory consistency. Gandiva [57] uses a checkpoint-restore approach, relying on library support.

9 Conclusion

DGSF is a platform that enables serverless functions to use GPUs through API remoting. DGSF disaggregates GPU resources from CPU resources, simplifiying scaling and resource management. DGSF enables GPU sharing, to increase GPU utilization, serving many functions with few GPUs.



(a) Worst fit scheduling.

(b) Best fit scheduling.

(c) Best fit scheduling with migration.

Figure 6: Memory and GPU utilization for a scenario where two NLP applications and two image classification are launched simultaneously, on a server with two GPUs and four API servers using different scheduling policies.

DGSF handles GPU utilization imbalance by migrating execution across GPUs transparently. DGSF provides performance comparable to, and often better than native by offsetting disaggregation overheads with optimizations specialized for the serverless environment.

Acknowledgements. This work is supported in part by NSF grants CNS-1846169, CNS-2006943, CNS-2008321 and CNS-1900457, and the Texas Systems Research Consortium.

References

[1] ArcFace. (Accessed: October 2021).

[2] Best practices for GPU-accelerated instances. (Accessed: May, 2023).

[3] Deploy GPU-enabled container instance - Azure Container Instances | Microsoft Learn. (Accessed: May, 2023).

[4] End-to-End Solutions for AI/ML Workloads |VMware. (Accessed: October, 2021).

[5] NVIDIA GRID. (Accessed: October 2021).

[6] OpenFaaS - Serverless Functions Made Simple. (Accessed: January 2021).

[7] ShahinSHH/COVID-CT-MD : A COVID-19 CT Scan Dataset Applicable in Machine Learning and Deep Learning. (Accessed: October, 2021).

[8] Underutilizing Cloud Computing Resources. (Accessed: October 2021).

[9] M. Amaral, Jordà Polo, David Carrera, N. Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D. D'Amora, A. Youssef, and M. Steinder. Drmaestro: orchestrating disaggregated resources on virtualized datacenters. Journal of Cloud Computing, 10:1–20, 2021.

[10] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In 14th USENIX OSDI 2020, pages 499–514. USENIX Association, November 2020.

[11] Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. In Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, pages 185–194. Citeseer, 1999.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In CVPR 09. IEEE, 2009.

[13] Jiankang Deng, Jia Guo, Xue Niannan, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In CVPR, 2019.

[14] Jiankang Deng, Jia Guo, Zhou Yuxiang, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. Retinaface: Single-stage dense face localisation in the wild. In arxiv, 2019.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[16] K. M. Diab, M. M. Rafique, and M. Hefeeda. Dynamic sharing of gpus in cloud systems. In 2013 IEEE ISPA, Workshops and Phd Forum, pages 947–954, 2013.

[17] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High Performance Network Virtualization with SR-IOV. Journal of Parallel and Distributed Computing, 72(11):1471–1480, 2012.

[18] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In Workshop on I/O Virtualization, 2008.

[19] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware's hosted I/O architecture. ACM SIGOPS Operating Systems Review, 43(3):73–82, 2009.

[20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In International

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 467–481. ACM, 2020.

[21] José Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA Acceleration Within Virtual Machines Using rCUDA. In Proceedings of the 2011 18th HIPC, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. Dgsf: Disaggregated gpus for serverless functions. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 739–750, 2022.

[23] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. Euro-Par 2010-Parallel Processing, pages 379–391, 2010.

[24] Anubhav Guleria, J Lakshmi, and Chakri Padala. Quadd: Quantifying accelerator disaggregated datacenter efficiency. In 2019 IEEE 12th International CLOUD, pages 349–357, 2019.

[25] Fan Guo, Yongkun Li, John C. S. Lui, and Yinlong Xu. Dcuda: Dynamic gpu scheduling with live migration support. In Proceedings of the ACM SoCC, page 114–125, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated Virtual Machines. In Proceedings of the 3rd ACM Workshop HPCVirt, pages 17–24, New York, NY, USA, 2009. ACM.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE CVPR, pages 770–778, 2016.

[28] B. Hu and C. J. Rossbach. Altis: Modernizing gpgpu benchmarks. In 2020 IEEE ISPASS, pages 1–11, 2020.

[29] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[30] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for GPU inference. In Thirty-second Conference on Neural Information Processing Systems, 2018.

[31] Tahereh Javaheri, Morteza Homayounfar, Zohreh Amoozgar, Reza Reiazi, Fatemeh Homayounieh, Engy Abbas, Azadeh Laali, Amir Reza Radmard, Mohammad Hadi Gharib, Seyed Ali Javad Mousavi, Omid Ghaemi, Rosa Babaei, Hadi Karimi Mobin, Mehdi Hosseinzadeh, Rana Jahanban-Esfahlan, Khaled Seidi, Mannudeep K. Kalra, Guanglan Zhang, L. T. Chitkushev, Benjamin Haibe-Kains, Reza Malekzadeh, and Reza Rawassizadeh. Covidctnet: an open-source deep learning approach to diagnose covid-19 using small cohort of ct images. npj Digital Medicine, 4(1), December 2021.

[32] Hee Seung Jo, Myung Ho Lee, and Dong Hoon Choi. Gpu virtualization using PCI direct pass-through. In Information, Communication and Engineering, volume 311 of Applied Mechanics and Materials, pages 15–19. Trans Tech Publications Ltd, 5 2013.

[33] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In Proceedings SoCC 2017, pages 445–451, New York, NY, USA, 2017. ACM.

[34] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pages 1–12, June 2017.

[35] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. Gpu enabled serverless computing framework. In 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pages 533–540, 2018.

[36] U. Kurkure, H. Sivaraman, and L. Vu. Virtualized gpus in high performance datacenters. In 2018 HPCS, pages 887–894, 2018.

[37] Kuan-Ching Li, Keunsoo Kim, Won W. Ro, Tien-Hsiung Weng, Che-Lun Hung, Chen-Hao Ku, Albert Cohen, and

Jean-Luc Gaudiot. On migration and consolidation of vms in hybrid cpu-gpu environments. In Jengnan Juang and Yi-Cheng Huang, editors, Intelligent Technologies and Engineering Systems, pages 19–25, New York, NY, 2013. Springer New York.

[38] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In 11th USENIX HotCloud 19, Renton, WA, July 2019. USENIX Association.

[39] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. Journal of Parallel and Distributed Computing, 139:32–42, 2020.

[40] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. MDev-NVMe: a NVMe storage virtualization solution with mediated pass-through. In 2018 USENIX ATC, pages 665–676, 2018.

[41] Javier Prades and Federico Silla. Gpu-job migration: The rcuda case. IEEE Transactions on Parallel and Distributed Systems, 30(12):2718–2729, 2019.

[42] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. CoRR, abs/1606.05250, 2016.

[43] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In Proceedings of the 20th HPDC, page 217–228, New York, NY, USA, 2011. Association for Computing Machinery.

[44] Carlos Reaño, Antonio J. Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S. Quintana-Ortí. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. 20th Annual International Conference on High Performance Computing, 0:1–10, 2012.

[45] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[46] Mehdi Sheikhalishahi, Richard M. Wallace, Lucio Grandinetti, José Luis Vazquez-Poletti, and Francesca Guerriero. A multi-dimensional job scheduling. Future Generation Computer Systems, 54:123–131, 2016.

[47] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. IEEE Trans. Comput., 61(6):804–816, June 2012.

[48] Jike Song, Zhiyuan Lv, and Kevin Tian. KVMGT: a Full GPU Virtualization Solution. In KVM Forum, volume 2014, 2014.

[49] State of the cloud report. https://www.rightscale.com/lp/state-of-the-cloud. (Accessed: January, 2021).

[50] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. Gloop: An event-driven runtime for consolidating gpgpu applications. In Proceedings SoCC 2017, page 80–93, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Kun Tian, Yaozu Dong, and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In 2014 USENIX ATC, pages 121–132. USENIX Association, June 2014.

[52] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In Proceedings of the Third ACM Symposium on Cloud Computing, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In Proceedings of the Eleventh ACM European Conference in Computer Systems (EuroSys), New York, NY, USA, 2016. Association for Computing Machinery.

[54] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In Proceedings of HPC Symposium, pages 2:1–2:8, 2014.

[55] Lei Xia, Jack Lange, Peter Dinda, and Chang Bae. Investigating virtual passthrough I/O on commodity devices. ACM SIGOPS Operating Systems Review, 43(3):83–94, 2009.

[56] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized GPU environment. In Proceedings of the 12th IEEE/ACM CCGrid, pages 124–131, 2012.

[57] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In 13th USENIX 2018 OSDI, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[58] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In Proceedings of the 1st MOTA, New York, NY, USA, 2016. Association for Computing Machinery.

[59] Shuo Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Wider face: A face detection benchmark. In 2016 IEEE CVPR, pages 5525–5533, 2016.

[60] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated virtualization of accelerators. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 807–825. ACM, 2020.

[61] Hangchen Yu and Christopher J Rossbach. Full Virtualization for GPUs Reconsidered. In 14th WDDD, ISCA, 2017.

[62] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, PLMR 20, volume 2, pages 98–111, 2020.