DGSF: Disaggregated GPUs for Serverless Functions

Henrique Fingler*, Zhiting Zhu*, Esther Yoon*, Zhipeng Jia*, Emmett Witchel*†, Christopher J. Rossbach*†
* The University of Texas at Austin, Department of Computer Science
† Katana Graph
{hfingler, zhitingz, esthery, zjia, witchel, rossbach}@cs.utexas.edu

***Abstract*—Ease of use and transparent access to elastic resources have attracted many applications away from traditional platforms toward serverless functions. Many of these applications, such as machine learning, could benefit significantly from GPU acceleration. Unfortunately, GPUs remain inaccessible from serverless functions in modern production settings. We present DGSF, a platform that transparently enables serverless functions to use GPUs through general purpose APIs such as CUDA. DGSF solves provisioning and utilization challenges with disaggregation, serving the needs of a potentially large number of functions through virtual GPUs backed by a small pool of physical GPUs on dedicated servers. Disaggregation allows the provider to decouple GPU provisioning from other resources, and enables significant benefits through consolidation. We describe how DGSF solves GPU disaggregation challenges including supporting API transparency, hiding the latency of communication with remote GPUs, and load-balancing access to heavily shared GPUs. Evaluation of our prototype on six workloads shows that DGSF's API remoting optimizations can improve the runtime of a function by up to 50% relative to unoptimized DGSF. Such optimizations, which aggressively remove GPU runtime and object management latency from the critical path, can enable functions running over DGSF to have a lower end-to-end time than when running on a GPU natively. By enabling GPU sharing, DGSF can reduce function queueing latency by up to 53%. We use DGSF to augment AWS Lambda with GPU support, showing similar benefits.**

***Index Terms*—cloud computing, serverless, FaaS, GPU, API remoting**

I. Introduction

There has been an exodus of event triggered applications from traditional application deployment on infrastructure as a service (IaaS) toward deployment using serverless functions [1], [2], [3] (or function-as-a-service, FaaS) due to benefits like fast access to elastic resources and offloading of operational concerns like infrastructure management. Many applications that are a good fit for serverless could enjoy significant performance improvements from GPU acceleration. Unfortunately, current access to accelerators from serverless functions is nonexistent in production services. Some providers enable GPU acceleration *indirectly* through specialized library APIs (e.g. AWS Elastic Inference exposes ML APIs like TensorFlow) but such services are not accessible to serverless functions and, moreover, do not provide general purpose access to GPU programming frameworks like CUDA.

The simplest way for an infrastructure provider to support serverless GPUs is to provision some subset of machines in a cluster with GPUs (e.g., by deploying CUDA-enabled containers [4], [5]). However, this quickly leads to a sticky provisioning challenge for the provider. Installing GPUs in all machines is prohibitively expensive and will lead to significant under-utilization of GPUs because many functions do not use them. Installing GPUs in just some of the 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 [6], [7], [8], [9]. 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 alternative, which is the focus of this paper, is to disaggregate 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 turning a 2-dimensional problem into two 1-dimensional problems. Without disaggregation, scheduling is more complex because 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 handful of different challenges which we address in this paper:

C1 Preserving the serverless programming model: remote GPU should appear local. Requesting and utilizing a GPU should not require any management or knowledge of its location.

C2 Preserving the performance promise of GPU acceleration in the face of overheads introduced by remote execution.

C3 Balancing load and maximizing utilization of remote GPUs.

We present DGSF, a platform for enabling general, transparent and disaggregated GPUs for serverless functions. DGSF makes use of API remoting [10], [11] *specialized for serverless*, allowing a virtual GPU for a serverless function to be backed by part of a physical GPU on a remote server, selected from a disaggregated pool. DGSF uses API remoting-based GPU virtualization to share those servers across potentially many functions, consolidating GPUs to increase utilization. DGSF solves **C1** by transparently supporting the GPU run-

time 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 library to the serverless setting through special handling of interposed API calls. For example, the GPU runtime context and common handles are precreated and pooled to reduce initialization overhead. DGSF solves **C3** using online load balancing through transparent migration across GPUs. By collecting information during runtime, DGSF's GPU server can fix GPU load imbalance 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 (§VII) that use the CUDA API directly or use GPU-enabled libraries, like CuPy, OpenCV, TensorFlow and ONNX Runtime. This paper makes the following contributions.

- DGSF uses novel tecniques to specialize API remoting for the serverless environment. DGSF's API remoting 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 APIs in a novel way to preserve identical address space mappings at the destination GPU.
- We explore the policy space to increase utilization under different server loads. During a burst of functions, DGSF increases average GPU utilization by 16%.
- We evaluate DGSF on six workloads (§VII). On a heavy load of GPU functions, DGSF with GPU sharing can complete all requests in 20% less time relative to DGSF without GPU sharing.

II. Background

Removing the burden of developers to manage infrastructure to execute applications is a key motivator for serverless functions. During function deployment the developer publishes its code, dependencies, and how much memory the function requires to run. After a function is deployed to a serverless provider, many concurrent requests to that function can be satisfied for developer and user, with all the management being done by the provider. Providers impose limitations on the functions, like small temporary storage, removal of external network addressability, limited execution time and lack of support to external resources like accelerators. We argue that missing accelerator support is untenable.

GPUs are vital for meeting the performance requirements of modern computationally demanding workloads. For example, recent work in machine learning shows that as model complexity increases, so does the total time to perform inference [12]. CPUs are unable to meet the performance goals for these complex models [12]; they require hardware acceleration, but cloud-accessible GPUs currently are cut off from the serverless ecosystem. One recent study found that small batch sizes can lead the GPU to utilization under 15% [12]. Other accelerators such as Google's tensor processing units are similarly underutilized [13]. Techniques to effectively share GPUs across serverless functions are urgently needed.

TABLE I: Feature comparison of existing API remoting systems.

System	API Remoting	GPU Sharing	Live Migration	Disaggregation	Supports Serverless
AvA [11]	✓	✓	✓	✓	×
rCUDA [10]	✓	✓	×	✓	×
DCUDA [23]	×	✓	✓*	×	×
Bitfusion [22]	✓	✓	×	✓	×
Gandiva [24]	×	✓	✓*	×	×
Antman [25]	×	✓	×	×	×
Kim et al. [4]	×	✓	×	×	✓
OSCAR [5]	✓	✓	×	✓	✓
DGSF	✓	✓	✓	✓	✓

The closest commercial solution to provide GPUs in a serverless fashion is AWS Elastic Inference (EI), which provides accelerators that are backed by a fraction or an entire GPU, but are *exclusive* to machine learning inference acceleration. EI requires applications to use modified ML libraries that interpose library functions and remote function calls through gRPC to servers with GPUs. EI is only supported in limited environments: EC2 VMs, SageMaker environments and ECS containers. There is no direct support for serverless functions (Lambda).

DGSF's goal is to enable the use of GPUs by serverless functions, while not adding more limitations to the user and not making management harder for the provider. Using DGSF, the developer specifies the amount of GPU memory a function requires just like it does for host memory. GPU support for serverless 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 utilization is notoriously difficult to consolidate [14], [15], [16], [17], [18]. DGSF consolidates optimistically by scheduling applications that fit in one GPU. But DGSF has a contingency plan in case the scheduling choice was not optimal: migrate execution to a different GPU.

Virtualization through remote execution removes the need for GPUs to be physically in the same machine that will execute a function and allows late binding of physical GPUs to functions. There are many flavors of remoting, such as remoting the PCI bus[17] and remoting driver and/or API calls [19], [20], [21], [11], [22]. DGSF supports GPUs for serverless functions by virtualizing GPUs at the runtime API layer (CUDA), allowing many serverless functions to use a small number of remote GPU-provisioned VM instances. For the provider, the 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. Moreover, it increases utilization of GPU-capable machines by sharing them. DGSF is optimized to ensure GPUs accessed using API remoting are as fast as possible, using techniques that range from generic batching to serverless-specific optimizations such as pre-initialization of remote GPUs to hide startup latency.

Table I shows a feature comparison of existing API-

remoting and DGSF. Live migration is only partially provided by existing work; these have an asterisk by the checkmark. For example, Gandiva [24] supports migration at the library level, relying on library functions that can snapshot-restore its state, e.g. TensorFlow's `train.Saver`, which snapshots a training session than can then be restored at a different location. DCUDA's migration [23] on the other hand, uses peer memory accesses to provide migration. When an application is migrated to another GPU, DCUDA does not explicitly move the data to the destination GPU's memory: application memory accesses may - and will - page fault and require data to be read on-demand from the peer GPU which has the data. These approaches are not fit for serverless because they lack generality. When migrating it is desirable to move data explicitly as to possibly create enough space for another function to utilize. At the time of writing of this paper, DGSF is the only GPU live migration approach that uses CUDA's low level memory management to maintain the application's virtual address space, but with physical allocations able to move between physical GPUs transparently. 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.

III. Motivation: Case Study

Some of the authors participated in a large (100+ students) graduate-level class where one of the assignments required the students to implement a project using CUDA. The students needed access to GPUs to write code, test and measure so they could write their report. Most of the university's lab machines had no GPUs, and manually creating accounts and scheduling student access to the very few machines with GPUs that the research group had was not feasible. The best commercial alternative was an external service which provides a web-based IDE that runs on top of a GPU-enabled container. Even when a container was not being actively used, the increased price of the container with GPU was being charged. The class was large and students were constantly using the service, so it was eventually cutoff due to an explosion in cost to the provider, leaving the students with no way to run their CUDA code. With DGSF, the IDE could use regular cheap containers and, when any code that uses the CUDA runtime API was executed, a serverless function would be invoked to handle it. This can reduce cost considerably since only GPU active use time is billed.

IV. Scope

This work (Figure 1) explores how a function realizes API remoting to a remote API server which has physical GPUs and how API servers running inside a GPU server are managed to increase GPU utilization. A GPU server is defined as a disaggregated GPU machine: it contains GPUs and a few CPUs and exclusively handles incoming API remoting. No functions or external code are executed by a GPU server. Scaling up GPU servers in DGSF is simple. A GPU server only needs the address of the central serverless backend

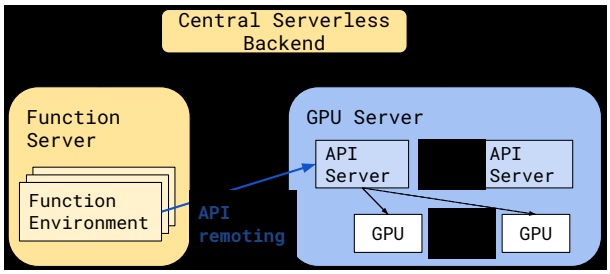


Fig. 1: Architecture of a serverless deployments using DGSF. Components in blue are the scope of this work. Components in yellow were developed during this work but are out of the scope of this paper.

to signal it's availability. After it is initialized and its API servers created, it annouces it is ready to handle functions and becomes a choice when a function requests a GPU. In our prototype, GPU servers runs within a NVIDIA container to simplify CUDA runtime versioning issues.

Outside the scope of this work is general serverless function management, such as scheduling functions requests to execution environments and execution environment management (creating and destroying environments for functions to be executed on). Reducing startup time is a vital goal for serverless research because each function's execution requires the initialization of a runtime that uses a deep software stack. Because this area is well explored [26], [27], DGSF factors it out by always using warm execution contexts. However DGSF optimizes GPU startup latencies by initializing remote GPU contexts in advance (§V-B).

Our DGSF prototype uses a fixed policy to choose, given a function requesting a GPU, which GPU server to use. Different policies can be used in a commercial deployment, such as choosing the least loaded GPU server to optimize latency or the opposite to increase utilization. Such policies would depend on the quantity of GPU servers and their size. For our evaluation we use one GPU server with four GPUs, but AWS provides machines with up to eight GPUs.

V. Design

This section details DGSF's system architecture. DGSF is agnostic to the serverless functions platform the functions run on, so we describe the implementation platform in Section VI.

A. Serverless GPUs

DGSF uses API remoting to virtualize GPUs for serverless functions. On a traditional machine with physically attached GPUs, application processes access GPUs through vendors' runtime libraries, such as CUDA libraries for NVIDIA GPUs. With API remoting, a shim (**guest library**) is inserted to interpose and intercept every API call which, through RPC, are executed at a remote server (**API server**).

In DGSF, the guest library captures all CUDA and CUDA related libraries, such as cuDNN and cuBLAS, and forwards them through TCP connections. API servers run on a host with physically attached GPUs (GPU server) and executes API calls on behalf of the guest application, via the vendor's

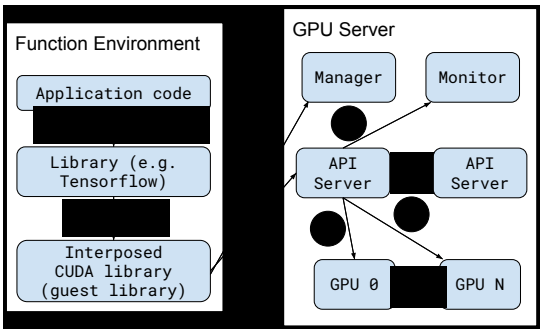


Fig. 2: Internal architecture of DGSF.

runtime library. By interposing at 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** consist of a set of physical GPUs, a **centralized manager**, a **monitor** and **API servers**. When a GPU server is provisioned, the first piece that runs is the manager, which is responsible for setting up the environment, checking the available GPUs and creating the monitor and the initial idle API servers. Once set up, it sends the serverless backend a message announcing that it is ready and how many functions it can handle (one per API server created). The manager then idles until it is shut down, passing all responsibilities to the monitor. The monitor is the main piece of the GPU server, maintaining statistics about the state of each GPU and API server and handling incoming function GPU requests by using scheduling policies to choose an appropriate API server. The monitor tracks how much memory is allocated by each API server and the memory and processor utilization of each GPU. Using its local information, the monitor can observe each application's behavior and decide whether or not to rearrange the API servers and their assigned GPUs. To illustrate this utility, consider a scenario where a GPU is being overwhelmed with API calls from two API servers, while there is an idle GPU. The monitor can decide to move one of the workers to the idle GPU to balance the load. This process is called *API server migration* and is described in Section V-D along with a scenario in which imbalance happens.

An API server is a process that handles exclusively one serverless function at a time and executes them on an actual physical GPU. An API server is a process created by the manager. It is initially assigned to one of the physically available GPUs. While handling requests for a function, the GPU assignment can change using migration. A serverless function may make any number of GPU API calls. Migration occurs at API call boundaries. When the current serverless function finishes, the API server changes its current GPU to the originally assigned one, if a migration happened. API servers are processes, thus multiple can share a physical GPU.

API calls done by the function are interposed and forwarded to an API server through TCP, which then handles the call either 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. In subsection V-B we describe the other API calls that require special handling.

Before the API functions of a serverless function start being remoted, the guest library must first ① talk to the monitor of a GPU server (which was chosen by the serverless backend) to receive the address of an API server. With the address of an API server, the guest library ② sends information about its kernels and can start remoting. The API server constantly ③ sends updates messages to the monitor so that it can keep track of utilization of each GPU. An API server is initially bound to a GPU ④ and will execute all remoted APIs in that GPU. During message exchange between the API server and the monitor, the latter can decide to move the former to another GPU ⑤.

B. API Remoting Specializations for Serverless

API calls made by the function are interposed by DGSF's custom preloaded library that is running underneath the application. There are two classes of APIs: remotable and localizable. Localizable APIs are not forwarded since they can be immediately responded by the guest library using internally cached information or can be safely ignored. Remotable APIs require the guest library to use our TCP-based RPC mechanism to talk to the API server and request execution. Some remotable APIs are forwarded but not realized as-is. For example, for `cudnnCreate`, which simply creates a handle necessary to use cuDNN, the API server can precreate a pool of these handles and simply return one of them when the API is called. For remotable API calls, some require special attention: ones that do **memory management**, **kernel launching** and **device management functions**.

Memory management. APIs such as `cudaMalloc` and `cudaFree` are captured and handled in a special way because DGSF does not use general device memory allocation functions. Instead, DGSF manually creates memory allocations on GPUs, reserves 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 necessary because if an API server is moved from one GPU to another, it must keep the same virtual addresses 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 ensure 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. A kernel launch takes as parameter a kernel function pointer. These function pointers are unique to each

CUDA context, thus, different for each GPU. For this reason the API server must make sure it is using the correct pointer for the current context in case the API server has migrated. 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 of the applications and libraries in our workloads follow this requirement without modifications.

Device management functions. Because GPU access is transparent to the function, it must not see the entire hardware of the GPU server. When a function starts execution, it is assigned to an API server and the API server to a GPU. TensorFlow for example, first asks how many GPUs there are, gets their properties and makes the best fitting one active. The API server always responds there's one GPU (index 0), notwithstanding the fact that the GPU server probably has more. 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. DGSF does not support applications that utilize multiple CUDA contexts. None of our workloads make use of multiple contexts and we believe such applications are not common. 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. Such applications, like ML training, are usually long running and would benefit from dedicated GPUs. However, there is no fundamental issue preventing DGSF from being extended for multiple GPUs. Such extensions would be straight forward.

C. API Remoting Optimizations for Serverless

Startup optimizations. Each GPU node maintains a pool of GPU API servers with their GPU runtime initialized. The GPU API server initializes the CUDA runtime because it takes the application-independent initialization cost off the execution path for the serverless function. In our experiments (§VIII-E) 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. A CUDA runtime context occupies ∼303 MB of device memory.

Each API server pre-creates a set of cuDNN and cuBLAS handles, which are be returned directly to serve the corresponding API calls (e.g. cudnnCreate). A cuDNN handle takes on average 1.2 seconds to be created on the machines used in our evaluation, and occupies around 386 MB of device memory. A cuBLAS handles takes ∼0.2 seconds to be initialized and occupies around 70 MB. 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.

These optimizations significantly improve the time of initializing machine learning models. The impact of such optimizations are presented in §VIII-C. Native GPU applications cannot pre-initialize their own runtime. GPU applications initialize the GPU and the GPU runtime when they start, and initialization maps data structures such as command rings into the process's address space. As a consequence, pre-initialization of the runtime is an optimization that works only if there is a separate server process to perform that initialization.

Guest library. DGSF precreates cuDNN-specific descriptors (e.g. cudnnConvolutionDescriptor_t) on the guest library side. APIs that create these descriptors are called often and simply allocate memory on the host side to hold the opaque structure. By pooling descriptors on the guest side, the remoting of corresponding APIs can be avoided, 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. GPU APIs are designed for local use, not for use over a network, so many programming idioms make frequent calls to GPU functions. DGSF optimizes frequently called GPU APIs in a few ways. First, DGSF's runtime directly emulates some GPU APIs (e.g., __cudaPushCallConfiguration and cudaPointerGetAttributes) without remoting them to the GPU API server. The semantics of such API functions are preserved through other mechanisms. The attributes of a pointer, for example, can be retrieved by the guest library without remoting because the guest library tracks the addresses returned by device memory allocation functions. Other functions such as pushing kernel launch configurations are not necessary since these configurations are piggybacked onto kernel launching APIs. 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. DGSF's GPU API servers maintain pools of frequently created CUDA descriptors such as cuDNN descriptors, thus remoting APIs for creating them will be much faster since the API server just returns one from the pool. Figure 4 shows that these optimizations can reduce inference time by up to 59%. Our optimizations could be applied to most *API remoting* systems, which include non-disaggregated ones. Importantly, they would not work for applications that use GPUs natively.

D. Migration

Choosing the best API server to handle a function is difficult since the only information the scheduler knows about a function is how much GPU memory it needs. Poor visibility makes scheduling vulnerable to poor decisions, and such decisions can affect the performance of the applications and/or cause load imbalance in GPU utilization. Consider two GPUs and two short and two long running applications. If the two short applications are scheduled to the same GPU, the long applications will share the other GPU until they finish. After the short functions finish, there will be an idle GPU and one

being contended by two applications. This scenario is explored in §VIII-E.

To avoid GPU load imbalance, DGSF monitors the API server assigments and GPU utilization. When the monitor notices imbalance, it may request an API server to move to another GPU. On migration, the API server must stop all of its threads that handle API calls and wait for completion of all pending operations. Then each of the application's memory allocations must be copied to the target GPU.

In order for the application to run correctly on the new assigned GPU, the 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 manually maintains the application's virtual address space by using CUDA's low-level memory management functions. For example, cuMemCreate allocates unmapped device memory, cuMemAddressReserve reserves virtual address ranges and cuMemMap maps device memory to a reserved virtual address. To move data to the destined GPU, DGSF creates temporary virtual addresses to access current data, allocates memory on the target GPU and maps it to the current virtual address. Finally the data is copied between the two devices and the previous device's allocations can be cleaned up. After all data is copied the API server can resume execution.

When migrating GPUs, the API server must switch CUDA context since it changed GPUs. This requires all context-dependent data to be moved and translated to the new context. For example, if an application is using a CUDA stream that was manually created, that stream's handle will not be valid after a migration. To work around this, the API server preemptively creates streams on each context when one stream is created and keeps a translation map of the stream handle returned to the client to internal stream handles in other contexts. This way, if the API server migrated, it can use this map to fetch a valid stream that has the same properties as the original one. This approach is also required for cuda events and cuDNN and cuBLAS library handles.

VI. Implementation

The GPU server of DGSF consists of approximately 134 thousand lines of c++: 778 for the manager and monitor, 80 thousand for the API server and 78 thousand for the guest library. Most of the code is automatically generated: we list all APIs and generate code for both sides of the API remoting system.

Although our current prototype provides applications with a CUDA runtime version 10.1, the API server runs on NVIDIA containers with CUDA 10.2 because the low level memory management functions are only present on this version and above. Supporting later versions of the CUDA runtime for both the guest library and API server is left as future work. For our workloads, the library versions used are ONNXRuntime v1.8.0 [28], TensorFlow v1.14.1, CuPy 9.2.0 and OpenCV 4.5.2. The CUDA libraries cuDNN and cuBLAS are versions 7.6.5 and 10.2, respectively.

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. Such can be accomplised with LD_PRELOAD or library path manipulation. In a real deployment with multiple GPU servers, the function scheduler would need to be augmented to choose a GPU server for functions that require GPU (§IV). Our prototype uses OpenFaaS [29] v0.21.1 as serverless platform. To demonstrate DGSF's flexibility, we also deployed our workloads and DGSF's guest library on AWS Lambda.

All of the data required by each function, such as models and inputs are downloaded from AWS S3. This would be the case in general, even without DGSF. For all our measurements we assume a warm start, where the cost of creating a container is avoided, for all functions. This is done by setting the minimum amount of replicas for each function.

VII. Workloads

K-means. K-means is a commonly used clustering algorithm. We use the CUDA K-means implementation in the Altis [30] GPU benchmark suite. The CPU version is a hand-optimized implementation in C using pthreads. We fix the input size to one million points in a 16-dimensional space, and group into 16 clusters for 2000 rounds. The total input size is 235.3 MB.

Covid Detection using CT scans. CovidCTNet [31] is an open-source application that uses deep learning to diagnose and differentiate covid-19 infection to other lung diseases, using CT scans as input. Because the original dataset is not publically available, we use an open source dataset [32] that has 305 labelled CT scans. The two models have a combined size of 47.3 MB. At each run, we do inference on two CT scans, which have an approximate total size of 155.5 MB. Although this application requires less memory than the entirety of a GPU, on a function invocation we request the memory of an entire GPU. This is because the application uses two TensorFlow models, whose custom memory allocators, for a brief moment during execution, allocates a large amount of memory: 13538MB. If we didn't oversize the function requirements, this workload would fail due to an out of memory error.

Face Detection. Face detection identifies candidates faces' using the RetinaFace neural network [33]. We use RetinaFace with ResNet50 backbone on top of ONNXRuntime. The input images are from WIDER FACE validation data set [34]. At each run, 256 images from the dataset are chosen, totalling approximately 30 MB. Batch size used is 16. The model has a total size of 104.4 MB.

Face Identification. Face identification uses ArcFace [35], a face recognition deep neural network, to compare a candidate face to the reference face. We use the LResNet100E-IR variant of ArcFace model [36] on top of ONNXRuntime. The input consists of 6000 face pairs from Labeled Faces in the Wild [37]. The input to each run is randomly chosen 256 faces

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

	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

from the dataset, totalling around 17 MB. The batch size used is 16. The model has a total size of 249 MB.

Question answering (NLP). We use the Bert model [38] from MLPerf [39] to perform SQuAD [40] question answering tasks using ONNXRuntime. At each run 512 inputs, which are questions created by crowdworkers on Wikipedia articles, are chosen from the dataset. The batch size used is 16. The inputs are approximately 61.7 MB and the model is sized at 1.2GB.

Image Classification. Image classification takes in an input image and selects a label that best describes it. We use ResNet-50 v1.5 [41] model from MLPerf [39] on ONNXRuntime. The input images come from ImageNet 2012 validation data set [42]. At each run 2048 preprocessed images from the dataset in NumPy format are chosen (approximately 1.2 GB) and processed using a batch size of 16. The model has a total size of 97.4 MB.

VIII. 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.

A. Testbed

Experiments were performend on AWS EC2 using two p3.8xlarge machines. We run the function server and the GPU server on identical virtual hardware to avoid performance variability due to different machine specifications, such as CPU and network and storage bandwidth. Ideally, function servers would be deployed on compute optimized machines, which would substantially reduce cost. However, using different machine types for experiments introduces methodological issues, making measurements incomparable. We have run DGSF on heterogeneous virtual hardware and bare-metal machines and the benefits reported by this work remain essentially the same.

Each p3.8xlarge machine has 4 NVIDIA V100 GPUs, each with 16GB of memory, 32 vCPUs of an Intel Xeon E5-2686, 244 GB of memory and a network interface of up to 10Gbps. The OS is Ubuntu 18.04, kernel version 5.4.0. The NVIDIA driver version is 495.29.

B. API Remoting

We measure our workloads when executed natively (the baseline) and under DGSF's API remoting mechanism (Table II). Comparison between GPU and CPU execution is presented to show scale and to demonstrate that DGSF preserves GPU acceleration benefits. 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 into phases: CUDA context initialization, input and model download time, 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 entirely from pre-creating the CUDA context. Other workloads also benefit from the optimizations described in §V-C. Face detection workload, for example, takes 11.7 seconds of processing running in DGSF, and 9.1 seconds when running natively, an increase of 28%, due to the overhead of remoting APIs over the network. However, communication overheads are compensated by other optimizations, including model loading and runtime pre-initialization. Model loading comprises a large number of calls to cuDNN APIs which are amenable to handle pre-creation and batching (see §VIII-C). Consequently, DGSF loads the model in 1.1 seconds and removes CUDA runtime initialization from the critical path, while natively the model loads in 1.7 seconds and requires 3.2 seconds for CUDA runtime initialization. All of our workloads follow this pattern.

When our workloads are executed on AWS Lambda using DGSF's API remoting, in workloads that require more network transfers, such as NLP and image classification, there is a spike in total execution time. This is because of lower bandwidth and larger variance in the network. Other workloads behave similar to our deployment of OpenFaaS.

C. 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 V-C, comparing against native execution. We remove from the comparison the times taken to download input and model files

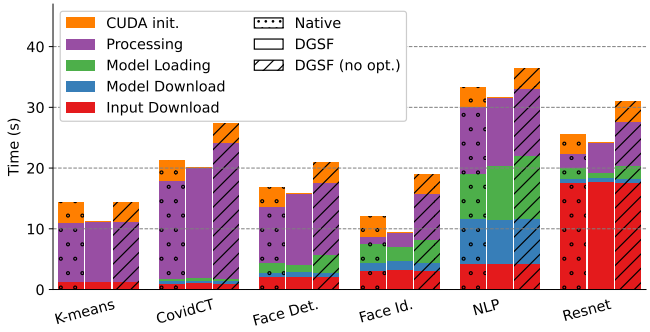


Fig. 3: Breakdown of each step of our workloads when running natively, remoted through DGSF with and without optimizations.

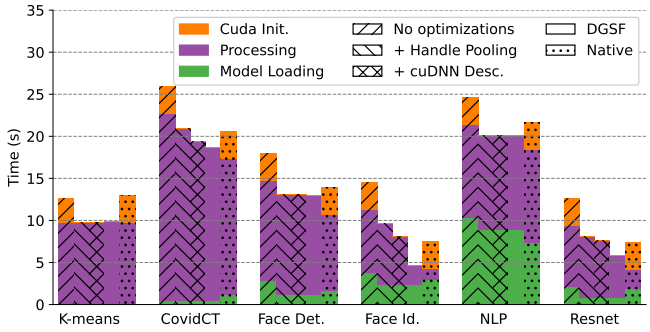


Fig. 4: Ablation study of DGSF's optimizations compared to running natively.

from remote storage (S3) into local storage, since these are not optimized by DGSF hence, 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. For face identification, total processing time (runtime initialization, model loading and inference) is 14.5 seconds with DGSF using no optimizations. Handle pooling reduces total processing time to 9.6 seconds, removing 4.9 seconds, which is approximately the time required to initialize the CUDA, cuDNN and cuBLAS libraries (3.2, 1.2 and 0.2 seconds respectively). The optimization that avoids remoting cuDNN descriptor APIs reduces inference time from 7.2 to 5.7 seconds. Adding the last layer of optimizations, batching APIs and avoiding other 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. Benefits are workloads dependent. K-means does not use any of the optimized APIs, and only benefits from CUDA runtime pre-creation. Face detection and NLP have a borderline improvement with DGSF's optimizations because fewer optimized APIs are called.

D. Mixed workloads

For all experiments in this section, we mix all six workloads, varying function invocation intervals. Scheduling at the GPU server enforces a first-come first-serve policy per serverless function. This 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.

First, we use a poisson distribution to emulate a real sequence of function invocations. We launch ten instances of each workload in a random (but consistent) order. On average our workloads utilize 12 seconds of GPU. Launching functions with an interval of 3 seconds on a GPU server with four GPUs fully utilizes the server with minimal queueing.

To emulate a GPU server under heavy load we launch functions at 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 ($\lambda = 0.5$). We report the end-to-end time as seen from the provider side, the average, standard deviation and the sum of all the functions' queuing and execution delay, which includes waiting when GPU requests cannot be satisfied due to all API servers being busy.

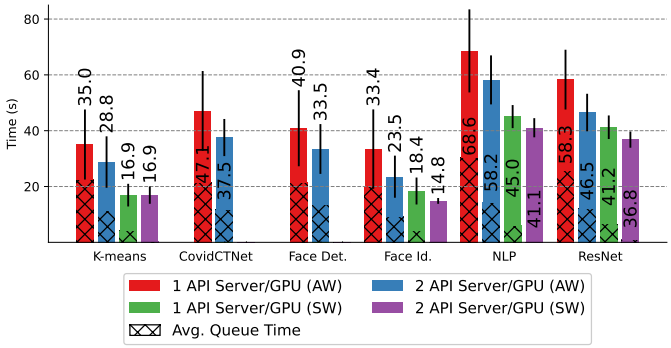


Fig. 5: Per workload queueing and execution delay when the GPU server is under a high load, running two different subset of workloads: all workloads (AW) and the four workloads with smaller memory footprints (SW). Time between function launches is from an exponential distribution with rate equal to 2.

If there is no queueing on the GPU server, the end-to-end time for each workloads will not have a large variance and will be close to the uncontended runtime (see Table II). Serverless functions with latency longer than the uncontended case reflect queuing latency at the GPU server, clearly observable in Table III. Table III shows the end-to-end time in seconds as seen from the provider (the time to handle all functions) and the sum of all function's end-to-end time (E2E, time from launch to completion) of a series of requests of all of our workloads and for only the four workloads with smaller memory requirements. The time between function launches is from an exponential distribution with rate equal to 2. Without sharing, the GPU server has a much larger function queue, which is also observed in the total function execution time sum: sharing can reduce it by 20%.

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.

TABLE III: Time in seconds of the provider's end-to-end time and sum of all function's end-to-end time when the system is under a low load.

	All Workloads		Smaller Workloads	
	End to end	Function E2E Sum	End to end	Function E2E Sum
No sharing	223.6s	2789.3s	127.3s	1178.5s
Sharing (Two) Best Fit	206.7s (-7%)	2304.8s (-17%)	121.0s (-5%)	1056.3s (-10%)
Sharing (Two) Worst Fit	206.2s (-8%)	2235.6s (-20%)	121.4s (5%)	1058.5s (-10%)

TABLE IV: Time in seconds of the provider's end-to-end time and sum of all function's end-to-end time when the system is under a high load.

	4 GPUs		3 GPUs	
	End to end	Function E2E Sum	End to end	Function E2E Sum
No sharing	242.6s	1512.2s	282.5s	2506.13s
Sharing (Two) Best Fit	240.3s (-1%)	1473.8s (-2%)	253.7s (-10%)	1832.7s (-27%)
Sharing (Two) Worst Fit	240.6s (-1%)	1446.5s (-4%)	253.8s (-10%)	1810.9s (-28%)

By increasing the rate of our exponential distribution to 3 (function launch every three seconds, on average) we emulate a GPU server under light load, where there should be less queueing and GPUs can possibly be idle between function requests. The end-to-end time as seen from the provider with four GPUs, with and without sharing is the same since there is no queueing, thus sharing doesn't have any benefit. Spreading the functions across the GPUs is the best choice for performance, although with only a marginal difference. This is observed for no sharing and sharing using a worst fit scheduler in Table IV, which shows the end-to-end time as seen from the provider and the sum of all function's end-to-end time of a series of requests of all of our workloads using three and four GPUs.

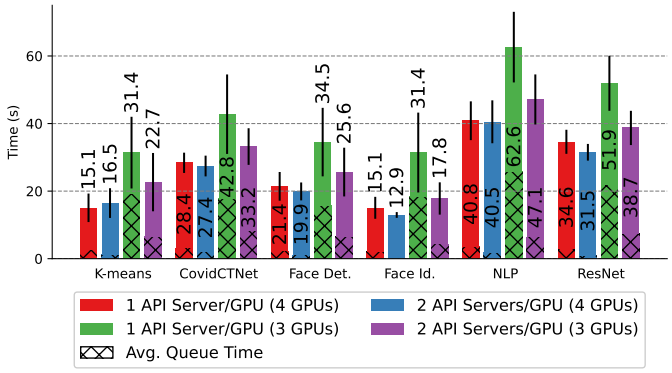


Fig. 6: Per workload queueing and execution delay when the GPU server is under a low load. Time between function launches is an exponential distribution with rate equal to 3.

In a low load case the provider could reduce the number of GPUs available to the GPU server and either leave it idle or use it for a different service. By using three instead of four GPUs under a low load with sharing, the time taken by the provider to handle all function requests increases by 5.5%.

The average end-to-end time of each of our workloads does not suffer significant changes with and without sharing with four GPUs (Figure 6). When this number is reduced to three, we see the benefit of sharing: in a contended environment, sharing reduces queueing latency of all functions and can reduce the time taken to handle a function by up to 25%.

We experiment with a bursty sequence of functions, where we launch all six workloads at once (a burst) ten times, with an interval of two seconds between each burst. Without sharing, the time taken to complete all function requests is 220 seconds. With two API servers per GPU and a best-fit policy, the end-to-end time is on average 200 seconds, a reduction of 9%. Adding more workers to GPUs yields no significant improvement because each workload uses most of the GPU's memory.

We measure the average utilization of the 4 GPUs during a burst. Figure 7 shows the results. Utilization data is acquired from NVIDIA's NVML every 200 milliseconds and is defined as the percentage of time over the past sample period that one or more kernels were being executed. For GPUs used in our evaluation, the sample time is 167 milliseconds. The figure shows a moving average window of size 5. With sharing enabled, we see a slightly higher utilization of the GPUs since they are idle less often. The average utilization for no-sharing during a burst is 31.8%, while with sharing we see an average of 37.1%, an increase of 16%. Utilization is not close to 100% for either approach because of how NVML measures utilization, as explained above. The workloads have substantial state that must be transferred or recreated on migration: loading models and recreating them on the GPU through cuDNN calls.

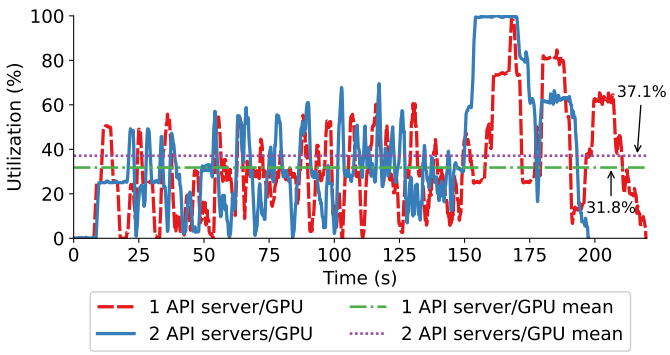
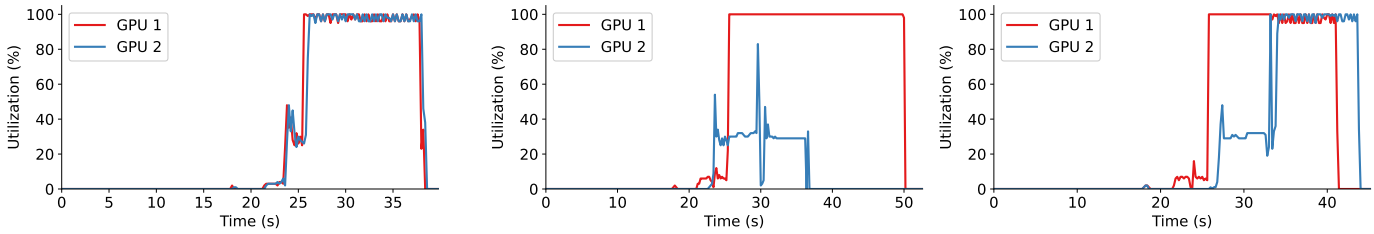


Fig. 7: Moving average of window size 5 of GPU utilization during a burst of GPU functions for a baseline with no sharing and DGSF using best-fit policy with two API servers per GPU.

E. Migration

The primary benefit of workload 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. If a provider wants to reduce cost through maximizing function packing, such as using a best-fit policy, scheduling decisions can leave some GPUs idle while others are over-subscribed. Migration helps mitigating possible performance issues by moving API servers between GPUs. We study this case below.



(a) Worst fit scheduling. (b) Best fit scheduling. (c) Best fit scheduling with migration.

Fig. 8: Memory and GPU utilization for a simple microbenchmark where two NLP applications are launched and, after three seconds, two image classification are launched using two GPUs and different scheduling and migration policies.

We study a scenario where migration is used using the NLP and image classification workloads. Using only two GPUs over DGSF remoting, each with 15GB of free memory (∼1GB is used by the API servers' contexts), we launch two NLP workloads and two image classification workloads. Because the image classification workloads require more data to be downloaded, the NLP workloads start using the GPUs first. The baseline comparison is API remoting without 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. This scenario takes 43.6 seconds to finish.

With GPU sharing enabled, more scheduling options become available. The best scenario is using a worst-fit scheduler, where one image classification and one NLP workload share a GPU. This takes 38.9 seconds, an improvement of about 11% over the baseline. With best-fit scheduling, we see the worst scenario: the two NLP workloads share a GPU; the total experiment takes 50.6 seconds to complete. Because they 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 behavior can be seen in Figure 8b, where the utilization for GPU 2 falls to zero while GPU 1 stays at 100% for over 24 seconds.

It is clear from Figure 8b that we could improve utilization and, consequently the runtime, by migrating one of the applications running on GPU 1 to GPU 2. The utilization for best fit policy with migration enabled is shown in Figure 8c. As soon as the second image classification workload finishes, the monitor notices an imbalance and moves one of the NLP workloads. This mechanism improves the end-to-end runtime to 42.6 seconds, a 16% improvement over best fit with no migration.

We create a synthetic workload that allocates a fixed size, single array of GPU memory, zeroes the array using cudaMemset and launches two kernels that perform simple arithmetic operations on the array elements. This is the worst case for migration since there is a single large array, which means memory copying can not be parallelized. The memory amount chosen for comparison are from three of our workloads' memory requirements. To measure the overhead of migrating, which requires stopping the API server from handling API requests and copying memory from one GPU to another, we forcefully migrate this application right before the second kernel is called. Results are presented in Table V. When running natively the dominant factor is initializing the CUDA runtime, which takes approximately 3 seconds, 95% of the end-to-end time for the largest array we measured with. The elimination of the CUDA runtime initialization latency is only possible with API remoting. The API servers are pre-created and pooled, so their initialization still occurs but, in contrast to running natively, the initialization is not on the critical path. With a migration between the two kernels we observe an increase in end-to-end time, due to the API server having to synchronize and wait for the first kernel to be done and copy the entire array between devices before resuming normal operation. The migration process cost increases as more memory needs to be moved and is around 78% of the end-to-end time for the largest memory allocation evaluated.

TABLE V: Average times in seconds of three runs of an application that allocates an array and launches 2 kernels that touch all elements.

	Native	DGSF	DGSF +migration	
	End-to-end	End-to-end	End-to-end	Migration
323 MB	3.04	0.04	0.25	0.50
3514 MB	3.06	0.06	0.70	0.53
7802 MB	3.10	0.10	1.38	1.19
13194 MB	3.11	0.12	2.34	2.12

IX. Related Work

GPU virtualization. Cloud providers expose GPUs to virtual machines using PCIe passthrough which dedicates the hardware interface directly to the guest, prohibiting sharing and causing underutilization [43]. Full-virtualization [44], [45], mediated pass-through (MPT) [45], [46], [47], para-virtualization [48] and SR-IOV [49], [50], [51] techniques have limitations that have hampered adoption in production cloud environments [52].

Accelerator virtualization specialized for serverless functions is a relatively new research space. Existing literature CUDA-enabled containers [4] and API-remoting [5] simply expose GPUs to serverless functions; unlike DGSF, they do not optimize API-remoting specfically for the serverless setting or support migration. Other accelerator types such as FPGAs through OpenCL can also be used by serverless functions [53].

API remoting [54], [21], [22], [11] is a virtualization technique that interposes a user-mode API, forwarding calls

to a user-level framework [55] on an appliance VM [56], or remote server [20]. API remoting is attractive for serverless because it decouples accelerator resource management from other resources'. Scheduling in such scenarios is easier and allows for several optimizations for heterogeneous workloads [7], [6], [57], [58].

GPU consolidation. Although plenty of literature exists, sharing GPUs is difficult and is not a solved problem [14], [15], [59], [18]. 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 [60]. For example, PipeSwitch [61] manually switches context of applications in GPU to ensure high utilization, while Gandiva [24] implements time-slicing.

GPU migration. Execution migration across GPUs is another heavily studied area of research [24], [62], [23] and is tightly coupled with consolidation. NVIDIA's GRID supports live migration of VMs, which is not the case when API remoting is in place. DCUDA [23] uses peer-to-peer GPU memory accesses to migrate kernel executions without manually moving data. We tried this approach for `DGSF` but found that it can incur large overheads depending on the kernel launching pattern, likely from the CUDA runtime ensuring safety and memory consistency. Gandiva [24] uses a checkpoint-restore approach, relying on library support, e.g. TensorFlow `train.Saver`.

X. 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. `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] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st MOTA*. New York, NY, USA: Association for Computing Machinery, 2016.

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

[3] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings SoCC 2017*. New York, NY, USA: ACM, 2017, pp. 445–451.

[4] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "Gpu enabled serverless computing framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 533–540.

[5] D. M. Naranjo, S. Risco, C. de Alfonso, A. Prez, I. Blanquer, and G. Molt, "Accelerated serverless computing based on gpu virtualization," *Journal of Parallel and Distributed Computing*, vol. 139, pp. 32–42, 2020.

[6] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proceedings of the Third SoCC*. New York, NY, USA: Association for Computing Machinery, 2012.

[7] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh EuroSys*. New York, NY, USA: Association for Computing Machinery, 2016.

[8] M. Sheikhalishahi, R. M. Wallace, L. Grandinetti, J. L. Vazquez-Poletti, and F. Guerriero, "A multi-dimensional job scheduling," *Future Generation Computer Systems*, vol. 54, pp. 123–131, 2016.

[9] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Citeseer, 1999, pp. 185–194.

[10] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "Enabling CUDA Acceleration Within Virtual Machines Using rCUDA," in *Proceedings of the 2011 18th HIPC*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10.

[11] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "AvA: Accelerated virtualization of accelerators," in *ASPLOS*. ACM, 2020, pp. 807–825.

[12] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic space-time scheduling for GPU inference," in *Thirty-second Conference on Neural Information Processing Systems*, 2018.

[13] 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)*, June 2017, pp. 1–12.

[14] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th HPDC*. New York, NY, USA: Association for Computing Machinery, 2011, p. 217228.

[15] Y. Suzuki, H. Yamada, S. Kato, and K. Kono, "Gloop: An event-driven runtime for consolidating gpgpu applications," in *Proceedings SoCC 2017*. New York, NY, USA: Association for Computing Machinery, 2017, p. 8093.

[16] K.-C. Li, K. Kim, W. W. Ro, T.-H. Weng, C.-L. Hung, C.-H. Ku, A. Cohen, and J.-L. Gaudiot, "On migration and consolidation of vms in hybrid cpu-gpu environments," in *Intelligent Technologies and Engineering Systems*, J. Juang and Y.-C. Huang, Eds. New York, NY: Springer New York, 2013, pp. 19–25.

[17] H. S. Jo, M. H. Lee, and D. H. Choi, "Gpu virtualization using PCI direct pass-through," in *Information, Communication and Engineering*, ser. Applied Mechanics and Materials, vol. 311. Trans Tech Publications Ltd, 5 2013, pp. 15–19.

[18] K. M. Diab, M. M. Rafique, and M. Hefeeda, "Dynamic sharing of gpus in cloud systems," in *2013 IEEE ISPA, Workshops and Phd Forum*, 2013, pp. 947–954.

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

[20] C. Reaño, A. J. Peña, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí, "CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution," *20th Annual International Conference on High Performance Computing*, vol. 0, pp. 1–10, 2012.

[21] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng, "Transparent accelerator migration in a virtualized GPU environment," in *Proceedings of the 12th IEEE/ACM CCGrid*, 2012, pp. 124–131.

[22] "End-to-End Solutions for AI/ML Workloads — VMware," (Accessed:

October, 2021). [Online]. Available: https://www.vmware.com/products/vsphere/ai-ml.html
[23] F. Guo, Y. Li, J. C. S. Lui, and Y. Xu, "Dcuda: Dynamic gpu scheduling with live migration support," in *Proceedings of the ACM SoCC*. New York, NY, USA: Association for Computing Machinery, 2019, p. 114125.
[24] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX 2018 OSDI*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610.
[25] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on GPU clusters for deep learning," in *14th USENIX OSDI 20*. USENIX Association, Nov. 2020, pp. 533–548.
[26] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS*. ACM, 2020, pp. 467–481.
[27] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX HotCloud 19*. Renton, WA: USENIX Association, Jul. 2019.
[28] "ONNX Runtime: cross-platform, high performance scoring engine for ML models," (Accessed: October 2021). [Online]. Available: https://github.com/microsoft/onnxruntime
[29] "OpenFaaS - Serverless Functions Made Simple," (Accessed: January 2021). [Online]. Available: https://www.openfaas.com/
[30] B. Hu and C. J. Rossbach, "Altis: Modernizing gpgpu benchmarks," in *2020 IEEE ISPASS*, 2020, pp. 1–11.
[31] T. Javaheri, M. Homayounfar, Z. Amoozgar, R. Reiazi, F. Homayounieh, E. Abbas, A. Laali, A. Radmard, M. Gharib, S. Mousavi, O. Ghaemi, R. Babaei, H. Mobin, M. Hosseinzadeh, R. Jahanban-Esfahlan, K. Seidi, M. Kalra, G. Zhang, L. Chitkushev, B. Haibe-Kains, R. Malekzadeh, and R. Rawassizadeh, "Covidctnet: an open-source deep learning approach to diagnose covid-19 using small cohort of ct images," *npj Digital Medicine*, vol. 4, no. 1, Dec. 2021.
[32] "ShahinSHH/COVID-CT-MD : A COVID-19 CT Scan Dataset Applicable in Machine Learning and Deep Learning," (Accessed: October, 2021). [Online]. Available: https://github.com/ShahinSHH/COVID-CT-MD
[33] J. Deng, J. Guo, Z. Yuxiang, J. Yu, I. Kotsia, and S. Zafeiriou, "Retinaface: Single-stage dense face localisation in the wild," in *arxiv*, 2019.
[34] S. Yang, P. Luo, C. C. Loy, and X. Tang, "Wider face: A face detection benchmark," in *2016 IEEE CVPR*, 2016, pp. 5525–5533.
[35] J. Deng, J. Guo, X. Niannan, and S. Zafeiriou, "Arcface: Additive angular margin loss for deep face recognition," in *CVPR*, 2019.
[36] "ArcFace," (Accessed: October 2021). [Online]. Available: https://github.com/onnx/models/tree/master/vision/body_analysis/arcface
[37] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.
[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[39] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2019.
[40] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100, 000+ questions for machine comprehension of text," *CoRR*, vol. abs/1606.05250, 2016.
[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE CVPR*, 2016, pp. 770–778.
[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *CVPR 09*. IEEE, 2009.
[43] "Underutilizing Cloud Computing Resources," (Accessed: October 2021). [Online]. Available: https://www.gigenet.com/blog/underutilizing-cloud-computing-resources/
[44] J. Song, Z. Lv, and K. Tian, "KVMGT: a Full GPU Virtualization Solution," in *KVM Forum*, vol. 2014, 2014.
[45] K. Tian, Y. Dong, and D. Cowperthwaite, "A Full GPU Virtualization Solution with Mediated Pass-Through," in *2014 USENIX ATC*. USENIX Association, Jun. 2014, pp. 121–132.
[46] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: a NVMe storage virtualization solution with mediated pass-through," in *2018 USENIX ATC*, 2018, pp. 665–676.
[47] L. Xia, J. Lange, P. Dinda, and C. Bae, "Investigating virtual passthrough I/O on commodity devices," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 83–94, 2009.
[48] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.
[49] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High Performance Network Virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.
[50] Y. Dong, Z. Yu, and G. Rose, "SR-IOV Networking in Xen: Architecture, Design and Implementation." in *Workshop on I/O Virtualization*, 2008.
[51] "NVIDIA GRID," (Accessed: October 2021). [Online]. Available: https://www.nvidia.com/en-us/data-center/virtual-gpu-technology/
[52] H. Yu and C. J. Rossbach, "Full Virtualization for GPUs Reconsidered," in *14th WDDD, ISCA*, 2017.
[53] M. Bacis, R. Brondolin, and M. D. Santambrogio, "Blastfunction: an fpga-as-a-service system for accelerated serverless computing," in *2020 DATE Conference Exhibition*, 2020, pp. 852–857.
[54] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated Virtual Machines," in *Proceedings of the 3rd ACM Workshop HPCVirt*. New York, NY, USA: ACM, 2009, pp. 17–24.
[55] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.
[56] L. Vu, H. Sivaraman, and R. Bidarkar, "GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor," in *Proceedings of HPC Symposium*, 2014, pp. 2:1–2:8.
[57] M. Amaral, J. Polo, D. Carrera, N. Gonzalez, C.-C. Yang, A. Morari, B. D. D'Amora, A. Youssef, and M. Steinder, "Drmaestro: orchestrating disaggregated resources on virtualized data-centers," *Journal of Cloud Computing*, vol. 10, pp. 1–20, 2021.
[58] A. Guleria, J. Lakshmi, and C. Padala, "Quadd: Quantifying accelerator disaggregated datacenter efficiency," in *2019 IEEE 12th International CLOUD*, 2019, pp. 349–357.
[59] U. Kurkure, H. Sivaraman, and L. Vu, "Virtualized gpus in high performance datacenters," in *2018 HPCS*, 2018, pp. 887–894.
[60] P. Yu and M. Chowdhury, "Fine-grained gpu sharing primitives for deep learning applications," in *PLMR 20*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 98–111.
[61] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *14th USENIX OSDI 2020*. USENIX Association, Nov. 2020, pp. 499–514.
[62] J. Prades and F. Silla, "Gpu-job migration: The rcuda case," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2718–2729, 2019.