EDAS: Enabling Fast Data Loading for GPU Serverless Computing

HAN ZHAO*, Shanghai Jiao Tong University, Shanghai, China
WEIHAO CUI*, Shanghai Jiao Tong University, Shanghai, China
QUAN CHEN, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
ZIJUN LI, Shanghai Jiao Tong University, Shanghai, China
ZHENHUA HAN, Unaffiliated, Shanghai, China
NAN WANG, Alibaba Cloud Computing, Shanghai, China
YU FENG, John Hopcropt Center, Shanghai Jiao Tong University, Shanghai, China
JIERU ZHAO, Shanghai Jiao Tong University, Shanghai, China
CHEN CHEN, Shanghai Jiao Tong University, Shanghai, China
JINGWEN LENG, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
MINYI GUO, Computer Science, Shanghai Jiao Tong University, Shanghai, China

Integrating GPUs into serverless computing platforms is crucial for improving efficiency. Many GPU functions, such as DNN inferences and scientific services, benefit from GPU usage, which requires only tens to hundreds of milliseconds for pure computation. Under these circumstances, fast data loading is imperative for function performance. However, existing GPU serverless systems face significant data stall issues, leading to extremely low GPU efficiency.

Faced with the above problems, we observe opportunities to optimize data loading, such as data preloading and deduplicated data loading. However, these optimizations are impossible in existing GPU serverless systems due to the lack of insights into data information, such as data sizes and read-write attributes of function inputs. To address this, we propose a novel GPU serverless system, EDAS. EDAS first enhances user request specifications, allowing users to annotate data retrieved by GPU functions from the database with additional attributes. Based on this, EDAS takes over data loading from GPU functions and proposes two innovative data loading management schemes: a parallelized data loading scheme and a multi-stage resource exit scheme. Our experimental results show that EDAS reduces function duration by 16.2× and improves system throughput by 1.91× compared to the state-of-the-art serverless platform.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Additional Key Words and Phrases: GPU, Serverless, Function Startup

*Both authors contributed equally to this research.

Authors' Contact Information: Han Zhao, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhao-han@cs.sjtu.edu.cn; Weihao Cui, Shanghai Jiao Tong University, Shanghai, China; e-mail: weihao@sjtu.edu.cn; Quan Chen, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: chen-quan@cs.sjtu.edu.cn; Zijun Li, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: lzjzx1122@sjtu.edu.cn; Zhenhua Han, Unaffiliated, Shanghai, China; e-mail: hzhua201@gmail.com; Nan Wang, Alibaba Cloud Computing, Shanghai, China; e-mail: kenan.wn@alibaba-inc.com; Yu Feng, John Hopcropt Center, Shanghai Jiao Tong University, Shanghai, China; e-mail: y-feng@sjtu.edu.cn; Jieru Zhao, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhao-jieru@sjtu.edu.cn; Chen Chen, Shanghai Jiao Tong University, Shanghai, China; e-mail: chen-chen@sjtu.edu.cn; Jingwen Leng, Department of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: leng-jw@cs.sjtu.edu.cn; Minyi Guo, Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: guo-my@cs.sjtu.edu.cn.

ACM 1544-3973/2025/6-ART
https://doi.org/10.1145/3743137

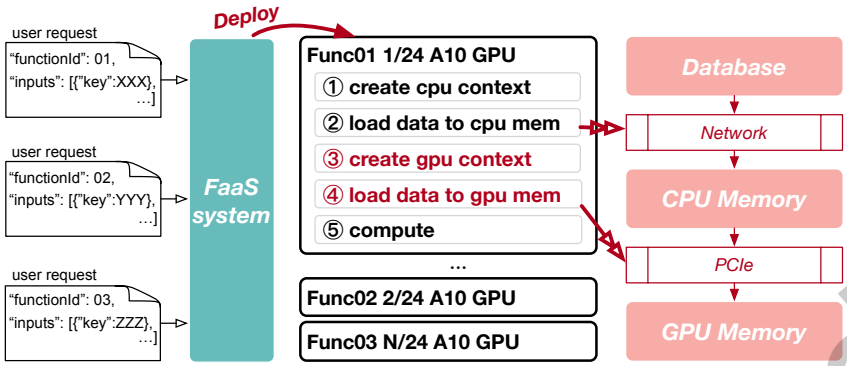

Fig. 1. Workflow of FixedGSL (instance-fixed GPU serverless computing mode).

1 Introduction

Emerging cloud computing services, such as image processing [30] and medical testing [45], rely on GPUs for computation. These cloud services often experience unstable loads, which may only a few to dozens of user requests per minute. In this scenario, a traditional GPU-based inference system [15, 16, 29, 32] will encounter inefficiencies because it needs to reserve GPU resources all the time. Faced with this problem, serverless computing [5, 21, 34] has been proven efficient for handling relatively low loads using only CPUs.

Some cloud providers, such as Alibaba Function Compute [3] and Azure Function [5], have also made efforts to integrate GPUs into serverless systems. Existing systems all adopt an instance-fixed GPU serverless computing mode (denoted by *FixedGSL* in Figure 1). When a GPU function is triggered by a user request, the system allocates a fixed size of GPU resources to it, such as 1/24 A10 GPU (1GB memory usage granularity and corresponding time slices). After the resource allocation, the system hands over data loading and function computation to the GPU function.

Since GPUs are used to accelerate computation, many GPU functions (such as DNN inference [27] and scientific services [45]) only consume tens to hundreds of milliseconds for pure computation. Under these circumstances, fast data loading is imperative for GPU function performance. However, we observe that existing GPU serverless systems face significant data stall issues, which lead to very low GPU efficiency.

On one hand, existing GPU serverless systems require every GPU function to create a GPU context before the actual data loading into GPUs. It significantly blocks the data preparation when launching a GPU function, even when the data to load is known before GPU context is ready (e.g., model weight). Our experiments in §3.2.1 reveal that GPU context creation requires around 100 milliseconds, leading to a long stall to actual data loading and computation.

On the other hand, all GPU functions need to load data to CPU memory over the network and load data to GPU memory via PCIe. They inevitably encounter network and PCIe contention. Experimental results in §3.2.2 show that GPU functions, on average, suffer from 6.1× longer data loading times in shared clusters compared with solo-runs. Moreover, the popularity of Large Language Models (LLMs) brings heavier transfer requirements, further exacerbating these contentions. Meanwhile, the prolonged loading time results in extended resource occupation without GPU computation, which also leads to a low function throughput.

Faced with the above problems, we observe opportunities to optimize data loading, such as data preloading and deduplicated data loading. Specifically, a serverless system could perform actual data loading before the GPU context is created. The parallelization of these two stages then speeds up the overall data loading. Also,

many services have read-only data, such as weights in DNN inference [18, 27] and parameters in scientific computing [45]. Enabling read-only data sharing among multiple requests of a GPU function could eliminate repetitive data loading, alleviating network and PCIe contention.

However, these optimizations are impossible in existing GPU serverless systems due to the lack of insights into data required by GPU functions. For example, data preloading and deduplicated data loading require data positions, data sizes, and read-write attributes of the function inputs, which are unknown in existing systems that treat GPU functions as black boxes. Furthermore, even when data can be preloaded, it is still difficult to map the prepared data to the GPU functions due to the unknown linkage.

To this end, we propose a novel GPU serverless system, EDAS. EDAS first designs an enhanced request specification and two data access interfaces for GPU functions developed from scratch. These interfaces enable the system to take over data loading. For domain-specific functions relying on specific frameworks (such as PyTorch), EDAS provides an automated conversion tool to leverage this design.

Specifically, we allow developers to annotate data retrieved by these functions from the database with additional attributes, such as input data size and read/write properties (read-only or writable). Once these data attributes are exposed, EDAS proposes a unified memory daemon and a set of data access interfaces, namely (`EdasLoadToGPU()` & `EdasDumpToDB()`), to facilitate GPU function programming. Using these interfaces, the memory daemon takes over the actual data transfer and establishes linkages between data and the launched GPU functions. EDAS then proactively optimizes data loading for GPU functions by using the data attributes in the enhanced user request.

Two primary optimization schemes are implemented to boost the startup of GPU functions. First, using the attribute of data size, EDAS devises a parallel setup scheme to speed up the data loading time for a cold start function. It directly instructs the memory daemon to load data to the GPU in parallel with GPU context creation. Upon accessing the preloaded data through the interfaces inside the GPU function, the linkages between the loaded data and the function are captured to ensure the correctness of function execution.

Second, with the attribute of read-write property, EDAS designs a multi-stage resource exit scheme, reducing the data loading pressure and alleviating network and PCIe contention. It releases writable data, GPU context, and read-only data in sequence. In this way, EDAS's function runtime keeps the read-only data warm for a longer period, maximizing opportunities to share them across multiple requests. Additionally, EDAS also temporarily caches read-only data in larger and more cost-efficient CPU memory to minimize the overhead of fetching data from remote databases.

We have implemented and deployed EDAS on a cluster with A10 GPUs. A series of GPU functions are integrated into EDAS, including 7 traditional DNN inferences [18, 26–28, 44, 46, 47, 56], 3 scientific computing tasks [45], and 2 large language models [31, 49]. These GPU functions using DNN models are generated from our automatic conversion tool, the GPU functions using LLM are ported from LLaMa.cpp [23] (details in subsection 5.4). The GPU functions for scientific computing tasks are built from the scratch.

Our experimental results show that EDAS reduces the function duration by 16.2× and improves system throughput by 1.91× compared with state-of-the-art serverless systems. The main contributions of this paper are as follows:

- **Identifying the root causes of long data loading and low throughput in existing GPU serverless systems.** Based on the investigations, we propose EDAS with appropriate data loading boost and fine-grained data-sharing methods to address the issues.
- **Designing an elegant mechanism that efficiently parallelizes actual data loading and context creation.** EDAS utilizes it to accelerate the overall data loading time, when there are no available resources for reuse.
- **Proposing approaches that reduce contention of the data-loading paths for improving the function data loading and system throughput.** EDAS achieves the goal through fine-grained data sharing, especially the multi-stage resource exit scheme.

Table 1. The benchmarks used for investigation.

Task	Task Type
vgg11 [44], inception3 (incep3) [47] resnet50 (resnet) [27], nasnet [56]	Computer Vision
deepspeech (speech) [26] seq2seq (s2s) [46]	Speech Recognition
bert [18], llama-7B-INT8[49] bloom-2.8B-FP16 [31]	Natural Language Processing
lbm, mrif, tpacf [45]	Scientific Computing

2 Related Works

There are a few prior works [3, 5, 19, 22, 25, 38] on integrating GPU in serverless computing. The usable GPU serverless platforms are available in Alibaba Cloud [3] and Microsoft Azure [5]. For tradeoffs between GPU accessibility and the portability of existing serverless platforms, they schedule the GPU function at the granularity of size-fixed GPU instances (FixedGSL). ORSCA [38], DGSF [22] provided GPU functions on CPU-only nodes with pre-created GPU context on remote GPUs. It is achieved through API-remoting. The significant memory required for pre-warming and communication overhead make these methods impractical for production use. We compare EDAS with FixedGSL and DGSF in § 8. In FaST-GShare [25], it focuses on spatio-temporal GPU sharing, whereas our work addresses a different challenge—efficient data loading. FaST-GShare is orthogonal to EDAS's goal, and they can be combined together for optimizations.

Some prior works support fine-grained memory and computation management through hijacking driver-level API, such as Alibaba cGPU [11], Tencent qGPU [9], vCUDA [43], qCUDA [35] and GPUshare [24]. These technologies provide isolated GPU memory and computing capacity. NVIDIA also has its own GPU fine-grained resource-sharing solutions, such as MPS [8]. Some works have proposed GPU-sharing solutions in fixed scenarios, such as deep learning applications [17, 54, 55]. These techniques are not aware of serverless's characteristics and do not help accelerate the setup for existing GPU serverless frameworks. However, they can be integrated into EDAS to provide the mechanism of better memory and computation isolation.

There are also many prior works on optimizing the startup time in serverless computing [34, 37, 39, 50–52]. Replayable Execution [53], Firecracker [13], and Catalyzer [20] are snapshot and fork-based optimizations. SAND [14] used a multi-level sandboxing mechanism to improve the performance of the application. These works only accelerate the host-side operations, making GPU functions still suffer from long setup and extra memory consumption.

3 Background and Motivation

In this section, we begin by presenting state-of-the-art GPU serverless systems, along with the benchmarks used. Next, we analyze the inefficiencies in existing serverless systems and identify the opportunities.

3.1 GPU Serverless and Benchmarks

Existing serverless systems, such as Alibaba Function Compute [3] and Azure Function [5], all adopt instance-fixed GPU serverless computing mode (FixedGSL mode). Different serverless systems may use different minimum GPU usage granularities. Alibaba Function Compute supports the usage of 1/24 A10 GPU or 1/12 T4 GPU (1 GB memory usage granularity and corresponding time slices) [3]. Azure Function only supports the usage of 1 A10 GPU or 1 T4 GPU.

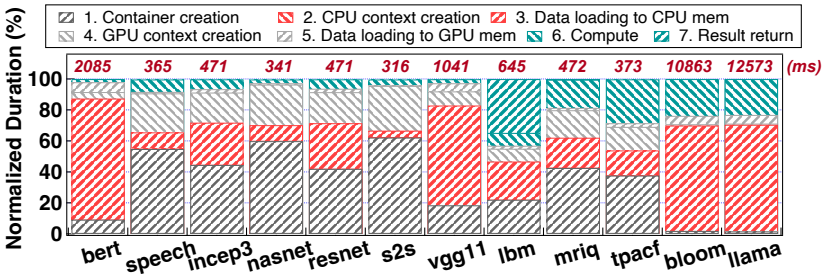

Fig. 2. The duration breakdown of all GPU functions.

When a GPU function is triggered by a user request, the system allocates a fixed size of GPU resources to it. Upon obtaining resources, the serverless system delegates the processing of requests entirely to GPU functions, covering data loading and computation. After completing the request, the GPU function stays alive for a while, facilitating warm startup for subsequent requests. If no requests arrive within a certain timeframe, such as 2 minutes [34], the system will release the resources occupied by the function.

Since GPU serverless is suitable for a variety of fields, we choose multiple DNN inference tasks, and scientific computing tasks as the benchmarks. As shown in Table 1, we use seven DNN models [18, 26–28, 44, 46, 47, 49, 56], and three scientific tasks [45]. The DNN models span computer vision, speech recognition, and natural language processing, while the scientific tasks include fluid mechanics (*lbm*), medical assistance (*mrif*), and astrophysics (*tpacf*). Meanwhile, *Llama-7B-INT8* and *Bloom-2.8B-FP16* are also included to represent the large language model (LLM). These functions are implemented by Rammer [36], Parboil [45], and LLaMa.cpp [23]. We run the benchmarks on a node with one Nvidia A10 GPU (24GB memory), which is one of the most commonly used inference-oriented GPUs among cloud providers. The detailed hardware and software configurations are described in Table 3 of § 8.1.

3.2 Inefficiencies of FixedGSL

To demonstrate the inefficiencies, we examine the performance of GPU functions both with and without resource contention.

3.2.1 End-to-end Execution without Contention. In order to measure GPU function duration without impact of network and PCIe contention, we generate user requests in a closed-loop manner for each GPU function running with FixedGSL. Figure 2 shows the duration breakdown of all functions. The end-to-end duration for GPU functions could be segmented into seven stages: 1) container creation; 2) CPU context creation (includes the network setup like grpc[10]); 3) data loading from database to CPU memory; 4) GPU context creation (includes the library initialization like cuDNN); 5) data loading from CPU memory to GPU memory; 6) function computation; and 7) results return.

In the figure, the stages before computation account for 88.7% of the end-to-end duration on average. Among these stages, container creation is the most time-consuming (49.7% of end-to-end latency on average). Since there is no big difference in launching a GPU container and a CPU container, GPU container creation time can be directly optimized using the pre-warm techniques proposed for CPU containers [3, 34, 39].

Data loading dominates function duration. When the container creation time is eliminated, we refer to the four stages from stage 2-5 as the overall data loading stages. The pure computation time of benchmarked functions shows that, excluding LLM models, the average computation time of these functions is 48.6 milliseconds. As for the LLM models, it ranges from hundreds to thousands of milliseconds. Across all benchmarks, computation

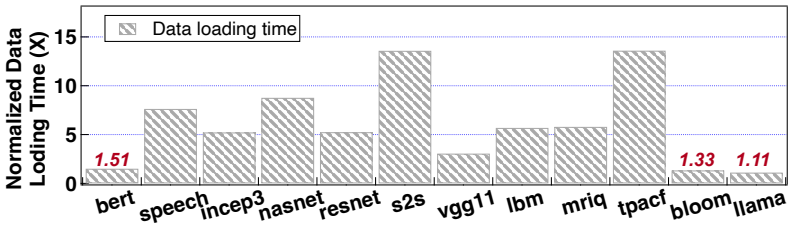

Fig. 3. The data loading time of all applications normalized to the solo-run case under FixedGSL.

still only takes up 11.1% of the duration. On the contrary, the overall data loading time takes up 49.7%. Given these observations, fast data loading is crucial for function performance. Otherwise, the overall data loading may dominate the end-to-end function duration.

Blocking Creation of GPU Context. Delving deeper into the time breakdown of the overall data loading stages, we find that the actual data loading takes up 63.2%, and the GPU context creation takes up 36.8%. For the non-LLM benchmarks, the actual data loading takes up 56.1%, and the GPU context creation takes up 43.9%. Access to the GPU requires an established GPU context. When the serverless system hands over all data loading stages to GPU functions, loading data to GPU memory inside the GPU function is blocked by the GPU context creation, even if the data is ready in CPU memory. Thus, GPU functions are suffering prolonged data loading time due to the blocking nature of GPU context creation.

3.2.2 Data Loading Duration With Contention. We then measure the data loading time of GPU functions under resource contention. In this scenario, requests are generated with a Poisson distribution and scheduled in an open-loop manner. To better illustrate data loading under contention, we benchmark one function at a time. Requests for the same functions contend for PCIe and network resources. Simultaneously, we gradually increase the load until FixedGSL fails to process received requests due to GPU resource limits.

Poorer function performance under contention. Figure 3 shows the average data loading duration of all functions with FixedGSL under the peak load. This time is normalized to the data loading duration without contention. As shown, each function in FixedGSL suffers 6.1× data loading time on average compared with no contention case. This demonstrates that multiple concurrent GPU functions encounter serious network and PCIe contention. While PCIe have greater bandwidth than network, we further conduct the experiments without network requirement. Experimental results show that each function still suffers 4.8× data loading time on average.

3.3 Opportunities and Challenges

In a nutshell, while the serverless system hands over the data loading to the GPU functions, they suffer from the long data loading stages and the severe resource contention in the datapaths. Faced with the significant data loading issues, we observe some opportunities when delegating data loading to the serverless system.

- The serverless system could perform actual data loading before the GPU context is created. The parallelization of these two stages can speed up the overall data loading.
- Many functions have read-only data, like weights in DNN inference [18, 27]. The serverless system could eliminate repetitive data loading across multiple requests of the same GPU function, thereby alleviating resource contention.

However, existing serverless systems fail to capitalize on these opportunities due to the absence of pertinent data information. The above opportunities all require the linkages between the data and GPU functions, i.e., data positions of function inputs and their properties (sizes and read-write requirements). Data preloading is not

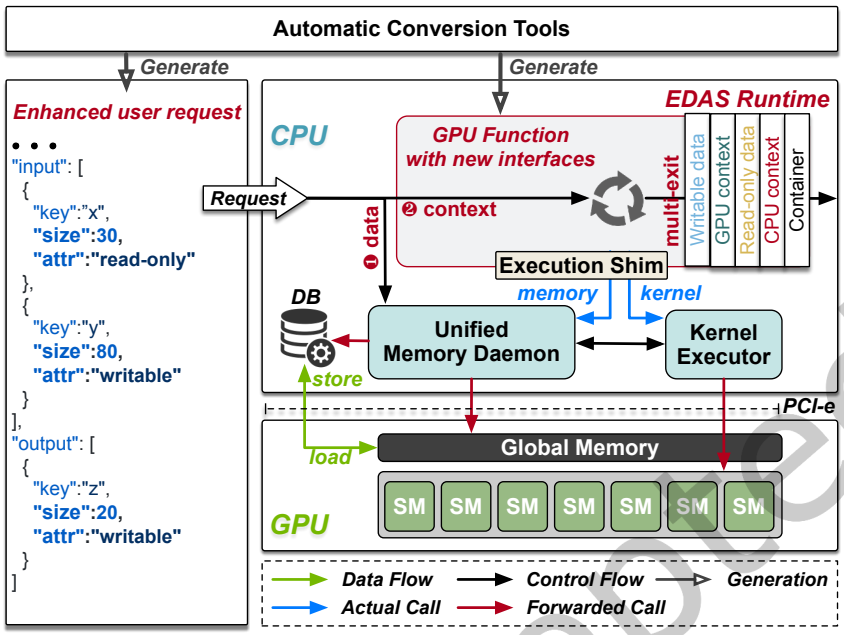

Fig. 4. The overview of EDAS.

possible in existing serverless systems without such information. Moreover, when multiple GPU functions share the same input data, such linkage can help serverless systems to detect and exploit the sharing opportunities. Our design goal is to improve the programming interface of GPU serverless systems to easily express the data linkage and optimize the contention of GPU functions with the exposed knowledge.

4 Design Overview

To capitalize on the opportunities outlined above, we propose EDAS, a GPU serverless system that takes over the data loading from GPU functions. Figure 4 depicts the overview of EDAS. The left of Figure 4 shows an example of EDAS's enhanced user request. The right shows EDAS's serverless runtime, which proactively optimizes the data management for the hosted GPU functions.

Enhanced user request. As shown on the left of Figure 4, EDAS enhances the user request by annotating the data attributes of function inputs, namely data size and read-write attributes. These attributes are essential for proactive optimization of data management. While awareness of these attributes allows for optimizations, EDAS still needs to ensure correct usage of managed data by perceiving the data access within each GPU function. `EDASLoadToGPU()` and `EDASDumpToDB()` are proposed for GPU functions to access the data managed by EDAS. EDAS establishes linkages between the managed data and functions via analysis of the invocations of these interfaces (§5.3).

Execution flow of EDAS's serverless runtime. EDAS can now host GPU serverless functions with proactive data management. EDAS's runtime consists of three modules: the execution shim, the *unified memory daemon*, and the *kernel executor*. When a request arrives at runtime, it extracts the data attributes by parsing the enhanced request specification, which are then forwarded to the memory daemon for data loading. Concurrently, the runtime initiates the function for creating context. During the function's execution, it uses the provided interfaces to access the data loaded by the memory daemon. The execution shim intercepts all GPU calls and forwards

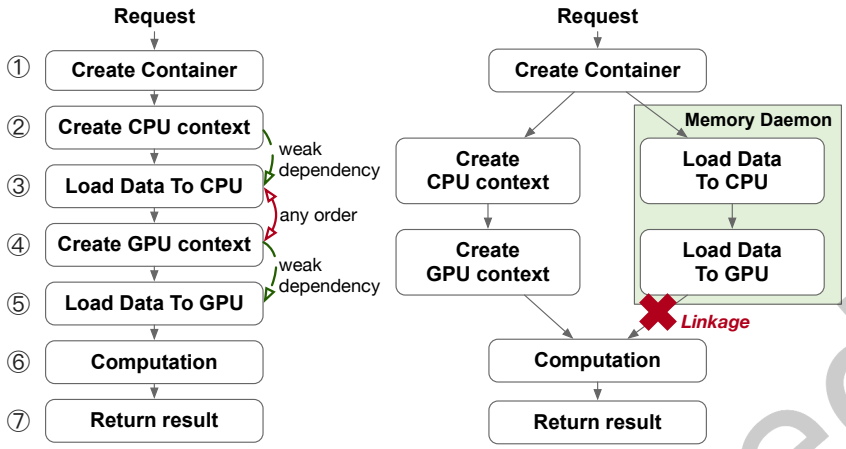

Fig. 5. Function execution flow in FixedGSL and EDAS.

them to the memory daemon or the kernel executor, respectively. The executor communicates with the memory daemon to ensure that related data loading is completed correctly. Meanwhile, the runtime employs a multi-stage resource exit scheme to support better data sharing across requests for the same function. It releases the function's resources in the order of writable data, GPU context, and read-only data.

Automatic conversion tool. Imposing additional development burdens on programmers is often unacceptable. Fortunately, many functions are developed using programming frameworks, such as PyTorch [40] for DNN applications, TorchPhysics [48] for physics-based applications, and PyHalide [41] for stencil applications. To address this, we provide an automatic conversion tool for PyTorch-based DNN applications by extending the DNN compiler, Rammer [36]. This automated tool also seamlessly supports physics applications developed with TorchPhysics. At the same time, we can support the automated conversion of stencil applications using PyHalide in a similar way.

5 Parallelized Data Loading

With the enhanced user request, we first present our method for optimizing the overall data loading time when a GPU function is first triggered by a user request.

5.1 Rationale for Decoupling Data and Context

As mentioned in §3.2.1, a GPU function consists of seven stages. Figure 5 outlines the common execution flow of these stages. Among these stages, stages 3 and 4 could be executed in any order due to no dependency. Stages 5 and 6 could also occur in any order because the GPU memory allocation and the GPU kernel launch could happen at any time. In the existing GPU serverless systems, these stages are all handed over to the GPU functions. Due to the fact that the data loading stages are blocked by the context creation stages, the overall data loading time is long for GPU functions (§3.2.1).

To address this issue, EDAS tries to decouple data and context for stage parallelization. We observe that GPU functions always load data from external sources. For instance, DNN inferences commonly read model inputs and weights from database, while scientific computing services require input data or updated data to be sourced from database. The data to be loaded for invocating the GPU function is already ready when the function is triggered. This means that the data loading and context creation are not firmly intertwined and could be decoupled for parallelization.

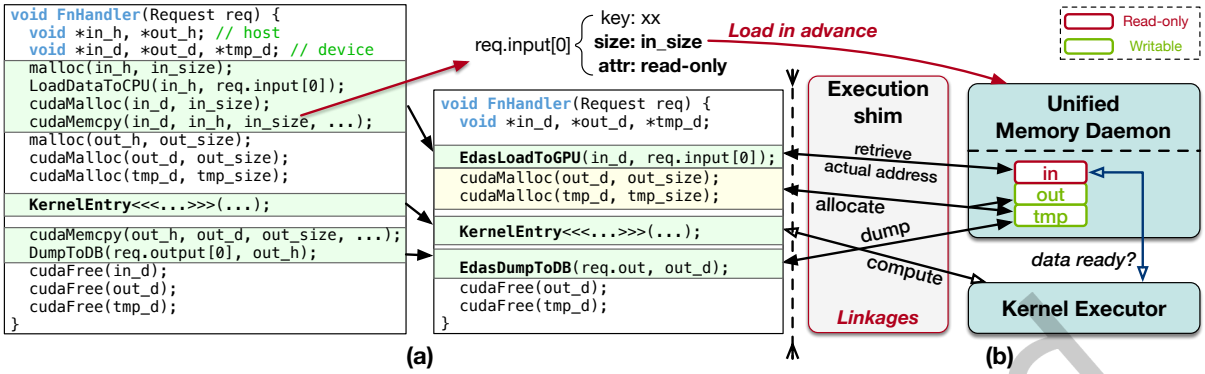

Fig. 6. Figure-(a) presents an example of adapting existing GPU application to a EDAS supported GPU function; Figure-(b) illustrates the workflow of execution shim to build the linkages for correctness ensurance. Note that, GPU contexts are typically created implicitly. In the original code, a context is instantiated automatically when cudaMalloc() is invoked for the first time, as it is required before any memory allocation can proceed. In our design, we also make this process explicit: the GPU context is created during the first call to EdasLoadToGPU(), ensuring that subsequent memory allocations are supported within this context. The loose "data ready" state refers to the condition where only memory allocation has been completed.

The serverless system could perform data pre-loading with a pre-created context when the GPU function is creating its own context. The data loaded by the serverless system will be shared with the GPU function when it actually needs it. This is achievable through inter-process communication, which is both supported on CPU and GPU [4].

5.2 Parallelizing Data and Context

Based on the above analysis, data loading is delegated to the serverless runtime in EDAS. EDAS reconstructs the dependency graph for the GPU function workflow. In the left of Figure 5, EDAS divides the overall data loading stages into two distinct parts: context creation and actual data loading. The GPU function and the memory daemon are responsible for these two parts respectively. The parallelization of these two parts could accelerate overall data loading.

Traditionally, the function loads the data to GPU memory, and gets the correct data address immediately. However, the decoupling of data and context breaks the linkage between the correct data address and GPU function, shown in the right of Figure 5. When a function needs to perform computation, the unified memory daemon does not perceive the right data address that the function computation needs. Even if we can intercept the memory-related interfaces, we still could not build the linkage. This is because GPU functions may request the input data in user requests in any order. Parallelization of data and context disrupts the original programming model.

To address this problem, EDAS proposes two interfaces: `EdasLoadToGPU()` and `EdasDumpToDB()` for GPU functions to access the data managed by EDAS. Users could define a GPU function for EDAS by using these two interfaces and the enhanced user request. Figure 6-(a) depicts an illustrative example of adapting a traditional GPU function to an EDAS-supported GPU function.

Originally, `LoadDataToCPU(req.input[0])` is used by the function handler to load data to CPU memory, which then uses `cudaMemcpy()` to load data to GPU memory. It uses the input id to build the linkage between data and function implicitly. In an EDAS-supported GPU function, the function handler resorts to

`EdasLoadToGPU()` for data loading and `EdasDumpToDB()` for data writeback, which are related to the database. Similarly, `EdasLoadToGPU()` requires the input id, which helps build the linkage between GPU functions and the data pre-loaded by the memory daemon. Note that CUDA native calls like `cudaMalloc()` could still be used.

Except for building the linkages, EDAS achieves two advantages through the two interfaces. First, when the GPU function calls `EdasLoadToGPU()`, the memory daemon returns a correct data address without waiting for the data loading to be completed. This is because the GPU function may need an asynchronous data transfer. Second, when the GPU function calls `EdasDumpToDB()`, the memory daemon directly dumps the data into the database, which also does not wait for completion.

5.3 Correctness Support with Execution Shim

Asynchronous data transfer brings another challenge. Traditionally, synchronization is performed inside the GPU function to ensure that the data is truly ready. However, the serverless system should not hack into the function. To this end, EDAS proposes an execution shim to intercept the common GPU function calls, like `cudaMalloc()` and kernel launching. Memory-related calls are forwarded to the memory daemon, and the kernel calls are forwarded to the executor. The memory daemon and the kernel executor cooperate to ensure data correctness.

Figure 6-(b) demonstrates how the shim interacts with the memory daemon and kernel executor. When the kernel executor receives the kernel call, it communicates with the memory daemon to verify whether all the data required by the kernel has been prepared. If all data is ready, the kernel executor can directly launch the kernel to the GPU. If the data is not ready, the kernel executor waits for a while and retries. When the function exits, EDAS also checks if all intercepted calls complete. Based on the above method, no matter the function requires synchronous data loading or asynchronous data loading, EDAS supports them well.

5.4 Automatic Conversion for Mainstream Applications

New GPU functions developed from scratch can directly utilize the data interfaces mentioned, with only minimal differences from traditional memory management functions. For GPU functions that rely on specific frameworks—such as PyTorch-based DNN applications—we provide an automated conversion tool to streamline the adaptation process.

In the case of **DNN applications built with PyTorch** [40], we provide an automatic conversion tool based on the DNN compiler, Rammer [36]. This tool first converts the PyTorch DNN model into an ONNX file and then directly generates deployable functions (written in CUDA) along with an enhanced user request based on the ONNX file.

Specifically, we design a function template for the DNN models, and we just need to extract the necessary information from the ONNX file to populate this template. As shown in Figure 7, this template consists of two parts: the main function, which handles network setup and request triggering, and the `functionEntry()`, which is triggered by the user request and executes the computation. The `functionEntry()` can be easily wrapped to support binding to the Python side, providing a callable interface. The process of generating code from this template can be divided into five steps.

- **Step 1:** We identify the parameter tensors in the ONNX file and classify them as read-only data. Meantime, we identify the input tensors classify them as writable data, while all other tensors are also treated as writable.
- **Step 2:** Since only the input tensors and parameter tensors need to be fetched from external databases, we use this information to generate the enhanced user request.
- **Step 3:** We generate data loading code for the input tensors and parameter tensors. As shown in the data loading section of Figure 7, we replace the original `cudaMalloc()` and `cudaMemcpy()` interfaces with `EdasLoadToGPU()`. For other tensors, we still rely on `cudaMalloc()`.

```
void functionEntry(Request req) {
    void *in_d, *weight_d, *out_d;
    // -------------------------- data loading section
    EdasLoadToGPU(in_d, req.input[0]);
    EdasLoadToGPU(weight_d, req.input[1]);
    cudaMalloc(out_d, req.output[0]);
    // -------------------------- kernel execution section
    kernelExecute();
    // -------------------------- result writeback section
    EdasDumpToDB(out_d, req.output[0]);
}
void main() {
    setupNetwork();
    RequestTrigger(functionEntry);
}
```

Fig. 7. The GPU function template used by the automatic conversion tool.

- **Step 4:** We identify the output tensors in the ONNX file and use `EdasDumpToDB()` to write the results back, as illustrated in the result-writeback section of Figure 7.
- **Step 5:** As shown in the kernel execution section of Figure 7, we generate the corresponding kernel code and wrap it in a `kernelExecute()` function. The entire GPU function is then ready for deployment.

Notably, large language models (LLMs) cannot be directly processed by current DNN compilers. Fortunately, LLaMa.cpp is a widely used open-source framework. We make slight modifications to LLaMa.cpp [23] to enable it to independently manage weights, the KV cache, and other intermediate results. Specifically, each time an LLM request is triggered, LLaMa.cpp allocates the maximum possible memory required for the KV cache and requests additional memory for other tensors based on actual needs. Using this method, we also support the integration of large language models within the EDAS system.

For **other framework-based applications**, such as the physics applications developed with TorchPhysics [48], the core computations also depend on PyTorch. Therefore, the same automated conversion workflow used for PyTorch-based DNNs can be applied to TorchPhysics-based applications with minimal modification. On the other hand, stencil applications that rely on PyHalide [41] for CUDA code generation require adjustments to PyHalide's code generation pipeline. By desigining a new template and modifying the generation process, we can also enable the automatic integration of data access interfaces. This functionality is part of our future work and will further expand the scope of automated GPU function support.

6 Sharing-based Memory Management

In addition to optimizing the data loading of the first request, we focus on optimizing the data loading of subsequent requests in this section. While existing serverless systems [1, 2] all adopt container reuse to optimize function startup, they do not concern the datapath contention problem of GPU functions. Therefore, we first analyze the data loading of mainstream GPU functions, and then present our multi-stage resource exit scheme to alleviate the datapath contention.

6.1 Data Loading Analysis

Existing serverless systems [1, 2] all adopt container reuse to optimize function startup. Subsequent requests can reuse the container of the previous request (including data and context), and this container will remain warm for a period of time. However, existing optimization methods cannot solve the datapath contention problem of GPU functions.

Table 2. The memory usage of GPU functions (MB).

benchmark	context	read-only	writable	read-only loading	writable loading
bert	254	1282.5	60.1	1282.5	0.1
speech	254	24.8	6.9	24.8	0.2
incep3	254	91.1	11.7	91.1	1.0
nasnet	254	20.3	11.8	20.3	0.6
resnet	254	97.7	11.9	97.7	0.6
s2s	254	6.1	0.1	6.1	0.1
vgg11	254	506.8	38.0	506.8	0.6
lbm	254	0	165	0	165
mrif	254	0	22	0	18
tpacf	254	0.1	28.3	0.1	28.1
bloom	254	5839	1028	5839	2
llama	254	6810	1028	6810	2

To reduce resource contention in the data loading paths, we investigate the data loading requirements of mainstream GPU functions. We find that the memory of a GPU function can be divided into three categories: *GPU context*, *read-only data*, and *writable data*. The memory for GPU context used to store the runtime data, is implicitly allocated during the initial execution of the function. Writable data denotes the memory usage for storing input, intermediate, and output results that may change during the execution. Read-only data, on the other hand, represents the unchangeable data. The memory usage of the three types of data are comparable.

Table 2 shows the GPU memory usage of all the benchmarks. As observed from the table, GPU context occupies the most memory usage except for *bert*, *vgg*11, and the two LLM models, which is 80.2% of the memory usage on average. For *bert*, *vgg*11, and the two LLM models, GPU context occupies 13.6% of the total memory usage on average. Although GPU context occupies a large amount of memory, experimental results show that the context creation time does not change when multiple functions create their contexts simultaneously. This implies that the GPU function's context creation does not interfere with each other.

In addition to GPU context, explicit memory accounts for 42.1% of the overall memory usage. Inside the explicit memory usage, the data needs loading from external sources accounts for 83.9%. Therefore, the GPU functions have large data loading requirements. As demonstrated in § 3.2.2, GPU functions encounter severe resource contention during data loading. The benchmark functions suffer 6.1× data loading time compared to the solo-run case.

However, existing GPUs only offer basic data transfer APIs through PCIe, which could only handle data transfer tasks sequentially. Meanwhile, the network bandwidth used by CPU loading is also limited. There lack of sophisticated interfaces to manage the data loading of multiple functions to improve efficiency. Faced with these limitations, we can only resort to other methods to optimize data loading.

6.2 Multi-stage Resource Exit

Inspired by the sharing-based container reuse method, we optimize data sharing between user requests to solve the data loading contention between functions. While read-only data remains unmodified during function execution, sharing it between multiple user requests does not cause correctness issues or performance problems. It should be noted that multiple user requests are serviced by the same function, which comes from the same service provider. There will also be no malicious memory attack on the GPU.

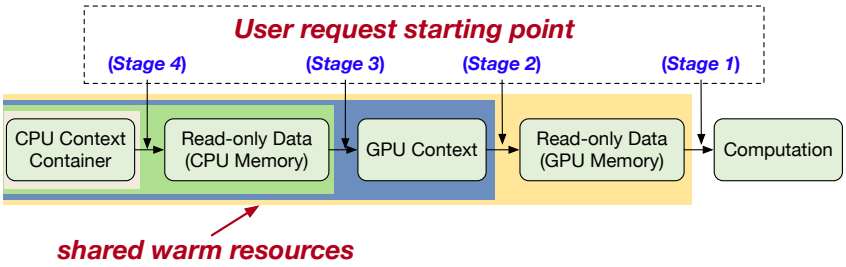

Fig. 8. The multi-stage resource exit mechanism. The shared resources are marked with different background colors for different stages.

In this case, sharing read-only data can effectively reduce data loading requirement, further reducing resource contention in the data loading paths. Therefore, the GPU function's data loading time could be improved. It is worth noting that read-only data is not limited to the weight of ML inference tasks. The *tpacf* benchmark in astrophysics also has read-only data. Any read-only data in the GPU function could be shared by multiple user requests.

Since the enhanced user request contains the read-write attributes of function inputs, EDAS can distinguish between read-only data and writable data. Based on this, EDAS proposes a multi-stage resource exit scheme based on data sharing. First, EDAS only prepares a single copy of read-only data for one GPU function, and multiple requests of the same function share the read-only data. Second, EDAS releases writable data, GPU context, and read-only data in sequence according to their different usage characteristics.

Specifically, we design the multi-stage resource exit scheme based on one key principle: maximizing sharing possibility to reduce the loading requirement. Based on this, EDAS's resource management has five stages as shown in Figure 8:

- **Stage 1:** When multiple requests of a function arrive simultaneously, the memory daemon returns the same read-only data address and prepare the writable data for these requests respectively. When the user requests complete the computation, they release the writable data immediately.
- **Stage 2:** When no user request arrives for a period of time, the GPU function releases the GPU context. This is because the GPU context does not bring data loading contention, and the GPU context creation time is stable.
- **Stage 3:** When no user requests come for another period of time, the GPU function then releases the read-only data on GPU and caches it on the CPU side. This is because CPU memory is a cheaper and larger storage medium, which is very suitable for caching the read-only data.
- **Stage 4:** If the GPU function keeps idle for a period of time, the GPU function then releases the read-only data in CPU memory.
- **Stage 5:** EDAS releases the CPU context and the container. Any requests come later will set the function to Stage 1.

Leveraging the above multi-stage exit scheme, we are able to attain a more flexible management of the tradeoff between GPU resource cost and warm function runtime. Moreover, it is noteworthy that the time interval of each stage is adjustable. For instance, configuring each stage's interval to the previous time interval would enable the GPU function to maintain a longer warm state at a lower cost. Alternatively, configuring the overall duration of the five stages to the previous time interval would allow the serverless platform to support a similar warm effect

Table 3. Hardware and software setups in the experiment.

	Configuration
CPU	Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz, Disk: SSD 100G, Cores: 16, DRAM: 60GB
GPU	Nvidia A10, 24GB HBM, PCIe 4.0 Nvidia A100, 40GB HBM, PCIe 4.0
Software	Operating system: Ubuntu 20.04 with kernel 5.4.0 Docker version: 20.10.21, Nvidia driver: 470.161.03
Container	Image: nvcr.io/nvidia/pytorch:22.09-py3 CUDA: 11.8, cuDNN: 8.6.0, cuBLAS: 11.11.3

at a lesser cost. In this paper, EDAS configures each stage's interval to the previous time interval, which is set as 30 seconds [33, 34].

7 Implementation and Security Model

7.1 Implementation

We implement a prototype of EDAS for components described in §4. Our current implementation supports both GPU functions developed from scratch and domain-specific GPU functions that rely on frameworks. We use Docker [12] and Nvidia Container Toolkit [7] for the application sandbox, GRPC [10] for message passing. We also deploy MinIO [6] an S3-compatible object store for data storing.

The Memory Daemon and Interfaces. The memory daemon runs directly on the host, outside the GPU function containers. When the daemon manages all memory allocation and data transfers for GPU functions, the GPU functions access their data addresses from the memory daemon via data access interfaces. This process is facilitated by CUDA IPC [4], which supports communication between the inside and outside of containers without requiring special handling.

7.2 Security Model

In EDAS, user-uploaded function handlers and data access are considered untrusted elements within the guest environment. We analyze the security model of EDAS from two aspects. On the one hand, we adopt the same security model as traditional CPU-side serverless used [14], using sandboxes to isolate untrusted guest environments to prevent malicious users from threatening hosts and other tenants.

On the other hand, tenants from different sandboxes access GPU memory via EDAS's unified memory daemon, which enforces security boundaries through GPU context isolation. Specifically, the memory daemon maintains a dedicated GPU context for each tenant, while functions belonging to the same tenant share the same context. Although the daemon may manage multiple contexts simultaneously, we mitigate the associated memory overhead by tuning context parameters using $cuCtxSetLimit()$ (§8.10).

If a GPU function crashes, it does not impact the execution of other GPU functions. This isolation is ensured by running each function in its own GPU context. We validate this behavior by inducing GPU function crashes under various scenarios, such as forced termination and incorrect memory deallocation. In all cases, the failure is confined to the crashing function, with no observable effect on other concurrently executing GPU functions.

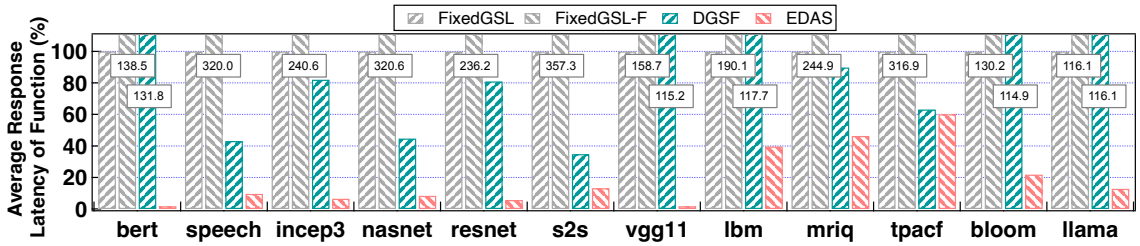

Fig. 9. The average function response latencies of FixedGSL-F, DGSF and EDAS, normalized to FixedGSL.

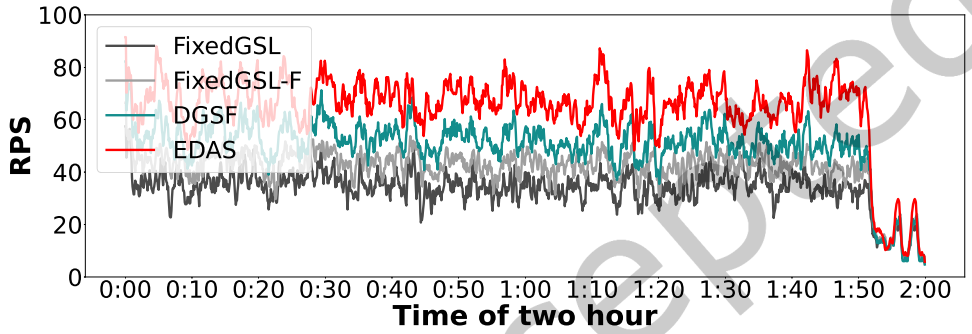

Fig. 10. The system throughput (RPS, requests per second) of FixedGSL, FixedGSL-F, DGSF, and EDAS.

8 Evaluation

8.1 Experimental Setup

Testbed. We mainly evaluate EDAS on a server equipped with an Nvidia A10 (24GB) GPU. We also evaluate EDAS on a cluster of servers equipped with the Nvidia A10 (24GB) GPU in §8.7. Meanwhile, we evaluate EDAS on a server equipped with an Nvidia A100 (40GB) GPU in §8.8. The detailed hardware and software setup are shown in Table 3.

Benchmarks and workload. We use seven DNN applications, three scientific computing applications, and two large language models for evaluation. These applications cover diverse domains such as object recognition, speech generation, natural language processing, fluid mechanics, and real-time medical imaging. Detailed information about the benchmarks can be found in Table 1 of § 3. In all experiments, we randomly extract the function traces from Microsoft Azure Functions (MAF) [42] and map our applications to them. We replay these traces for 2 hours.

Baselines. We use FixedGSL, FixedGSL-F, and DGSF as the baselines. FixedGSL is only capable of allocating memory for functions at a granularity of 1 Gigabyte. We extend FixedGSL to FixedGSL-F, which supports flexible memory allocation for functions. Additionally, we also choose DGSF [22] as another baseline. DGSF pre-creates two GPU contexts for each function to eliminate GPU context preparation time. When the pre-created GPU contexts are available, the user requests could use them. When the contexts are occupied, the later user requests need to construct new GPU contexts [22]. Note that, to better distinguish our effectiveness from previous works on the CPU side [13, 14, 33], we enhance all the systems with the pre-warmed container. Specifically, the user requests from the same function could share the container, which could optimize the container creation time.

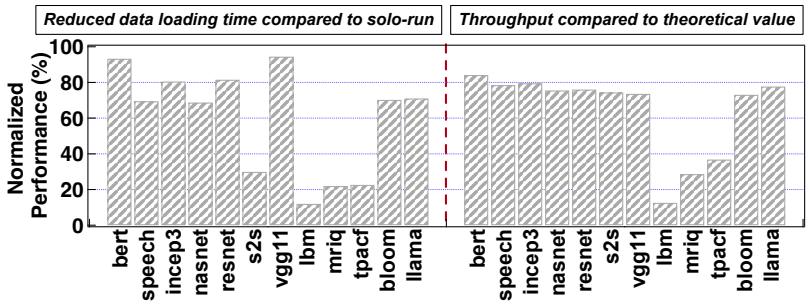

Fig. 11. The throughput performance of of EDAS normalized to theoretical throughput and the reduced setup time reduction of EDAS compared to solo-run case.

8.2 Overall Performance

In this subsection, we collect the average response latency of functions and system throughput for FixedGSL, FixedGSL-F, DGSF, and EDAS. The load of user requests is 20-200 requests per minute, which is randomly chosen from the MAF traces [42].

Figure 9 illustrates the average response latency of each benchmark with FixedGSL, FixedGSL-F, DGSF, and EDAS. On average, EDAS outperforms FixedGSL, FixedGSL-F, and DGSF by 16.2×, 31.1×, and 16.6×, respectively. EDAS outperforms FixedGSL, FixedGSL-F, and DGSF on the minimum by 2.16×, 3.47×, and 1.11×, respectively. Besides, we also collect the 99%-ile latency of each benchmark, which is not shown due to page limitation. For the 99%-ile latency of all functions, EDAS outperforms FixedGSL, FixedGSL-F, and DGSF by 7.9×, 16.6×, and 9.3×, respectively.

FixedGSL and FixedGSL-F do not prioritize the latency performance of functions. They only schedule the function invocations with the memory usage in a fixed granularity. In DGSF, although GPU contexts are pre-created to reduce setup, the benefit can easily diminish. This is because DGSF only pre-creates 2 contexts for the same function due to the high memory overhead. When the function load bursts, only the first several invocations can enjoy the pre-created contexts. Moreover, it ignores the resource contention of the data-loading path, still suffering long data preparation. EDAS takes both of these factors into account and is highly effective in reducing latency in all circumstances.

Figure 10 illustrates the system throughput of FixedGSL, FixedGSL-F, DGSF, and EDAS. EDAS's system throughput outperforms FixedGSL, FixedGSL-F, and DGSF by 1.91×, 1.55×, and 1.29×, respectively. The throughput improvement comes from the fast data loading of EDAS. While GPU functions do not need to wait for data for a long time, they could complete the compution and release the GPU resources in a short time. Therefore, EDAS achieves a higher system throughput.

We can also observe that FixedGSL-F performs worse than FixedGSL in both latency and throughput. This is because FixedGSL-F has larger function concurrency due to the flexible memory usage. While FixedGSL and FixedGSL-F do not concern the data loading management, more function concurrency means more serious and unrestrained contention of the data-loading paths. Therefore, FixedGSL-F performs worse than FixedGSL in both latency and throughput.

8.3 Supported Peak Throughput

In this subsection, we use the same experimental setup as § 3 to evaluate the function's data loading time and the system throughput for EDAS. The left part in Figure 11 shows the normalized reduced response latency of each function using EDAS compared to the solo-run case. The right part in Figure 11 shows the throughput of each

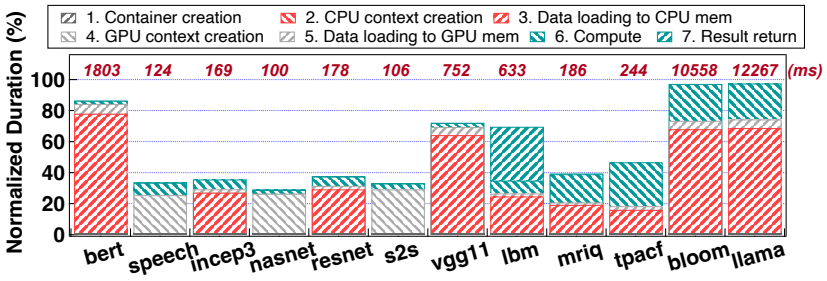

Fig. 12. The duration breakdown using the parallelized data loading scheme only.

Table 4. Latency breakdown of *resnet*50 running with multi-stage resource exit scheme.

Time (ms)	SOTA	stage 1	stage 2	stage 3	stage 4
end-to-end	438.2	8.9	95.6	155.8	438.2
return data	0.1	0.1	0.1	0.1	0.1
compute	7.1	7.1	7.1	7.1	7.1
GPU data	10.5	0.2	**10.5**	*10.5*	*10.5*
GPU context	88.4	0	*88.4*	**88.4**	**88.4**
CPU data	138.1	1.5	**1.5**	*138.1*	*138.1*
CPU context	1	0	0	0	1
Container	193	0	0	0	193

function using EDAS normalized to the theoretical one. The theoretical throughput is computed based on the function's pure computation time. For example, *bert* needs 42 milliseconds to complete the function computation, its theoretical throughput is 1000/42 = 23.8 requests per second.

Experimental results show that EDAS reduces the data loading time of function invocations by 60.3% on average. As shown from the figure, the DNN workloads have a greater effect than the scientific services. This is because DNN workloads have more read-only data, which could benefit more from the multi-stage resource exit scheme. Scientific services could only benefit from parallelized data loading and the last stages of the multi-stage resource exit scheme. Meanwhile, the system throughput of functions is 64.2% of its theoretical value on average. This comes from the reduced data loading time and shorter GPU resource occupation.

Note that, there is also a significant gap between EDAS and the theoretical value because of the resource contention. Although EDAS could optimize the data loading by deduplicating the repetitive ones, some data loading could not be avoided. These data loading still could encounter the datapaths contention.

8.4 Ablation of Optimization Mechanisms

Effectiveness of parallelized data loading. To evaluate the effectiveness of the parallelized data loading scheme, we measure the end-to-end duration of the GPU functions like the experiment in §3.2.1. The function invocations are generated in a close-loop manner. Figure 12 shows the duration breakdown of all the benchmark functions with EDAS. Excluding two LLM models, EDAS reduces the data loading time of all the functions by 29.1% on average compared with the results in Figure 2. The benchmark functions either have the GPU context preparation time or the actual data loading time. The parallelized function data loading scheme hides one part, thus improving the data loading performance.

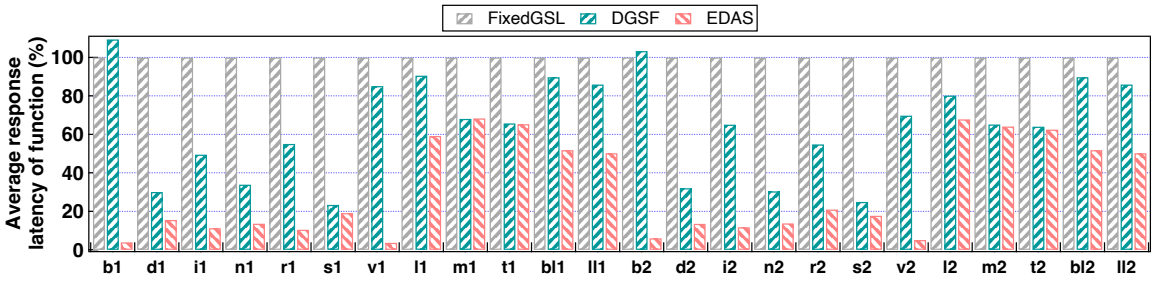

Fig. 13. The average function response latencies of FixedGSL, DGSF and EDAS with 24 functions. Results are normalized to FixedGSL. The x-axis shows the 24 GPU functions. For instance, $b1$ and $b2$ represent the two bert functions.

Effectiveness of multi-stage exit. We use Resnet50 as the representative benchmark to show the breakdown in EDAS's multi-stage exit mechanism. As shown in Table 4, compared with the traditional method which only includes warm container state, EDAS contains four stages for fine-grained management. In the table, the bold values and the zero values are hidden by the italic value in EDAS. For example, the GPU data loading time (10.5) and the CPU data loading time (1.5) are hidden by the GPU context creation time (88.4) in stage 2. In stage 4, EDAS is also able to hide data loading latency along with GPU context creation. The only difference between stage 3 and stage 4 lies in the container creation step. EDAS improves the function's latency by an average of 10.6× in the other three states compared to the cold state (baseline), with a minimum of 1.6×. All other benchmarks show similar results. Experimental results show that EDAS reduces 91.7% of the queries in the cold state.

Note that, LLaMA and BLOOM benefit more from the multi-stage resource exit scheme due to their larger data footprints and longer data loading times. Once these models enter stage 3, a substantial amount of time is spent loading their parameters into CPU memory and GPU memory. As a result, the ability to defer memory release becomes especially valuable. Moreover, LLaMA and BLOOM impose greater memory pressure on the GPU, which in turn increases the need for multi-stage exit among other co-located functions sharing the same resources.

8.5 Efficiency with More Functions

In this subsection, we use more functions to further demonstrate the effectiveness of EDAS. While the 12 functions in §8.2 have a load of 80-200 requests per minute, we add 12 functions with a lower load in this section. Specifically, we use $b1, b2$ to represent two different *bert* benchmarks. These 12 functions have a load of 10-20 requests per minute, and the traces are also randomly mapped from the MAF trace.

Figure 13 shows the average response latency of 24 functions under FixedGSL, DGSF, and EDAS. EDAS outperforms FixedGSL and DGSF by 7.4× and 5.3×. EDAS outperforms FixedGSL and DGSF in terms of system throughput by 1.85× and 1.23×, respectively. EDAS still achieves considerable improvements compared to FixedGSL and DGSF. This demonstrates EDAS's effectiveness in dealing with more GPU functions. However, EDAS have less improvement compared to results in §8.2. This is because the added functions bring more data loading, which brings more resource contention. Meanwhile, the GPU functions encounter more function cold startup due to limited GPU memory, which benefits little from the multi-stage resource exit scheme.

8.6 Scaling EDAS Out

To assess EDAS's scalability, we conduct the performance tests on a cluster with 4 A10 devices on 4 nodes. Figure 14 illustrates the average latency of the ten applications using FixedGSL, DGSF, and EDAS on the cluster. On average, EDAS surpasses FixedGSL and DGSF by 10.5× and 9.5×, respectively. Experimental results also

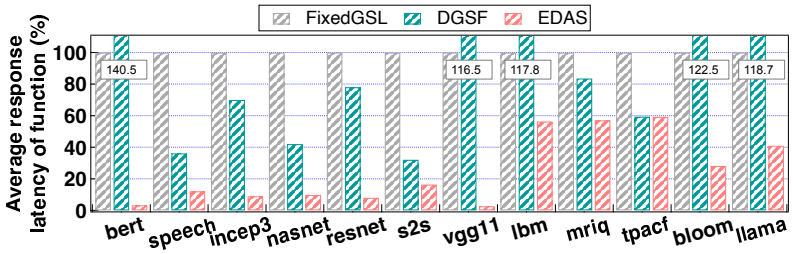

Fig. 14. The average function response latencies of FixedGSL, DGSF and EDAS on a cluster with 4 A10 GPUs, normalized to FixedGSL.

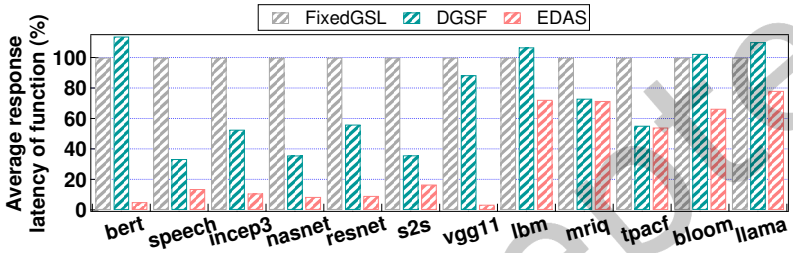

Fig. 15. The average function response latencies of FixedGSL, DGSF and EDAS with local database.

show that EDAS delivers a 1.79× and 1.24× increase in system throughput compared to FixedGSL and DGSF, respectively.

Based on the aforementioned data, it can be deduced that EDAS has excellent scalability. Although we randomly dispatch the function invocations to GPU, EDAS could still improve the overall system's average latency and system throughput. This is because the proposed optimization methods of EDAS are orthogonal to the task distribution strategy at the cluster level. EDAS can integrate any cluster-level scheduling strategies proposed for specific scenarios to further achieve performance improvements.

Note that, EDAS maintains its performance advantage even in larger-scale environments. This is because the system continues to employ the same load-balanced request distribution strategy regardless of scale. As a result, the workload pattern observed by each GPU remains consistent, ensuring that performance does not degrade as the system scales.

8.7 PCIe Contention Only

The database could be hosted on the local disk (SSD) or accessed through the remote network. The main experiments are all conducted on the network with 20Gbit/s (2.5 GB/s) bandwidth. Meanwhile, the PCIe 4.0 has a bandwidth of 32 GB/s. The GPU functions first encounter the network bandwidth contention. In this subsection, we set the database in the same node, proving the effect of PCIe contention.

Figure 15 presents the function response latency of FixedGSL, DGSF, and EDAS, which are all normalized to FixedGSL. On average, EDAS surpasses FixedGSL and DGSF by 8.3× and 6.1×, respectively. Meanwhile, EDAS outperforms FixedGSL and DGSF regarding system throughput by 1.55× and 1.14×, respectively. This is because the baselines still suffer from severe PCIe contention except from the data loading contention on the host. This implies the data loading management schemes in EDAS also mitigate the data loading contention in PCIe.

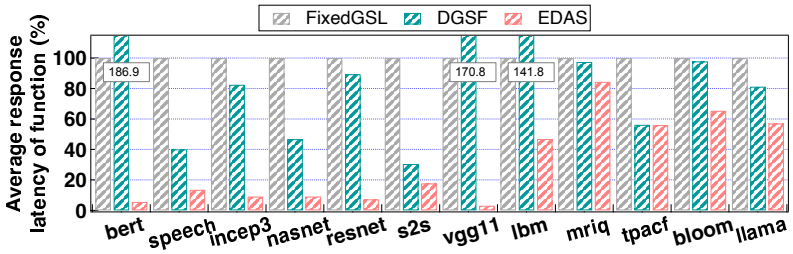

Fig. 16. The average function response latencies of FixedGSL, DGSF and EDAS on a server with 1 A100 GPU, normalized to FixedGSL.

8.8 Evaluation on Different GPUs

In this subsection, we conduct the experiments on the node equipped with an Nvidia A100 GPU. Nvidia A100 GPU has greater computing power and large memory space than Nvidia A10 GPU.

Figure 16 presents the function response latency of FixedGSL, DGSF, and EDAS, which are all normalized to FixedGSL. On average, EDAS surpasses FixedGSL and DGSF by 10.3× and 9.8×, respectively. Meanwhile, EDAS outperforms FixedGSL and DGSF regarding system throughput by 1.73× and 1.22×, respectively. Although Nvidia A100 brings more function concurrency, it still faces the same datapath bandwidth. Therefore, EDAS still benefits from the innovative data loading management schemes. This further leads to better response latency of functions and higher system throughput.

8.9 Overhead of EDAS

The overhead of EDAS arises from the runtime interception of functions and the communication between the kernel executor and memory daemon. We test the runtime interception of functions for a million times, compute the average overhead, which is about 5 microseconds. Meantime, we use the shared memory to support the communication between the executor and daemon. We also test the communication overhead, which is less than 10 microseconds. Therefore, the overhead of EDAS is negligible.

8.10 Cost-benefit Analysis

To support parallel function setup and multi-stage resource teardown, EDAS introduces a memory daemon responsible for data loading and GPU memory management. However, the memory daemon requires GPU contexts to manage actual GPU allocations, which introduces memory overhead. On an A10 GPU, the default memory size is 254 MB per context. To mitigate this overhead, we optimize the context configuration parameters using *cuCtxSetLimit*(), reducing the per-context memory to approximately 120 MB. Assuming ten users concurrently deploying functions on a single GPU, a separate GPU context is instantiated for each user, resulting in a total memory overhead of 1.2GB. Despite this modest overhead, EDAS reduces function latency by 16.2× and improves system throughput by 1.91× compared to the state-of-the-art serverless platform.

9 Conclusion

In this paper, we identify that the data loading delegated to GPU functions is a critical bottleneck in the existing GPU serverless systems. The lack of data attributes is preventing the system from optimizing the data loading, leading to high function latency and low throughput. To address this issue, we propose EDAS, a serverless system with proactive data management for GPU functions. EDAS enhances the request specification by allowing users to specify the data attributes including the data size, data read-write property. Leveraging the data attributes, EDAS

optimizes the GPU functions with two proactive data management mechanisms. With the data size attribute, EDAS accelerates the loading process of a cold GPU function by parallelizeing the data loading and function setup. With the data read-write property, EDAS maximize the data reuse among multiple requests for the same function through a multi-stage resource exit scheme. Our experimental results show that EDAS reduces function duration by 16.2× and improves function throughput by 1.91×, compared with state-of-the-art solutions.

Acknowledgments

This work is partially sponsored by the National Key Research and Development Program of China (2024YFB4505703), the National Natural Science Foundation of China (62302302, 62232011), and Natural Science Foundation of Shanghai Municipality (24ZR1430500). Quan Chen is the corresponding author.

References

[1] [n. d.]. Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org. Accessed: January 23, 2025.
[2] [n. d.]. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: January 23, 2025.
[3] [n. d.]. Best practices for GPU-accelerated instances. https://www.alibabacloud.com/help/en/functioncompute/fc-3-0/product-overview/instance-types-and-usage-modes. Accessed: January 23, 2025.
[4] [n. d.]. *CUDA Interprocess Communication.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=interprocess#interprocess-communication Accessed: January 23, 2025.
[5] [n. d.]. GPU-Enabled Docker Image to Host a Python PyTorch Azure Function. https://github.com/puthurr/python-azure-function-gpu. Accessed: January 23, 2025.
[6] [n. d.]. MinIO. https://min.io/. Accessed: January 23, 2025.
[7] [n. d.]. NVIDIA Container Toolkit. https://github.com/NVIDIA/nvidia-container-toolkit. Accessed: January 23, 2025.
[8] [n. d.]. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: January 23, 2025.
[9] [n. d.]. qGPU Overview. https://www.tencentcloud.com/document/product/457/42973. Accessed: January 23, 2025.
[10] [n. d.]. Remote Procedure Call (RPC) framework. https://grpc.io/. Accessed: January 23, 2025.
[11] [n. d.]. What is the cGPU service. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/what-is-the-cgpu-service. Accessed: January 23, 2025.
[12] Year of the Docker release or last update. Docker. https://www.docker.com/. Accessed: January 23, 2025.
[13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
[14] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.
[15] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[16] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2071–2084.
[17] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. Pipeswitch: Fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 499–514.
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[19] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 797–813.
[20] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.
[21] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.
[22] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 739–750.

[23] Georgi Gerganov. [n. d.]. ggerganov/llama.cpp: Inference of Meta's LLaMA model (and others) in pure C/C++. https://github.com/ggerganov/llama.cpp. Accessed: January 23, 2025.
[24] Anshuman Goswami, Jeffrey Young, Karsten Schwan, Naila Farooqui, Ada Gavrilovska, Matthew Wolf, and Greg Eisenhauer. 2016. GPUShare: Fair-sharing middleware for GPU clouds. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1769–1776.
[25] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*. 635–644.
[26] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).
[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[29] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 257–262.
[30] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 141–154.
[31] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2023. Bloom: A 176b-parameter open-access multilingual language model. (2023).
[32] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.
[33] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund
[34] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through {Inter-Function} Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 69–84.
[35] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. 2019. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 95–102.
[36] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 881–897.
[37] Anup Mohan, Harshad S Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. *HotCloud* 2019, 10.5555 (2019), 3357034–3357060.
[38] Diana M Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel and Distrib. Comput.* 139 (2020), 32–42.
[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 57–70.
[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8024–8035.
[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
[42] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423* (2020).
[43] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2011. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61, 6 (2011), 804–816.
[44] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[48] Derick Nganyu Tanyu, Jianfeng Ning, Tom Freudenberg, Nick Heilenkötter, Andreas Rademacher, Uwe Iben, and Peter Maass. 2023. Deep learning methods for partial differential equations and related parameter identification problems. *Inverse Problems* 39, 10 (2023), 103001. doi:10.1088/1361-6420/ace9d4

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[50] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[51] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. 2017. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*. 1–4.

[52] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. 2005. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 148–162.

[53] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[54] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning.. In *OSDI*. 533–548.

[55] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-grained GPU sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems* 2 (2020), 98–111.

[56] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.

Received 23 January 2025; revised 3 April 2025; accepted 7 May 2025