Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

GPUless – Serverless GPU Functions

Master Thesis

Lukas Tobler

Wednesday 19th January, 2022

Advisors: Prof. Dr. Torsten Hoefler, Marcin Copik
Scalable Parallel Computing Lab, ETH Zürich

Abstract

For many years, serverless has been an emerging computing paradigm, with Function-as-a-Service (FaaS) being especially popular. When it comes to GPU-enabled machine learning applications, commercial options for FaaS are limited. GPU execution nodes are not typically available because of their high cost and the difficulty of efficiently sharing them between tenants in isolated environments.

Multi-Instance GPU (MIG) is a new feature of the NVIDIA A100 device of the Ampere architecture that provides performance and security isolation by partitioning one physical GPU into multiple GPU instances of configurable size. MIG opens up the possibility to build serverless platforms with stronger isolation than what was possible previously.

We present the GPUless system, a prototype client-server CUDA execution service based on MIG isolation. The client intercepts the CUDA API and is compatible with current-generation PyTorch machine learning applications. The server provides dynamic resource management for MIG devices of requested size for clients and provides an execution environment. We present a novel way of transporting CUDA API calls over the network: Aggregating call traces and only synchronizing with the remote executor when necessary, reducing network overheads.

We show that in cold-start scenarios, our system can be very effective. We apply some optimizations to overcome implementation inefficiencies in machine learning frameworks that lead to a cold-start performance of our system that is even faster than native execution in some cases. We can also show that in a hot execution setting (model is pre-initialized), we can achieve performance close to native execution while still being orders of magnitudes faster than execution in the AWS (Amazon Web Services) Lambda environment. An analysis of bandwidth requirements shows that our system will perform well if at least 1 Gbps of network bandwidth is available.

Contents

Chapter 1

Introduction

The serverless computing model has been a big trend for the global IT industry. Serverless is a computing paradigm, where code is run on dynamically allocated cloud resources, with the goal to lift the burden of managing server infrastructure from developers. The move towards cloud-based computing has been accelerating for many years [43]. Function-as-a-Service (FaaS) is a programming paradigm based on event-driven function execution in fully dynamically allocated runtime environments. FaaS has become a very popular model, as it can be very cost-effective: a pay-as-you-go model is usually applied, where only actual execution time is billed. However, when it comes to GPU applications, few options are available for FaaS. As of 2021, none of the big commercial providers (Amazon, Microsoft, Google, and IBM) offer GPU-enabled FaaS options. GPU nodes are readily available as a Infrastructure-as-a-Service (IaaS) option, but this is a different paradigm that negates many of the advantages FaaS can bring: the advantage of providing computing resources that are fully dynamic, and completely opaque to the application developer, meaning that developers have no insight or control over where their functions are actually run. The reason why GPUs are available for FaaS computing is that many of the fine-grained sharing and isolation mechanisms that are available for CPU execution are not available for GPUs. Sharing GPUs between tenant is not efficient: GPUs have to be attached to containers and reinitialized frequently. If performance and security isolation is desired, allowing concurrent use by many tenants is very challenging.

Machine learning (ML) training and inference has become an important workload across many industries. To efficiently run ML inference in the cloud, GPUs are required, as they speed up these tasks by orders of magnitude. The flexible model of serverless is very attractive for machine learning and has been shown to be cost effective and fast [22]. However, for real world applications, such as IoT devices that rely on ML inference for making decisions, there still are challenges. Previous work [21] shows that FaaS is

a viable option for machine learning inference, but the latencies involved, especially in cold start scenarios, still leave a lot to be desired when trying to meet strict service-level-agreements (SLAs).

A challenge for using GPU accelerator devices in the cloud is that to achieve high utilization, the devices need to be shared between multiple tenants fairly and efficiently and with strong isolation. The KNIX platform [40] provides such a system by virtualizing device access at the CUDA API level. However, the KNIX system still introduces significant overhead compared to executing GPU functions natively. Another idea is to use NVIDIAs most recent technology for concurrent device usage, Multi-Instance GPU (MIG). In MIG, devices can be physically partitioned, giving strong isolation guarantees. This work leverages MIG to share a single device among multiple clients, without software-based virtualization techniques.

In this thesis, we propose GPUless: a GPU-native runtime system for GPU FaaS functions, based on intercepting the NVIDIA CUDA API and executing it on a remote, dynamically allocated GPU environment. GPUless uses NVIDIAs MIG technology to achieve:

- Low latencies for cold and hot execution of machine learning inference tasks.
- High utilization when sharing GPUs between multiple tenants.
- The ability to run native CUDA applications remotely without a real GPU being installed on the host system.

Chapter 4 outlines the design of the system and its components, chapter 5 gives insight into the implementation details of the system, and chapter 6 will evaluate the system against a set of benchmarks, including a selection of popular machine learning models.

Chapter 2

Background

In this chapter, we will introduce the concepts of serverless computing in more detail. We will also discuss the CUDA programming model and NVIDIAs MIG technology and how to apply it to a FaaS system.

2.1 Serverless

Serverless is an emerging paradigm for cloud computing. The goal is to remove the responsibility of server deployment and management from developers and let a cloud provider deal with it. Low-level details (such as deployment, configuration, and scheduling) are abstracted away, and resources are made available on-demand and with an efficient pay-as-you-go model. This alleviates the problem of scaling and load-balancing for developers, who would have to rely on over-provisioning models in the past.

The authors of "The Rise of Serverless Computing" [11] define the Serverless paradigm as follows:

> *"Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running."*

Function-as-a-Service (FaaS) is an immediate application of serverless computing, where small blocks of code ("functions") are executed on dynamically allocated environments, with execution triggered by event-driven architectures. Typically, execution time is limited to a few minutes.

There are many commercial FaaS providers, like AWS Lambda [5], Google Cloud Functions [3], Microsoft Azure Functions [2], and IBM Cloud Functions [4]. Additionally, there are Open Source efforts to build FaaS platforms such as Apache OpenWhisk [1].

In the next paragraphs, we will briefly introduce the main components and characteristics of FaaS.

Execution environment. In FaaS, the developer has no control over where their code is run. This deliberate abstraction ensures developers do not have to manage any lower-level resources themselves. The runtime systems usually execute code in sandboxes built with containers or microVMs. Some platforms only provide a very high-level API for application development, and others give users the ability to submit their own container images. Cold-start latencies tend to be high in FaaS: "Peeking Behind the Curtains of Serverless Platforms" [44] reports a median for cold start latencies in commercial platforms of a few hundred milliseconds, with maximums of up to multiple seconds.

Triggers. In FaaS systems, code is typically pre-registered with the service. An execution is invoked by *triggers* as part of an event-driven architecture. Such triggers can be HTTP API gateways for manual invocation or cloud event sources. An example of such a cloud event source in the Amazon ecosystem would be new data appearing in AWS S3 storage, Amazon's object data storage service.

State and Storage. FaaS systems use stateless functions because the functions can be run anywhere and can be canceled or migrated at any time for fault tolerance reasons. However, of course, there is a need for storage for most applications. Typically, there are three storage options available for serverless functions:

- Embedding of data or libraries in the container image. Very limited in size (For example in AWS: 50 MB).
- Temporary storage (`/tmp`) on the execution environment. Limited in size.
- External storage systems, such as AWS S3 in the Amazon ecosystem. Elastic storage size.

There have been efforts to integrate GPUs in such cloud infrastructures (see sections 3.2.1, 3.2.2), but these have not been especially successful at achieving the low latencies ML inference tasks usually require.

This work explores the provisioning of GPUs in a FaaS context while reducing the environment setup to the bare minimum. We can achieve much better cold start latencies than current commercial offerings by relying on NVDIDIAs MIG technology for isolated execution environments and forgoing a typical CPU runtime environment.

2.2 GPU Programming Model

We will briefly introduce the architecture of NVIDIAs Ampere generation of computing devices, the A100 in particular. This device was chosen because it is the only one supporting Multi-Instance GPU (MIG) as of this GPU generation.

Ampere architecture. NVIDIAs Ampere architecture at the highest level is composed of Graphics Processing Clusters (GPC) containing Streaming Multiprocessors (SM). Each SM comprises execution units: CUDA cores and Tensor cores. On these, so-called kernels are executed in SIMT (Single Instruction, Multiple Threads) model.

The A100 device has 7 GPCs containing 12 SMs each for a total of 84 SMs. Each SM contains 128 CUDA cores.

CUDA API. CUDA is NVIDIA's so-called programming model and interface for interacting with a GPU device. Execution happens in so-called *kernels*, which define the parallel work. When launching a kernel, it needs to be configured with block and grid dimensions. A CUDA thread block is a group of threads executed on a specific SM. A CUDA grid is a group of thread blocks scheduled for execution on many SMs on a CUDA-capable GPU. Generally, an SM can schedule many thread blocks at the same time.

This design makes CUDA kernels scalable: A kernel can be executed on different size GPUs without prior knowledge of where the code will run due to partitioning work into thread blocks and grids.

Programming example. To illustrate how the CUDA API is used, we provide a simple example for vector addition in Listing 2.1. A block size of one is used in the example, so a thread is spawned in a separate thread block for every value in the vector. The `vadd` function is defined as `__global__`, a C++ extension that marks the function as callable from both host and device. The CUDA compiler will generate the appropriate machine code for the function to execute on the device. Note also the `vadd<<<N,1>>>(...)` special syntax for launching a kernel with given grid and thread block dimensions in the triple brackets. This syntax is also a CUDA-specific C++ extension.

CUDA extension libraries. Additionally to the basic CUDA API, NVIDIA also provides a set of extension libraries that implement many essential primitives. Necessary for our work are cuDNN [30] and cuBLAS [29] because they are extensively used in PyTorch [32], a machine learning framework our benchmarks are based on. cuDNN provides primitives for standard deep neural network applications such as forward and backward convolution,

```
#define N 8

__global__ void vadd(const float *a, const float *b,
                     float *c, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

// allocate memory on the host
float *host_a = malloc(sizeof(float) * N);
float *host_b = malloc(sizeof(float) * N);
float *host_c = malloc(sizeof(float) * N);

// allocate memory on the device
float *device_a, *device_b, *device_c;
cudaMalloc(&device_a, sizeof(float) * N);
cudaMalloc(&device_b, sizeof(float) * N);
cudaMalloc(&device_c, sizeof(float) * N);

// initalize host memory
...

// copy host memory to device
cudaMemcpy(device_a, host_a, cudaMemcpyHostToDevice);
cudaMemcpy(device_b, host_b, cudaMemcpyHostToDevice);

// execute a kernel
vadd<<<N, 1>>>(device_a, device_b, device_c, N);

// copy device memory back to host
cudaMemcpy(host_c, device_c, cudaMemcpyDeviceToHost);
```

Listing 2.1: CUDA programming example

pooling, normalization, and activation layers. cuBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) [9] for the CUDA runtime and provides implementations for operations such as matrix multiplication. The benefit of these NVIDIA-provided libraries is that they are highly optimized and tuned for many generations of accelerator devices.

Typical CUDA applications. To illustrate how typical applications use CUDA, we analyze the ResNet50 [19] image recognition model implemented using the PyTorch framework. The following table shows the anatomy of the application from a CUDA standpoint. There are not just a few kernel invocations but over two hundred and a substantial number of host-to-device

(H2D) memory copies. Another important take-away is that PyTorch makes heavy use of cuDNN, to the point where most executed kernels are shipped with cuDNN and not PyTorch. cuBLAS is only used as an auxiliary library in this example and is only called a few times.

Kernel launches	Unique kernels	cudaMalloc	H2D	D2H	cuBLAS API calls	cuDNN API calls
219	27	17	321	2	5	1349

2.3 NVIDIA Multi-Instace GPU (MIG)

Multi-Instance GPU (MIG) [31] is a GPU partitioning technology developed by NVIDIA for the A100 product family of the Ampere architecture. Its goal is to provide a mechanism for sharing a GPU between multiple users who do not use the entire device independently. A MIG device partition is supposed to provide complete fault, performance, and security isolation.

MIG is the latest of NVIDIAs concurrency mechanisms and supplements CUDA streams [38] and the Multi-Process Service (MPS) [27]. Both streams and MPS are part of the CUDA API, while MIG is a device-level technology. This means that MIG can be used in conjunction with the older streams and MPS features. However, MIG gives stronger isolation guarantees than previous concurrency mechanisms: Memory protection, memory bandwidth Quality of Service (QoS), and error isolation. There is no error isolation in MPS, so one failing process can lead to failures of concurrently running processes [31].

We will assume features and configurations present on the NVIDIA A100 device when discussing MIG because this is currently the only MIG-enabled device available. Future device generations will likely have some significant differences.

In MIG, a device can be partitioned into at most seven so-called *GPU instances (GI)*, one for each Graphics Processing Cluster (GPC). Each GI has a certain number of streaming multiprocessors (SM) and allocated memory. It does not share memory bandwidth with other GIs. Figure 2.1 gives an overview of how a MIG device can be partitioned. Blocks highlighted in green are selected partitions. A partitioning scheme is only possible if there is no overlap in selected partitions. Additionally to partitioning into GPU instances, there is also the ability to further partition these into *Compute instances.*. Compute instances do not provide performance isolation and are used for logical partitioning of applications inside a GPU instance.

Note that compute capabilities are divided into 7^{ths}, but memory is divided into 8^{ths}. This mismatch is a source of some inefficiency because, in some configurations, ⅛ of memory is left unused.

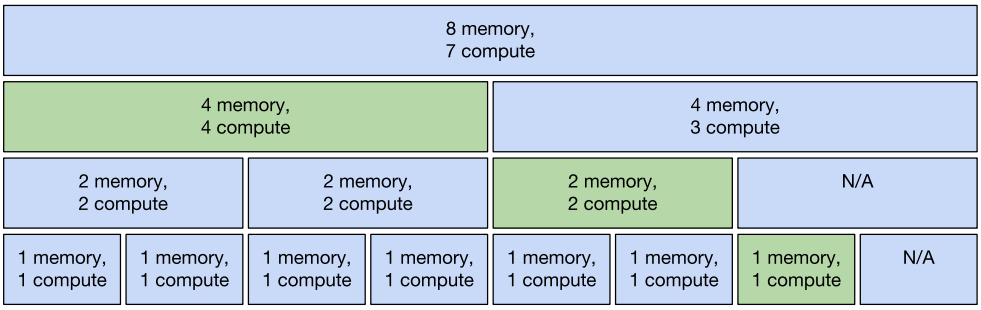


Figure 2.1: Example MIG partitioning scheme

Application to Serverless. It is easy to see how MIG could be applied to serverless computing: Tenants could request a partition of the CUDA device that suits the demands of their applications. Many machine learning applications will not make use of an entire A100 device, as it is a very large accelerator with up to 80 GB of memory. Few models require this much memory.

Chapter 3

Related Work

This section will discuss work related to CUDA device virtualization (Section 3.1), GPU device sharing in clouds (Section 3.2), and previous work that made use of NVIDIA's MIG technology (Section 3.3).

3.1 CUDA Device Virtualization

3.1.1 DS-CUDA

Distributed-Shared CUDA [23] is a solution for virtualizing a large number of GPU-equipped nodes in a cluster as a single device as middleware. Virtualization allows running a CUDA application on many GPUs without having to use frameworks such as MPI, simplifying the deployment of CUDA applications drastically. An additional goal of DS-CUDA is to increase reliability: Redundant execution of specific fractions of work is used to detect errors. Error detection was crucial for accelerator devices of that era because error rates of up to 10% were reported at the time [18].

DS-CUDA is implemented as a middleware between native CUDA applications and execution nodes using the `LD_PRELOAD` library interception technique. CUDA API calls are intercepted and forwarded to an execution node using basic TCP or RDMA transport or NVIDIAs GPUDirect RDMA technology.

The downsides of DS-CUDA are that it only supports the outdated CUDA 6.5 API and relies on a custom CUDA extension that requires source code access to the CUDA application. Source code modification disqualifies DS-CUDA immediately for machine learning applications, as the majority of these use frameworks such as PyTorch [32] or Tensorflow [7]. Also, DS-CUDA is not concerned with GPU device sharing, i.e., multiple applications using *one* device, which is a focus of this work considering the considerable size of a current accelerator like the NVIDIA A100.

The redundancy features have also become much less relevant, as many clusters these days use enterprise-grade CUDA devices that support ECC memory, instead of commodity GPUs.

3.1.2 rCUDA

rCUDA [15] is a middleware solution like DS-CUDA but significantly more mature. Claimed benefits of rCUDA include:

- More GPUs are available for a single application.
- Idle GPU resources in a node can be utilized despite the CPU being used.
- Several VMs can concurrently access the same GPU in a shared manner.

The downsides of rCUDA are similar to that of DS-CUDA: Finer-grained device sharing is not something that it is concerned with; the main focus is giving applications access to *many* accelerator devices. Additionally, it is unclear how well APIs like cuBLAS and cuDNN are supported, which are increasingly popular libraries in modern applications. Lastly, the latest supported CUDA version by rCUDA is 9.0, which is unsuitable for current machine learning frameworks.

Recent follow-up work [26] to rCUDA applies rCUDA to build an accelerated serverless computing platform. While similar to our work, it focuses on the architectural aspects of building a cloud computing system. The downsides of rCUDA remain: It is concerned with providing heterogeneous access to many accelerators in a cluster and much less with finer-grained sharing of a device at a smaller scale.

Also, rCUDA is not concerned about providing platform for hot-invocations of machine learning applications. Overheads are relatively large for every function invocation due to model loading costs. It would be desirable to support keeping models hot and initialized in a GPU environment to achieve much lower hot-invocation latencies.

A major problem of rCUDA is its nature as proprietary software, limiting its further research and deployment. It is therefore not evaluated further in this work.

3.1.3 Pagoda

Pagoda [47] is a system for efficiently sharing a GPU among many "narrow" tasks, i.e., tasks using fewer than 512 threads. Pagoda implements a full runtime with a scheduling system written entirely in CUDA kernels. Especially for workloads that spawn more than 32 tasks (NVIDIAs HyperQ [10] system supports 32 work queues), Pagoda can improve device utilization

significantly and claims to achieve a geometric mean speedup of 1.76X over HyperQ.

The downside of the Pagoda system is that it *only* supports narrow tasks and lacks genericity. Additionally, applications need to be specifically adapted to work with the Pagoda system, so libraries and applications must be ported to the Pagoda runtime system.

3.2 CUDA Device sharing in Clouds

3.2.1 KNIX

X et al. present an evolution [40] of the SAND/KNIX platform [8], which was renamed to KNIX. KNIX can execute Python functions in sandboxes using NVIDIA GPU resources. Isolation is achieved using GPU Manager [17], a component of the GaiaGPU project, which is based on the `LD_PRELOAD` API interception technique. Containers are provided with vGPUs (virtual GPUs) that can be shared between containers and enables control over memory and compute ability allocation.

KNIX allows sharing of GPUs between containers without modifications to Kubernetes code or container images and provides isolated allocations for functions. However, reported overheads are still significant compared to bare-metal execution.

The major problem for machine learning inference tasks is that in a serverless setting, latency can become prohibitive due to having to spin up container instances for every function execution. This is not fully addressed in KNIX, and approaches like rCUDA that do not require expensive environment setup are dismissed on the grounds of "lacking fault tolerance" [40, Section 3.2].

3.2.2 GaiaGPU

GaiaGPU [17] is a project that virtualizes GPUs for shared use by multiple containers, allowing flexible allocation of memory and compute resources. The system achieves very low overhead overall and significantly improves resource allocation.

GaiaGPU is implemented as a device plugin for Kubernetes. The GPU virtualization is realized using a `LD_PRELOAD` preload library that acts as a middleware between applications making CUDA API calls and the actual device.

The downside of GaiaGPU is that it is unclear if it can compete with the potentially much stronger properties that NVIDIA MIG can provide, such as complete memory bandwidth isolation. In exchange, there is more flexibility.

Additionally, GaiaGPU does not concern itself with the concept of function execution specifically, being more generic.

3.2.3 GPU Enabled Serverless Computing Framework

Kim et al. present a computing framework for serverless [24], that is based on attaching GPUs to an open source function execution environment. GPUs are used individually by each execution container, using the NVIDIA Docker Container Toolkit [28]. This does not address the core challenges for GPU-enabled FaaS services: sharing devices efficiently and providing isolation for tenants.

3.2.4 GPU-Enabled Serverless Workflows for Efficient Multimedia Processing

Risco et al. present a workflow for multimedia processing in serverless workflows [39]. Comparing to other works mentioned here, it is an entire system architecture based on the Amazon cloud platform. GPU acceleration is achieved using Amazon EC2 instances with the AWS Batch system.

3.3 Applications of NVIDIA MIG

3.3.1 Serving DNN Models with Multi-Instance GPUs

"Serving DNN Models with Multi-Instance GPUs" studies approaches to configure MIG partitions for serving DNNs such that throughput is maximized. The authors define this problem as the *Reconfigurabe Machine Scheduling Problem (RMS)* and propose an algorithmic solution. They report that their algorithm pipeline can save up to 40% GPUs while providing the same throughput. The work focuses on fulfilling service-level agreements (SLOs) in production deployments while optimizing throughput at the same time. Our work differs from this in that we focus on being able to execute arbitrary GPU functions on MIG partitions of client-requested size, taking the role of a service provider.

3.3.2 Contention-Aware GPU Partitioning and Task-to-Partition Allocation for Real-Time Workloads

"Contention-Aware GPU Partitioning and Task-to-Partition Allocation for Real-Time Workloads" [48] is an application of NVIDIAs MIG technology for solving timing constraints of modern real-time applications. It gives an algorithm that decides how to partition a GPU for use by different kernels such that interference between the different kernels is reduced.

3.3.3 NVIDIA Triton

NVIDIA Triton inference server is an NVIDIA product for serving machine learning models in production environments. It supports all major machine learning frameworks and can be used in various deployment settings. From a research perspective, the downside of this system is its focus on production systems: It only serves models from a pre-configured model repository. GPUless aims to support generic CUDA functions.

Chapter 4

Design

This chapter will discuss the goals (Section 4.1), an introduction to traced execution of CUDA (Section 4.2), and an overview of the design (Section 4.3) of the GPUless system.

4.1 Goals

We set the following goals for our GPU-native runtime for FaaS:

- Ability to share current generation GPUs (Ampere and later) among multiple tenants using hardware-level isolation and sharing mechanisms
- Strong performance and security isolation without operating system level sandboxing.
- Transparent API that doesn't require modifications to existing CUDA applications.
- Improve efficiency by allocation only GPU tasks and avoiding blocking the GPU when functions use the CPU for preprocessing

4.2 Trace execution

A fundamental problem in previous CUDA API middleware implementations like DS-CUDA [23] is that every CUDA API call is forwarded, which can result in tens of thousands of invocations over the network. This design leads to a large amount of network overhead and is nontrivial to implement if the exact asynchronous properties of CUDA are supposed to be retained. In GPUless, we will extend on the basic concepts of CUDA middleware implementations by introducing the concept of aggregated trace execution.

```
1 cudaMalloc(...) // sync
2 cudaMalloc(...) // sync
3 cudaMemcpy(hostToDevice, ...)
4 cudaMalloc(...) // sync
5 cudaMemcpy(hostToDevice, ...)
6 cudaLaunchKernel(...)
7 // repeat cudaLaunchKernel 23 times
8 cudaLaunchKernel(...)
9 cudaMemcpy(deviceToHost, ...) // sync
```

Listing 4.1: Trace of hotspot benchmark

The key idea is that not every API call has to be executed immediately to execute a CUDA application. It is possible to define "synchronization points", where execution has to happen because data is written back to the host. API calls between those points can be recorded, and executed only when required. This works because only some CUDA API calls have effects that can influence the flow of the host program, namely those that write back result data. For an example, see listing 4.1 that is an execution trace of the *hotspot* thermal simulation benchmark from the Rodinia [12] suite. There are two CUDA API calls that will write back data to the host: `cudaMalloc` (writes back a pointer to the allocated address), and `cudaMemcpy` (writes back data to host memory when using the device-to-host variant). Note that especially the kernel launches do not write back data, so all of these calls can be recorded and submitted to the execution environment only when synchronizing on the last `cudaMemcpy`. In this example, only 4/31 API calls actually require synchronization.

Trace analysis of machine learning inference application shows that the number of such synchronization points is low: In a typical resnet50 [19] image recognition inference task, only about 3% of API calls need to be synchronized. Other machine learning tasks like BERT [14] or 3D U-Net [49] also range from 3–6% synchronized calls. For more details on synchronization statistics, see Table 6.3 in chapter 6.

For an illustration, see the histogram of synchronization points across the execution timeline in our system with the amount of data transferred in Figure 4.1. happen can see that most synchronizations happen at the start, where memory is allocated, and at the end, where memory is copied back from the device to retrieve computation results. For synchronization histograms off all benchmarks evaluated in chapter 6, see appendix A.1.

Of interest is also the *size* of these aggregated trace executions. Were synchronization points spread uniformly, the "package-size" of a synchronization would still relatively small. Figure 4.2 shows an empirical cumulative distribution function (ECDF) plot of the size of the trace at synchronization

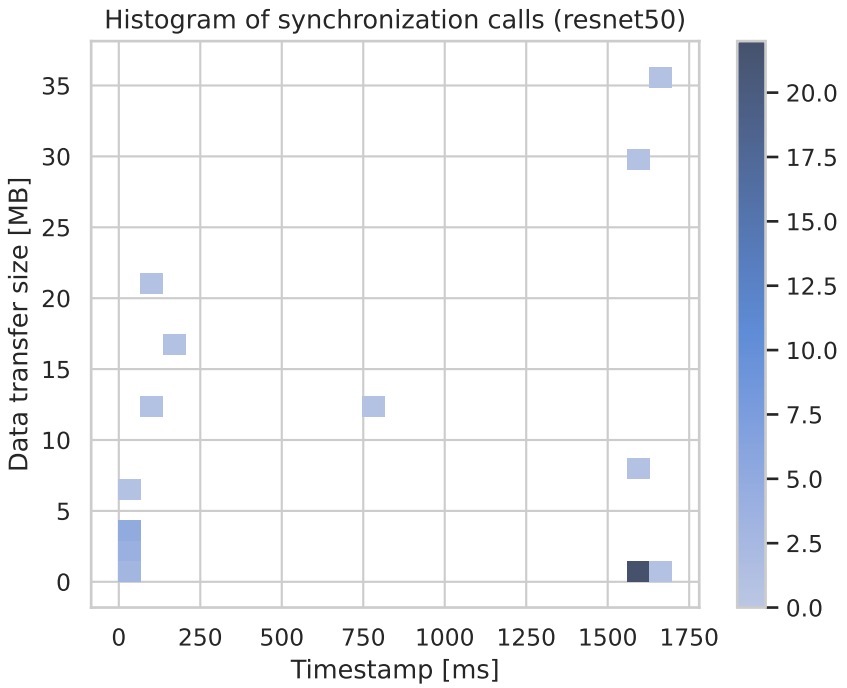


Figure 4.1: Histogram of synchronization calls in resnet50 image recognition inference (cold start)

for a ResNet50 invocation. We see that there are a large number of smaller synchronizations, with about 80% being smaller than 50 calls. However, there is also a relatively long tail of larger sychronizations with the largest containing more than 350 calls.

Note that return values of CUDA API calls (a status flag of `cudaError_t`) cannot reasonably be used to drive host computation. Firstly, they mostly indicate failures that cannot be recovered from (such as failure to initialize the CUDA driver), and secondly, errors reported by return value often do not originate from that specific call due to CUDAs asynchronous nature.

For a more detailed description of synchronization requirements of CUDA, cuDNN, and cuBLAS, see Chapter 5.

4.3 Design

Figure 4.3 gives an overview of the system.

1. A resource request for a device of a certain size is made.
2. One or many trace execution requests are sent directly to a manager process that owns the device.

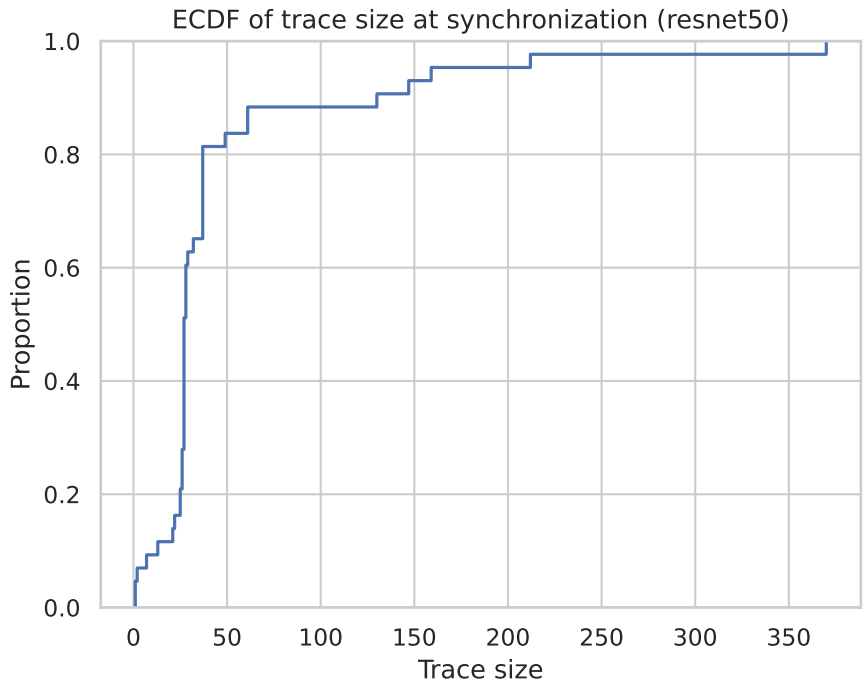


Figure 4.2: Histogram of trace sizes in synchronizations for resnet50 image recognition inference (cold start)

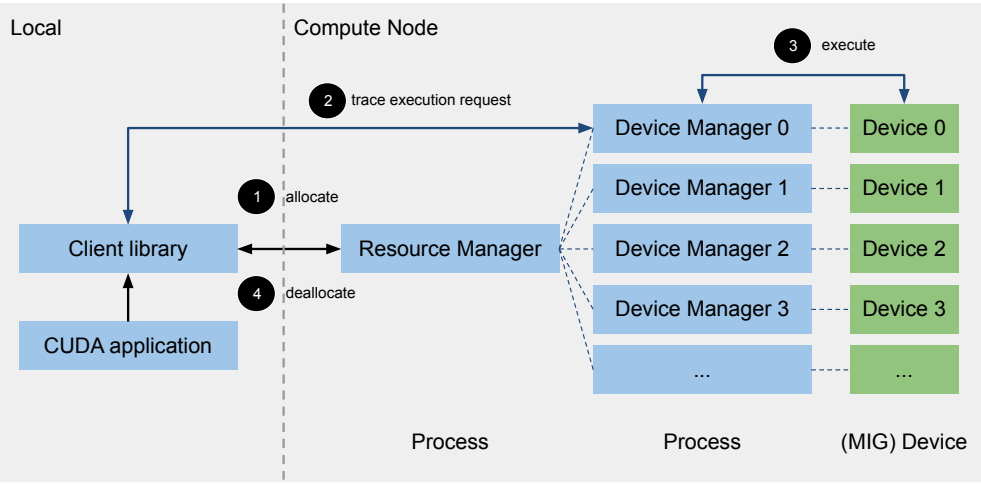


Figure 4.3: GPUless architecture overview

3. CUDA tasks/kernels are natively executed on the device.
4. And the end of the program, resources are deallocated.

Details of how these steps are designed will be be discussed in the following subsections.

Components. GPUless consists of the following major components:

- A client library that interfaces with CUDA applications.
- A server process that manages device partitions and allocations (resource manager).
- Secondary server tasks that manages each device partition and executes CUDA API traces on them (device manager).
- Protocols that enable communication between these components.

Client. There needs to be a generic way of integrating with existing code, because our client should be compatible with existing frameworks. While we could modify PyTorch source code directly to achieve our goals, it would not be portable to other frameworks and a substantial effort. We decide to use the `LD_PRELOAD` technique [6], which works by defining symbols of functions that should be intercepted and overridden and specifying the library in the `LD_PRELOAD` environment variable. It This approach has already been used in DS-CUDA [23] and KNIX [40] for intercepting the CUDA API. A downside of this approach is that it requires the CUDA library to be dynamically linked at compile-time, but in frameworks such as PyTorch, this is the default configuration.

Server. The server consists of two components, one resource manager and many device managers (see also Figure 4.3), which run the network protocols discussed next. Each device manager needs to be run as a separate OS process because CUDA manages its state on a per-process level. Figure 4.4 shows a comparative timeline between a native invocation and an invocation in the GPUless system. Both applications start with uninitialized CUDA devices to ensure a fair comparison, allowing us to measure our system's overheads accurately. Note that it is also possible to keep an OS process with an attached and initialized CUDA device ready in a pool. This approach is comparable to a "warm" start in serverless.

4.3.1 Resource allocation

For resource allocation the protocol shown in Figure 4.5. It follows the following basic steps:

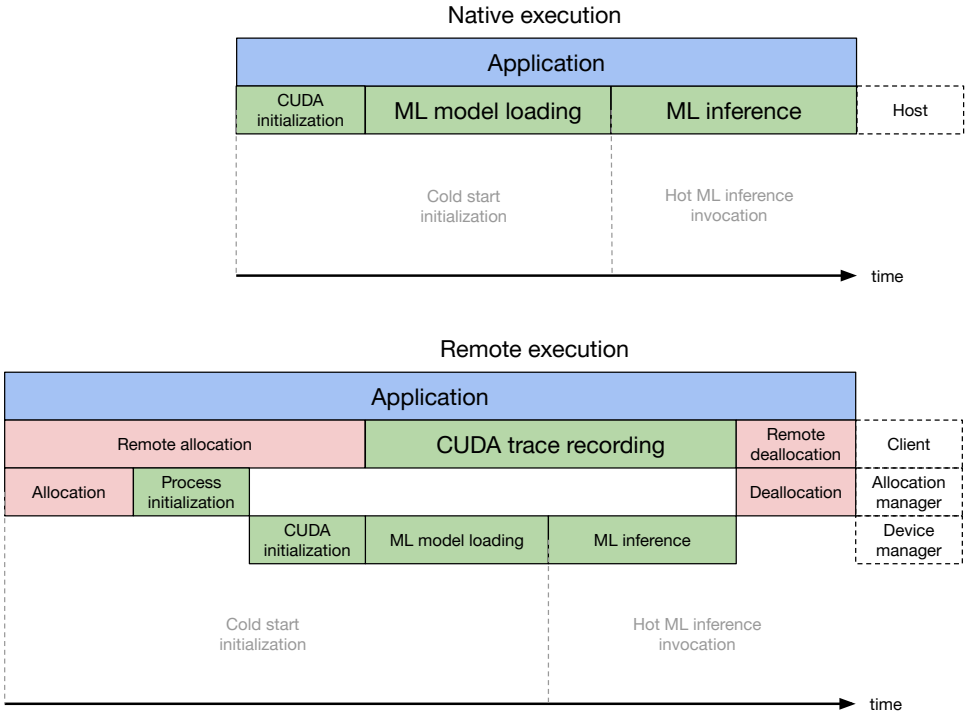


Figure 4.4: Timeline of execution in the GPUless system compared to a native execution

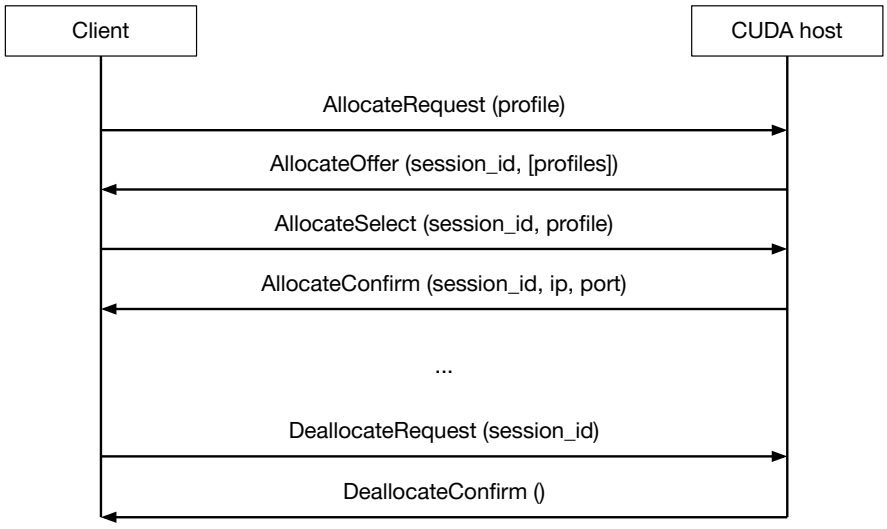


Figure 4.5: Allocation protocol

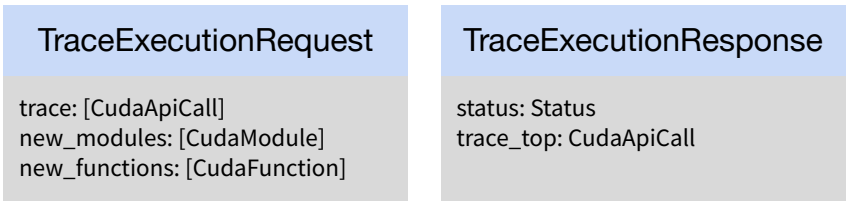


Figure 4.6: Structure of trace execution requests and responses

1. Request a session of a given profile. The profile is a MIG instance size.
2. The manager process answers with a list of available instances, according to the clients request.
3. The client can then select a suitable profile for the list.
4. The manager confirms the allocation, and issues a port and IP where the client can submit trace execution requests.

Additionally, a deallocation request is available for terminating the session before the timeout.

Protocol. To execute CUDA API traces on a remote host, the CUDA kernels and their arguments contained in that trace need to be packaged and transferred to it. Additionally, the execution service also needs the GPU device code. These requirements are implemented with a simple request-response protocol illustrated in Figure 4.6.

An execution request contains the trace of CUDA API call up until the current synchronization point. It also has to contain the modules (binary device code) and functions (symbol names) that should be loaded to execute that trace. Typically, a CUDA application will call the same kernel many times. By tracking what module code and functions were already transmitted previously at the client, bandwidth can be saved compared to deciding whether a module must be loaded at the server.

The `CudaApiCall` structures contain all necessary data to execute it on a native device, and if data is written back, it is stored too. The response will only include the most recent `CudaApiCall` because this is the synchronization point. It has return data required by the client, while all previous calls do not.

4.4 Summary

In summary, this design requires implementations for the components: Server processes, communication protocols, and `LD_PRELOAD` interception libraries

with trace execution capabilities.

Chapter 5

Implementation

Our prototype system is implemented in about 5000 lines of C++. It consists of a client library that is used as a `LD_PRELOAD` library and a server. Network serialization is handled using Flatbuffer [16] schemas, using standard TCP data transfers. An overview of the essential components is shown in figure 5.1. At the client-side, there are three API interception modules (see Section 5.1) for the main CUDA API targets: the CUDA runtime library, cuDNN, and cuBLAS.

There is an additional binary analysis module (see also Section 5.5) that provides insight on the types of kernel function parameters. The trace execution module supports the interception modules and handles serializing the traces and running the protocols described in Chapter 4.

On the server-side, there is two types of managers: A manager for allocating resources at the request of clients and device managers that will serve as execution node for traces. The device manager also holds its CUDA context, virtual handles for CUDA library structs (see Section 5.4), and registries for CUDA modules and functions that are referenced by the API calls in the traces.

In Section 5.2, the mechanism for submitting kernels is discussed. Section 5.3 shows the calls that require synchronization, Section 5.4 shows some optimizations that we applied, Section 5.5 discusses PTX analysis, and Section 5.6 shows an example for a system execution.

5.1 Library interception

At the core of the implementation lies the interception of library calls to CUDA, cuBLAS, and cuDNN. In its stage as a prototype, GPUless currently does not aim to provide full coverage of these APIs. For a complete list of supported APIs, see appendix A.3. Listing 5.1 shows an example of how

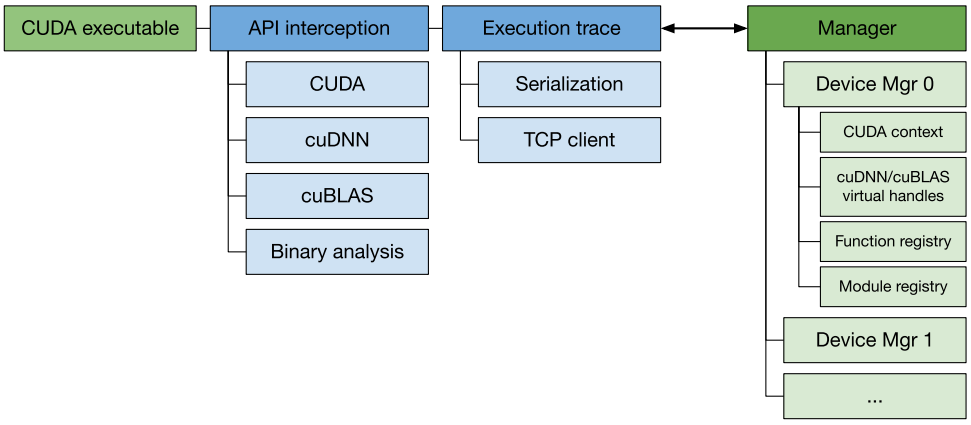


Figure 5.1: Architecture overview

such a library call interception is implemented, using `cudaMalloc` for an example. `cudaMalloc` is a synchronization point because a memory address is returned to the client, which is then commonly used as a base pointer for many memory copy operations. First, the call is stored in the current trace, and then synchronization is performed. The returned data required by the client (the pointer to device memory) is retrieved from the trace history and written back to the client using the out-parameter argument.

```
1  cudaError_t cudaMalloc(void **devPtr, size_t size) {
2      // store the call in the trace record
3      auto api_call = std::make_shared<CudaMalloc>(size);
4      getCudaTrace().record(api_call);
5
6      // perform a synchronization
7      getTraceExecutor()->synchronize(getCudaTrace());
8      auto top = getCudaTrace().historyTop();
9      auto api_call_executed
10         = std::static_pointer_cast<CudaMalloc>(top);
11
12     // write back data to the application
13     *devPtr = api_call_executed->devPtr;
14
15     return cudaSuccess;
16 }
```

Listing 5.1: Example CUDA API override

5.2 Kernel submission

An important aspect of a serverless system is submission of executable code. In the GPU context, this means we need to extract and transfer the GPU

```
typedef struct {
    int magic;
    int version;
    const unsigned long long* data;
    void *filename_or_fatbins;
} __fatBinC_Wrapper_t;
```

executable code to the execution host. To extract the code at runtime from a CUDA application, we intercept the `__cudaRegisterFatBinary` function from the CUDA runtime API, which registers binary modules embedded in the executable by the NVIDIA CUDA compiler.

Fat binary modules are of the following structure:

Note that this structure does not contain the size of the data, but it is embedded as header and can be extracted like this:

```
uint64_t *p = (uint64_t*) data;
size_t data_len = ((p[1] - 1) / 8 + 1) * 8 + 16;
```

Additionally, we hijack `__cudaRegisterFunction` which contains the fat binary handle where the function is contained. This is recorded, and once we intercept a `cudaLaunchKernel` call, we look up what module contains that function symbol and attach the module data for loading on the remote executor at the next synchronization.

5.3 Synchronization requirements

The GPUless project is implemented as a prototype that does not aim to achieve full coverage of CUDA, cuBLAS and cuDNN. To support our suite of benchmarks, only a small subset of theoretical synchronization points need to be supported:

```
cudaGetDeviceProperties
cudaFuncGetAttributes
cudaMalloc
cudaMemcpy(DeviceToHost)
cudaMemcpyAsync(DeviceToHost)
cudaFree
cudnnGetConvolutionForwardAlgorithm_v7
cudnnGetConvolutionBackwardDataAlgorithm_v7
cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize
cudnnGetBatchNormalizationTrainingExReserveSpaceSize
```

Some API calls do return data, but synchronization can be avoided. For more details on this optimization, see the following section.

5.4 Optimizations

Virtualized handles. Some API calls that return data do not require synchronizations, namely API calls that initialize handles to structures in cuBLAS and cuDNN such as `cudnnCreate`. Because the implementation details of these structures are private in the NVIDIA APIs, applications only use their handles as arguments to API calls but never access their data directly. This opaqueness allows us to virtualize these handles, generating new virtual handles in the client library upon their creation and mapping them to real handles on the server-side. The creation of the following handles are virtualized this way:

```
cudnnHandle_t
cudnnTensorDescriptor_t
cudnnFilterDescriptor_t
cudnnConvolutionDescriptor_t
cublasHandle_t
cublasLtHandle_t
cublasLtMatmulDesc_t
cublasLtMatrixLayout_t
```

Using the example of `cudnnCreate`, handle virtualization is implemented like this:

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle) {
    auto virt_handle = nextCudnnHandle();
    *handle = reinterpret_cast<cudnnHandle_t>(virt_handle);
    getCudaTrace().record(
        std::make_shared<gpuless::CudnnCreate>(virt_handle));
    return CUDNN_STATUS_SUCCESS;
}
```

Lazy module loading. When using the CUDA runtime API, the CUDA compiler will package kernels into *modules*, that contain kernel *functions*. For each of these modules and functions, the CUDA compiler will inject a call to `__cudaRegisterFatBinary` or `__cudaRegisterFunction`. Libraries like PyTorch include tens of thousands of kernel functions, so significant time is spent registering these modules and functions during initialization. In the GPUless system, we intercept these calls but do not forward the registration calls. Instead, we keep track of the module data and the symbols located within them, and once a kernel launch call is registered, the module is shipped with the next synchronization if it is not available at the server yet. Benchmark evaluation(see chapter 6) shows that this optimization can even lead to a performance gain compared to native execution.

5.5 PTX analysis

A major challenge when serializing a call to a kernel function is that all of its arguments need to be shipped as well. The CUDA API for launching a kernel (`cudaLaunchKernel`) does not include any information about the types of the arguments or even how many there are. The CUDA provides no public API for querying the arguments of a kernel function at runtime. To resolve this problem, we let the user provide our system with the binary that contains the CUDA code and analyze it using the CUDA binary utilities (`cuobjdump`). While this can be an expensive operation, we only need to extract the list of parameters and their sizes. The results can be cached easily, as libraries such as PyTorch are not updated frequently.

5.6 Example execution

To illustrate how our prototype system can be used, we show how a Python machine learning application (such as the example in listing 5.2, an image classification invocation on ResNet50) can be launched in our system. Arguments are given using environment variables, and most importantly our client library is preloaded using the `LD_PRELOAD` environment variable. We note that the core part of PyTorch interface has not been changed:

```
MANAGER_IP=127.0.0.1                        \
MANAGER_PORT=8002                           \
CUDA_BINARY=/usr/libtorch/libtorch_cuda.so  \
LD_PRELOAD=~/libgpuless.so                  \
python run.py
```

```
import torch
from PIL import Image
from torchvision import transforms

model = torch.hub.load('pytorch/vision:v0.10.0',
            'resnet50', pretrained=True)
model.eval()

input_image = Image.open('dog.jpg')
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]),
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0)
input_batch = input_batch.to('cuda')
model.to('cuda')

with torch.no_grad():
    output = model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)
top_prob, top_catid = torch.topk(probabilities, 1)
top_catid = top_catid[0].item()
top_prob = top_prob[0].item()
```

Listing 5.2: Sample python application machine learning application (run.py)

Chapter 6

Evaluation

This chapter will evaluate the GPUless system against a set of benchmarks. We chose a selection of applications from the Rodinia [12] benchmark suite for scientific computing, and machine learning application from the MLperf [37] benchmark suite and reference models from PyTorch hub [33]. The evaluation is split into hot and cold invocations. A cold invocation means that no data is initialized prior to execution, the client-server system has not negotiated an allocation yet, and the GPU devices are uninitialized at the server. We also verify NVIDIAs claims about MIG and show that indeed MIG can provide sound performance isolation.

6.1 Evaluation platform

All GPU executions, unless stated otherwise, are executed on a node equipped with an AMD EPYC 7742 64-Core Processor @ 2.25 GHz, 512 GB of main memory, and 4 NVIDIA A100 GPUs with 40 GB of GPU memory. This node is used for both native execution benchmarks and for running the GPUless server. In benchmarks that use the GPUless client to run applications remotely on the GPUless server, a node with an AMD EPYC 7742 @ 2.25GHz and 512 GB of memory are used. The two nodes are interconnected with a 100 Gbit network interface. There is no switch between nodes, and TCP/IP ping-pong latency is about 15 μsec.

6.2 MIG performance isolation

To evaluate MIG performance isolation, we use the STREAM [13] benchmark. Figure 6.1 shows the performance implications of running the STREAM benchmark concurrently in MIG compute instances, which are not supposed to provide performance isolation. We use the largest available MIG GPU-instance partition (spanning the whole device) and configure it with a 1+3+3

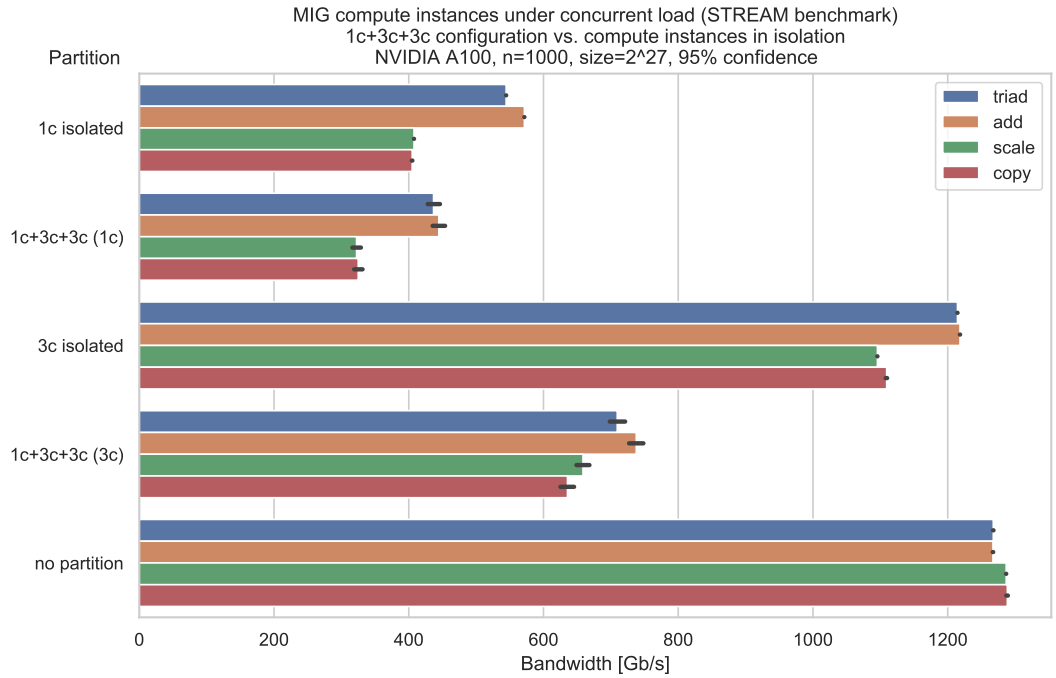


Figure 6.1: Performance isolation for compute instances in MIG under STREAM benchmark

compute instance configuration. For example, the "1c isolated" row shows throughput running only a single benchmark in a 1c compute instance, while "1c+3c+3c (1c)" shows the STREAM throughput in a 1c compute instance, while STREAM is run concurrently in the other two instances. Throughput under concurrent load is significantly reduced when sharing the device using compute instances. We conclude there is no bandwidth isolation when executing in compute instances.

Figure 6.2 shows the throughput in setting where the device is shared with GPU instances instead of compute instances, which is supposed to provide complete performance isolation. Measurements confirm that this holds to some degree, but we see a slight degradation in performance under concurrent load, especially in the largest (4g) instance. We conclude that GPU instance do provide some bandwidth isolation, but there is cases where there is still measurable impact of running applications concurrently.

6.3 Scientific Computing: Rodinia

For an evaluation of more traditional CUDA applications from the scientific computing domain, we evaluate a selection of benchmarks from the Rodinia [12] suite in the domains of simulations, graph algorithms, and linear algebra.

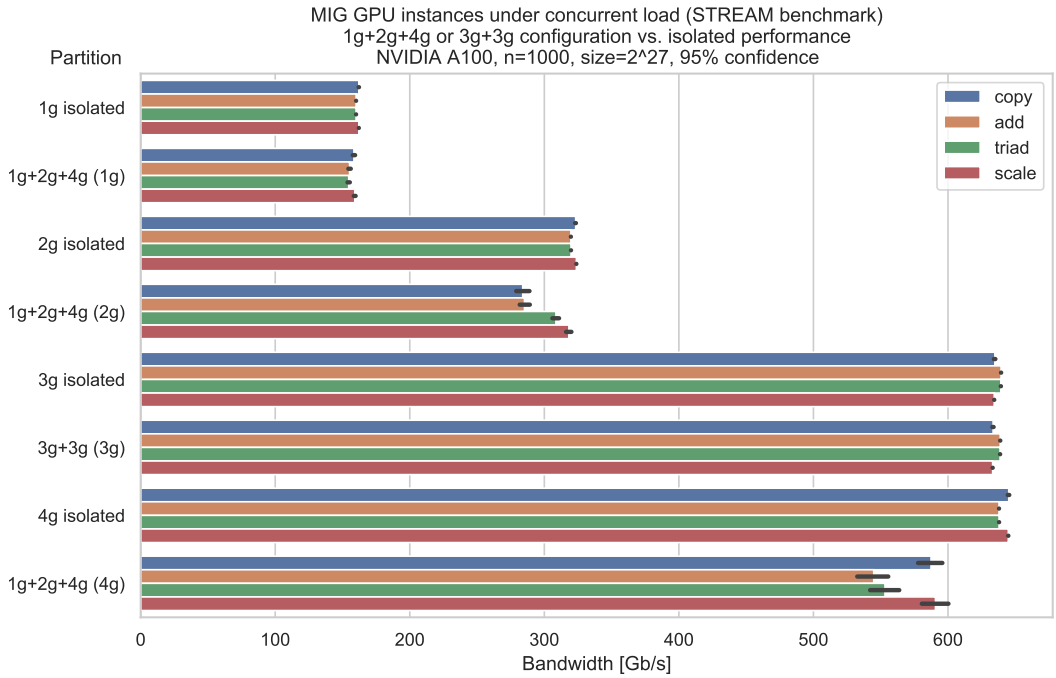


Figure 6.2: Performance isolation for GPU instances in MIG under STREAM benchmark

6.3.1 Benchmarks

- **bfs**. An implementation of breadth-first search (BFS) for traversing a graph.
- **hotspot**. HotSpot is a thermal simulation application that estimates processor temperature based on an architectural floorplan of a microchip.
- **pathfinder**. PathFinder is a dynamic programming application that finds the path in a matrix from the bottom row to the top row with the smallest accumulated weights.
- **srad_v1**. Speckle Reducing Anisotropic Diffusion (SRAD) is an image processing application that is used to remove locally correlated noise in ultrasonic and radar imaging.
- **gaussian**. Gaussian is an implementation of the Gaussian elimination algorithm for solving systems of equations.
- **myocyte**. Myocyte is a biological simulation application that models a cardiac myocyte (heart muscle cell).

For statistics on the number of kernel launches, unique kernels, memory allocations, and copies, see Table 6.1.

Benchmark	Kernel launches	Unique kernels	cudaMalloc	H2D	D2H
bfs	24	2	7	18	13
hotspot	25	1	3	2	1
pathfinder	5	1	3	2	1
srad_v1	502	6	12	5	201
gaussian	2046	2	3	3	3
myocyte	3900	1	4	7800	7800

Table 6.1: Statistics on CUDA API usage of Rodinia benchmarks

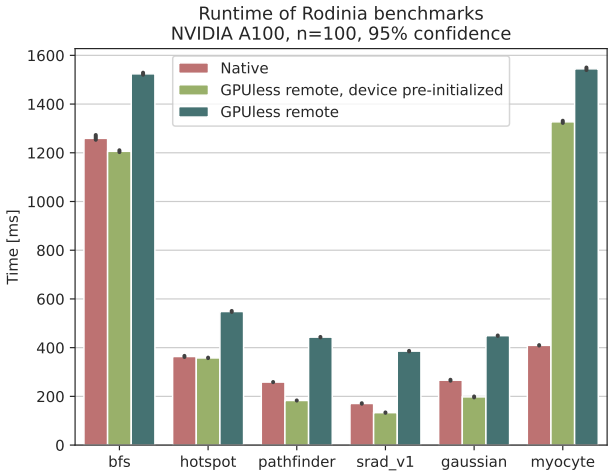


Figure 6.3: GPUless execution time vs. native in Rodinia benchmark

6.3.2 Results

Figure 6.3 shows the runtime of the Rodinia benchmarks. We see a performance of our system that is generally close to native performance. If CUDA devices are pre-initialized before being allocated for this task, we see a slight gain in performance due to the relatively short benchmarks. The one outlier is the *myoyte* application, which has much-reduced performance due to its many device-to-host memory copies that cause expensive synchronizations.

6.4 Machine Learning inference

For machine learning inference, we evaluate a selection of benchmarks included in the *mlperf* [37] suite and pre-trained models publicly available on PyTorch Hub [33].

6.4.1 Benchmarks

- **alexnet [25]**. AlexNet is a model for image recognition that won the ILSVRC 2012 task.
- **vgg19 [41]**. vgg-nets is a convolutional network for image recognition that makes use of a very large network depth, placing 1^st in the ILSVRC 2014 task.
- **resnet50 [19]**. ResNet is a residual learning framework for image recognition that won the 1^{st} place on the ILSVRC 2015 classification task.
- **resnext50, resnext101 [46]**. ResNeXt is an evolution of the ResNet framework that placed 2^{nd} in the ILSVRC 2016 classification task.
- **yolop [45]**. YOLOP (You Only Loop Once for Panoptic Driving Perception) is a driving perception system that performs traffic object detection, drivable area segmentation and lane detection simultaneously.
- **MiDaS [36, 35]**. MiDaS is model for monocular depth estimation based on vision transformers.
- **3d-unet-kits19**. 3D-UNet [49] is a model for 3D image segmentation, performing the KiTS 2019 kidney tumor segmentation task [20], used in the *mlperf* benchmark.
- **BERT-SQuAD**. BERT [14] (Bidirectional Encoder Representations from Transformers) is a language representation model that can be fine-tuned for a wide range of tasks. Our benchmark is configured for the SQuAD [34] question-answering task.

For statistics on the number of kernel launches, unique kernels, memory allocations, and copies, see Table 6.2. For reference histograms of synchronizations and CDFs of trace sizes at sychronization, see Appendix A.1 and A.2.

6.4.2 Results

Cold invocation. Figure 6.4 shows the runtime of machine learning inference benchmarks of native execution, compared to execution in the GPUless system for cold execution, where nothing is in cache, and the GPU is not initialized. The entire model is loaded, and an inference invocation is made. The image recognition models perform well, with GPUless being slightly faster than native due to optimizations discussed in chapter 5. Models with longer run times (3d-unet-kits19, BERT-SQuAD) do not benefit from these optimizations as much, and they tend to have more overhead because more data is transferred (see Table 6.3).

Benchmark	Kernel launches	Unique kernels	cudaMalloc	H2D	D2H
resnet50	219	27	17	321	2
resnext50	458	24	12	321	2
resnext101	1435	21	27	627	2
alexnet	38	12	5	17	2
vgg19	96	16	12	39	2
yolop	762	63	111	546	247
midas	660	38	96	369	1
3d-unet-kits19	6850	17	22	82	2
BERT-SQuAD	714	22	71	397	3

Table 6.2: Statistics on CUDA API usage of machine learning benchmarks

Benchmark	API calls	Syncs	Syncs %	Up [MB]	Down [MB]
resnet50	2086	43	2.0%	180.64	0.019
resnext50	2083	39	1.8%	178.59	0.020
resnext101	4047	49	1.2%	434.36	0.017
alexnet	165	13	7.8%	311.23	0.006
vgg19	431	24	5.5%	641.57	0.010
yolop	4644	398	8.5%	339.93	0.067
MiDaS	2284	119	5.2%	1509.73	7.525
3d-unet-kits19	39314	2248	5.7%	1689.99	679.760
BERT-SQuAD	1974	75	3.7%	1491.03	0.011

Table 6.3: Statistics on synchronization

Hot invocation. Figure 6.5 shows the latency of one ML inference call, comparing native execution to the GPUless system, FaaS+GPUless, and AWS Lambda. FaaS+GPUless is a setup where a basic HTTP server takes invocation requests and executes them using a GPUless remote execution. Note that AWS Lambda does not support GPU execution, so it will naturally be much slower than running on a GPU (AWS Lambda benchmarks are executed on CPU only). Native executions are always faster, but GPUless is still quite close. In the case of FaaS+GPUless, we see quite a bit of overhead introduced due to the use of the HTTP protocol, but especially for longer running benchmarks (yolop, MiDaS, BERT-SQuAD), this constant overhead is of much less importance.

Bandwidth limiting. Figures 6.6 and 6.7 show the influence of the available bandwidth on the execution latency, for both cold and hot executions. This evaluation was run on a workstation computer using an Ampere generation GPU (RTX 3060), an Intel i7-9700KF CPU, and 32GB of main memory. Both server and client are run on the same host, using the loopback network

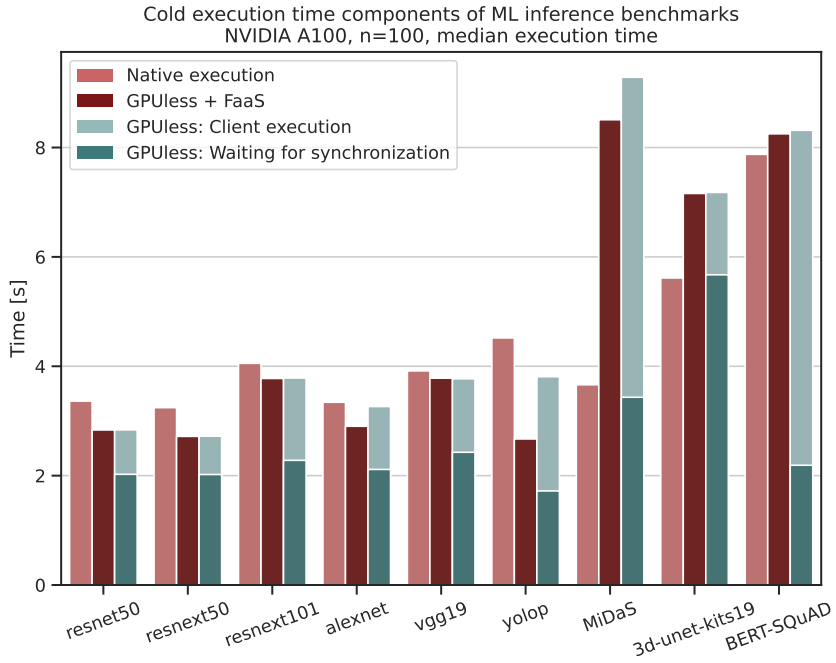


Figure 6.4: Cold execution time for machine learning benchmarks

interface. The bandwidth is limited by limiting the loopback interface to a specific bandwidth. As expected, performance degrades with reduced bandwidth. In general, we can say that to retain good performance compared to native execution, 1-10 Gbps of bandwidth should be available.

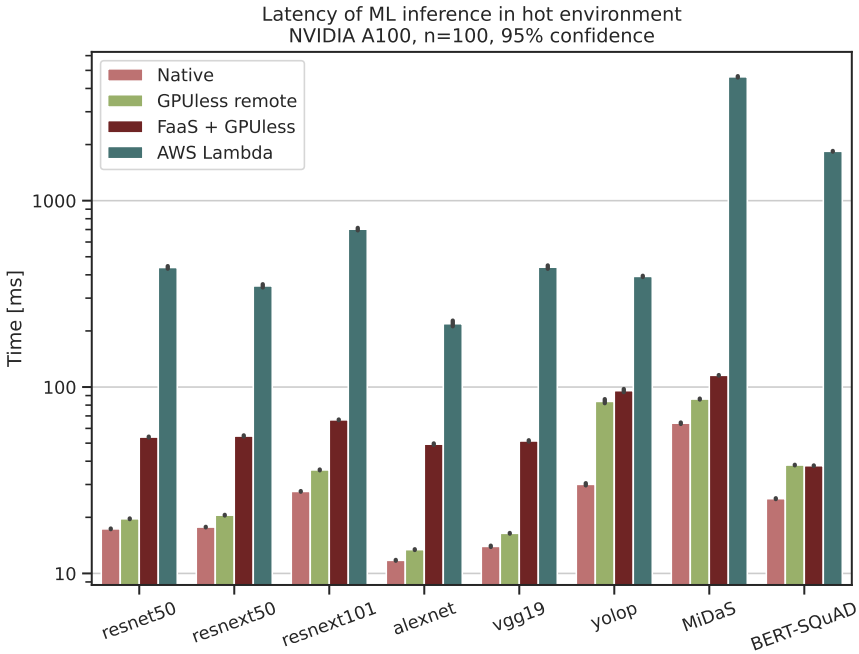


Figure 6.5: Hot execution time for machine learning bechmarks

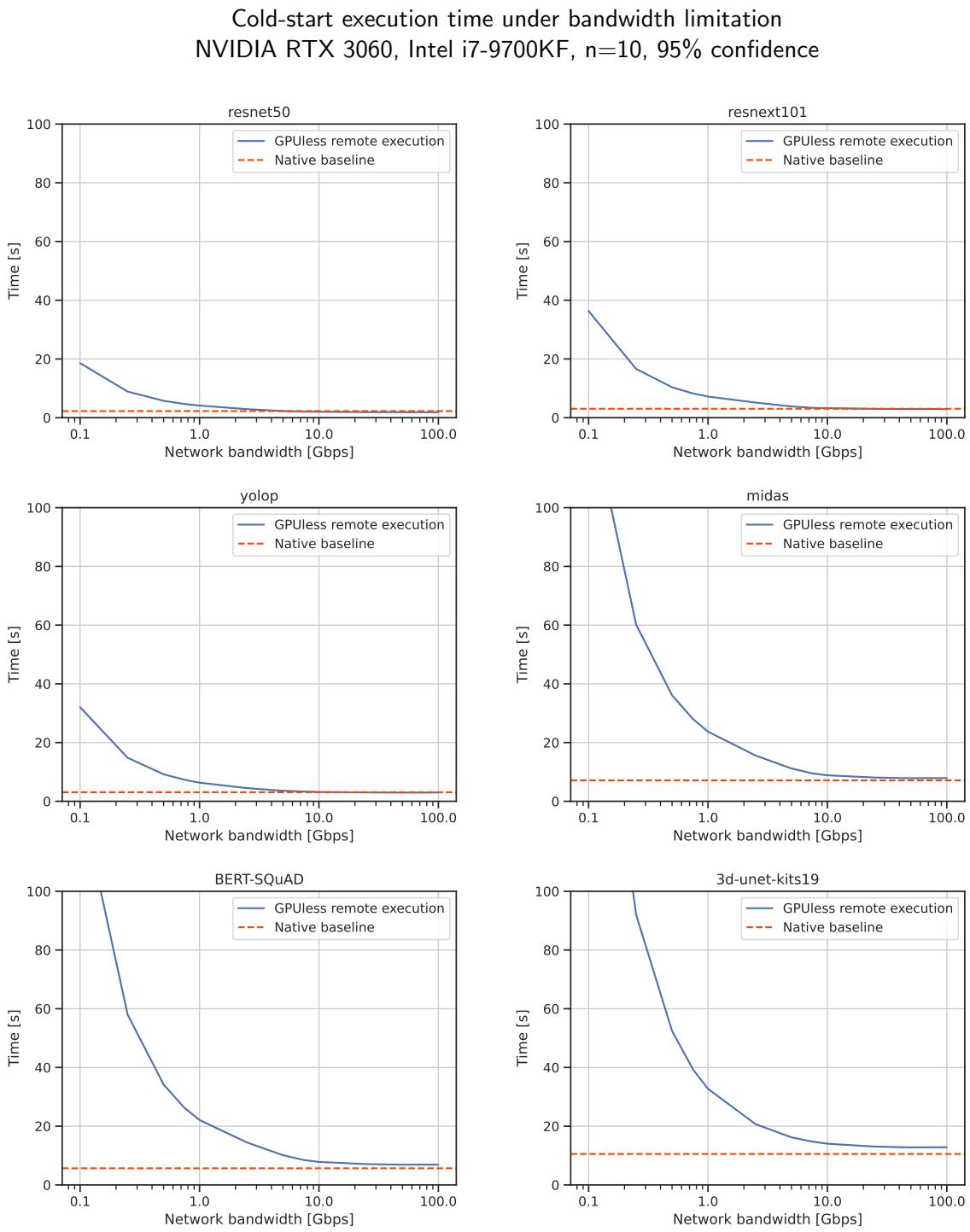


Figure 6.6: Influence of network bandwidth on cold-start performance

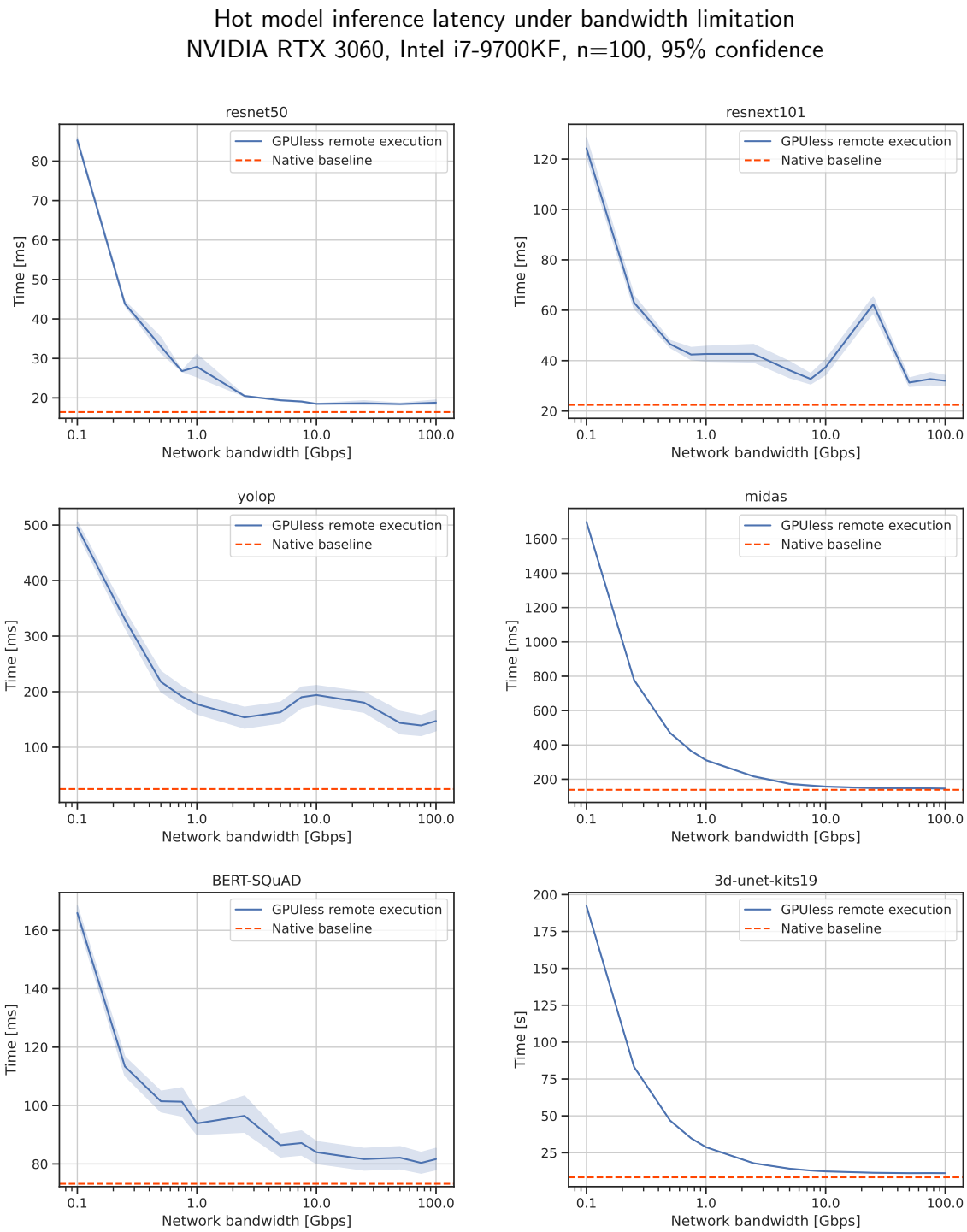


Figure 6.7: Influence of network bandwidth on hot model inference performance

Chapter 7

Conclusions and Future Work

In this work, we presented GPUless, a prototype system for GPU-native function execution. We introduce traced execution, a new approach to transporting CUDA API calls over the network where execution traces are collected and executed in the aggregated form at specific synchronization points. Our prototype supports a subset of CUDA, cuDNN, and cuBLAS, which allows us to evaluate the system against the Rodinia benchmark suite and select machine learning applications. Due to some optimizations applied in the implementation, we see good cold-start performance for machine learning benchmarks compared to native execution. For hot invocations, the performance penalty is more significant but not unreasonable. We also analyzed the bandwidth requirements of our system, and conclude that a network with a bandwidth of at least 1 Gbps is required.

Opportunistic execution. Functions tend to be regular, i.e., their execution structure often does not change between invocations. In such cases, traces could be stored on the FaaS executor without the client transmitting the full transcript for every invocation. This optimization would be optimistic, as we assume the kernel structure for the function does not change. Synchronizations could be limited to host-to-device and device-to-host memory copy operations that carry new data. Correctness could be verified by a mechanism that checks that the recorded structure agrees with client execution.

Warming data. A problem with GPU function execution is that containers cannot be kept "warm", as is commonly down in commercial FaaS systems, where containers will reside in memory for some time after an invocation. This is more challenging with GPUs because every tenant will need a separate process with an attached GPU device to achieve isolation. When such a process is created and the device initialized, the device is reset fully, leaving no data behind. However, it could be possible to use main memory (RAM)

storage to offload the state from the GPU and later re-instantiate it. If we could determine what parts of the application stay constant and what parts depend on user input, the constant data could be cached in the main memory between invocations. This scheme could benefit applications with high initialization costs but much smaller costs for repeated invocations with different user data, such as machine learning applications. These applications incur a high cost for loading the models, but inference operations have a much shorter run time and require much fewer data transferred.

Function format. Based on the success of the ideas in the previous two paragraphs, it could be possible to achieve the goal of true cloud-based GPU functions. A classic FaaS function consists of code (typically Python/-JavaScript), libraries, and data loaded into main memory. For GPU functions, a function definition could consist of the kernels (i.e., PTX code), the CUDA API trace, synchronization points, and constant data that is loaded to GPU memory.

Implementation improvements. A problem in the prototype implementation is that only a small fraction of the CUDA, cuBLAS, and cuDNN libraries are covered. The entire API would have to be covered to support any application fully. An approach to achieve full coverage with high degrees of correctness could be to generate the code from an API specification automatically. This would be a significant software engineering problem, but likely would not require additional much more additional insight.

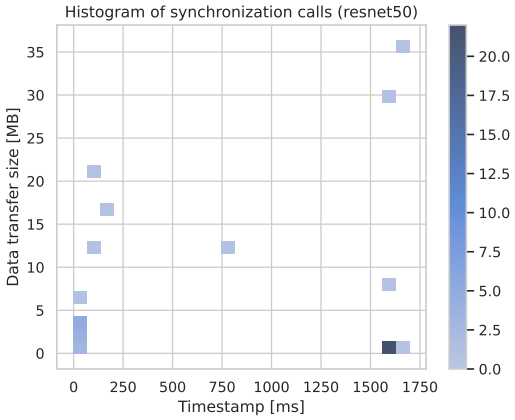
Currently, the selection of a MIG instance size is left entirely to the client. However, previous work has investigated how to optimize throughput by more advanced scheduling mechanisms [42]. Such scheduling algorithms could potentially also be applied to our system.

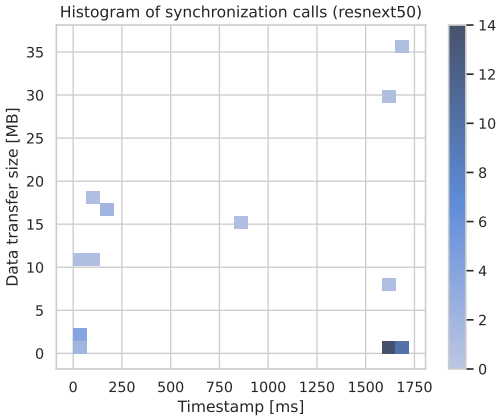
Our prototype implementation is based on basic TCP transport protocols. However, it would be possible to accelerate network operations by using asynchronous RMDA transport that provides higher bandwidth. A further option could be to support NVIDIAs GPU-Direct technology, which allows using RDMA for direct access to GPU memory.

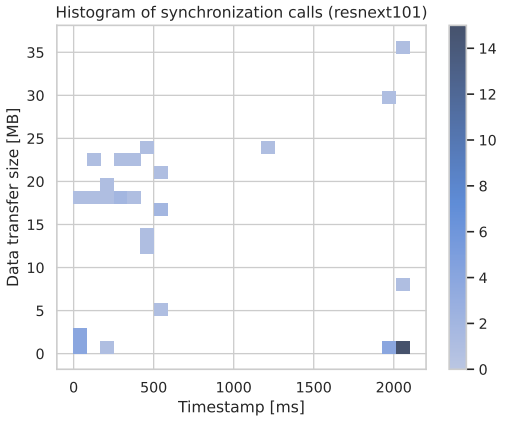
Appendix A

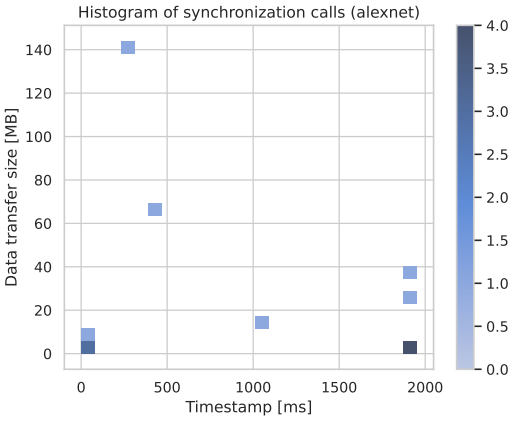
Appendix

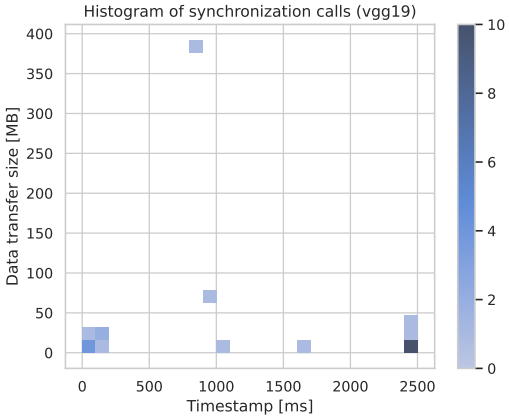
A.1 Synchronization histograms (vs. transfer size)

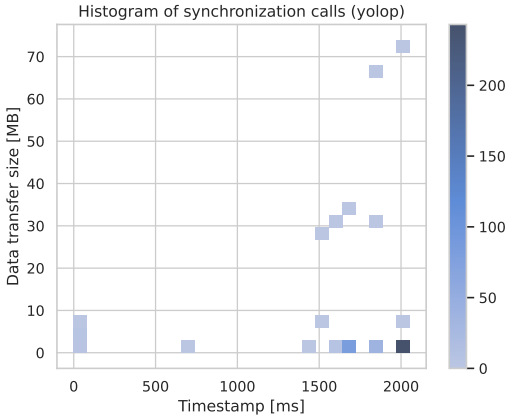


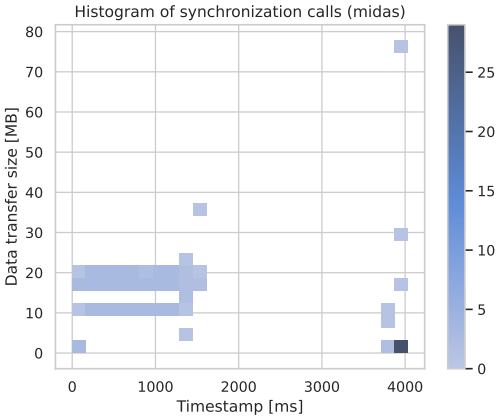


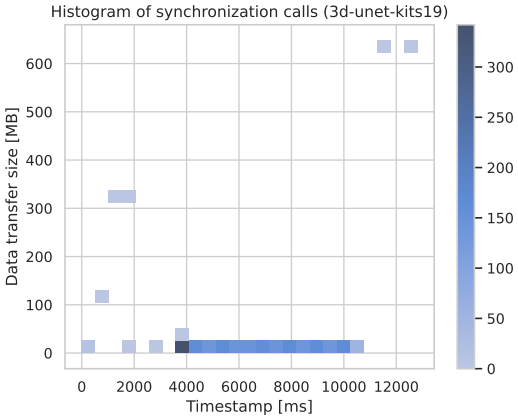


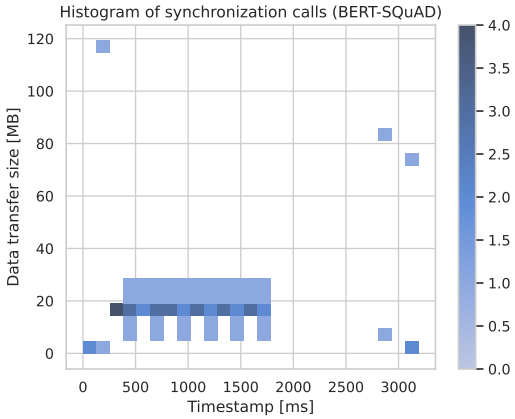



Histogram of synchronization calls (vgg19)
Data transfer size [MB]
Timestamp [ms]

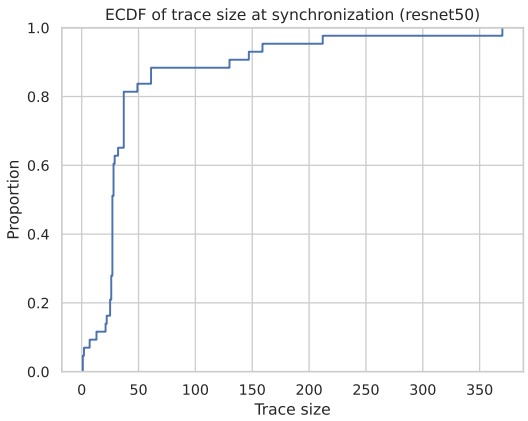

Histogram of synchronization calls (yolop)
Data transfer size [MB]
Timestamp [ms]

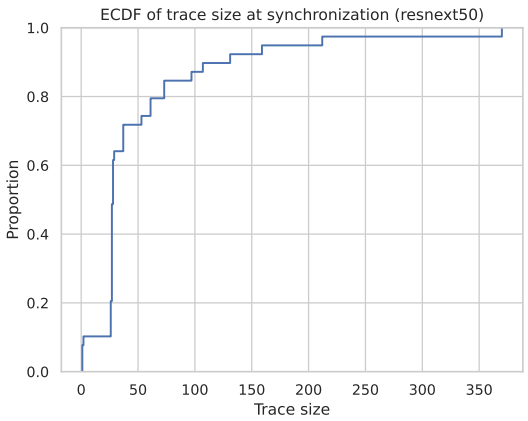

Histogram of synchronization calls (midas)
Data transfer size [MB]
Timestamp [ms]

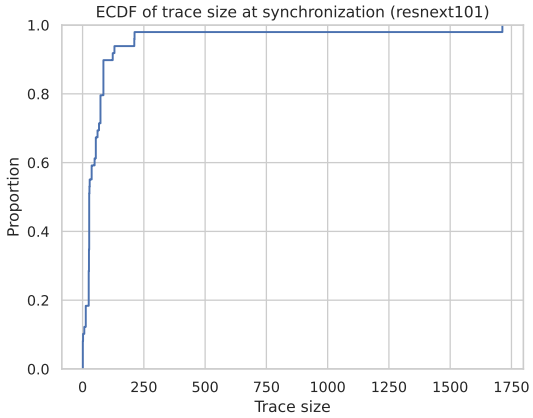

Histogram of synchronization calls (3d-unet-kits19)
Data transfer size [MB]
Timestamp [ms]

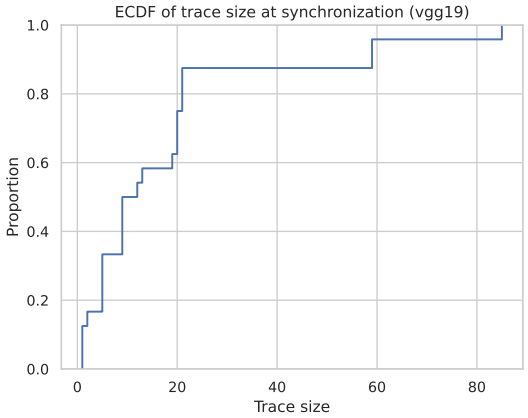

Histogram of synchronization calls (BERT-SQuAD)
Data transfer size [MB]
Timestamp [ms]

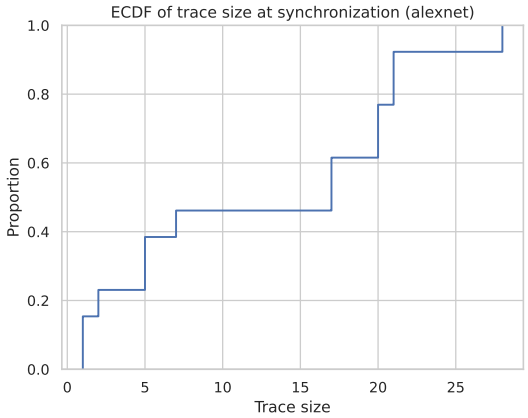
A.2 Synchronization trace size ECDF

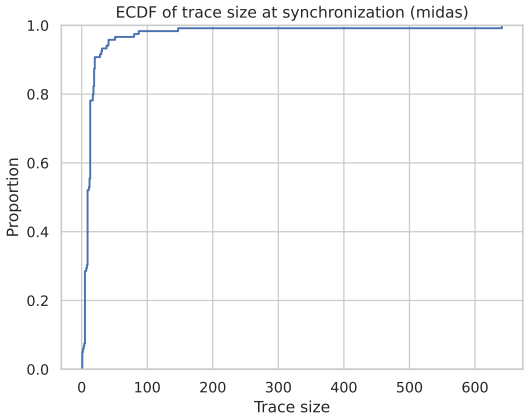


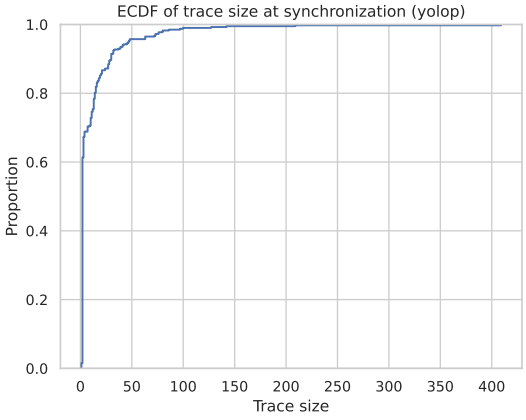


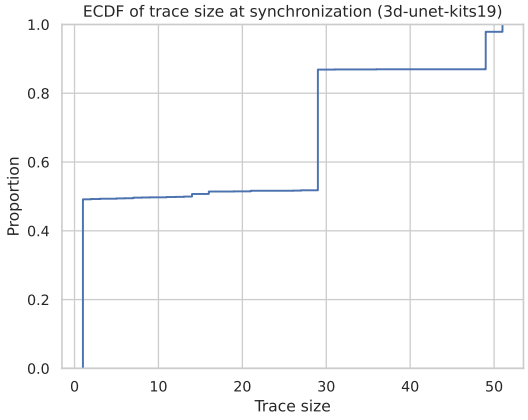


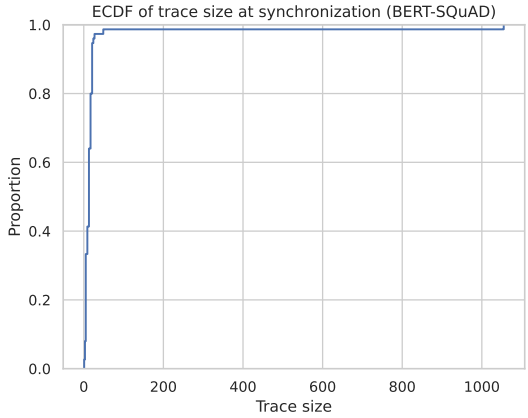







ECDF of trace size at synchronization (yolop)
Proportion
Trace size
0.0
0.2
0.4
0.6
0.8
1.0
0
50
100
150
200
250
300
350
400


ECDF of trace size at synchronization (3d-unet-kits19)
Proportion
Trace size
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50


ECDF of trace size at synchronization (BERT-SQuAD)
Proportion
Trace size
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000

A.3 CUDA API coverage

```
cudaMalloc
cudaMemcpy
cudaMemcpyAsync
cudaLaunchKernel
cudaFree
cudaStreamSynchronize
cudaThreadSynchronize
cudaDeviceSynchronize
cudaStreamCreateWithFlags
cudaStreamIsCapturing
cudaGetDevice
cudaSetDevice
cudaGetDeviceCount
cudaGetDeviceProperties
cudaGetLastError
__cudaPushCallConfiguration
__cudaPopCallConfiguration
__cudaRegisterFatBinary
__cudaRegisterFatBinaryEnd
__cudaRegisterFunction
__cudaRegisterVar
__cudaUnregisterFatBinary
cudnnCreate
cudnnSetStream
cudnnCreateTensorDescriptor
cudnnSetTensorNdDescriptor
cudnnCreateFilterDescriptor
cudnnSetFilterNdDescriptor
cudnnCreateConvolutionDescriptor
cudnnSetConvolutionGroupCount
cudnnSetConvolutionMathType
cudnnSetConvolutionNdDescriptor
cudnnGetConvolutionForwardAlgorithm_v7
cudnnConvolutionForward
cudnnConvolutionBackwardData
cudnnGetConvolutionBackwardDataAlgorithm_v7
cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize
cudnnGetBatchNormalizationTrainingExReserveSpaceSize
cudnnBatchNormalizationForwardInference
cudnnBatchNormalizationForwardTrainingEx
cudnnDestroyConvolutionDescriptor
cudnnDestroyFilterDescriptor
cudnnDestroyTensorDescriptor
cublasCreate_v2
cublasLtCreate
cublasSetStream_v2
cublasSetMathMode
cublasSgemm_v2
cublasLtMatmulDescCreate
cublasLtMatmulDescDestroy
cublasLtMatmulDescSetAttribute
cublasLtMatmul
```

```
cublasLtMatrixLayoutCreate
cublasLtMatrixLayoutDestroy
cublasLtMatrixLayoutSetAttribute
cublasSgemmStridedBatched
```

Bibliography

[1] Apache OpenWhisk. https://openwhisk.apache.org/. Accessed: 2022-01-19.

[2] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2022-01-19.

[3] Google Cloud Functions. https://cloud.google.com/functions/. Accessed: 2022-01-19.

[4] IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed: 2022-01-19.

[5] AWS Lambda. https://aws.amazon.com/de/lambda, 2014. Accessed: 2022-01-19.

[6] *ld.so (8) Linux User's Manual*, 5.13 edition, 2021. Accessed: 2022-01-19.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow, large-scale machine learning on heterogeneous systems, 11 2015.

[8] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards

high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[9] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[10] Thomas Bradley. Hyper-Q Example. https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2013. Accessed: 2021-12-11.

[11] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, nov 2019.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[13] Ben Cumming. Stream benchmark in cuda c++. https://github.com/bcumming/cuda-stream.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[15] Jose Duato, Antonio J. Pena, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. IEEE, June 2010.

[16] Google. Flatbuffers. https://google.github.io/flatbuffers.

[17] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. Gaiagpu: Sharing gpus in container clouds. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 469–476, 2018.

[18] Tsuyoshi Hamada, Rio Yokota, Keigo Nitadori, Tetsu Narumi, Kenji Yasuoka, and Makoto Taiji. 42 tflops hierarchical n-body simulations on

gpus with applications in both astrophysics and turbulence. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 11 2009.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[20] Nicholas Heller, Niranjan Sathianathen, Arveen Kalapara, Edward Walczak, Keenan Moore, Heather Kaluzniak, Joel Rosenberg, Paul Blake, Zachary Rengel, Makinna Oestreich, et al. The kits19 challenge data: 300 kidney tumor cases with clinical context, ct semantic segmentations, and surgical outcomes. *arXiv preprint arXiv:1904.00445*, 2019.

[21] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform, 2018.

[22] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. *Proceedings of the 2021 International Conference on Management of Data*, Jun 2021.

[23] A. Kawai, Kenji Yasuoka, K. Yoshikawa, and Tetsu Narumi. Distributed-shared cuda: Virtualization of large-scale gpu systems for programmability and reliability. *The Fourth International Conference on Future Computational Technologies and Applications*, pages 7–12, 01 2012.

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

[25] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014.

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

[27] NVIDIA. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2021-12-18.

[28] NVIDIA. NVIDIA Container Toolkit. https://github.com/NVIDIA/nvidia-docker. Accessed: 2022-01-19.

[29] NVIDIA. NVIDIA cuBLAS Documentation. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2022-01-05.

[30] NVIDIA. NVIDIA cuDNN Documentation. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html. Accessed: 2022-01-05.

[31] NVIDIA. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. Accessed: 2021-12-18.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[33] PyTorch. PyTorch Hub. https://pytorch.org/hub/. Accessed: 2022-01-02.

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

[35] René Ranftl, Alexey Bochkovskiy, and Vladlen Koltun. Vision transformers for dense prediction. *ArXiv preprint*, 2021.

[36] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020.

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

[38] Steve Rennich. CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed: 2021-12-18.

[39] Sebastián Risco and Germán Moltó. GPU-enabled serverless workflows for efficient multimedia processing. *Applied Sciences*, 11(4):1438, February 2021.

[40] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. Efficient GPU sharing for serverless workflows. ACM, June 2020.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[42] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem, 2021.

[43] Rick Villars, Holly Muscolino, Wayne Kurtzman, Serge Findling, Ritu Jyoti, Dan Vesset, Mario Morales, Jennifer Cooke, Deepak Mohan, Jonathan Lang, Al Gillen, Mickey North Rizza, Carrie MacGillivray, and Ashish Nadkarni. Idc futurescape: Worldwide it industry 2021 predictions. 2020.

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[45] Dong Wu, Manwen Liao, Weitian Zhang, and Xinggang Wang. Yolop: You only look once for panoptic driving perception, 2021.

[46] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017.

[47] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. Pagoda. 6(4):1–23, December 2019.

[48] Houssam-Eddine Zahaf, Ignacio Sanudo Olmedo, Jayati Singh, Nicola Capodieci, and Sebastien Faucou. Contention-aware gpu partitioning and task-to-partition allocation for real-time workloads, 2021.

[49] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 3d u-net: Learning dense volumetric segmentation from sparse annotation, 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

GPUless - Serverless GPUless functions

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Tobler	Lukas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Wil SG, 15.01.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.