PyPlover: A System for GPU-enabled Serverless Instances

Ryan Yang, Nathan Pemberton, Jichan Chung

Abstract

Demand for GPUs has grown exponentially since the onset of machine learning workloads. However, the cost of an efficient GPU still remains very high. For a machine without a GPU, one solution is to send the GPU workload to a dedicated cluster of GPU-enabled instances for processing. However, without the proper knowledge, this method turns out to be very inefficient due to improper load balancing and instance tuning. We propose PyPlover, a serverless GPU framework that allows the user to send kernels and inputs to a serverless provider without needing to worry about set-up costs and load balancing.

1 Introduction

Serverless computing is a cloud-computing execution model that enables users to deploy applications in the form of a function, without the necessity of a server. The main advantage of the service is that it enables fine-grained control of pricing by allowing the user to pay for the actual amount of resources they use for the task, rather than using pre-purchased units of capacity. Another advantage is that the user does not have to worry about configuring the server, making it easy to deploy and scale up. The framework is also beneficial for the cloud service provider by allowing it to maximize utilization using load balancing, thanks to fine-grained resource allocation. Due to these advantages, Cloud providers (e.g., AWS Lambda, Google Cloud Functions) and open source projects (e.g., OpenLambda [4]) have developed an infrastructure for serverless computing, and many applications are built on top of these services (e.g., PyWren[3]) as a way to utilize these low-cost and scalable compute resources.

Currently, these publicly available services only provide CPU access. With the rise of Deep Learning applications, the necessity for large-scale computations utilizing other types of accelerators such as GPUs have emerged. Considering the advantages of serverless computing, it is natural to extend this framework for these applications. One implementation of this idea is to attach a GPU to the already existing Lambda container [1]. While the container based service has an advantage in that the user can easily deploy applications that can be run on the local machine, the main shortcoming of this approach is the overhead due to software dependencies being packed into the container. This makes execution time bigger and makes it hard to deploy it large-scale. Also, the container based approach requires a whole GPU to be assigned to an application, limiting the fine-grained control and multi-process access. Moreover, in most modern GPU workloads, CPU usage is limited to data loading, memory allocation, and invoking the GPU, leaving most of the time idle. In these workloads, providing options to use GPUs without CPU allocation can reduce the price for the user and can improve machine utilization for the provider. Cloud providers are also not constrained to supporting a diverse range of customers for our design. Rather, they are given the freedom to design and deploy the GPU server however they want.

In this project, we propose a new serverless framework that allows users to execute their GPU applications in the cloud. The user can request execution of their application in the form of kernels.

To summarize, our contributions are as follows:

- Designed generic function signature that can accomodate most existing GPU applications.
- Implemented server and client programs for the serverless GPU framework.

- Ported vector addition operation and a Convolutional Neural Network to be run in our framework and measured the performance.

2 Background and Motivation

2.1 Serverless

Serverless has become somewhat of a buzzword among the systems community when discussing running large workloads at scale. The most popular serverless frameworks as of right now include AWS Lambda and Google Cloud Functions. Serverless providers hide much of the gritty set-up of dedicated instances behind a veil, allowing the user to directly send requests to provider without having to worry much about setting up an instance for running these requests. In addition, having users share a single serverless provider allows a greater usage of the resource hidden behind the provider. Sharing hardware behind a single serverless provider enables greater resource usage since all aspects of the hardware can be used on demand as user requests come in.

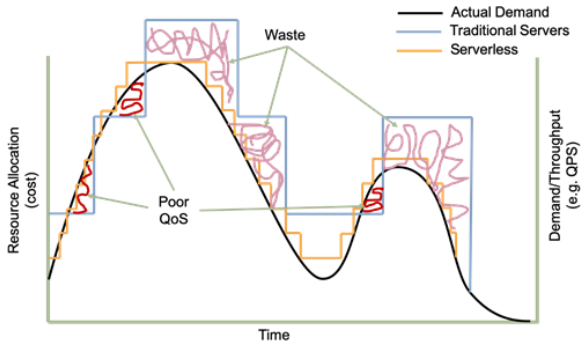

Figure 1: Graph of how traditional servers allocate resources versus how serverless allocates resources. Serverless architecture allows for much more fine-grained allocation

Another benefit of serverless benefits both the client and the provider: by having requests done in more fine-grained chunks, the provider is able to more exactly match the demand given by its users. This concept is illustrated in Figure 1. When the resource usage falls below the demand line, this gives its users low quality of service. When the resource usage goes above the demand line, this means that the provider is wasting resources by allocating too much for the current task. Hence, the solution to this problem would be to allocate resources in smaller chunks to more exactly match the demand line. This is exactly what serverless providers do when they divide the resources that they own into containerized instances and automatically load balance requests between them.

2.2 PyWren

PyWren [3] is a system developed by exposing a Python map primitive on top of AWS Lambda. By doing this, it allows the user to easily run embarrassingly parallel functions using AWS Lambda, making distributed computing very easy. However, PyWren is a very general framework in that it does not target any specific kind of workload. Hence, due to this generality, its performance across all workloads is suboptimal. We build on top of the PyWren framework by specifying that we are targeting GPU workloads, therefore optimizing our code specifically for these kinds of workloads. By removing the generality constraint of PyWren, we are effectively focusing specifically on GPUs, which allow us to strip away some of the restrictions that PyWren put on the inputs. But by restricting the inputs only to GPU workloads, we introduce more constraints that will be discussed in a later section

2.3 Motivation

Based on current serverless designs, PyPlover would enable a user to run GPU workloads on a machine without any GPUs attached by sending requests to a serverless provider. This provider would automatically load balance these requests to the appropriate containers and return the result of the GPU execution once it is finished. Figure 1 illustrates the resource allocation benefits of using serverless vs. serverful approaches. Rather than allocating resources in larger chunks, as with serverful,

serverless chooses to separate these resources into smaller chunks and allocate them per user demand.

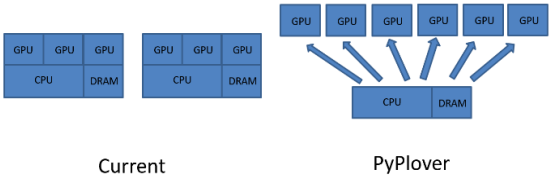

Figure 2: Illustration of traditional methods of processing GPU workloads on a non-GPU machine versus the PyPlover method of dealing with this issue. The current method allocates a whole instance, whereas PyPlover abstracts small containers behind a serverless provider

Figure 2 illustrates the traditional way of running a GPU workload vs. our proposed method of running a GPU workload. Currently, a user would allocate some GPU-enabled instances on a remote server, such as AWS EC2, and would need to send multiple instances of data such as the GPU workload, the kernel functions, web servers, database clients, and anything else needed to get a server running. Much of this sent data is extraneous with the serverless approach. Once the workload has finished running, the chosen instance to run will return the result. However, this approach introduces some problems with regards to scaling, resource usage, and set-up costs. First off, the user is responsible for the set-up and management of these instances. This means that the user needs to be knowledgeable in methods to setting up an EC2 instance that is able to run GPU workloads efficiently and needs to be able to load balance requests to the cluster of instances correctly. These requirements are not something that the average user wanting to run a GPU workload possesses. Failing to meet these requirements will result in suboptimal performance on this cluster of dedicated instances. In particular, failing to load balance the requests between instances correctly will result in greatly reduced performance and under-utilization of total resources due to all of the requests being overloaded onto a single instance. In our design, this set-up and load balancing is abstracted away from the user. Therefore, the user does not need to possess knowledge in EC2 instance set-up and cluster management. All these benefits are also in addition to the finer-grained resource allocation discussed in the previous section.

3 Related Work

GPU attached to CPU Lambda. Prior work has been done on attaching a GPU to the conventional Lambda framework [1]. Our approach differs from this in that we completely decouple the GPU from the CPU, allowing the user to send GPU-only requests through the serverless provider. The GPU attached to Lambda approach poses a few downsides. By doing this, the provider is essentially doing the same thing as the traditional serverful approach, just on a smaller scale. By decoupling the GPU from the CPU, the provider is able to rent out CPU usage as well in addition to the GPU if the user is not using the CPU. However, if the user wants to use the CPU along with the GPU, with other approach, it is not possible.

Using CUDA Without Environment Setup. Prior work has been done on remote GPU acceleration of GPU-enabled workloads, similar to our project but without the serverless approach. To the user, nothing changes. The user is still able to make calls to CUDA through a CUDA driver. However, on the backend of the CUDA driver, it is actually sending CUDA requests over the network . One downside of this approach is the lack of fault tolerance. Through serverless, there is no need to worry about this issue since the request would just time out and the user can resubmit it. The authors called this special CUDA driver rCUDA [2].

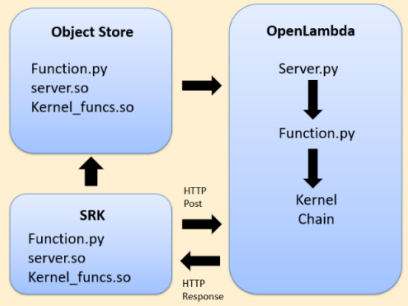

Figure 3: PyPlover Design. SRK uploads objects to the object store and sends data to OpenLambda. OpenLambda runs our server code to process the kernels and sends the result back to SRK, which returns it back to the user

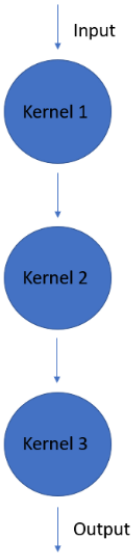

Figure 4: PyPlover Kernel Execution. A kernel chain in our system must process data in a chain (i.e output of previous kernel is input of next kernel)

4 PyPlover Overview

PyPlover enables users to run GPU workloads cheaper and more efficiently than if they were to run the same workload on a cluster of dedicated GPU-enabled EC2 clusters. To the best of our knowledge, there does not exist any major serverless provider that allows GPU workloads. Prior work has explored attaching GPUs to the current serverless instance framework [1]. Our work is different from this in that we our design does not have an attached CPU for each GPU core. On the one hand, this allows us to decouple GPU processes from CPU processes but also decreases the amount of flexibility we give to users when they run their kernel functions on the GPU. Another similar work [2] has explored remote GPU acceleration, using CUDA without any explicit environment set-up, which is extremely important for our project since it enables users to directly run CUDA code without setting up any environment on the serverless provider's end. Overall, our approach builds upon these two papers and overcomes several challenges that make it difficult to provide serverless GPU support.

4.1 Design

The overarching design for PyPlover starts by having a user upload a set of GPU kernel functions that follow a predefined function signature and execution structure. The user then uploads an input in JSON format, in which the serverless provider responds by allocating a container such that the load balance specifications are met and directly running the uploaded GPU functions. The output of these GPU functions are then returned to the user through HTTP. We have separated these tasks into server tasks and client tasks. Figure 3 illustrates our design.

Server. On the server side we use OpenLambda [4], a project that can simulate a serverless provider by automatically load balancing requests to containers that it allocates. OpenLambda pulls the objects (the function that it runs and the shared objects containing the CUDA functions) from a predefined object store. It runs the function and the corresponding kernels provided to OpenLambda. After execution, the server code returns the corresponding output back to the user.

Client. The client side of PyPlover utilizes the Serverless Research Kit (SRK) [5], a project still under development, to abstract away the complexity of uploading objects to an object store and passing function invoke requests to the server side. Since this was already under development, we decided to use this project to reduce the complexity of implementing it ourselves.

4.2 Challenges and Contributions

We identified three main challenges to implementing serverless GPUs: (1) Identifying a general function signature that is able to encompass all kernel functions and all kernel inputs. (2) Determining the execution model for a set of kernels on the server side. (3) Setting the amount of flexibility to give to the user.

4.2.1 General Function Signature

One of the challenges we encountered while designing our framework was identifying a general

function signature that is able to encompass all kernel functions and kernel inputs. As of the current CUDA version, the most complex aspect of GPU functions remains to be tuning the dimensions of the CUDA kernel to be optimal for the user's purposes. Therefore, we ensured that the user was able to specify the dimensions of each CUDA kernel he defines. The general structure we ended up using for the inputs to the CUDA kernel is discussed in Section 5.

4.2.2 Execution Model

On the server side of PyPlover, the order in which we execute the kernels and the restrictions we set on execution strategy was a challenge to determine. Contemporary functions that call CUDA kernels are able to perform a variety of different execution strategies. A large part of our project was determining an execution model that is able to encompass a majority of these execution strategies for CUDA kernels. Section 4 discusses the model PyPlover uses.

5 Client

5.1 Input Structure

```
struct state_t {
  buf_t input
  buf_t constant
  buf_t tmp
  buf_t out
}
```

```
struct buf_t {
  uint8_t* dat
  size_t len
}
```

Figure 5: State and buffer structures. All kernel functions that our system run take in a state object

The input structure design was one of the largest challenges in this project. To the best of our knowledge, there have not been studies on a generic data structure that can be used as the sole input to a CUDA kernel. Therefore, we had to design one based on the requirements that needed to be met. This input structure was designed with the structure of the kernel execution discussed in 5.2 in mind. The buf_t struct is the generic buffer object that our kernels consume. It consists of a pointer to a byte

array allocated on the GPU (since byte arrays can be casted to anything in C) and the size of the byte array, sometimes needed to copy and iterate through the byte array. The wrapper around the buf_t struct, state_t, consists of four buffers: input, constant, tmp, and out.

- Input: Populated per kernel chain invocation with the input of the kernel chain
- Tmp: utilized by the user for any temporary data storage that needs to happen between kernel function calls
- Constant: holds any static data that is populated with the strategy discussed in 5.1.3
- Out: holds the final output of the kernel chain and is returned to the user

5.2 Kernel Structure

```
typedef void (*kernelFp) (state_t *s, int, int);
```

Figure 6: Function signature for each kernel function. We had each kernel function take in as input a state object and the dimensions for the kernel.

```
typedef void (*kernelFp) (state_t *s, int, int);

kernelFp* init(const char* so_path, char** function_names,
               state_t* state, int num_funcs, size_t tmp_len,
               size_t out_len)

state_t* init_func(uint8_t* constants, size_t constant_len)

buf_t* run(kernelFp* function_ptrs, buf_t *input, state_t* state,
           int *block_sizes, int *grid_sizes, int num_funcs)
```

Figure 7: The functions that are contained in the internal execution of the server code. Load static is not included since it is written as a python function in the python script run on the server end.

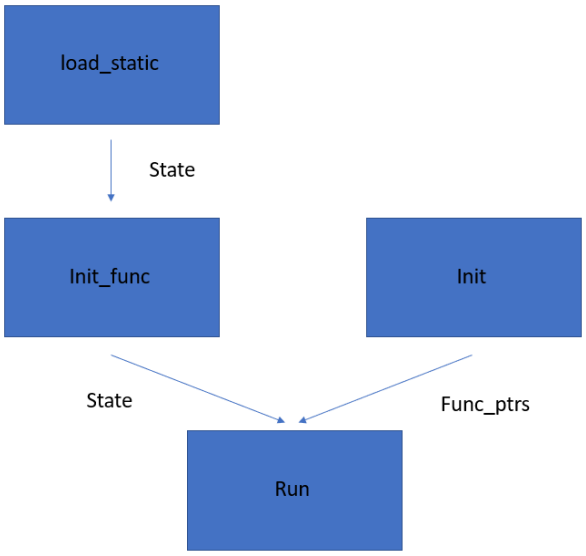

Figure 8: The execution pipeline of the PyPlover server code.

The kernel takes in a single object, a state that is populated with the required fields. Here, we gave the user an enormous amount of flexibility. We gave the user the freedom to serialize and deserialize the data however he wants. All the data in our state object is represented in byte arrays. The user is free to do whatever he wants with this byte data. As a result, this leads to PyPlover being able to support nearly any kernel input as long as the user is able to serialize it into byte data. The contents of the kernel can consist of anything as long as it does not return program control back to the CPU, since this destroys the idea of decoupling a GPU from a corresponding CPU.

5.3 SRK

The serverless research kit [5] is a core part of our project. It abstracts away the user input part of PyPlover, allowing us to mainly focus on the innovations, such as the code that the server runs and the design decisions, such as the input and kernel structures. SRK is able to take in objects from

the user and upload them to some object store that OpenLambda can pull from. In this case, we just chose to use our local filesystem as the object store. In addition, SRK is responsible for taking in input from the user and forwarding it to OpenLambda through an HTTP post request. SRK retrieves the result from OpenLambda by waiting for a corresponding HTTP response through the same port.

6 Server

6.1 Internal Execution Structure

The internal execution structure of PyPlover consists of four main functions: (1) init. (2) init_func. (3) load_static. (4) run. Init, init_func, and load_static pass inputs to each other in a systematic way, and their inputs are forwarded to run, which runs the specified kernel chain. Figure 5 illustrates the way in which these four functions are linked.

6.1.1 Init Implementation

The most important task of the init function is to load in the user defined shared object containing the CUDA kernels and parse the object for the given function names. To do this, init uses dlopen to open the shared object and retrieve a pointer to the object handler, and it uses dlsym to parse the object for the given function name to retrieve the symbol corresponding to that given function name. Init returns a list of pointers to the retrieved symbols in the user defined shared object.

6.1.2 Init_func Implementation

The init_func function is responsible for allocating memory on the GPU for the tmp and out buffers in the state object that the kernel functions operate on. The init_func function makes use of the cudaMalloc CUDA function to allocate memory on the GPU. But for this to work, init_func must take in as input the sizes of the tmp and out buffers that the user plans to allocate. This function returns a state object containing the allocated tmp and out buffers.

6.1.3 Load_static Implementation

The load_static function is responsible for loading in any static data that does not vary between function invocations. One example of this data would be weights in a deep neural network. Uniquely, this function is only called a single time upon kernel function creation, when the container with the kernel function is created. This function initially creates the state object, upon which it allocates space on the GPU for the static data and uses cudaMemset to copy the static data to the GPU. By separating static data from input data, a portion of time and resources is saved, since only a fraction of the total data the kernels take in as input needs to be copied over to the GPU. Load_static returns the state object containing the allocated constant buffer.

6.1.4 Run Implementation

The run function is responsible for running the kernel functions, given the state object and the pointers to the kernel functions to run. Run starts by allocating space for the input buffer on the GPU using cudaMalloc and copying the input data over to the GPU with cudaMemset. After the kernel chain returns, run fetches the contents of the out buffer containing the output of the kernel chain and returns it to the user.

6.2 Kernel Execution Structure

The structure in which the kernel runs holds some limitations, but, in general, it is very flexible about which types of kernel execution strategies that it can support. Kernel chains in PyPlover are restricted to run in a chain, in which one kernel is chained after the other without any CPU runtime between the kernels. This structure prevents the user from performing any CPU operations on the temporary data between kernels, but also optimizes runtime of the kernels by not allowing for any overhead of

memory changes between CPU and GPU. We feel that this tradeoff is necessary for an optimal runtime of a kernel chain on the GPU and to prevent any unexpected memory changes due to faulty CPU code running between kernels. In addition, this structure also supports the method of entirely decoupling a GPU from a CPU. Figure 4 illustrates the structure that a kernel chain must take.

7 Evaluation

At the beginning of our project, we defined our metrics of success as:

1. A total runtime on our system, of some kernel, that is comparable to the runtime on a regular GPU-enabled instance
2. A total resource consumption, of some kernel, that is comparable, if not better, than the resource consumption on a regular GPU-enabled instance

We were only able to evaluate metric 1 thoroughly through two different kernels. Metric 2 is unmeasured and is considered future work.

7.1 vecAdd Kernel Results

	SRK/Openλ	Local
Load Kernel	0.0015s	0.079s
Load Constants	0.0620s	
Run Kernel	0.0208s	
SRK/Openλ Overhead	2.8307s	
Total	2.915s	

Figure 9: Results for the first vecAdd experiment. This experiment simply took two vectors and parallelized their addition using CUDA.

The first kernel we tested our system on was the vecAdd kernel. This kernel is very simple and was essentially a test to see if our system worked correctly from end-to-end. The static data in this function is a vector consisting of n=1000000 elements. The input data in this function is a vector consisting of n=1000000 elements. For this test, we had the input vector equal the constant data for simplicity. The system was run locally on a machine with two Nvidia K20C GPUs and a single Intel Core i7-4765T CPU.

The results from this first experiment were interesting. We noticed that the overhead for this kernel was absolutely massive. Originally, we were under the impression that this overhead was the warm start, in which case the system could be considered infeasible due to the massive overhead involved. However, after looking into SRK further, we realized that SRK was shutting down the OpenLambda connection each time that the experiment finished. Therefore, this overhead is always the cold start overhead, which include the massive time to start up a container and perform all the start-up procedures. In addition, since the overhead is still quite large compared to the actual time needed to run the kernels, we decided to perform the same timing experiment on a more complex kernel chain, leading up to the CNN kernel example.

7.2 CNN Kernel Chain Results

	Cold	Warm	Hot	Local
Load Kernel	0.015ms	0.015ms	0.015ms	20ms
Load Constants	0.62ms	0ms	0ms	
Run Kernel	0.208ms	0.208ms	0.208ms	
SRK/OpenLambda Overhead	2154.9ms	93.3ms	0.2ms	
Total	2155.74ms	94.14ms	1.04ms	

Figure 10: Results for the CNN Kernel Chain Experiment. The local timing is greater than the hot start-up timing since local timing also includes some overheads that a hot start-up ignores.

Layer	Configuration
1	Convolutional layer with 6 filters with size (5,5), stride 1, no padding
2	Convolutional layer with 1 filter with size (4,4), stride 4, no padding
3	Fully connected layer with input size 6*6*6 and output size 10

Table 1: Architectural details of CNN. This CNN takes in an input from the MNIST dataset and returns a one-hot vector with the calculated label for the input.

The next experiment we performed was to take an already implemented CNN on CUDA and run the kernel chain specified in the code through our system. The experiment was performed on the same machine as the vecAdd experiment. This CNN takes input from the MNIST dataset and predicts a label for this input.

The architectural details of CNN implementation that we use is shown in detail in Table 1. We modify the original code to be run with state and buffer structure shown in Figure 5. In detail, we first declare each layer of the CNN as a function kernel to be run. When initializing kernels, the weight of each layer is loaded to the 'constant' buffer in function state. At runtime, the input image is copied to 'input' buffer and the layer kernels are run sequentially, writing output (activation) matrix in 'tmp' buffer to be read by the next layer's kernel invocation. When all the kernels finish running, the output activation are written in 'out' buffer of the state, and returns to the user.

Building on top of our previous experiment, we determined that we needed to keep track of the timing for the cold, warm, and hot starts of our system. The overhead differs drastically between the different starts, mostly due to the setup of the container in which the kernel function runs. For a cold start, the container has to be allocated from scratch and set-up, which takes a significant amount of time. For a warm start, the container is only paused, which allows a much faster start-up. For a hot start, the container is already running on some previous input and can immediately take the next input, leading to an almost negligible overhead.

8 Future Work

Both of the kernels we tested our system on were relatively simple kernels in terms of runtime. In order to show that the overhead for SRK/OpenLambda could be negligible even on a cold start, it is important to test our system on a more complex kernel chain.

Adding onto making the overhead for SRK/OpenLambda negligible, it would be beneficial toward our project to replace this combination with something more efficient. We have taken steps toward discussing integrating Droplet, another project at RISELab similar to OpenLambda, into PyPlover.

It would also be useful to see how easy it is to attach an existing machine learning framework, such as TVM [6], to our system. Since TVM auto-generates its CUDA code, we would need to tune this auto-generation system to generate PyPlover comptatible CUDA code instead.

One metric that we did not have time to measure is resource usage compared between running a kernel locally versus through our system. This would be an important metric to measure since if PyPlover resource allocation is much better compared to local resource allocation, it could be a worthwhile gain for the overhead time cost of setting up containers with OpenLambda.

9 Conclusion

PyPlover is a system we designed to abstract away the set-up of GPU-enabled instances toward the processing of GPU workloads. By doing this, the

client is able to get potentially better resource utilization than setting up a cluster of dedicated GPU instances himself. In addition, through PyPlover, scaling GPU workloads becomes trivial and even fault tolerance becomes less of an issue. Although the results for our system were not necessarily optimal, the most important takeaway of our project is the design of the general kernel function and the kernel chain structure we implemented on the server side.

10 Individual Contribution

Since this is a group project, Ryan Yang needs to list individual contributions:

- Wrote nearly this entire report
- Implemented the server-side code for PyPlover
- Created the poster for the poster session
- Collected timing results for vecAdd and a few for the CNN kernel chain
- Helped create the design for the buffer and state structures

References

[1] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, Daeyoung Kim, et al. GPU Enabled Serverless Computing Framework. https://ieeexplore.ieee.org/abstract/document/8374513/

[2] Jose Duato, Antonia J. Pena, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, et al. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. https://ieeexplore.ieee.org/document/5547126/

[3] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, et al. Occupy the Cloud: Distributed Computing for the 99%. https://arxiv.org/abs/1702.04024

[4] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani , Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, et al. Serverless Computation with OpenLambda. https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf

[5] Nathan Pemberton. Serverless Research Kit. https://github.com/serverlessresearch/srk

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. https://arxiv.org/abs/1802.04799