On the Acceleration of FaaS Using Remote GPU Virtualization

Diana M. Naranjo Delgado
Manuel Contreras
Germán Moltó
Sebastián Risco
Ignacio Blanquer
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Valencia, España
dnaranjo@i3m.upv.es
gmolto@dsic.upv.es
srisco@i3m.upv.es
iblanque@dsic.upv.es

Javier Prades
Federico Silla
Dpto. Informática de Sistemas y Computadores
Universitat Politècnica de València
Valencia, España
japraga@gap.upv.es
fsilla@upv.es

ABSTRACT

Serverless computing and, in particular, Function as a Service (FaaS) has introduced novel computational approaches with its highly-elastic capabilities, per-millisecond billing and scale-to-zero capacities, thus being of interest for the computing continuum. Services such as AWS Lambda allow efficient execution of event-driven short-lived bursty applications, even if there are limitations in terms of the amount of memory and the lack of GPU support for accelerated execution. To this aim, this paper analyses the suitability of including GPU support in AWS Lambda through the rCUDA middleware, which provides CUDA applications with remote GPU execution capabilities. A reference architecture for data-driven accelerated processing is introduced, based on elastic queues and event-driven object storage systems to manage resource contention and GPU scheduling. The benefits and limitations are assessed through a use case of sequence alignment. The results indicate that, for certain scenarios, the usage of remote GPUs in AWS Lambda represents a viable approach to reduce the execution time.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

Serverless, CUDA, GPU, FaaS, AWS Lambda

ACM Reference Format:
Diana M. Naranjo Delgado, Manuel Contreras, Germán Moltó, Sebastián Risco, Ignacio Blanquer, Javier Prades, and Federico Silla. 2023. On the Acceleration of FaaS Using Remote GPU Virtualization. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584933

1 INTRODUCTION

Serverless [6] has revolutionized computing in the last years with its ability to use managed services offered by Cloud providers to support event-driven computations on finely-grained auto-scaled platforms. Function as a Service (FaaS), the main service model that supports serverless, is most notably exemplified by AWS Lambda [3], Azure Functions [5] and Google Cloud Functions [14].

AWS Lambda allows executing user-defined functions in response to events such as file uploads to an object storage system (e.g. Amazon S3 [4]) or HTTP-based requests via Amazon API Gateway [1]. Its highly-elastic capacity allows executions of up to 3000 concurrent invocations for up to 15 minutes using at most 10240 MB of RAM in 1-MB increments. By providing scale-to-zero capabilities, no infrastructure pre-provision is required, thus dynamically scaling computational workloads using unprecedented levels of elasticity. Finally, its fine-grained billing approach, in milliseconds of execution time and depending on the amount of RAM allocated, introduces advantages for applications that show disparate computational requirements over time. The generous free tier allows developers to deploy applications in the Cloud at zero cost when they are not being used, which can dynamically scale to support heavy workloads and remain at zero cost if the free tier is not exceeded.

These advantages have propelled the adoption of serverless computing in different areas such as inference of Machine Learning (ML) models [30], linear algebra [37] and microservices-based applications [17], among many others. It is an essential layer in the computing continuum.

However, these advantages come with certain limitations. An ephemeral storage of [512,10240] MB leads to the creation of stateless functions with no affinity with the underlying hardware platform. This storage may be shared among invocations of the same function, depending on the provider's resource provisioning strategy. Stateful persistence of data generated in each Lambda function invocation can be achieved via a durable managed shared file system provided by Amazon EFS [33], which provides NFS (Network File System) as a service with a non-negligible additional cost.

In turn, GPU computing has demonstrated its ability to accelerate intensive computations in areas as diverse as genetic algorithms [8], medical image reconstruction [11], and privacy-preserving machine learning [39], among many others. It is precisely this hardware acceleration that led to the proliferation of these use cases since the usage of CPUs made it unfeasible to perform data processing in reasonable times.

However, one of the main limitations of FaaS services is the lack of support to use accelerated hardware, such as GPUs, to speed up the executions, thus preventing users from exploiting the versatility and high elasticity of these managed platforms to run the aforementioned applications. It is possible to deploy Virtual Machines (VMs) using Kubernetes-based managed services such as Amazon EKS [34] or Azure AKS [19] with instances (Virtual Machines) that support GPUs and then install existing open-source FaaS platforms such as OpenFaaS [12], Nuclio [15] or OSCAR[1] [23]. However, this approach relies on traditional auto-scaling approaches for virtual machines, thus hindering the ability to rapidly scale upon sudden workload increases.

With the advent of remote GPU virtualization techniques, such as those provided by rCUDA[2] [16], new avenues open up to accelerate the execution of FaaS-based invocations for workloads that can benefit from this support, such as sequence alignment or the inference of Machine Learning (ML) models. Indeed, rCUDA allows remote GPU sharing across function invocations. Therefore, it is essential to evaluate potential integration techniques to enable remote GPU virtualization on managed FaaS platforms to understand its benefits and limitations for accelerating applications running on public FaaS-based services such as AWS Lambda.

After the introduction, the remainder of the paper is structured as follows. First, section 2 describes the related work in using GPUs to accelerate serverless applications. Next, section 3 describes the components and introduces the proposed architecture to exploit remote GPUs by developing a technology demonstrator that involves AWS Lambda and rCUDA. Then, section 4 evaluates the platform with a use case of sequence alignment using GPU-Blast and discusses its advantages and drawbacks. Finally, section 5 summarizes the main contributions and points to future works.

2 RELATED WORK

Several works in the area are related to the usage of GPU-based computing in serverless platforms. A previous work in the area, by Risco et al. [29] provided the integration of the SCAR[3] framework for container-based computing in AWS Lambda and AWS Batch [2], to dynamically deploy virtualized elastic clusters with GPU support. This allowed to support event-driven workflows to efficiently process multimedia data where the short jobs are executed in AWS Lambda, and long-running jobs requiring GPU acceleration are executed in AWS Batch. Another work, by Naranjo et al. [20], introduced support for GPU resources on dynamically provisioned clusters configured with the OSCAR serverless platform, where resources were made available via PCI passthrough to accelerate inference on deep learning models for image classification.

[1]OSCAR - https://oscar.grycap.net
[2]rCUDA - http://www.rcuda.net/
[3]SCAR - https://github.com/grycap/scar

Several works introduce support for GPU computing into open-source serverless frameworks. This is the case of the work by Kim et al. [18], where the Iron.io framework is used to integrate NVIDIA-Docker as the underlying mechanism to use GPUs. The work by Satzke et al. [31] extends the open-source KNIX serverless framework to execute functions on shared GPU cluster resources. They execute Python KNIX functions by using NVIDIA GPU resources.

Previous work is also related to designing improved approaches for GPU scheduling, with potential applications to FaaS frameworks. For example, the work by Garg et al. [13] proposes a solution for the scheduling of predefined GPU functions in heterogeneous multi-GPU setups. They created a discrete-event simulator-based prototype, but no open-source implementation was provided. The work by Prakash et al. [26] uses a software-based task slicing technique to determine task sizes for scheduling on vGPUs so that most of the functions can successfully complete by the deadlines. The approach is evaluated with different heuristics, but no direct application to an existing FaaS framework is provided.

This paper proposes an architecture and a reference implementation in AWS Lambda to exploit remote GPUs using the rCUDA middleware. To the best of the authors' knowledge, this is the first paper that evaluates the feasibility of using remote GPU virtualization techniques to accelerate workloads that run on a managed FaaS service, in this case, AWS Lambda. Indeed GPU support is not available in the public FaaS services. Still, there is a need to accelerate new workloads being deployed in AWS Lambda, such as AI/ML model inference. Indeed, the lack of GPU support is identified in the work by Christidis et al. et al. [9] as a limiting factor in serverless platforms.

3 COMPONENTS AND ARCHITECTURE

Figure 1 summarises the main approach envisaged to support remote GPU virtualization for accelerating function invocations in a managed FaaS service. We focus on event-driven data-processing applications to adopt a widely used computing paradigm. A user uploads a file to an object storage system in the Cloud (step 1), which enqueues a job execution request in an elastic queue (step 2) to process that file with a user-defined set of GPU-based requirements. The GPU-aware Resource Contention Manager is responsible for contacting the remote GPU virtualization layer to determine if there are free resources, in which case it reserves them (step 4) and it triggers the invocation of a previously-defined function in the FaaS service (step 5). The function's application code uses CUDA and profits from the reserved remote GPU resources to accelerate its execution (step 6), releasing the resources once the execution has finished. The output data files are stored in the object storage for long-term persistence (step 7).

From this initial view, this section describes the primary components used to support remote GPU virtualization in AWS Lambda through the rCUDA framework. Notice that a similar approach can be extrapolated to alternative FaaS services such as Microsoft Azure Functions or Google Cloud Functions. For the sake of clarity, Figure 2 provides an interaction diagram.

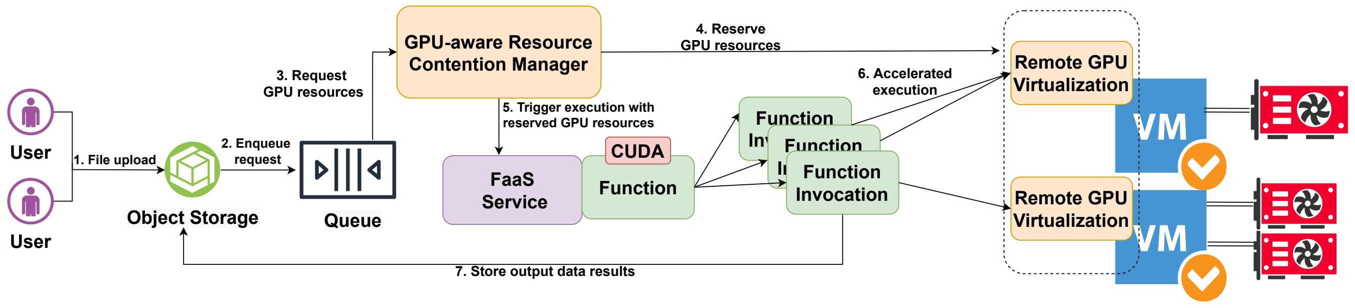

Figure 1: Generic approach to integrate remote GPU virtualization support in a managed FaaS service.

3.1 SCAR (Serverless Container-aware ARchitectures)

SCAR [22] is an open-source framework for building event-driven file-processing serverless applications on public FaaS platforms, supporting AWS Lambda. SCAR simplifies running container-based applications in AWS Lambda as event-driven applications triggered via an HTTP call to API Gateway or in response to file uploads to a file storage system, such as Amazon S3. Created in 2017, it pioneered using Docker images as a runtime in AWS Lambda. Functions in SCAR are created using a Functions Definition Language (FDL) in a YAML configuration file, where the parameters required for creating the SCAR function are specified.

Recall that AWS Lambda has strict computational limits regarding the amount of RAM, ephemeral storage potentially shared across invocations, and execution time. For those applications that do not satisfy these requirements, SCAR provides seamless integration with AWS Batch. This AWS Elastic Cluster-as-a-Service product runs applications packaged in Docker containers. It can grow and shrink based on the number of jobs in the queue and the available resources. This allows exceeding the maximum execution limit of 15 minutes imposed by AWS Lambda or even the use of GPU for an accelerated execution [29] by resorting to executing on dynamically provisioned virtual machines from Amazon EC2.

In this work, SCAR is employed to facilitate the deployment of the Docker-based Lambda functions, as shown in Figure 2. Therefore, it is a convenient tool, but not mandatory, to introduce the support for remote GPU via rCUDA in AWS Lambda functions.

3.2 rCUDA

rCUDA [38] is a middleware that allows using remote devices compatible with CUDA (Compute Unified Device Architecture) [40] in a completely transparent way to the programmer without the need to modify the source code written for CUDA. This way, CUDA applications being executed in a computer without a GPU can use one located in a remote computer. In this regard, rCUDA virtualizes GPUs that are physically located on a remote machine, thus turning that computer into a GPGPU (General Purpose Computing on Graphics Processing Units) server [27]. rCUDA presents a client-server architecture that supports most CUDA APIs and operations. On the client side, rCUDA is a library with the very same interface as the NVIDIA CUDA library. Thus, the CUDA application is unaware of using a remote GPU managed by rCUDA. On the server side, there is a daemon that receives requests from the client side and forwards them to the physical GPU. Communication between the client (CUDA application being executed in a node without GPU) and the server (remote node where the physical GPUs are located) can be done through TCP/IP (Ethernet) or Infiniband [24] to minimize network latency [28]. One of the most prominent features of rCUDA is concurrent remote GPU sharing among multiple applications, which encourages the development of a multitenant environment and the efficient use of GPUs [20].

rCUDA's scheduler (SSGM, Simple Scheduler for GPU Management) allows allocating GPU resources to jobs [25], and it is not part of the client-server rCUDA architecture described above. The SSGM scheduler consists of three processes that run on different nodes:

- **SSGMD** is the scheduler daemon. This process implements the GPU allocation logic based on the configured policy. It is responsible for the allocation of resources. This daemon is executed in one of the nodes of the cluster of machines (regardless of where physical GPUs are located) and gathers allocation requests from the SSGM process and GPU status information from the SSGM GPU Monitor (see below). With the information from the SSGM GPU Monitor and the allocation requests from the SSGM process, the SSGMD daemon performs the best job-resource matching available at that moment. Once the matching is carried out, the SSGMD daemon provides the required information to the SSGM process. The SSGMD daemon has two working modes: blocking and non-blocking. In non-blocking mode, the scheduler immediately returns a response, which can be either positive if the requested resources were allocated or negative if it could not assign the requested resources. In the blocking mode, the scheduler does not provide a response until the allocation is carried out (notice that in case an impossible allocation is requested, then the scheduler immediately provides a negative answer).
- **SSGM GPU Monitor** is a small daemon that runs on every node with GPUs to be used by the rCUDA middleware in the set of machines. A given SSGM GPU Monitor daemon running in a node communicates to the SSGMD, in real-time, information related to the GPUs available in that node, such as the number of jobs running in the GPU, available GPU memory, GPU utilization, etc. As stated earlier, the SSGMD scheduler uses that information to carry out the

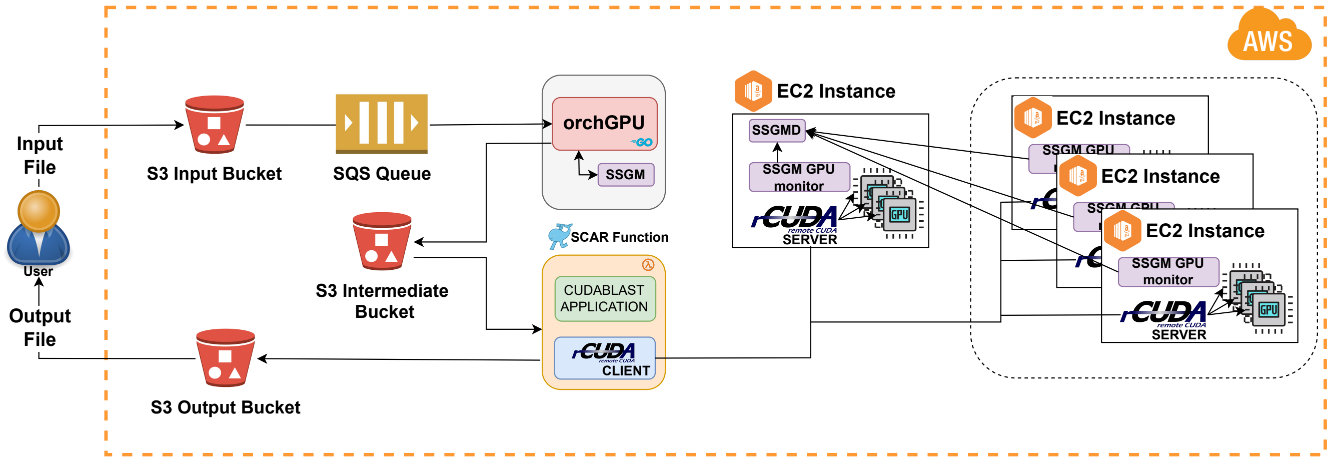

Figure 2: Architectural approach to support GPU virtualization on AWS Lambda via rCUDA.

match between jobs and GPUs according to the scheduling policy configured in the cluster.

- **SSGM** is the interface between the SSGMD daemon and users. The SSGM process acts as a client of the SSGMD scheduler daemon and can be used to modify configuration parameters in the scheduler as well as to submit requests to it. In the latter case, the SSGM program is executed in the node where the CUDA application using a remote GPU will be used. In this scenario, every time a user wants to submit a job to the scheduler, the user leverages the SSGM application to send the request information to the scheduler daemon (amount of requested GPUs, amount of requested GPU memory, etc). The SSGM application waits for the answer from the scheduler. If the resource allocation can be performed, the data received from the SSGMD daemon is used by the requesting application to determine, among other parameters, the IP addresses of the remote rCUDA servers to be used.

In this work, the rCUDA server is employed to virtualize the remote GPUs. There is one instance of the rCUDA server per node with GPUs, while there is only an instance of the rCUDA's scheduler (SSGM) for all the group of nodes to be used. Even if the rCUDA framework was initially designed to virtualize GPUs in physical resources it can seamlessly run on GPU-enabled virtual machines, such as the *p2* family of instances in AWS, which support up to 16 NVIDIA K80, 64 vCPU and 732 GiB of RAM with 192 GB of GPU memory. Notice that additional GPU-enabled instance types are available with different types of GPUs.

3.3 orchGPU: GPU orchestrator

OrchGPU is an open-source software developed by our research group in Go[4] that relies on asynchronous programming techniques to coordinate the GPU usage requests into the rCUDA scheduler for managing resource contention. Indeed, the highly-elastic capabilities of AWS Lambda, which can run up to 3000 concurrent invocations can rapidly introduce a bottleneck in the set of GPU-based computing resources.

[4]Go - https://go.dev/doc/

To this aim, orchGPU uses an elastic job queue created on Amazon SQS [35] that queues up the job execution requests that are triggered in response to file uploads to an Amazon S3 input bucket. After reserving the corresponding GPU resources via the rCUDA scheduler, it delegates the job request into an S3 intermediate bucket to trigger the Lambda function that uses the rCUDA client to use the reserved remote GPU resources. Note that orchGPU relies on Amazon S3 as the source and target of events in order to match the file processing model introduced by SCAR. However, it could be easily adapted to accommodate other event-driven services through the cloud provider's interfaces. This program is one of the fundamental elements of the designed architecture, since it is in charge of communicating several modules of the architecture. The functionalities of orchGPU can be summarized as follows:

- Initialize the AWS environment and get the URL of the job queue.
- Execute an infinite loop with the following actions:

(1) Get a message from the queue.
(2) Make a request to the rCUDA's scheduler (SSGM).
(3) Process the data received from the scheduler, specifically the number of available GPUs and the IP address of the EC2 where the GPUs are located. This information is needed by the rCUDA client to establish the connection with the rCUDA server.
(4) Save the information of the environment variables needed by the rCUDA client, as well as the input file, compressed in a zip file, which is stored in an intermediate bucket. This will trigger the Lambda function invocation to process the CUDA application that will use the remote GPU resources reserved.
(5) Wait for the result of the file processing.
(6) Release the GPU resources that were allocated for the execution of the application.
(7) Delete the message from the SQS queue.

The actions described above are executed continuously, so when there are several messages in the queue, orchGPU detects these messages and asks the scheduler for available resources. If GPU resources are available, orchGPU will execute the same procesfs of creating the zip file with the environment variables needed by the rCUDA client and the input file. This allows parallel execution of multiple jobs as long as computing resources are available.

Figure 2 shows the interaction among the aforementioned components to support remote GPU virtualization in AWS Lambda. This is based on SCAR for the deployment and processing of the serverless functions in AWS Lambda, rCUDA for the virtualization of the GPUs and orchGPU to create and manage the file-processing requests. In this prototype design, orchGPU runs on an Amazon EC2 instance and the current multi-threaded implementation partially alleviates the bottleneck introduced by orchGPU acting as a resource scheduler.

However, in order to overcome this issue, nothing prevents running orchGPU as a Lambda function that is triggered in response to the messages arriving at the SQS queue. Notice that SQS queues have a *visibility window*, defined in time units, so that messages that are extracted from the queue, but are not explicitly deleted, automatically reappear in the queue thus triggering again the Lambda function invocation. This would allow retrying GPU allocation requests to the rCUDA scheduler in case of a temporary lack of resources.

In fact, multiple EC2 instances can be used, each one reporting GPU resources to the SSGMD component. In combination with the auto-scaling capability of Amazon EC2 and the ability to scale on user-defined metrics in CloudWatch, the AWS monitoring service, the back-end of GPU-based compute resources can grow and shrink depending on the number of messages in the SQS queue (e.g. using the metric *ApproximateNumberOfMessagesVisible*). However, in this particular scenario, since Lambda functions are ephemeral, the release of GPU resources should be performed by the Lambda function invocation with the CUDA application, once it has finished the execution, thus requiring support by the application developer.

The design of this architecture enables GPU allocation support in executing Lambda functions, a capability not yet supported on AWS Lambda.

4 USE CASE: GPU-BLAST

In order to evaluate the benefits of the proposed approach in terms of jobs processed per time unit, a case study has been designed that uses the GPU-Blast[5] application. GPU-Blast is a version of the popular NCBI-BLAST[6] application that makes use of acceleration devices to improve the processing times of the original application. NCBI-BLAST is a local alignment search (BLAST) tool for finding regions of similarity between sequences. The program compares nucleotide or protein sequences with sequence databases and calculates the statistical significance of the matches [21]. The input file of GPU-Blast is a text file with information about the sequence and the output file generated after the execution process shows the result of the sequences in the database that are similar according to a weight matrix and the editing distance to the searched sequence.

Figure 3 shows the interaction diagram among the components of the proposed architecture. These are the steps taken during the processing of a file:

[5]GPU-Blast - https://sahinidis.coe.gatech.edu/gpublast
[6]BLAST - https://blast.ncbi.nlm.nih.gov

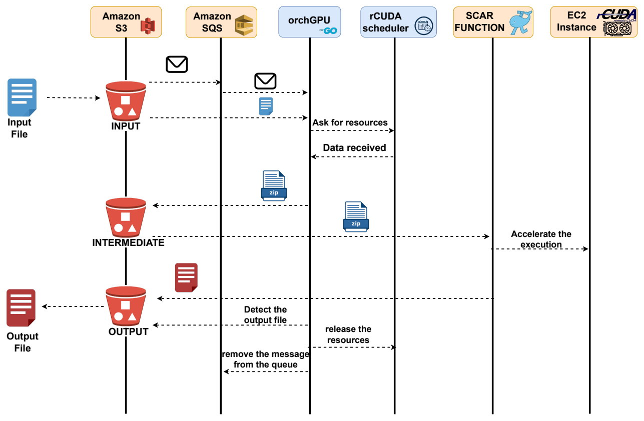

Figure 3: Interaction diagram to process a file.

(1) Through the SCAR command-line interface, the user creates the function with the application to process the file, in this case, GPU-Blast. It is necessary to define the input bucket together with the intermediate bucket, which is the one that will generate the event to trigger the processing of the input file. The output file can be any bucket selected by the user. The code shown in Figure 4 displays the YAML file needed to create this function. Note that the user defines the memory requirements and the container image available in Amazon ECR [32], a registry for Docker images.
(2) The user uploads a file to the input bucket to trigger the processing of the file, which introduces a message in the SQS queue indicating that there is a new input file ready to be processed. The orchGPU gets the message from the queue.
(3) orchGPU requests a reservation of GPU resources to the rCUDA scheduler (SSGM) for processing the input file. OrchGPU combines the data received by the scheduler, i.e. the connection details to the remote rCUDA server, and the input file and stores it in an intermediate bucket.
(4) Uploading this file to the intermediate bucket generates an event that triggers the execution of the Lambda function created by the user in step 1. This script first decompresses the file to obtain the necessary data to locate the rCUDA server. Then creates the necessary environment variables to make the connection between the rCUDA client and the rCUDA server, processes the input file and, finally, stores the result in the output bucket.
(5) Once orchGPU detects that the file resulting from the alignment process is in the output bucket, it releases the resources and removes the message from the queue.
(6) In the last step, the user downloads the output file generated after the inference process.

The objective of the experiment is fundamentally to assess the efficiency of the system in terms of execution time. To that end, files with several sequence lengths are used and the execution time is measured considering different scenarios:

- Scenario 1 (CPU): The execution is performed on the CPU provided by the Lambda runtime environment. This baseline

```
functions:
  aws:
    - lambda:
        runtime: image
        name: scar-rcuda-cudablast-orchgpu
        memory: 3072
        init_script: script.sh
        container:
          image: rcuda-lambda-cudablast
          create_image: false
        input:
        - storage_provider: s3
          path: scar-rcuda/intermediate
        output:
          - storage_provider: s3
            path: scar-rcuda/output
      ecr:
        delete_image: false
```

Figure 4: Functions Definition Language file to deploy the function through SCAR.

will be used to compare how much the processing time can be improved by using (remote) GPUs.
- Scenario 2 (Native GPU): The execution is carried out on a native GPU, specifically a 32GB Tesla V100 located in an on-premises cluster belonging to our research group, connected via PCI passthrough to a virtual machine with 8 vCPUs and 8GB RAM. It is important to point out that in this scenario, networking is not involved, so there will be no latency due to the communication between client and server, just pretending to illustrate the performance of a dedicated GPU interface for the processing of this use case.
- Scenario 3 (Lambda + SCAR + rCUDA): This refers to the architecture described in Figure 2. Execution times in this scenario will be compared with the worst expected times (Scenario 1, CPU only) and the best expected times (Scenario 2, native GPU).

GPU-Blast allows you to specify the number of threads in the alignment process, so we have run the tests in the three scenarios with both 1 thread and 6 threads, to determine the influence of multi-threading in the processing times. The upper limit to the number of threads is due to AWS Lambda assigning 6 vCPUS when allocating the maximum amount of RAM to the Lambda function [10].

As previously mentioned, rCUDA is a client-server middleware. For this case study, the rCUDA server is located on an Amazon EC2 *p2.xlarge* instance that has one GPU and is the cheapest based on price. This instance has 1 GPU, 4 vCPUs and 61 GiB of RAM and is priced at $0.9/hour in the N. Virginia region. In addition to the rCUDA server in the instance, the SSGM GPU Monitor for GPU monitoring is also executed, together with orchGPU. Notice that this approach allows performing a technology demonstration

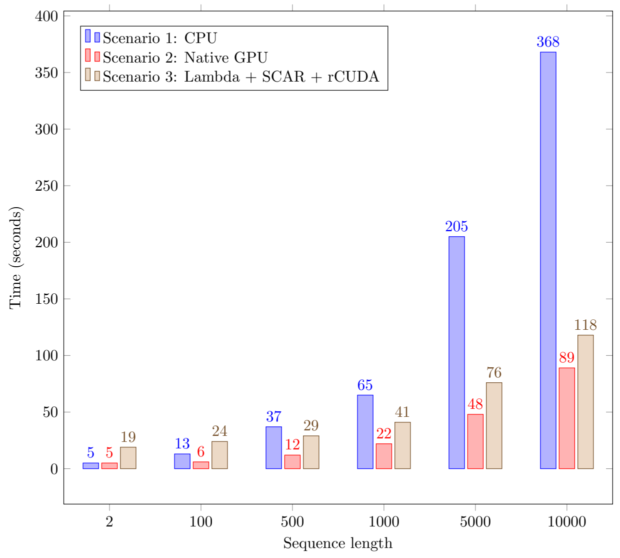

Figure 5: Execution times for different sequence lengths in the scenarios analyzed (using 1 thread).

with reduced costs, even if the centralized approach is not the best approach in terms of performance.

Moreover, a Lambda function is used in the proposed architecture with 3072 MB of RAM with a cost of $0.0000000333/ms. The amount of RAM required to execute the application was empirically estimated. The function runs the rCUDA client library and the GPU-Blast application. There is also a job queue in Amazon SQS to process incoming messages and three S3 buckets, one for input files, another for the output files and an intermediate bucket (its use is explained below). The three S3 buckets (input, intermediate and output) deliver a cost of $0.023/GB for data storage. Notice that all of these AWS services have a free tier that reduces the overall cost. For example, AWS Lambda allows 1M free requests per month and in Amazon S3 the first 12 months are free.

4.1 Results

In order to test the efficiency of the proposed approach, different sequence lengths were used to assess the behaviour with different execution times: 2, 100, 500, 1000, 5000, and 10000. The longer the sequence, the longer the execution time. Four executions were performed for each sequence length, so the times shown in the following graphs correspond to an average of the values obtained.

Figure 5 shows the times obtained in each of the considered scenarios when a single-thread configuration is used in the GPU-Blast application. Figure 6 shows the same experiment, but in this case, six threads were specified when executing the GPU-Blast application.

The results depicted in Figure 5 show that CPU processing, in Scenario 1, is less efficient than in Scenarios 2 and 3 where acceleration devices are used. The best processing times are obtained in Scenario 2 where the execution process is performed natively on the GPU. In Scenario 3, where the proposed architecture is used,

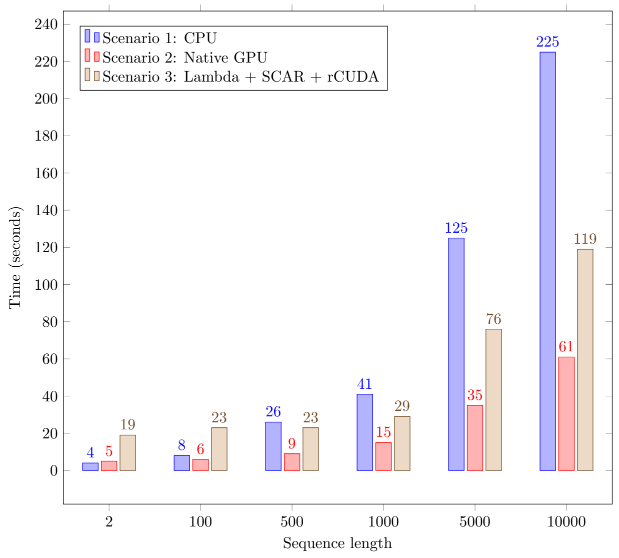

Figure 6: Execution times for different sequence lengths in the scenarios analyzed (using 6 threads).

the processing times greatly improve the times obtained in CPU, although they are larger than the times obtained in Scenario 2. Note that for the case where the length of the sequence of the input file is higher, the results obtained with the proposed architecture greatly improve CPU processing times and are quite close to the times obtained with the native GPU even if delays are introduced in the proposed architecture due to the component interaction.

The results obtained in Figure 6 are approximately the same as those obtained in the previous figure. Remember that AWS Lambda proportionally allocates the CPU power in correspondence with the selected memory, with a maximum of 10GB of memory with 6 vCPU [36]. We determined that, for the case study, three cores were allocated to the Lambda function. Notice that the usage of remote GPUs in Scenario 3 outperforms Scenario 1, while the latencies introduced by the usage of remote GPUs cannot deliver a reduced execution time compared to Scenario 2.

In addition to the experiments described here, other sequence lengths were used, but for these values, the processing time exceeded the 15-minute maximum execution time imposed by the AWS Lambda environment. Although the processing times obtained in Scenario 3 are higher than the times obtained where the processing is done on the native GPU, Scenario 2, the results show that GPU virtualization in AWS Lambda can broaden the range of applications that they can benefit from the FaaS model. Moreover, researchers usually search for sequences smaller than 10.000 nucleotides. As an example, the Spike protein of the SARS-COV-2 is on the order of 4.000 base pairs.

An analysis of execution time and cost was carried out. For the analysis of the cost when running exclusively in the CPU (Scenario 1), the execution time of the Lambda function is taken into account, and for the usage of a remote GPU (Scenario 3), the cost of the Lambda function and the EC2 instance with the GPU are considered.

For the highest execution time on CPU and GPU, in the case of executing with 1 thread, that is, 368 and 118 seconds, the cost per execution was determined to be 0.0184 USD and 0.0354 USD, respectively. For the case of executing with 6 threads in the CPU, the cost was 0.011 USD. The GPU processing times are lower than the times obtained for the CPU, especially for longer sequences, which require a greater execution time. However, the costs when using a GPU with respect to Scenario 1, where CPU is used, are higher, also taking into account the selected instance.

Although times obtained in AWS Lambda are longer than with native GPUs, the new architecture improves times in Lambda without investing in local GPUs. Indeed, this contribution has assessed that it is technically viable to introduce remote GPU support in the constrained execution environment provided by AWS Lambda. However, further analyses should be carried out to determine the cost-effectiveness of this approach. First, increasing the amount of RAM in the Lambda function increases the price (billed in milliseconds), but it also affects the underlying number of vCPUs and computing power, which also reduces the execution time. This is why tools such as AWS Lambda Power Tuning [7] help visualize and fine-tune the memory/power configuration of Lambda functions in terms of cost and speed.

Direct concurrent execution of multiple applications on a GPU is not always possible, justifying the need for a virtualization layer. Since rCUDA allows to share the GPUs among multiple Lambda function invocations, the cost of provisioning GPU-enabled EC2 instances can be also distributed among the invocations. Extending such tools to consider the cost of remote GPU sharing would be beneficial to determine the cost-effectiveness of this approach.

5 CONCLUSIONS AND FUTURE WORK

This article has proposed an architecture to support remote GPU execution in managed FaaS services, exemplified in AWS Lambda, a feature that is unavailable. The components involved are SCAR, which executes applications packaged in Docker containers as functions in AWS Lambda and which are triggered in response events such as file uploads to Amazon S3; rCUDA, a tool that allows the virtualization and sharing of GPUs; and orchGPU, to manage resource contention and scheduling on the available GPU resources.

Integrating GPUs as back-end resources is key for improving fast execution and resource provisioning in serverless functions. GPU resources are expensive, and one of the advantages of using the designed architecture is the ability to share the same GPU between multiple executions, thanks to the use of rCUDA and the developed tools. The proposed architecture extends the SCAR framework by including GPU virtualization in Lambda functions.

The implemented development constitutes a step forward in the adoption of the serverless model in relation to the use of acceleration devices for the execution process of applications that support CUDA. The processing times obtained for this type of application in GPU environments show an improvement in performance compared to the exclusive use of CPU. With the proposed design, processing time can be reduced without investing in local GPUs.

There are several fundamental lines in which we intend to continue to improve the design of the designed system. First, we plan to further automate the deployment of an auto-scaled back-end so that additional EC2 instances are deployed upon workload increments, measured as increments in the number of messages in the queue. Second, we plan to further explore additional applications that can benefit from the remote GPU acceleration within AWS Lambda, as is the case of inference of ML models. Finally, we plan to extend the cost analysis to additional instance types in order to find cost-effective scenarios that benefit from the integration of GPU support in AWS Lambda.

ACKNOWLEDGMENTS

Grant PID2020-113126RB-I00 funded by
MCIN/AEI/10.13039/501100011033 and Project PDC2021-120844-I00 funded by
MCIN/AEI/10.13039/501100011033 funded by the European Union NextGenerationEU/PRTR. The European Commission has partially funded this work under the H2020 grant 101016577 AI-SPRINT: AI in Secure Privacy-pReserving computINg conTinuum.

REFERENCES

[1] Amazon Web Services. [n. d.]. Amazon API Gateway. https://aws.amazon.com/api-gateway/
[2] Amazon Web Services. [n. d.]. AWS Batch — Easy and Efficient Batch Computing Capabilities. https://aws.amazon.com/batch/
[3] Amazon Web Services. [n. d.]. *AWS Lambda*. https://aws.amazon.com/lambda
[4] Amazon Web Services. [n. d.]. Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/
[5] Microsoft Azure. [n. d.]. Azure Functions. https://azure.microsoft.com/es-es/services/functions/{#}overview
[6] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1 arXiv:1706.03178
[7] Alex Casalboni. [n. d.]. AWS Lambda Power Tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning
[8] John Runwei Cheng and Mitsuo Gen. 2019. Accelerating genetic algorithms with GPU computing: A selective overview. *Computers and Industrial Engineering* 128 (2019), 514–525. https://doi.org/10.1016/j.cie.2018.12.067
[9] Angelos Christidis, Roy Davies, and Sotiris Moschoyiannis. 2019. Serving machine learning workloads in resource constrained environments: A serverless deployment example. *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019* (11 2019), 55–63. https://doi.org/10.1109/SOCA.2019.00016
[10] Robert Cordingly, Navid Heydari, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, and Wes Lloyd. 2021. Enhancing observability of serverless computing with the serverless application analytics framework. In *ICPE 2021 - Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM, New York, NY, USA, 161–164. https://doi.org/10.1145/3447545.3451173
[11] Philippe Després and Xun Jia. 2017. A review of GPU-based medical image reconstruction. *Physica Medica* 42 (oct 2017), 76–92. https://doi.org/10.1016/j.ejmp.2017.07.024
[12] Alex Ellis. [n. d.]. OpenFaaS. https://www.openfaas.com/
[13] Anshuj Garg, Purushottam Kulkarni, Umesh Bellur, and Sriram Yenamandra. 2021. FaaSter: Accelerated Functions-as-a-Service with Heterogeneous GPUs. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 406–411. https://doi.org/10.1109/HiPC53243.2021.00057
[14] Google Cloud. [n. d.]. Cloud Functions - Event-driven Serverless Computing. https://cloud.google.com/functions/
[15] Iguazio. [n. d.]. Nuclio. https://nuclio.io/
[16] Sergio Iserte, Javier Prades, Carlos Reaño, and Federico Silla. 2016. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 98–101.
[17] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, Vol. 15. ACM, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701
[18] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. 2018. GPU Enabled Serverless Computing Framework. In *Proceedings - 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2018*. Institute of Electrical and Electronics Engineers Inc., 533–540. https://doi.org/10.1109/PDP2018.2018.00090
[19] Microsoft. [n. d.]. Azure AKS. https://azure.microsoft.com/services/kubernetes-service/{#}overview
[20] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel and Distrib. Comput.* 139 (may 2020), 32–42. https://doi.org/10.1016/J.JPDC.2020.01.004
[21] National Library of Medicine. [n. d.]. BLAST: Basic Local Alignment Search Tool. https://blast.ncbi.nlm.nih.gov/Blast.cgi
[22] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. 2018. Serverless computing for container-based architectures. *Future Generation Computer Systems* 83 (jun 2018), 50–59. https://doi.org/10.1016/j.future.2018.01.022
[23] Alfonso Perez, Sebastian Risco, Diana Maria Naranjo, Miguel Caballer, and Germán Moltó. 2019. On-premises serverless computing for event-driven data processing applications. In *IEEE International Conference on Cloud Computing, CLOUD*, Vol. 2019-July. 414–421. https://doi.org/10.1109/CLOUD.2019.00073
[24] Javier Prades, Carlos Reaño, and Federico Silla. 2019. On the effect of using rCUDA to provide CUDA acceleration to Xen virtual machines. *Cluster Computing* 22, 1 (2019), 185–204. https://doi.org/10.1007/s10586-018-2845-0
[25] Javier Prades and Federico Silla. 2018. Made-to-Measure GPUs on Virtual Machines with rCUDA. In *The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018*. ACM, 19:1–19:8. https://doi.org/10.1145/3229710.3229741
[26] Chandra Prakash, Anshuj Garg, Umesh Bellur, Purushottam Kulkarni, Uday Kurkure, Hari Sivaraman, and Lan Vu. 2021. Optimizing Goodput of Real-time Serverless Functions using Dynamic Slicing with vGPUs. In *Proceedings - 2021 IEEE International Conference on Cloud Engineering, IC2E 2021*. Institute of Electrical and Electronics Engineers Inc., 60–70. https://doi.org/10.1109/IC2E52221.2021.00020
[27] Carlos Reano and Federico Silla. 2015. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 488–489. https://doi.org/10.1109/CLUSTER.2015.76
[28] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. 2015. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference on ZZZ - Middleware Industry '15*. ACM Press, New York, New York, USA, 1–7. https://doi.org/10.1145/2830013.2830015
[29] Sebastián Risco and Germán Moltó. 2021. GPU-Enabled Serverless Workflows for Efficient Multimedia Processing. *Applied Sciences* 11, 4 (feb 2021), 1438. https://doi.org/10.3390/app11041438
[30] Sebastián Risco, Germán Moltó, Diana M. Naranjo, and Ignacio Blanquer. 2021. Serverless Workflows for Containerised Applications in the Cloud Continuum. *Journal of Grid Computing* 19, 3 (sep 2021), 30. https://doi.org/10.1007/s10723-021-09570-2
[31] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. 2021. Efficient GPU Sharing for Serverless Workflows. In *HiPS 2021 - Proceedings of the 1st Workshop on High Performance Serverless Computing, co-located with HPDC 2021*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/3452413.3464785
[32] Amazon Web Services. [n. d.]. Amazon ECR. https://aws.amazon.com/ecr/
[33] Amazon Web Services. [n. d.]. Amazon EFS. https://aws.amazon.com/efs/
[34] Amazon Web Services. [n. d.]. Amazon EKS. https://aws.amazon.com/eks/
[35] Amazon Web Services. [n. d.]. Amazon SQS. https://aws.amazon.com/sqs/
[36] Amazon Web Services. 2020. AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions. https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/
[37] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *SoCC 2020 - Proceedings of the 2020 ACM Symposium on Cloud Computing*, Vol. 15. Association for Computing Machinery, Inc, New York, NY, USA, 281–295. https://doi.org/10.1145/3419111.3421287
[38] Federico Silla, Sergio Iserte, Carlos Reaño, and Javier Prades. 2017. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience* 29, 13 (2017), e4072. e4072 cpe.4072.
[39] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *Proceedings - IEEE Symposium on Security and Privacy*, Vol. 2021-May. Institute of Electrical and Electronics Engineers Inc., 1021–1038. https://doi.org/10.1109/SP40001.2021.00098 arXiv:2104.10949
[40] Manuel Ujaldón. 2016. CUDA achievements and GPU challenges ahead. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 9756. 207–217. https://doi.org/10.1007/978-3-319-41778-3_20