UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Computación Serverless basada en GPU en AWS

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y Altas Prestaciones

Autor: Manuel Ramón Contreras Ruiz

Tutor: Germán Moltó Martínez

Dirección experimental: Diana María Naranjo Delgado

Curso: 2020-21

Quiero dedicar este proyecto a :

Germán, Diana y otros miembros del grupo de investigación GRyCAP,
con los que he aprendido y disfrutado haciéndolo.

El equipo de rCUDA, especialmente a Federico y Javier,
por estar siempre dispuestos a prestar su ayuda.

A mi familia y mis amigos,
porque siempre sé que cuento con su apoyo.

Resumen

La computación serverless (sin servidor) ha surgido en los últimos años como una evolución de los modelos de ejecución de aplicaciones basados en el cloud. Esencialmente, el modelo serverless permite a sus usuarios despreocuparse completamente de la infraestructura subyacente a sus aplicaciones, pudiendo dedicar recursos a otras áreas más interesantes e incluso, en algunos casos, reducir los costes de despliegue y operación. En el núcleo del modelo serverless se encuentran los servicios FaaS (Function as a Service), entre los que se encuentra AWS Lambda, la propuesta de Amazon Web Services. Esta memoria refleja el proceso y las conclusiones obtenidas de un proyecto en el que se ha tratado de proporcionar características de computación GPU a AWS Lambda mediante el *framework* rCUDA, en el contexto de la inferencia de modelos de aprendizaje profundo.

Palabras clave: Serverless Computing, Amazon Web Services, GPU

Abstract

Serverless computing has risen in the recent years as an evolution of cloud-based application execution models. In essence, the serverless model allows users to deploy their applications without having to operate the underlying infrastructure, enabling them to dedicate resources to other areas, and in some cases, even reduce deployment and operational costs. In the core of the serverless model we find FaaS (Function as a Service) services, with AWS Lambda being the FaaS offer by Amazon Web Services. This document shows the process and results obtained in a project in which we have tried to integrate GPU computing capabilities intro AWS Lambda using the rCUDA *framework*, using deep learning inference as context.

Keywords: Serverless Computing, Amazon Web Services, GPU

Contenido

Figuras

Tablas

Parte 1. Introducción

En esta primera parte, se detallan la motivación para afrontar el proyecto y sus objetivos técnicos y académicos, así como la metodología definida para llevarlo a cabo.

1.1 Motivación

El panorama de los servicios cloud se encuentra en constante evolución para dar respuesta a las necesidades de los usuarios, quienes demandan constantemente una mayor variedad de soluciones que les permitan optimizar sus procesos de desarrollo y despliegue y así ser más competitivos. Aun así, es muy complicado que los grandes proveedores cloud puedan dar soporte a todos los casos de usos que sus clientes plantean, por lo que surge la necesidad de desarrollar soluciones personalizadas basadas en la tecnología existente.

AWS Lambda [1] es el líder global entre los servicios FaaS [2] y desde su introducción en 2014, ha presentado mejoras de funcionalidad considerables de forma regular. Sin embargo, a fecha de este proyecto, Lambda no cuenta con soporte nativo para computación basada en procesadores gráficos (GPU) de uso general [3], muy habitual en campos como el aprendizaje profundo. AWS propone que la inferencia se haga en Lambda usando únicamente CPUs, y que se consideren otros servicios como Elastic Container Service (ECS) [4], Elastic Kubernetes Service (EKS) [5] o SageMaker [6] para aquellos casos en los que sea indispensable hacer uso de GPUs. Este proyecto busca integrar herramientas desarrolladas en el grupo de investigación GRyCAP (Grupo de Grid y Computación de Altas Prestaciones) [7] con el framework rCUDA [8] para lograr la comunicación de funciones Lambda con dispositivos GPU remotos. Esto no solo ayudaría a avanzar la investigación de dichas herramientas, sino que demostraría la capacidad de Lambda para soportar nuevos flujos de trabajo que puedan beneficiarse de estos dispositivos de aceleración.

1.2 Objetivos

El principal objetivo de este proyecto es investigar de forma práctica la integración del procesamiento basado en GPU remota con funciones de AWS Lambda, para así evaluar las posibilidades y beneficios del uso conjunto de dichas tecnologías. En concreto, se han identificado los siguientes objetivos académicos y técnicos:

- **Objetivos académicos**
  - **Objetivo 1:** Ampliar los conocimientos adquiridos en asignaturas relacionadas con la computación cloud y serverless.
  - **Objetivo 2:** Poner en práctica habilidades trasversales como la auto-organización y la elaboración de documentación técnica.
  - **Objetivo 3:** Colaborar en líneas de investigación actuales.
  - **Objetivo 4:** Adquirir conocimientos sobre rCUDA, su scheduler y la ejecución en GPU remota.

- **Objetivos técnicos**
  - **Objetivo 1:** Adquirir experiencia implementando soluciones serverless que aborden casos de uso reales.
  - **Objetivo 2:** Demostrar la viabilidad de la ejecución en GPU remota para casos de uso relacionados con la inferencia de modelos de aprendizaje profundo y automatizar el proceso de inferencia mediante el desarrollo de un software específico.
  - **Objetivo 3:** Estudiar la relación coste/beneficio de las soluciones desarrolladas, y detallar las posibles limitaciones y casos de uso recomendados para las tecnologías implicadas.

1.3 Alcance

Este proyecto está orientado a investigar y desarrollar una prueba de concepto que demuestre que es posible la integración exitosa de las tecnologías involucradas. Independientemente del resultado, el proceso de desarrollo será una fuente de información sobre el nivel de madurez y funcionalidad de dichas tecnologías. Aunque se prestará atención al rendimiento y tiempos de ejecución de las pruebas que se realicen, conseguir implementaciones óptimas no se considera parte de los objetivos fundamentales, ya que sobre todo se busca adquirir conocimiento explorando soluciones alternativas a los problemas que se planteen. Por otro lado, el uso de modelos de aprendizaje profundo sirve principalmente para orientar el proyecto en un contexto realista posiblemente útil, pero se considera que la implementación de dichos modelos y la evaluación de sus prestaciones escapa al ámbito del proyecto.

1.4 Metodología

Desde el principio del proyecto, se tenía relativamente claro las tecnologías que se iban a usar y los objetivos a cumplir. Teniendo además en cuenta que se trataba principalmente de un trabajo de investigación y una gran parte de la dedicación se iba a destinar a solucionar

problemas técnicos inesperados, la metodología que se ha seguido ha sido dividir el proyecto en hitos que se iban definiendo y cumpliendo conforme se conseguía implementar versiones mayormente funcionales de los casos de uso planeados. Cada hito se identifica con un grupo de tecnologías y se lleva a cabo como consecuencia del anterior. Por ello, la tercera parte de esta memoria, "Desarrollo del proyecto" se divide en tres secciones, correspondiente a las tres fases (hitos) principales en los que se ha trabajado durante el proyecto.

A lo largo del proyecto se ha intentado aprovechar la capacidad de muchas de las herramientas utilizadas para ser configuradas de forma declarativas mediante ficheros de configuración, agilizando el prototipado y la integración de cambios. Para el versionado de estos cambios, se han utilizado varios repositorios Git: uno privado para almacenar los archivos de código y documentación del proyecto, y uno adicional, creado bajo la organización GRyCAP para almacenar los archivos del programa "orchGPU" [9], desarrollado durante la última fase del proyecto. En cuanto a la comunicación, se ha usado Microsoft Teams para intercambiar ideas e informar del progreso frecuentemente al tutor y a la directora experimental del proyecto. Para resolver dudas sobre el funcionamiento de algunas herramientas, se ha recurrido tanto a la comunicación directa con los responsables, como a los *issues* en el repositorio del software correspondiente.

Parte 2. Estado del arte

En esta segunda parte, se hace una introducción a las herramientas y los conceptos que han hecho posible este proyecto. También se comenta brevemente el estado del arte en el que dicho proyecto se enmarca.

2.1 Herramientas y conceptos clave

2.1.1 Programación de sistemas cloud

La computación cloud permite acceder a recursos remotos de procesamiento y almacenamiento. Aunque su popularización se produjera a comienzos del siglo XXI, algunos de sus conceptos teóricos tomaron forma durante las décadas de los 50 y 60, época en la que científicos pioneros como John McCarthy [10] ya especulaban con la posibilidad de ofrecer recursos de cómputo como un servicio. Hoy día, la computación cloud se integra con otras soluciones para crear arquitecturas complejas que dan soporte a los procesos de negocios numerosas organizaciones en todo el mundo.

A pesar de su ubicuidad, la programación de sistemas cloud presenta varios desafíos al compararla con modelos de programación más tradicionales. En primer lugar, es un tipo de programación limitada por la entrada/salida (*I/O bound*), es decir, la comunicación entre distintos servicios a través de la red suele ser el principal factor que afecta al rendimiento, ya que es órdenes de magnitud más lenta que los accesos a almacenamiento local y los cálculos llevados a cabo por el procesador. Esta limitación obliga frecuentemente a usar técnicas de programación asíncrona, en la que un sistema puede ejecutar tareas adicionales mientras espera la (lenta) respuesta de otro sistema. Los lenguajes más utilizados en entornos cloud (Python, Go, Node.js, Rust...) suelen incorporar herramientas como *callbacks*, corutinas y promesas (ver figura 1) para facilitar el desarrollo de programas que implementen estas técnicas. Dichos lenguajes suelen trabajar a un alto nivel de abstracción (orientación a objetos, recolección de basura, tipado dinámico o inferencia de tipos, etc.) que permiten a los desarrolladores centrarse en la lógica de aplicación en lugar de en los detalles de implementación subyacentes.

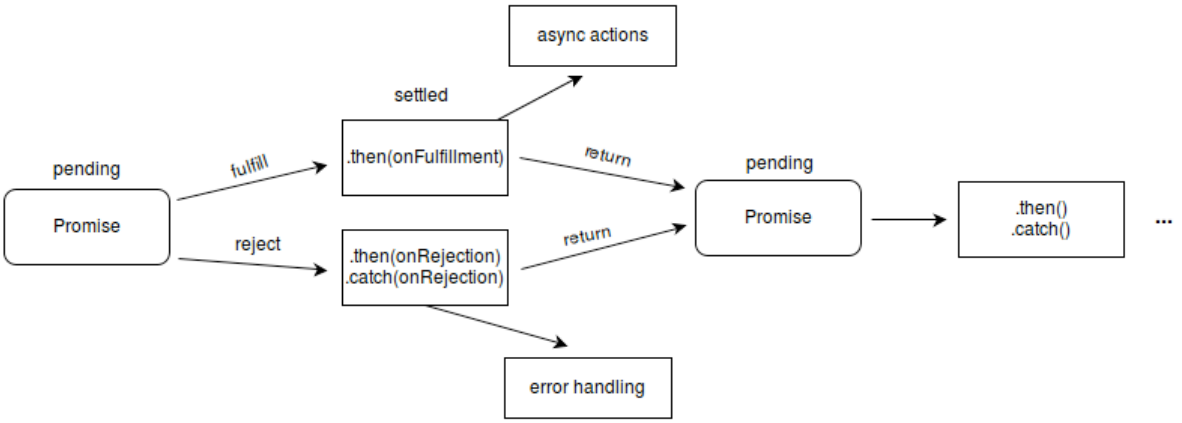

FIGURA 1 . FLUJO DE EJECUCIÓN DE UNA PROMESA
FUENTE: DOCUMENTACIÓN DE MOZILLA (HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE/GLOBAL_OBJECTS/PROMISE)

2.1.2 Serverless, FaaS y AWS Lambda

Dentro del ecosistema cloud se incluye el paradigma de la computación serverless (sin servidor), que propone un modelo de ejecución para aplicaciones en las que el usuario únicamente se encarga de la gestión a nivel de las funciones que las componen, es decir, sin prestar atención al aprovisionamiento de la infraestructura de cómputo subyacente.

Aunque debido a su corta edad, los términos serverless y FaaS suelen usarse indistintamente, generando cierto debate en torno a ello [11], se pueden considerar los servicios FaaS como los que forman el núcleo de la computación serverless y posibilitan el despliegue de aplicaciones siguiendo tal modelo. En un servicio FaaS, la unidad básica de ejecución son las funciones, cada una de las cuales representa la ejecución de una parte de la lógica interna de una aplicación. El usuario suele tener la opción de elegir un entorno de ejecución para sus funciones (se ofrecen para distintos lenguajes como Node.js, Python, Ruby, Java, Go o .NET Core [12]) o incluso implementar un entorno personalizado [13]. Una vez hecho esto, el usuario tan solo tiene que cargar el código de la función en el runtime y este lo ejecutará, generalmente en respuesta a eventos que ocurran en otras partes de la lógica de la aplicación [14]. Los servicios FaaS suelen ofrecer una facturación por tiempo de ejecución, escalado automático y características de seguridad, monitorización y *logging*. Con todo esto, los servicios FaaS pueden suponer una ventaja competitiva para sus usuarios, al evitar sobrecostes en aplicaciones con una demanda variable de recursos de cómputo y que requieren una mayor granularidad en su aprovisionamiento.

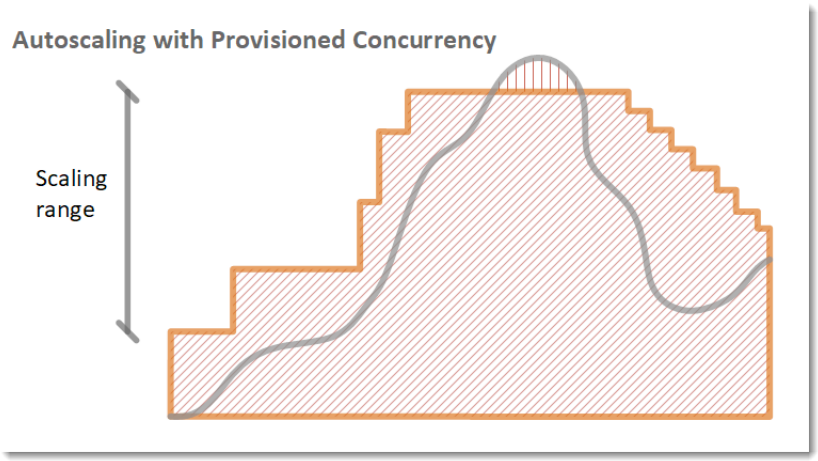

FIGURA 2 . APROVISIONAMIENTO ADAPTADO A UNA CARGA DE TRABAJO VARIABLE
FUENTE: DOCUMENTACIÓN DE AWS (HTTPS://DOCS.AWS.AMAZON.COM/LAMBDA/LATEST/DG/CONFIGURATION-CONCURRENCY.HTML)

AWS Lambda es el servicio FaaS de Amazon Web Services, que además es líder en un mercado en el que compite con Google Cloud Functions [15] y Azure Functions [16]. Su atractivo reside en características como la integración con otros servicios de AWS o la facturación contabilizada en milisegundos [17]. Además, Lambda permite optimizar el tiempo de ejecución de las funciones eligiendo entre varias configuraciones de memoria y CPUs virtuales. Cada función se ejecuta de forma aislada al resto, utilizando tecnologías similares a las usadas en contenedores y exponiendo solamente el directorio /tmp al usuario [18]. Actualmente, AWS Lambda impone varios límites a la ejecución de las funciones como un tamaño máximo de 512 MB para el mencionado directorio /tmp, 15 minutos de tiempo de ejecución máximo, o entre 128 y 10240 MB de memoria. Es interesante saber que la capacidad de cómputo de cada función se asigna automáticamente según la cantidad de memoria elegida para la misma, por ejemplo, a 1769 MB de memoria le corresponde el equivalente a 1 vCPU [19]. Este escalonamiento de los recursos de cómputo disponibles permite a los usuarios de Lambda buscar la configuración de recursos óptima en relación a los tiempos de ejecución y al coste.

El modelo de programación se AWS Lambda se basa en que las funciones sean *stateless* (no tengan estado interno) por lo que para recibir y almacenar datos, las funciones deben tener contacto con fuentes externas y otros servicios de AWS. Como se ha mencionado anteriormente, Lambda no cuenta con soporte nativo para ejecutar cargas de trabajo en GPU, sin embargo, en 2020 AWS anunció la introducción del soporte para contenedores [20], aumentando considerablemente la flexibilidad del servicio y reduciendo el coste del cambio a organizaciones cuyos flujos de trabajo ya estén basados en contenedores. Este soporte está sujeto a ciertas limitaciones, como que las imágenes para los contenedores deben estar alojadas en Amazon ECR en vez de Docker Hub. En este proyecto no se ha hecho uso del soporte para contenedores de Lambda en favor de colaborar con la investigación [21] de

herramientas *open-source* desarrolladas por GRyCAP como SCAR [22]. Las diferencias entre ambas opciones serán discutidas más adelante.

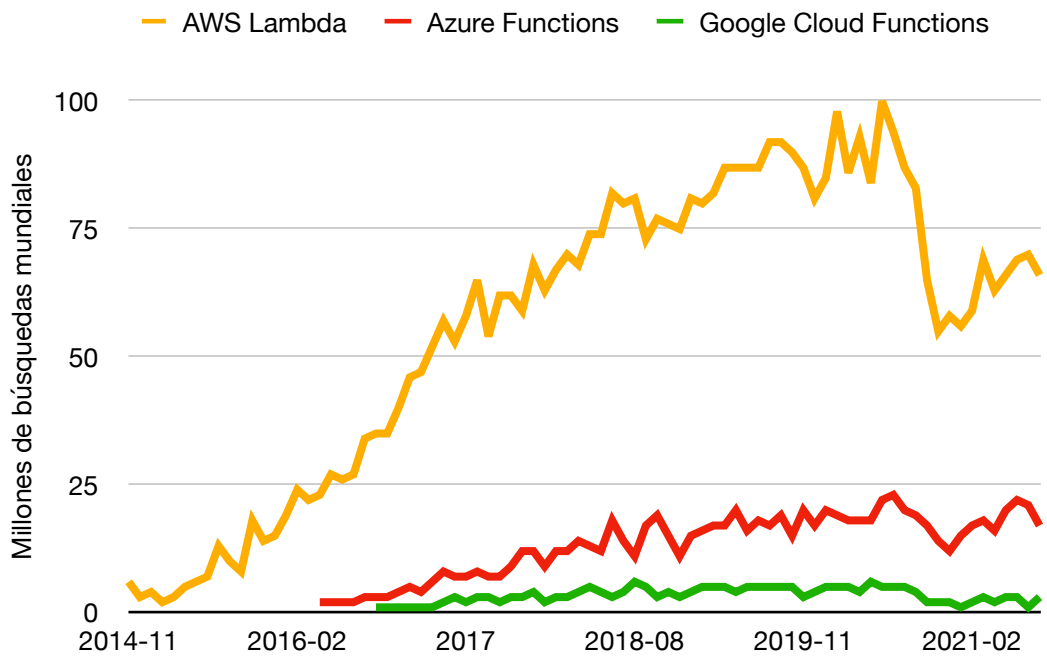

FIGURA 3 . POPULARIDAD DE LOS GRANDES SERVICIOS FAAS
FUENTE: PROPIA CON DATOS DE GOOGLE TRENDS

A priori, puede parecer que existe cierto solapamiento entre la funcionalidad de algunos de los más de 200 servicios ofrecidos por AWS, especialmente entre los relacionados con cómputo, como es el caso de Lambda. Es importante tener en cuenta que, aunque en algunos casos las tecnologías que soportan a los servicios sean parecidas, cada servicio cuenta con una filosofía única orientada a un conjunto de casos de uso concretos. A continuación, se resumen las características de algunos servicios de cómputo y contenedores que suelen ser fuente de confusión entre los usuarios:

- **Amazon EC2** (Elastic Compute Cloud) [23] es uno de los servicios básicos de AWS. Proporciona computación bajo demanda en forma de máquinas virtuales completamente configurables por el usuario.
- **Amazon ECS** (Elastic Container Service) es el servicio de orquestación de contenedores gestionado de AWS. Puede usar EC2 o Fargate como proveedores de cómputo, dependiendo del grado de gestión que quiera asumir el usuario.
- **AWS Fargate** [24] se utiliza para proporcionar servidores de cómputo completamente gestionado a otros servicios.
- **Amazon EKS** (Elastic Kubernetes Service) es el servicio para la creación de clústeres Kubernetes [25] gestionados. Al igual que ECS, soporta EC2 y Fargate.

- **Amazon ECR** (Elastic Container Registry) [26] es el registro de imágenes de contenedor de AWS.
- **AWS Batch** [27] facilita la ejecución de trabajos de procesamiento por lotes (*batch jobs*) en otros servicios como EC2 o Fargate sin coste añadido.
- **AWS CloudFormation** [28] es el servicio de infraestructura como código de AWS, permitiendo desplegar colecciones de recursos en AWS a partir de plantillas.
- **AWS Elastic Beanstalk** [29] está orientado a la creación sencilla de servicios web usando tecnologías populares.
- **Amazon Lightsail** [30] ofrece una experiencia simplificada para usar servidores privados virtuales (VPS) en la nube.
- **AWS Lambda** ofrece computación serverless para cargas de trabajo que requieran la ejecución de funciones breves y de forma completamente gestionada. Desde 2020, cuenta con soporte para la ejecución de contenedores bajo las mismas condiciones que sus funciones.

FIGURA 4 . LOGO DE AWS LAMBDA
FUENTE: DOMINIO PÚBLICO (HTTPS://EN.WIKIPEDIA.ORG/WIKI/AWS_LAMBDA#/MEDIA/FILE:AMAZON_LAMBDA_ARCHITECTURE_LOGO.PNG)

2.1.3 Contenedores, Docker y Kubernetes

Los contenedores [31] son una técnica de virtualización a nivel del sistema operativo que permite ejecutar procesos de manera aislada al resto de procesos del sistema. Este aislamiento se consigue para los varios recursos disponibles para el proceso (sistema de ficheros, *networking*, etc.) utilizando herramientas software proporcionadas por el propio sistema operativo. Los contenedores contienen los archivos de una aplicación y las dependencias estrictamente necesarias para su ejecución, que suele ser *stateless*, lo que permite a los contenedores ser replicados, terminados y levantados regularmente en un entorno de producción sin afectar al funcionamiento del servicio. Estas características han provocado que los contenedores sean una de las tecnologías más populares en la industria [32]. Comparados con las máquinas virtuales, los contenedores ofrecen una solución más ligera y fácil de distribuir (figura 5).

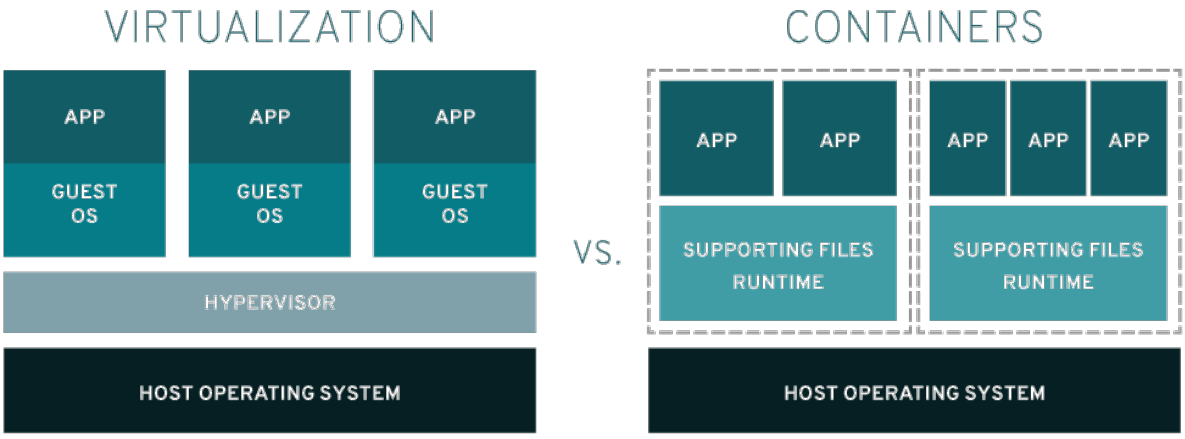

FIGURA 5 . CONTENEDORES VS MÁQUINAS VIRTUALES
FUENTE: RED HAT (HTTPS://WWW.REDHAT.COM/ES/TOPICS/CONTAINERS/CONTAINERS-VS-VMS)

Docker [33] es una plataforma de software *open-source* para la gestión de contenedores. Está disponible para varios sistemas operativos, aunque debido a que hace uso de varias herramientas del *kernel* Linux para conseguir el aislamiento entre contenedores (*chroot*, *cgroups*, *namespaces*) [34], requiere una capa adicional de virtualización para funcionar en otros sistemas operativos, así como permisos de *root* cuando se ejecuta de forma nativa sobre distribuciones Linux. Docker facilita la transferencia y despliegue de los contenedores mediante el concepto de "imagen". Estas imágenes almacenan el estado inicial de la ejecución de un contenedor y pueden ser creadas mediante archivos "Dockerfile", en los que se especifica dicho estado mediante una secuencia comandos Docker. Las imágenes de contenedores se suelen distribuir desde repositorios públicos, siendo el más conocido Docker Hub [35].

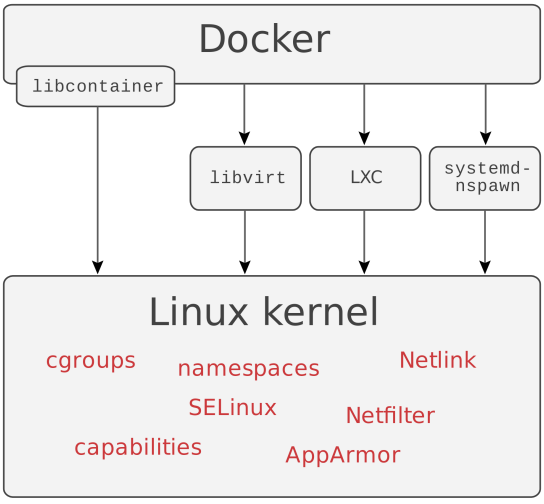

FIGURA 6 . INTERACCIÓN DE DOCKER CON EL KERNEL LINUX
FUENTE: WIKIPEDIA (HTTPS://EN.WIKIPEDIA.ORG/WIKI/DOCKER_(SOFTWARE)#/MEDIA/FILE:DOCKER-LINUX-INTERFACES.SVG)

Al usar contenedores para implementar la arquitectura de aplicaciones cada vez más complejas, surge la necesidad de contar con un *middleware* que se ocupe de gestionar varios aspectos del ciclo de vida de los mismos. Este tipo de *middleware* se conoce como software de orquestación de contenedores y el nombre más destacado en este área es Kubernetes [36], surgido como una implementación abierta de las ideas usadas por Google para su propio orquestador, Borg [37].

Kubernetes se ha convertido en una parte esencial del *stack* tecnológico de muchas empresas ya que proporciona características de resolución de nombres (DNS), balanceo de carga, seguridad, monitorización, auto-escalado, etc. Un clúster de Kubernetes en producción debe contar con un mínimo de tres nodos (físicos o virtuales) para resolver votaciones por mayoría, pero puede escalar a varios miles si es necesario [38]. A uno de los nodos se le asigna el rol de nodo principal y es el responsable de exponer la API del clúster al exterior (es a esta API donde se conecta el cliente de línea de comandos de Kubernetes, *kubectl*). El resto de nodos albergan los contenedores desplegados sobre un *runtime*, que puede ser Docker, aunque se encuentra sin soporte desde 2020 en favor de otros como containerd o CRI-O [39]. A pesar de su utilidad, Kubernetes posee cierta curva de aprendizaje debido a que introduce un nivel de abstracción adicional sobre la infraestructura y numerosos conceptos que los administradores deben conocer para poder gestionar un clúster de forma eficaz. Aun así, muchos estudios muestran altas tasas de adopción por parte de la industria [40], convirtiendo a Kubernetes en el estándar de facto para la orquestación de contenedores.

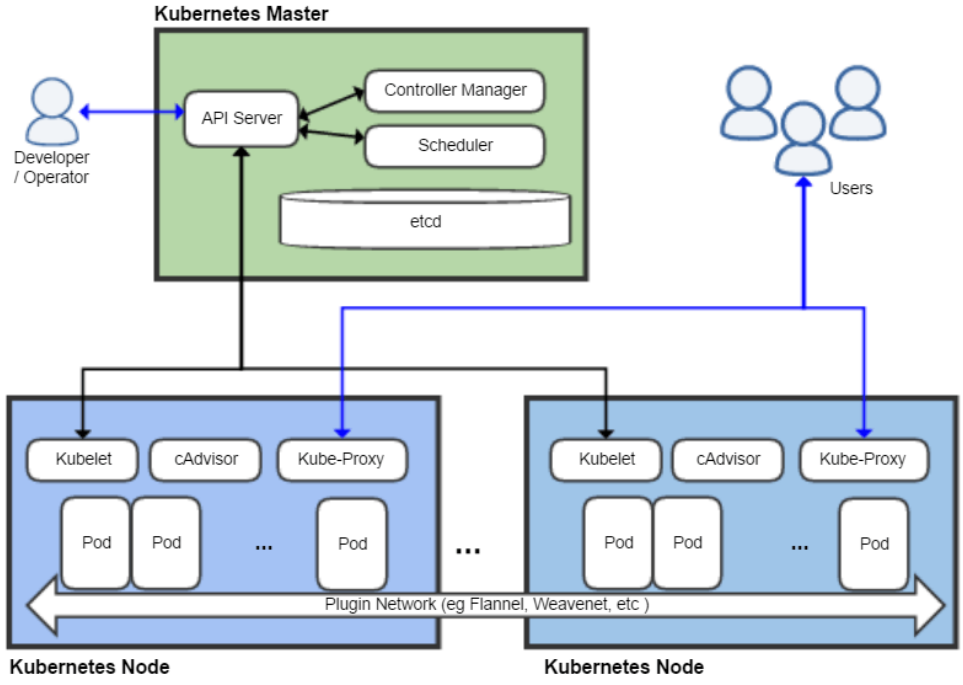

FIGURA 7 . ARQUITECTURA DE KUBERNETES
FUENTE: WIKIPEDIA (HTTPS://EN.WIKIPEDIA.ORG/WIKI/KUBERNETES#/MEDIA/FILE:KUBERNETES.PNG)

2.1.4 SCAR y OSCAR

SCAR (Serverless Container-aware ARchitectures) es una herramienta de línea de comandos (CLI) de código abierto desarrollada por el grupo de investigación GRyCAP para ejecutar contenedores Docker sobre AWS Lambda cuyo desarrollo comenzó en 2017. Esto permite aprovechar la inmensa cantidad de imágenes Docker disponibles actualmente de forma pública para su ejecución en un entorno serverless para crear flujos de trabajos dirigidos por eventos y orientados al procesamiento de archivos, en los que un archivo que se transfiere a un bucket de Amazon S3 desencadena la ejecución de una instancia de la función conectada a dicho bucket. La función procesa entonces el archivo y almacena el resultado en el bucket que se haya designado como salida. SCAR cuenta con características adicionales que pueden resultar muy útiles en ciertos flujos de trabajo, como la capacidad de modificar el comportamiento de cada invocación adjuntando un script, o la posibilidad de integrar un *endpoint* HTTP con una función de una manera muy sencilla.

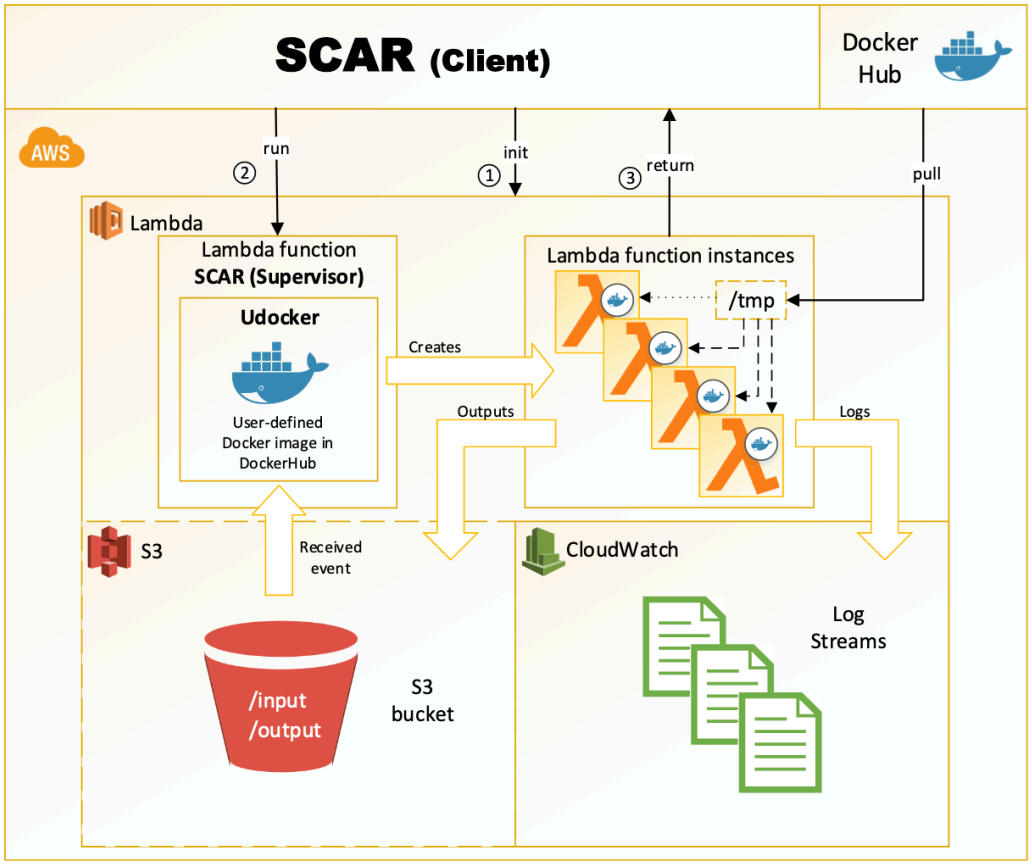

FIGURA 8 . ARQUITECTURA DE SCAR
FUENTE: PÉREZ ET AL., 2019. SERVERLESS COMPUTING FOR CONTAINER-BASED ARCHITECTURES. HTTPS://DOI.ORG/10.1016/J.FUTURE.2018.01.022

La manera habitual de proceder con SCAR es crear las funciones en Lambda especificando sus parámetros de configuración mediante los *flags* del comando correspondiente o de forma declarativa usando un lenguaje de descripción de funciones en un fichero de configuración YAML [41]. Esta última forma es mucho más recomendable para tener reproducibilidad y capacidad de realizar cambios fácilmente.

En la sección 2.1.2 se comentó que AWS ha incluido el soporte nativo para contenedores en Lambda. Es lógico preguntarse qué diferencia a la propuesta de SCAR ahora que pueden ejecutarse contenedores en Lambda de una manera sencilla y soportada oficialmente por el proveedor. Como parte del proyecto de la asignatura "Infraestructuras de cloud público" de este máster, se identificaron algunas diferencias que pueden llegar a ser significativas y que se explican de forma resumida a continuación:

- Las imágenes de SCAR están sujetas al tamaño máximo que Lambda permite para el directorio /tmp, que actualmente es de 512 MB. El soporte nativo para contenedores del servicio permite ejecutar imágenes de hasta 10 GB.
- Para ejecutarse en SCAR la lógica de aplicación debe estar adaptada al modelo del programación de la herramienta. Para hacerlo de forma nativa en Lambda, se debe hacer uso de su *Runtime API*. Dependiendo del funcionamiento del código existente, puede ser más recomendable elegir una opción u otra.
- Lambda solo permite descargar imágenes en ECR, mientras que SCAR permite hacerlo desde Docker Hub.
- Los tiempos de ejecución suelen ser más largos en SCAR, especialmente en la primera de las ejecuciones, ya que es necesario descargar la imagen de contenedor. Esto puede traducirse en un mayor coste.

```
ubuntu@rcuda-client:~/prueba_scar$ scar init -f scar-yolov3.yaml -t 900
Using latest supervisor release: '1.4.0'.
Using existent 'faas-supervisor' layer.
Creating function package.
Function 'alucloud24-scar-yolov3' successfully created.
Log group '/aws/lambda/alucloud24-scar-yolov3' successfully created.
ubuntu@rcuda-client:~/prueba_scar$ scar ls
AWS FUNCTIONS:
NAME                    MEMORY    TIME  IMAGE_ID                    API_URL    SUPERVISOR_VERSION
----------------------  --------  ------  --------------------------  ---------  --------------------
alucloud24-scar-yolov3       512     900  mrconrui/oscar-test:tiny    -          1.4.0
ubuntu@rcuda-client:~/prueba_scar$
```

FIGURA 9 . INICIALIZACIÓN DE UNA FUNCIÓN CON EL CLI DE SCAR
FUENTE: PROPIA

OSCAR (Open Source Serverless Computing for Data-Processing Applications) [42] es una herramienta que implementa el modelo de ejecución de funciones serverless en clústeres *on premises*, de manera similar a SCAR actuando sobre AWS. Cuenta con una interfaz web, así como una herramienta de línea de comandos, que es la que se ha usado principalmente en este proyecto, ya que admite el despliegue de funciones a partir de un fichero YAML como ocurre con SCAR. Para poder desplegar clústeres elásticos, OSCAR hace uso de varias tecnologías de código abierto desarrolladas por GRyCAP:

- **EC3** (Elastic Cloud Computing Cluster) [43], una herramienta para el despliegue de clústeres elásticos.
- **IM** (Infrastructure Manager) [44], una herramienta de provisión de infraestructura *multi-cloud*.
- **CLUES** (Cluster Energy Saving) [45], un gestor de recursos que escala horizontalmente los nodos del clúster de Kubernetes.

Una vez desplegado el clúster, OSCAR le proporciona capacidades de procesamiento de archivos mediante sus propios binarios, a la vez que también despliega MinIO [46] para obtener almacenamiento orientado a objetos y OpenFaaS [47] para crear funciones a partir de peticiones HTTP entrantes. OSCAR también puede utilizar los siguientes servicios externos de almacenamiento:

- **Amazon S3** (Simple Storage Service) [48], el servicio de almacenamiento orientado a objetos de Amazon Web Services.
- **MinIO**, un sistema de almacenamiento compatible con la API de S3 que OSCAR puede desplegar dentro del propio clúster.
- **Onedata**, la solución del almacenamiento usada por la infraestructura federada europea EGI Federated Cloud [49].

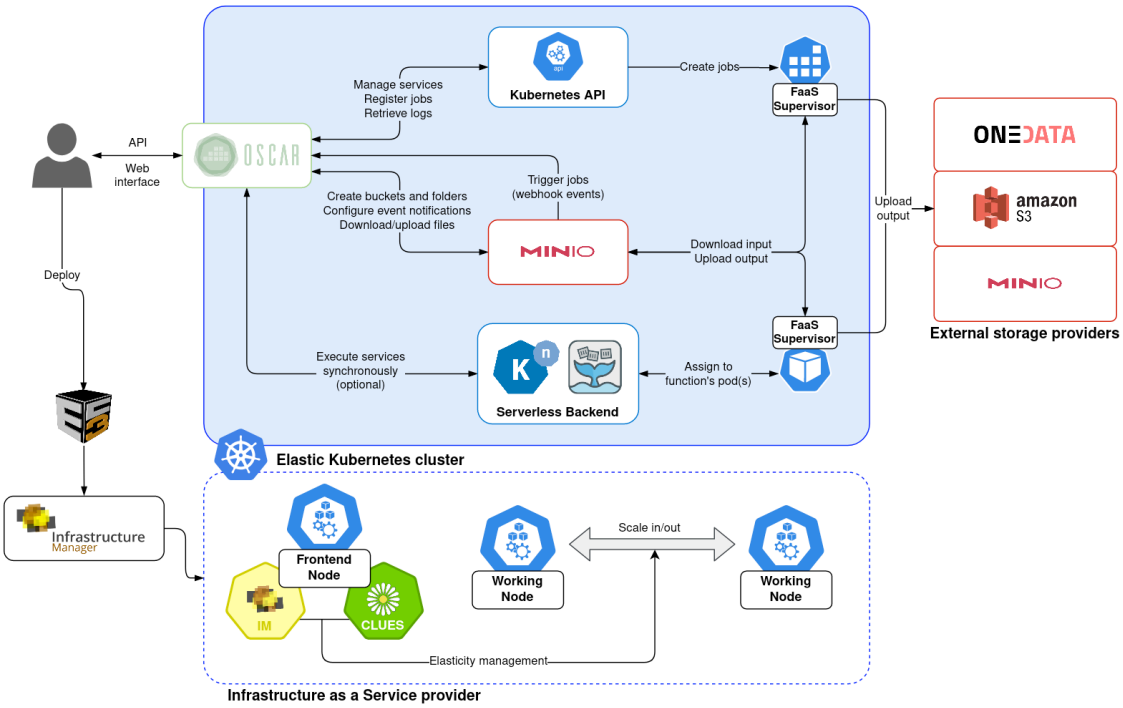

FIGURA 10 . ARQUITECTURA DE OSCAR
FUENTE: DOCUMENTACIÓN DE OSCAR (HTTPS://GRYCAP.GITHUB.IO/OSCAR/)

En el pasado, OSCAR se ha usado exitosamente en pruebas de concepto para la creación de plataformas serverless de computación acelerada mediante GPU [50].

2.1.5 Programación GPU y rCUDA

Un paradigma computacional que ha tenido también un gran éxito en los últimos años es la computación de propósito general sobre procesadores gráficos (abreviado como GPGPU, *general-purpose computing on graphics processing units*) [51]. Existen determinados procesos dentro del ámbito de la computación científica que se benefician enormemente de las características de las GPU. Estos dispositivos presentan un alto grado de paralelismo basada en miles de procesadores que, a pesar de ser más simples y tener menor frecuencia de reloj que los de una CPU convencional, pueden realizar ciertas operaciones sobre estructuras de datos de forma simultánea, lo que se traduce en un rendimiento muy superior.

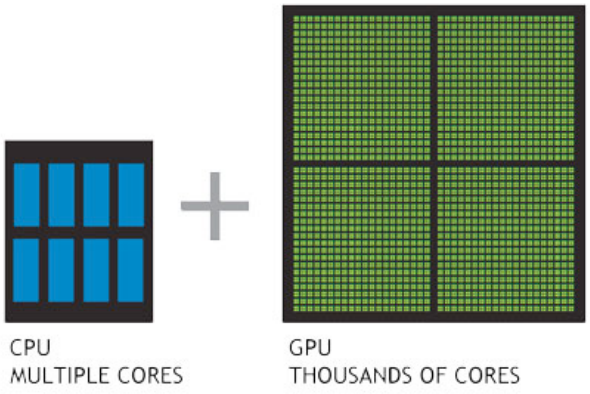

FIGURA 11 . NÚCLEOS DE PROCESAMIENTO EN CPU Y GPU
FUENTE: NVIDIA (HTTPS://WWW.NVIDIA.COM/ES-LA/DRIVERS/WHAT-IS-GPU-COMPUTING/)

Sin duda, uno de los campos que más ha popularizado el uso de dispositivos GPU para su propio beneficio es el aprendizaje profundo o *deep learning*, una disciplina dentro del aprendizaje automático (*machine learning*) que estudia el uso de redes neuronales artificiales para el reconocimiento y predicción de patrones en conjuntos de datos de diversa índole. El ajuste de dichas redes neuronales a los datos requiere un proceso iterativo de entrenamiento en el que se realizan numerosas operaciones de álgebra lineal sobre sus parámetros. Es este tipo de operaciones en las que una GPU tiene una gran ventaja con respecto a los procesadores CPU. Tal es el éxito de la computación GPGPU en este campo, que los fabricantes han empezado a ser conscientes de que añadir características orientadas al *deep learning* puede incrementar la demanda de sus productos e incluyen en sus diseños procesadores exclusivamente dedicados a aumentar el paralelismo en las operaciones más frecuentes del entrenamiento y la inferencia con redes neuronales. Este es el caso de los Tensor Cores [52] de las GPUs de Nvidia.

FIGURA 12 . NVIDIA TESLA V100, GPU PARA DATA CENTER
FUENTE: NVIDIA (HTTPS://WWW.NVIDIA.COM/ES-ES/DATA-CENTER/TESLA-V100/)

Un gran inconveniente que apareció en los primeros experimentos con GPU de propósito general es el hecho de que los algoritmos debían traducirse a las primitivas de programación necesarias para hacer uso de los núcleos de procesamiento gráfico. Para solucionar este problema, surgieron varios lenguajes de programación y plataformas dedicadas, siendo la propuesta más exitosa la de Nvidia, CUDA [53]. El núcleo de dicha propuesta son los lenguajes CUDA C/C++ y CUDA Fortran, que extienden a sus respectivos lenguajes de origen con operaciones dedicadas para la programación GPGPU, como transferencia de datos o creación y planificación de hilos de procesamiento. Para ejecutar un programa en CUDA, es necesario transformar una o varias de sus funciones en *kernels*, el término con el que CUDA denomina a las funciones que pueden ser ejecutadas mediante varios hilos en paralelo en la GPU. Los datos necesarios para la ejecución de un *kernel* deben ser transferidos a alguna de las memorias del dispositivo, que se organizan de forma jerárquica (privada, compartida y global). Adicionalmente, la plataforma contiene APIs de bajo y alto nivel, así como varias librerías de algoritmos. La compatibilidad entre las distintas versiones de CUDA y los dispositivos GPU del fabricante viene dada por un identificador (*Compute Capability*) que el fabricante asigna a cada nueva gama de productos, dependiendo de la arquitectura de su hardware.

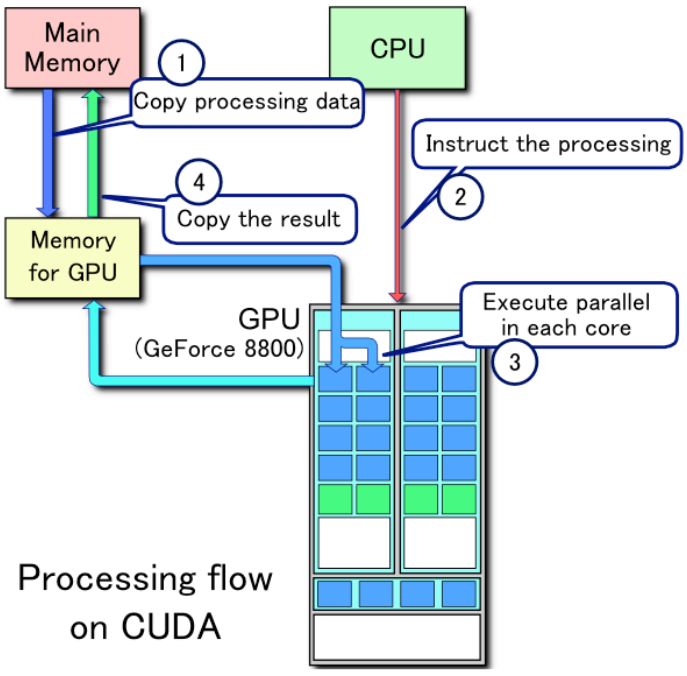

FIGURA 13 . FLUJO DE EJECUCIÓN DE CUDA
FUENTE: WIKIPEDIA (HTTPS://EN.WIKIPEDIA.ORG/WIKI/CUDA#/MEDIA/FILE:CUDA_PROCESSING_FLOW_(EN).PNG)

rCUDA es un *framework* que permite virtualizar GPUs remotas para la ejecución de programas CUDA de una manera completamente transparente al programador (no es necesario modificar el código fuente de los programas escritos para CUDA). rCUDA está activamente desarrollado por el grupo de Arquitecturas Paralelas de la Universitat Politècnica de València y, al igual que CUDA, se distribuye bajo licencia propietaria. Es compatible con la mayoría de las operaciones y APIs de CUDA y permite comunicaciones con nodos remotos tanto mediante TCP/IP (Ethernet) como RDMA (Infiniband o RoCE) [54], siguiendo una arquitectura cliente-servidor.

La configuración de rCUDA es relativamente sencilla. Tan sólo es necesario establecer la configuración en los nodos cliente y servidor, y arrancar los procesos del servidor de comunicaciones y de licencia. Al igual que CUDA, rCUDA permite usar uno o varios dispositivos GPU. La principal ventaja de rCUDA radica en que permite el uso simultáneo (compartición) de un mismo dispositivo GPU por parte de múltiples aplicaciones. En la figura 14 se muestra la arquitectura del *framework*.

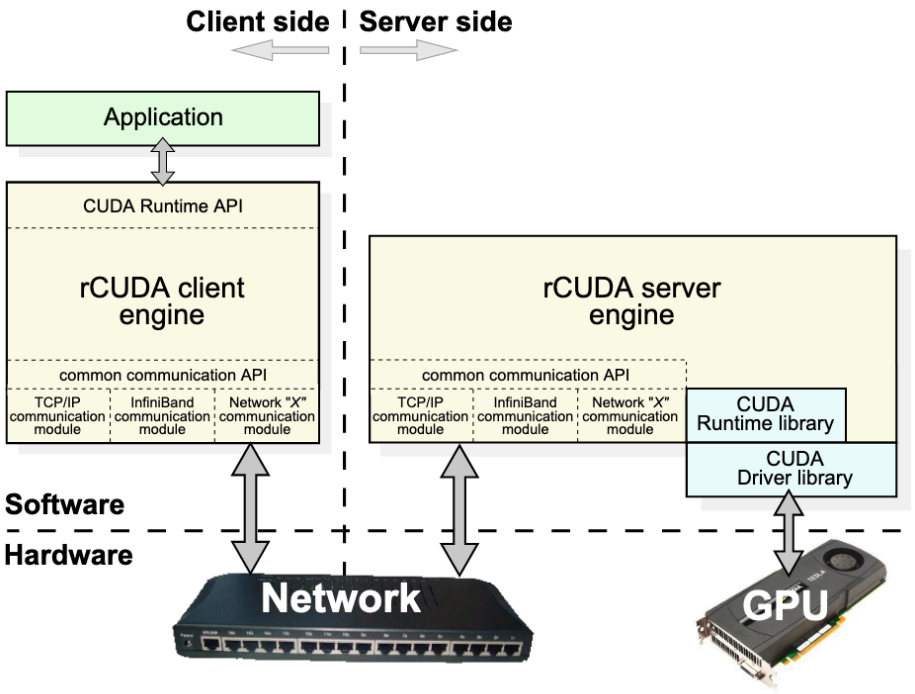

FIGURA 14 . ARQUITECTURA DE RCUDA
FUENTE: DOCUMENTACIÓN DE RCUDA (HTTP://RCUDA.NET/PUB/RCUDA_GUIDE.PDF)

2.2 Estado del arte actual

Como ya se ha comentado, AWS no cuenta actualmente con soporte nativo para GPUs en su servicio FaaS, AWS Lambda. Aún más sorprendente es que sus competidores directos, Google Cloud Functions y Azure Functions tampoco mencionan el soporte para dichos dispositivos en sus respectivas páginas de inicio. Esto coincide con la experiencia de otros usuarios [55] que acusan la falta de soporte para integrar el procesamiento GPU en la nube con sus aplicaciones. Aunque no parece haber una explicación clara al respecto, es posible que los proveedores no hayan incluido esta característica para diferenciar más claramente sus servicios. También es posible, aunque más improbable, que consideren que no existe suficiente cantidad o variedad de casos de uso como para justificar la inversión. En todo caso, en una industria que evoluciona tan rápido, es razonable pensar que el soporte para GPUs pueda llegar en un futuro a los servicios mencionados.

Por otro lado, es posible encontrar algunos proyectos para transferir cargas de trabajo a GPUs simulando el modelo serverless. Este es el caso de:

- PyPlover [56], una plataforma desarrollada por investigadores de UC Berkeley que permite el envío de *kernels* para su ejecución en GPU en una plataforma que imita el funcionamiento interno de un proveedor de servicios serverless. Al recibir un *kernel,* junto con la definición de la función y los datos de entrada, el *backend* de PyPlover levanta los contenedores necesarios y distribuye la carga entre los mismos utilizando OpenLambda [57]. Los autores consideran que la arquitectura serverless de PyPlover ofrece mayor tolerancia a fallos que rCUDA, ya que implementa mecanismos de timeout.
- Kuda [58], una herramienta desarrollada por Cyril Diagne para desplegar APIs sobre contenedores con acceso a GPUs desplegados en Knative [59], una plataforma para la ejecución serverless de contenedores sobre Kubernetes.

Parte 3. Desarrollo del proyecto

En esta parte de la memoria, se describen los detalles de implementación de las soluciones desarrolladas a lo largo del proyecto. Además, se muestran los resultados alcanzados desplegando dichas soluciones.

3.1 Fase 1 - Despliegue local

3.1.1 Diseño y arquitectura

La primera fase del proyecto se centró en adquirir familiaridad con rCUDA y conseguir hacer funcionar una prueba de concepto en un entorno controlado y con una arquitectura poco compleja, pero que contara con elementos que se reutilizarían en fases posteriores. Persiguiendo este objetivo, se planteó un escenario en el que un modelo de aprendizaje profundo realizaría un proceso de inferencia sobre una imagen, comunicándose con una GPU situada en un nodo diferente. Para evitar incurrir en costes innecesarios, los dos nodos implicados serían máquinas virtuales alojadas en clústeres on-premise a disposición del GRyCAP. Para simular mejor los escenarios posteriores, se decidió seguir la propuesta de la documentación de SCAR para *testing* local [60] y hacer uso de udocker para ejecutar el modelo dentro de un contenedor. Como se mencionó en la introducción, Docker requiere privilegios de *root* para ejecutarse sobre el *kernel* Linux. udocker [61] es una herramienta *open-source* que permite ejecutar contenedores Docker en espacio de usuario, sin necesidad de obtener permisos de administrador (y por lo tanto, perdiendo una parte de la funcionalidad total de Docker). El diagrama de arquitectura para esta primera fase es el que se muestra a continuación.

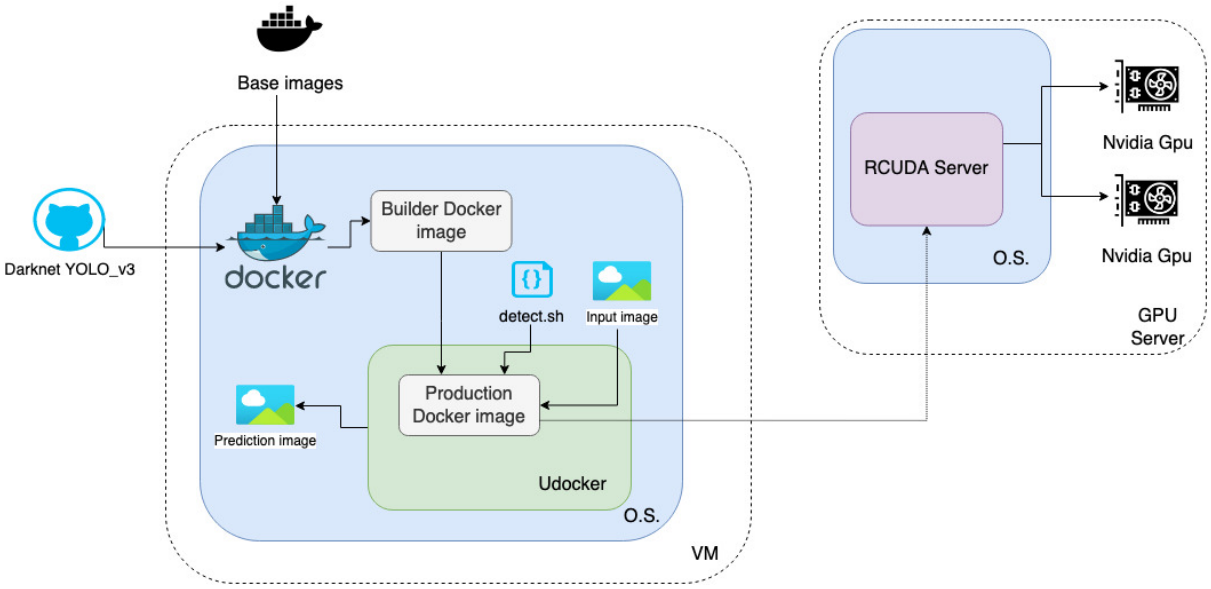

FIGURA 15 . DIAGRAMA FASE 1
FUENTE: PROPIA

En el diagrama se muestra la máquina virtual que contiene udocker y ejecuta el contenedor con el modelo de *machine learning* y la máquina virtual que ejecuta el servidor de rCUDA. Sobre dicho diagrama cabe mencionar que:

- Se han incluido varias GPUs por generalidad, en la práctica solo había un dispositivo.
- El servidor de rCUDA se ejecutaba dentro de un contenedor creado a partir de una imagen nvidia-docker [62] para conseguir virtualización de los dispositivos GPU físicos. La máquina sobre la que se ejecutaba dicho contenedor tenía configurada la redirección de puertos para que esto fuera transparente a las conexiones desde el exterior, por lo que no se ha mostrado en el diagrama.
- No se muestra el servidor de licencia de rCUDA, ya que se encontraba en un tercer nodo y apenas aporta al flujo de ejecución.

3.1.2 Implementación, despliegue y resultados

Una vez obtenidos los binarios de rCUDA y configurado el entorno de desarrollo, la mayor parte del esfuerzo se centró en crear el Dockerfile para la imagen que contendría el modelo de *deep learning*. Contar con un Dockerfile correcto era crítico en este punto, ya que proporcionaría una base estable sobre la que poder hacer cambios fases posteriores. Se decidió usar un *multi-stage build* [63], que permite repartir las tareas del proceso de construcción entre varias imágenes base, de forma que la imagen resultante contiene únicamente los ficheros necesarios para la ejecución, sin las dependencias intermedias. Las tareas contenidas en el Dockerfile se pueden resumir como sigue:

1. **Partir** de una imagen base de CUDA 9.0 desarrollada por Nvidia a partir de Ubuntu 16.04. Las imágenes basadas en versiones más recientes de Ubuntu no incluyen por defecto versiones compatibles con Cuda 9.0 de algunas librerías de compilación.
2. **Instalar** en dicha imagen utilidades como git y wget.
3. **Clonar** el repositorio git de la red YOLOv3: un popular modelo de aprendizaje profundo construido sobre el *framework* Darknet [64], que permite la detección de objetos en imágenes.
4. **Descargar** los pesos pre-entrenados para YOLOv3 usando wget.
5. **Compilar** CUDA con soporte para GPU, activando unos flags en el Makefile.
6. **Copiar** los archivos de CUDA a una nueva imagen base de menor tamaño. Esta vez desarrollada por Bitnami como una versión reducida de Debian [65] denominada "minideb".
7. **Instalar** algunas librerías que rCUDA tiene como dependencia.
8. **Copiar** los archivos de CUDA y YOLOv3 a la nueva imagen.
9. **Copiar** los archivos de rCUDA desde el almacenamiento local y establecer las variables de entorno correspondientes.

```
FROM nvidia/cuda:9.0-cudnn7-devel-ubuntu16.04 AS builder

RUN apt-get update -y && apt-get install -y git wget

RUN git clone --depth 1 --branch Yolo_v3 https://github.com/AlexeyAB/darknet.git
WORKDIR darknet

RUN sed -i "/compute_70/s/^#//g" Makefile
RUN sed -i "s/GPU=0/GPU=1/g" Makefile
RUN sed -i "s/CUDNN=0/CUDNN=1/g" Makefile
RUN make EXTRA_NVCCFLAGS=--cudart=shared
RUN wget https://pjreddie.com/media/files/yolov3.weights

FROM bitnami/minideb:jessie
RUN apt-get update && apt-get install -y libibverbs-dev
COPY --from=builder /darknet/ /darknet/
COPY bandwidthTest /bandwidthTest

run mkdir /rCUDA/
ADD rCUDA/ /rCUDA/
ENV PATH $PATH:/usr/local/cuda/bin
ENV LD_LIBRARY_PATH /rCUDA/lib
ENV RCUDA_DEVICE_COUNT 1
ENV RCUDA_DEVICE_0
```

FIGURA 16. DOCKERFILE GENERADO
FUENTE: PROPIA

El principal problema a la hora de crear el Dockerfile fue encontrar una combinación de versiones de Ubuntu, CUDA y librerías que fueran compatibles entre sí y con rCUDA. Una vez solucionado esto, el desarrollo fue una cuestión de iterar sobre los comandos para pulir errores derivados del hecho que se estaban ejecutando sobre un contenedor Docker y sin intervención manual. Por otro lado, también fue necesario crear un script bash conteniendo el comando de ejecución de Darknet para realizar la inferencia. Este script también se reutiliza para crear la función SCAR al hacer pruebas en AWS más adelante.

```
#!/bin/sh

cd /tmp

FILE_NAME=`basename $INPUT_FILE_PATH`
OUTPUT_FILE=$TMP_OUTPUT_DIR/$FILE_NAME

ln -s /darknet/cfg .
ln -s /darknet/data .
ln -s /darknet/yolov3.weights .

/darknet/darknet detect /darknet/cfg/yolov3.cfg /darknet/yolov3.weights $INPUT_FILE
_PATH

rm -rf cfg data yolov3.weights
```

FIGURA 17. SCRIPT GENERADO
FUENTE: PROPIA

Las pruebas realizadas usando el Dockerfile y script desarrollados se llevaron a cabo de forma relativamente exitosa: el proceso de inferencia da como resultado imágenes en las que el modelo consigue identificar los elementos que la componen, aunque se detectó cierta inconsistencia entre los elementos reconocidos en pruebas sucesivas con la misma imagen. Esto es más probable que se deba al funcionamiento/configuración del modelo de inferencia que al uso de una GPU remota, por lo que no se ha investigado más al respecto. En cuanto al tiempo de ejecución total, el uso de una GPU remota es considerablemente más lento que el de la CPU local debido al ancho de banda limitado que proporciona la conexión TCP/IP (en la sección 3.4 se hace una comparación de rendimiento más completa en la que se aprecia esta diferencia). Este resultado concuerda con las advertencias en la documentación de rCUDA, que aconseja que se use una tecnología de comunicación como Infiniband [66] o similar para obtener tiempos de ejecución razonables.

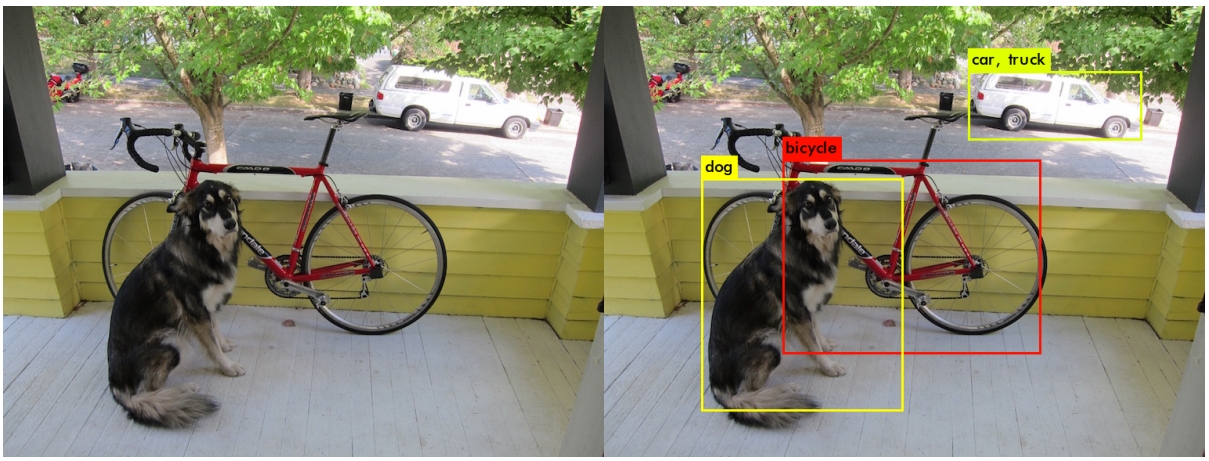

FIGURA 18. RESULTADO DE LA INFERENCIA USANDO YOLO_V3
FUENTE: PROPIA A PARTIR DE UNA IMAGEN DE MUESTRA INCLUIDA CON EL MODELO

```
/darknet/data/dog.jpg: Predicted in 44.219111 seconds.
dog: 81%    (left_x:  124   top_y:  220   width:  258   height:  298)
bicycle: 38%    (left_x:  229   top_y:  196   width:  330   height:  249)
car: 71%    (left_x:  465   top_y:   83   width:  222   height:   89)
truck: 42%
```

FIGURA 19. INFERENCIA DEL MODELO PARA LA IMAGEN ANTERIOR
FUENTE: PROPIA

3.2 Fase 2- Despliegue en OSCAR

3.2.1 Diseño y arquitectura

La continuación lógica tras la primera fase de desarrollo era investigar el comportamiento del servidor de rCUDA ante múltiples llamadas a la GPU remota, para lo que se creó un clúster OSCAR como entorno de pruebas. Como se comentó en la sección 2.4, OSCAR permite llevar el modelo de procesamiento orientado a ficheros de SCAR a un clúster elástico sobre Kubernetes. En este caso, se eligió desplegar el clúster sobre la infraestructura *on-premises* disponible y gestionarlo principalmente mediante la herramienta de línea de comandos "oscar-cli" [67] para poder probar su funcionamiento y descubrir posibles errores en la documentación. Para poder hacer el despliegue desde oscar-cli, es necesario especificar la configuración del despliegue en un archivo YAML usando un lenguaje de definición de funciones (FDL) similar al usado en SCAR. El archivo YAML desarrollado especificaba un número variable de funciones dentro del clúster, todos conectados a un mismo directorio de entrada y de salida, y ejecutando el mismo script de inferencia usando YOLOv3. De esta forma, al añadir una nueva imagen al directorio de entrada, los servicios lanzan de forma simultánea funciones que ejecutan el script dentro de un contenedor, al igual que lo haría SCAR. Para dicho contenedor se reutiliza la imagen de la fase anterior. A medida que las funciones van finalizando su ejecución, las imágenes resultantes se almacenan en el

directorio de salida, usando la hora como nombre de archivo para evitar colisiones. A continuación se muestra un diagrama de la arquitectura desplegada para esta fase.

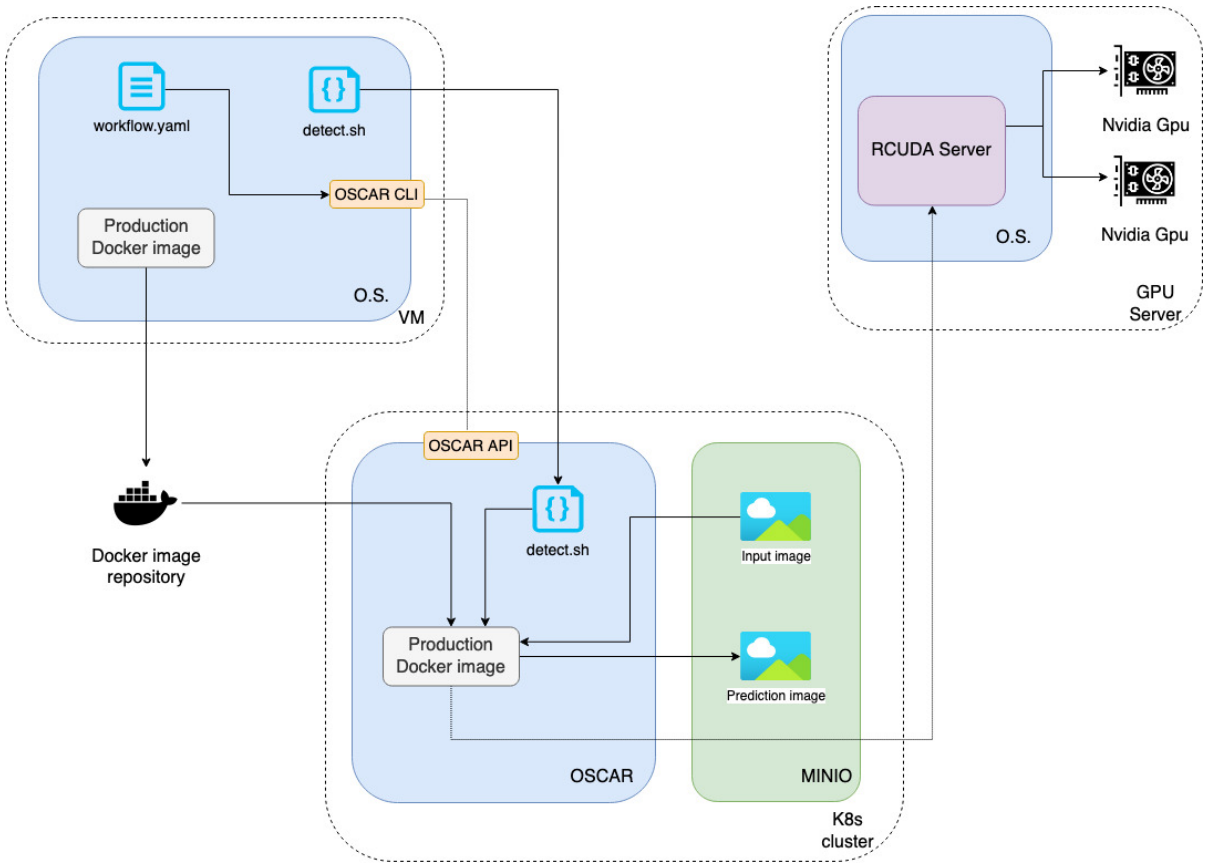

FIGURA 20. DIAGRAMA FASE 2
FUENTE: PROPIA

En el diagrama se puede ver la máquina virtual que sirve de entorno de desarrollo y en la que se instaló la herramienta oscar-cli, además se muestran el clúster OSCAR y el servidor de rCUDA. Se aplican también las mismas aclaraciones del diagrama anterior con respecto a las GPUs y los servidores de comunicaciones y licencia de rCUDA.

3.2.1 Implementación, despliegue y resultados

Tras instalar oscar-cli en el entorno de desarrollo y crear el fichero YAML (figura 21) se comenzó con la primera prueba en el clúster OSCAR, que contaba con un único nodo con dos CPUs virtuales y 4 GB de memoria. Esta primera prueba creaba varias funciones concurrentes a las que se les asignaba una vCPU y 2 GB de RAM. Al cargar la imagen de prueba en el directorio de entrada, una de las funciones pasaba a ejecutarse, mientras que el resto quedaba en estado pendiente. Tras discutir el resultado con la directora experimental del proyecto, se planteó la hipótesis de que el total de vCPU y RAM consumido por las funciones

debía ser estrictamente menor que las características correspondientes ofrecidas por el clúster, ya que una fracción de los recursos se debe destinar a la ejecución del sistema operativo, los procesos de Kubernetes, etc.

Para comprobar si la hipótesis era correcta, se hicieron algunas pruebas más: en la primera se repitió la configuración anterior pero reduciendo la asignación de recursos de cada función a 0.5 vCPU y 1 GB RAM. Con esta nueva configuración se pudieron ejecutar tres funciones concurrentes, lo cual tiene sentido, ya que añadir una nueva función haría que el total de recursos fuera igual a los disponibles en el clúster, lo que contradiría la hipótesis. Para la segunda prueba, el clúster se amplió a dos nodos de cómputo con 4 CPUs virtuales y 8 GB de memoria. Dicho clúster era elástico, por lo que el gestor CLUES incluido en OSCAR activaba o apagaba los nodos de forma autónoma según la carga de trabajo [68]. Con un nodo activo, el clúster admitió 7 funciones concurrentes (0.5 vCPU y 1 GB de RAM) y cuando CLUES activaba el segundo, el clúster admitió las 10 funciones especificadas en el YAML. Dado que todas las pruebas concuerdan con la hipótesis, se puede concluir que en su última configuración el clúster hubiera admitido hasta 15 funciones. Las imágenes obtenidas mostraban resultados de inferencia similares a los obtenidos en la fase anterior.

```yaml
functions:
  oscar:
  - oscar-rcuda:
      name: rcuda
      memory: 1Gi
      cpu: '0.5'
      image: mrconrui/oscar-test:latest
      script: detect.sh
      input:
      - storage_provider: minio.default
        path: test1/oscar-rcuda/in
      output:
      - storage_provider: minio.default
        path: test1/oscar-rcuda/out
  - oscar-rcuda:
      name: rcuda2
      memory: 1Gi
      cpu: '0.5'
      image: mrconrui/oscar-test:latest
      script: detect.sh
      input:
      - storage_provider: minio.default
        path: test1/oscar-rcuda/in
      output:
      - storage_provider: minio.default
        path: test1/oscar-rcuda/out
```

FIGURA 21. MUESTRA DEL FICHERO YAML GENERADO PARA ESTA FASE
FUENTE: PROPIA

3.3 Fase 3 - Despliegue en AWS Lambda

3.3.1 Diseño y arquitectura

Las últimas pruebas con OSCAR coincidieron aproximadamente con la recepción de los binarios de un scheduler de trabajos desarrollado por el equipo de rCUDA. A la misma vez, se colaboró con dicho equipo para diseñar una arquitectura serverless basada en SCAR y servicios de AWS en la que se pudiera incluir el scheduler para asignar los recursos de GPU a funciones mediante llamadas remotas. El diagrama de dicha arquitectura es el siguiente.

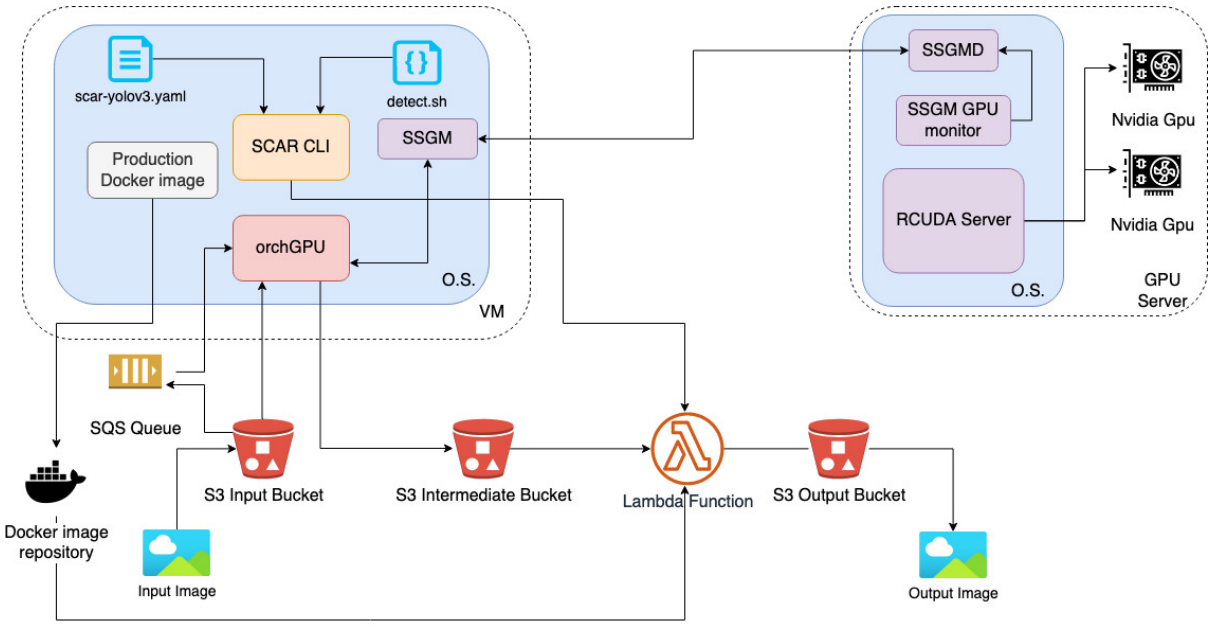

FIGURA 22. DIAGRAMA FASE 3
FUENTE: PROPIA, INSPIRADA EN LOS DISEÑOS DE DIANA M. NARANJO

Este diseño extiende la arquitectura típica de un despliegue basado en SCAR añadiendo la asignación de recursos GPU a los trabajos de inferencia por medio del scheduler de rCUDA . Los componentes del scheduler, desarrollados por el equipo de rCUDA, se representan en el diagrama como SSGM, SSGMD y SSGM GPU Monitor (cliente, servidor y monitor de hardware, respectivamente) y su funcionamiento se explica más adelante. Cuando una nueva imagen se almacena en el bucket S3 de entrada, este notifica el evento a una cola de trabajos en Amazon SQS [69]. Un script denominado "orchGPU", desarrollado específicamente para este proyecto, se encarga de crear trabajos a partir de los datos contenidos en los mensajes de la cola, lanzar funciones en SCAR para procesar dichos trabajos y reservar recursos GPU para dar soporte al proceso de inferencia de cada trabajo. orchGPU no tiene por qué estar en la máquina local del usuario, sin embargo es necesario que tenga acceso al ejecutable cliente del scheduler "SSGM". Una vez finalizado el proceso de inferencia, las imágenes resultantes

aparecen en el bucket de salida, como suele ser habitual en el modelo de SCAR. Este proceso se muestra de forma simplificada en el siguiente diagrama de secuencia:

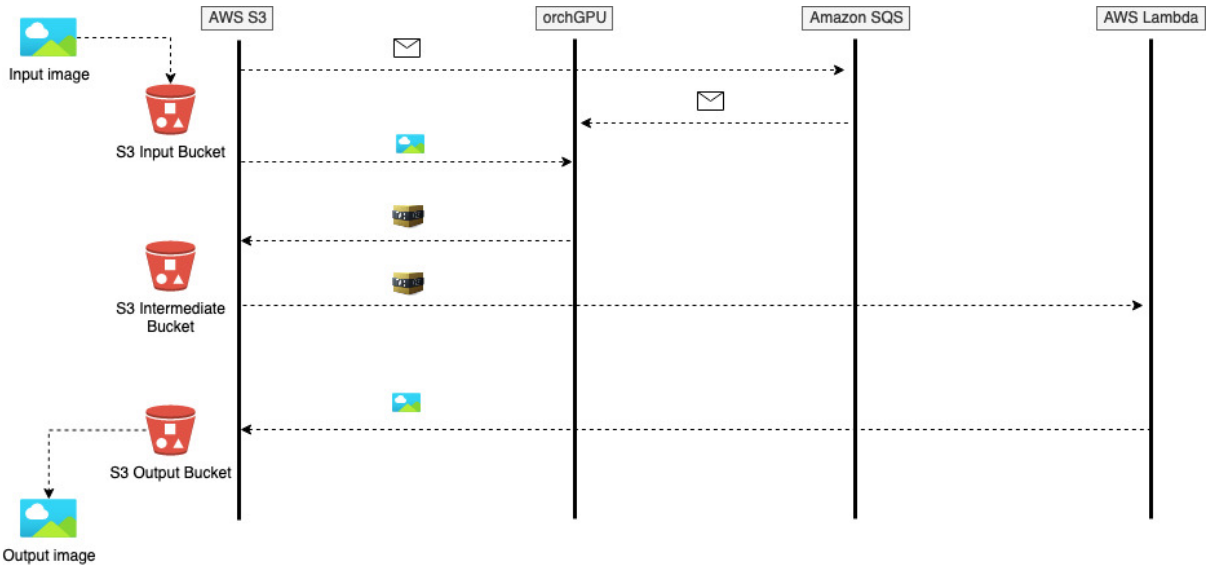

FIGURA 23. DIAGRAMA DE SECUENCIA FASE 3
FUENTE: PROPIA

En el diagrama anterior, el bucket de entrada de S3 envía un mensaje a SQS con la notificación de un nuevo fichero de entrada. El script orchGPU combina entonces algunos datos que recibe del scheduler (no se muestra en el diagrama) con la imagen de entrada en un archivo comprimido que transfiere a un nuevo bucket (bucket intermedio) para lanzar la función de SCAR en Lambda. El resultado de la función es almacenado en el bucket de salida. La decisión implementar este escenario usando un archivo comprimido y el bucket intermedio puede parecer arbitraria, pero surge en respuesta a la limitación de que SCAR no cuenta con la funcionalidad para enviar archivos adicionales junto con el script al hacer la invocación de una función. Haciendo uso de un proveedor de almacenamiento externo (S3) se puede aprovechar la capacidad de SCAR de ejecutar una función al cargar un archivo en un bucket para transferir dicho archivo al entorno de ejecución de la función.

En las dos siguientes secciones se explican las dos novedades tecnológicas más importantes que se añadido en esta fase: orchGPU y el scheduler de rCUDA.

3.3.1.1 orchGPU

La mayor parte de la lógica de esta arquitectura reside en el código de orchGPU, un binario desarrollado en Go expresamente para este proyecto que, como puede apreciarse en los diagramas anteriores, se comunica con el resto de entidades de la arquitectura incluyendo

varios servicios de AWS. Antes de explicar su funcionamiento es importante mencionar que orchGPU hace uso de técnicas de programación asíncrona para ejecutar algunas partes del código, ya que en caso contrario, la ejecución quedaría bloqueada en varios puntos a la espera de respuestas del resto de servicios y no sería posible ordenar la ejecución de varios trabajos concurrentes. Para implementar la asincronía se exploraron diferentes opciones (hilos, corutinas, async/await) en los lenguajes Python, Node.js, Rust y Go. Tras hacer algunas pruebas, se eligió implementarlo en Go [70], ya que permitía implementar la solución requerida con mínimos cambios al código existente. Go es un lenguaje compilado de propósito general desarrollado por Google que cuenta con un modelo de concurrencia basado en *goroutines* [71] que está siendo utilizado ampliamente para el desarrollo del *backend* de sistemas web, así como herramientas CLI y sistemas cloud. Las *goroutines* son hilos de ejecución extremadamente ligeros que son gestionados por el *runtime* de Go en vez de por el programador (en el caso de Go, el *runtime* no es una máquina virtual como la JVM de Java, sino una librería que se incluye en el binario y proporciona varios servicios al programa). El *runtime* asigna autónomamente cada *goroutine* a un hilo del sistema operativo según sea necesario, lo que resulta en una forma muy sencilla de obtener concurrencia, sacrificando el control fino sobre la misma. En el caso de orchGPU, existe un bucle que se ejecuta constantemente en la *goroutine* principal, y si recibe un mensaje de la cola (se hace un *polling* de la misma con frecuencia especificada por el usuario) y el scheduler confirma la reserva del trabajo, lanza una nueva *goroutine* que se ejecuta sin bloquear el bucle principal, permitiendo lanzar concurrentemente tantos trabajos como mensajes lleguen a la cola, siempre que el scheduler los admita.

```
//Check the SSGM error value
ssgm_error_value := rcuda_data_splits[0]
if ssgm_error_value == string(1) {
        fmt.Println("SSGM has return SSGM_ERROR=1")
} else {
        fmt.Println("SSGM has received rCUDA data containing: " + rcuda_data)
        //Invoke the SCAR function using the invoke_scar auxiliary function
        go invoke_scar(rcuda_data_splits, *sqs_job_id, *intermediate_bucket, *out
                *scheduler_address, *scheduler_port, *sqs_job_body, result_bucket
        //Delete the message from the queue
        fmt.Println("Deleting message from queue...")
        dMInput := &sqs.DeleteMessageInput{
                QueueUrl:      queueURL,
                ReceiptHandle: sqs_job_receipt_handle,
        }
        _, err := RemoveMessage(context.TODO(), client, dMInput)
        if err != nil {
                fmt.Println("An error happened while deleting the message:")
                fmt.Println(err)
                return
        }
```

FIGURA 24. CÓDIGO DE ORCHGPU LANZANDO UNA GOROUTINE (INVOKE_SCAR)
FUENTE: PROPIA

A continuación, se resume la funcionalidad de orchGPU, siguiendo aproximadamente el orden de ejecución:

1. **Procesar los *flags*** de línea de comandos pasados por el usuario.
2. **Inicializar la configuración** local de AWS y obtener la URL de la cola de trabajos de SQS cuyo nombre ha especificado el usuario.
3. **Ejecutar un bucle infinito** con las acciones 4-12:
4. **Obtener un mensaje** de la cola de trabajos y procesar su contenido. Si no hay ningún mensaje en la cola, esperar unos segundos (10 por defecto, configurables por el usuario) y reintentar hasta que lo haya.
5. **Hacer una petición al scheduler** para obtener recursos de GPU. Si el scheduler devuelve un error de tipo "recursos ocupados", esperar unos segundos antes de intentarlo otra vez (hasta que la reserva se realice correctamente).
6. **Procesar los datos de la respuesta** del scheduler y si no contiene un error, lanzar una nueva *goroutine* que lleve a cabo las acciones 7-12:
7. **Guardar los datos** de la respuesta del scheduler en un fichero de texto.
8. **Obtener del bucket de entrada en S3** la imagen que ha ocasionado el mensaje en la cola de SQS. El nombre de dicha imagen está en el cuerpo del mensaje.
9. **Crear un archivo comprimido** (.tar.gz) que contenga la imagen recibida y el fichero de texto con los datos del scheduler.
10. **Enviar el archivo comprimido** al bucket intermedio en S3. Esto provoca la ejecución de la función SCAR, ya que esta tiene definido al bucket intermedio como bucket de entrada.
11. **Esperar usando un bucle infinito** a que el resultado de la función SCAR esté en el bucket de salida, dado que la operación anterior no es bloqueante.
12. **Solicitar la liberación de los recursos** del scheduler, una vez el bucket contenga el resultado. Si en algún momento de la ejecución de la *goroutine* se produce un error, también se han de liberar dichos recursos.
13. **Borrar el mensaje** de la cola de trabajos

El código fuente de orchGPU está organizado en un paquete con tres archivos que al ser compilados forman un binario único. Estos archivos son:

- **main.go** contiene el bucle de eventos principal y una gran parte de la lógica del programa. Se comunica con la cola de SQS, pide recursos al scheduler y ordena a una función de Util.go el lanzamiento asíncrono de la función de SCAR.
- **util.go** contiene varias funciones de utilidad, principalmente relacionada con procesar los datos recibidos del scheduler y establecer la comunicación con SCAR para invocar la función.

- **api.go** contiene varias interfaces y funciones de utilidad para comunicarse con los servicios de Amazon Web Services, según aparece en la documentación del SDK de AWS para Go.

orchGPU se lanza desde el terminal con varios *flags* que permiten especificar la configuración de lanzamiento. A continuación, se muestran dichos *flags* junto con su descripción:

```
./orchGPU
          -q [Nombre de la cola de trabajos]
          -S [Dirección IP o hostname del scheduler]
          -P [Número de puerto del scheduler]
          -g [Número de GPUs a reservar]
          -v [Tiempo de invisibilidad del mensaje en la cola]
          -w [Tiempo de espera para recibir un mensaje]
          -s [Timeout para la reserva con el scheduler]
          -u [Timeout para cola vacía]
          -b [Timeout para el resultado de la inferencia]
          -m [Ruta del ejecutable de SSGM]
          -i [Directorio y nombre del bucket intermedio]
          -o [Directorio y nombre del bucket de salida]
```

FIGURA 25. FLAGS DE ORCHGPU
FUENTE: PROPIA

3.3.1.2 Scheduler de rCUDA

Por su parte, el scheduler está compuesto por tres procesos que se ejecutan en distintos nodos:

- **SSGMD** es el proceso principal del scheduler de rCUDA. En este caso se ejecuta en una máquina independiente al resto e implementa la lógica de asignación de los dispositivos GPU en función de la política configurada. En este proyecto, el scheduler se ha configurado con una GPU Nvidia Tesla V100, una política de asignación con *round-robin* (indiferente cuando solo hay un dispositivo) y un modo de respuesta no bloqueante, en el que el scheduler devuelve inmediatamente una respuesta si los recursos solicitados se encuentran ocupados. En tal caso, se devuelve una variable de error activada y en caso contrario, se devuelven los datos de la reserva, como la dirección del dispositivo GPU remoto asignado. Estos datos son usados por el programa solicitante (en este caso, el modelo de *machine learning*) para poder comunicarse con el dispositivo remoto. Los comandos que SSGM puede ejecutar sobre el scheduler son los siguientes:

- **Set** permite configurar parámetros del scheduler como la política de asignación de GPUs.
- **Alloc** solicita recursos de GPU al scheduler.
- **Dealloc** solicita la liberación de los recursos asignados.
- **Top** muestra el estado del scheduler en tiempo real.

- **SSGM** es el proceso que actúa como cliente del proceso servidor SSGM. Puede solicitar la asignación y liberación de recursos, así como el estado de los trabajos en ejecución.
- **SSGM GPU Monitor** es un proceso simple que se ejecuta en cada nodo con dispositivos GPU y comunica al proceso servidor del scheduler, SSGM, la carga de trabajo de dichos dispositivos.

```
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$ ./ssgm -S 158.42.104.198 -P 10000 -alloc -g 2
export SSGM_ERROR=1;
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$ ./ssgm -S 158.42.104.198 -P 10000 -alloc -g 1
export SSGM_ERROR=0;export JOB_ID=1;export RCUDA_DEVICE_COUNT=1;export RCUDA_DEVICE_0=horsemen16.i3m.upv.es:0;
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$ ./ssgm -S 158.42.104.198 -P 10000 -top

+-------------------------------------------------------------------------------------------+
| v2021.8.18           Wed Aug 18 18:09:44.728           Universitat Politecnica de Valencia |
| Scheduler Type: rCUDA    Blocking Mode: On     Node ID: Hostname        Policy: RR         |
+-------------------------------------------------------------------------------------------+
| GPU  Model       Node            dev            Memory             Utilization Power  Jobs |
+-------------------------------------------------------------------------------------------+
|   0  Tesla V100  horsemen16.i3m.   0    10MiB     0MiB 32502MiB       0%   0%   30W      1 |
+-------------------------------------------------------------------------------------------+
^C
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$ ./ssgm -S 158.42.104.198 -P 10000 -dealloc -j 1
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$ ./ssgm -S 158.42.104.198 -P 10000 -top

+-------------------------------------------------------------------------------------------+
| v2021.8.18           Wed Aug 18 18:09:52.941           Universitat Politecnica de Valencia |
| Scheduler Type: rCUDA    Blocking Mode: On     Node ID: Hostname        Policy: RR         |
+-------------------------------------------------------------------------------------------+
| GPU  Model       Node            dev            Memory             Utilization Power  Jobs |
+-------------------------------------------------------------------------------------------+
|   0  Tesla V100  horsemen16.i3m.   0    10MiB 32502MiB 32502MiB       0%   0%   30W      0 |
+-------------------------------------------------------------------------------------------+
^C
ubuntu@rcuda-client:~/ssgm_CUDA9_0_4a154b5$
```

FIGURA 26. RESERVA Y LIBERACIÓN DE UN JOB CON EL SCHEDULER
FUENTE: PROPIA

A modo ilustrativo, se muestra en la figura anterior el proceso de reserva y liberación de recursos usando el scheduler. En el primer comando, se reservan (*-alloc*) dos GPUs (*-g 2*) pero dado que el servidor solo cuenta con un dispositivo, devuelve un error. Al hacer la solicitud con una única GPU, el devuelve los datos de la reserva. Además, es posible consultar el estado del scheduler mediante la opción *-top*. La salida del comando con esta opción permite observar en tiempo real la asignación de trabajos a cada GPU, junto con información de interés como la activación del modo de operación bloqueante y la política de asignación de trabajos, que en este caso es *round-robin (RR)*.

3.3.1 Implementación, despliegue y resultados

La primera tarea llevada a cabo en esta fase fue conseguir que el scheduler funcionara correctamente en el entorno de pruebas, para lo que hubo que resolver algunas incompatibilidades con el sistema operativo y la versión de CUDA. Una vez comprobado que el scheduler funcionaba y tenía conectividad con el entorno de desarrollo, se realizaron algunas peticiones de recursos manualmente para descubrir el formato de las respuesta del scheduler. Esto era necesario para luego poder implementar una parte del comportamiento de orchGPU (línea 5 del pseudocódigo). El desarrollo de orchGPU centró la mayor parte del esfuerzo de esta fase, al ser necesario iterar sobre distintas alternativas y decisiones de diseño, especialmente las relacionadas con la comunicación entre orchGPU y otros elementos de la arquitectura. Tal es el caso de la elección del mecanismo para lanzar funciones SCAR (HTTP o CLI) o la manera óptima para hacer llegar los datos devueltos por el scheduler a las instancias del modelo de *machine learning* que los necesita para poder realizar la inferencia. Por otro lado, fue necesario crear un nuevo archivo YAML para esta fase, que proporcionara a SCAR la descripción de la función a lanzar. En concreto, este archivo contiene un nombre para la función, el script a ejecutar en cada lanzamiento, la imagen Docker a usar y los buckets de S3 que servirán a la función para transferir archivos con el exterior. Cabe destacar que el identificador del bucket de entrada en el YAML es "/intermediate" ya que en la arquitectura de esta fase, es orchGPU quien se comunica con el auténtico bucket de entrada. El script "detect.sh" indicado es prácticamente igual al usado en la fase 1.

```
functions:
  aws:
  - lambda:
      name: alucloud24-scar-yolov3
      init_script: detect.sh
      container:
        image: mrconrui/oscar-test:tiny
      input:
      - storage_provider: s3
        path: alucloud-lambda/intermediate
      output:
      - storage_provider: s3
        path: alucloud-lambda-out/24
```

FIGURA 27. DOCKERFILE GENERADO
FUENTE: PROPIA

Una vez que se contaba con todos los archivos necesarios, se hicieron numerosas pruebas usando SCAR y AWS Lambda para ir progresivamente corrigiendo fallos en la arquitectura y en el comportamiento de orchGPU. En el proceso se detectaron algunos fallos adicionales en otros elementos, por ejemplo, el tamaño de la imagen usada para el contenedor Docker

superaba los 512 MB que AWS Lambda tiene de límite para el directorio /tmp, y por lo tanto para el tamaño del contenedor que SCAR ejecuta en él, por lo que fue necesario modificar el Dockerfile para usar un conjunto de pesos de menor tamaño (YOLOv3-tiny). Tras la corrección de los errores, se consiguió llevar a cabo una ejecución correcta con un único trabajo e incluyendo todos los elementos de la arquitectura. El resultado de la inferencia también fue correcto, aunque los tiempos de ejecución fueron bastante elevados debido al ancho de banda limitado que se ha comentado anteriormente. En este momento se discutió si al situar el servidor de rCUDA en una instancia de AWS EC2, se podría reducir la duración del proceso de inferencia (esto se comprueba en la sección 3.4). También se investigaron otras alternativas, por ejemplo, en 2018 AWS introdujo soporte para Elastic Fabric Adapter (EFA) en EC2 [72], lo que permite conexión entre instancias basada en Infiniband. Sin embargo, Lambda no parece contar con este soporte todavía e investigar su uso en otros servicios como AWS Batch escapa al objetivo de este trabajo.

Una vez comprobado el funcionamiento de orchGPU con un job, se pasó a probar con dos trabajos concurrentes. Progresivamente, se fueron puliendo fallos de implementación hasta conseguir una prueba exitosa. En dicha prueba, orchGPU pudo gestionar correctamente la adquisición de los mensajes y la gestión de los recursos del scheduler tanto para el job en ejecución como para el que se estaba en espera. No hay razón para pensar que orchGPU no pueda gestionar más jobs que los dos comprobados hasta ahora: orchGPU no impone ningún límite al número de jobs concurrentes, por lo que, en principio, admite tantos trabajos como mensajes haya en la cola de SQS asociada.

En el anexo 1 se muestra el registro de Amazon CloudWatch que describe una ejecución completa de la fase 3.

3.4 Comparativa de rendimiento

Llegados a este punto es conveniente analizar si el rendimiento de las soluciones desarrolladas puede competir con el de las arquitecturas no basadas en el modelo serverless. Para ello, se ha tomado la imagen Docker con el modelo reducido YOLOv3 usado en la fase 3 y se le ha añadido el paquete GNU Time [73]. La prueba diseñada para realizar la comparativa de rendimiento es sencilla: plantear distintos escenarios de despliegue y medir los tiempos de ejecución de YOLOv3-tiny con una de las imágenes de prueba incluidas en Darknet. Los escenarios planteados son los siguientes:

- **Local CPU:** contenedor desplegado sobre Docker en la máquina local (2 núcleos Intel Xeon Skylake, 2GB RAM) realizando la inferencia sobre la CPU local.

- **Local GPU:** contenedor desplegado sobre Docker en el servidor de rCUDA (4 núcleos Intel Xeon Skylake, 8 GB RAM) realizando la inferencia sobre una GPU Nvidia Tesla V100 situada en el propio servidor.
- **Local GPU externa:** contenedor desplegado sobre Docker en la máquina local (2 núcleos Intel Xeon Skylake, 2GB RAM) realizando la inferencia utilizando una GPU Nvidia Tesla V100 situada en el servidor de rCUDA.
- **Lambda CPU:** contenedor desplegado sobre una función en AWS Lambda con 512 MB de memoria realizando la inferencia en CPU. Se ha elegido la cifra moderada de 512 MB ya que el interés principal de Lambda es usarla en conjunción con una GPU remota, y en ese caso, la memoria y la capacidad de cómputo de la función no debe ser muy influyente en el rendimiento total.
- **Lambda GPU externa:** contenedor desplegado sobre una función en AWS Lambda con 512 MB de memoria realizando la inferencia sobre una GPU Nvidia Tesla V100 situada en el servidor de rCUDA en hardware *on-premises*.
- **Lambda GPU EC2:** contenedor desplegado sobre una función en AWS Lambda con 512 MB de memoria realizando la inferencia sobre una GPU Nvidia Tesla K80 situada en una instancia EC2 de tipo p2.xlarge (4 vCPU, 61 GB RAM).

En la imagen inferior se muestra un diagrama simplificado con la arquitectura del entorno de pruebas, en el que se puede observar rápidamente la situación de cada elemento desplegado. El entorno es en realidad muy similar al de la fase 3, con la adición de la máquina virtual con GPU desplegada en EC2 (las máquinas que cuentan con GPU muestran en el diagrama el logo de Nvidia). Se debe tener en cuenta que no todas las conexiones dibujadas entre las máquinas se usan en todos los escenarios de prueba. La flecha en cada conexión indica la dirección en la que se hace la petición de cómputo CPU o GPU.

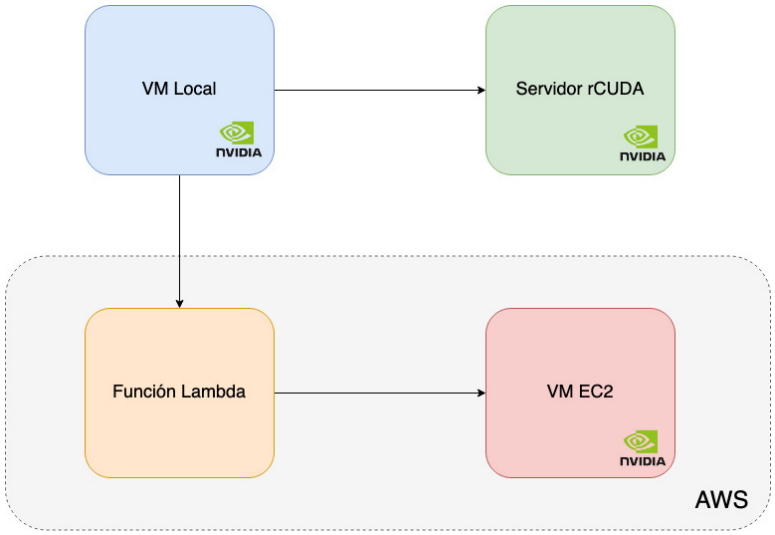

FIGURA 28. DIAGRAMA DE ARQUITECTURA DEL ENTORNO DE PRUEBAS
FUENTE: PROPIA

Para cada uno de los anteriores, se ha tomado el tiempo de inferencia reportado por Darknet (tiempo que dura la ejecución de la función de inferencia de Darknet) y el tiempo de ejecución medido por GNU Time (tiempo de ejecución total del comando dentro del contenedor). En la tabla inferior se muestran los resultados ordenados de menor a mayor tiempo de ejecución. Para los escenarios de menor duración, se muestran los tiempos medios entre varias ejecuciones, ya que son los más afectados por fluctuaciones en la red, etc. Adicionalmente, se ha incluido una columna con una estimación del coste en el que se ha incurrido en Lambda y EC2 para hacer un millón de invocaciones (ya que existe un sobrecoste si se hacen más). Esta estimación se ha conseguido a partir del producto del coste por tiempo de cada servicio por los segundos mostrados en la columna "GNU Time", que no tiene en cuenta algunos aspectos como el tiempo de descarga de la imagen del contenedor, por lo que el coste real será algo superior a las cifras mostradas en la tabla. Se han usado los precios para la región "E.E.U.U. Este (Norte de Virginia)" a fecha de la elaboración.

Tabla 1. Tiempos de ejecución y costes en AWS

	Tiempo de inferencia (s)	GNU Time (s)	Coste estimado Lambda (USD por 1 millón de ejecs.)	Coste estimado EC2 (USD por 1 millón de ejecs.)
Local CPU	2.31	3.78	-	-
Local GPU externa	0.27	7.28	-	-
Local GPU	0.23	6.98	-	-
Lambda GPU EC2	0.29	16.58	137.61	4145.00
Lambda CPU	8.78	88.96	740.44	-
Lambda GPU externa	45.23	813.48	6770.82	-

Los resultados obtenidos no muestran grandes sorpresas. Como es de esperar, la inferencia con GPU es más rápida que con CPU y los escenarios locales siempre obtienen un mejor rendimiento debido al ancho de banda limitado y elevada latencia entre el entorno local y la infraestructura física de AWS. Un aspecto a tener en cuenta es que la medición del tiempo de inferencia ofrecida por Darknet parece estar influida por la transferencia de datos, como se deduce al comparar dos escenarios que comparten GPU, como "Local GPU externa" y "Lambda GPU Externa". Examinando el código fuente de Darknet no parece haber una explicación sencilla a este hecho.

Por otro lado, mediante estas pruebas se ha podido confirmar lo que se discutió en la sección 3.3.1 sobre si contar con una GPU en una instancia en EC2 reduce los tiempos de ejecución con respecto a la aproximación híbrida en la que la función Lambda se comunica con una GPU en hardware *on-premises*. Efectivamente, se produce una reducción dramática en los tiempos al usar una GPU alojada en EC2 gracias a una mayor tasa de transferencia de datos. Con todo, la conclusión más importante que se puede extraer desde el punto de vista del rendimiento es que, al menos para el caso de la ejecución de un modelo de *machine learning* de pequeño tamaño, utilizar Lambda es recomendable sólo para aquellos dispuestos a renunciar a una parte del rendimiento con tal de obtener una mayor sencillez operativa. En ese caso, y desde un punto de vista exclusivamente centrado en el rendimiento, el uso de una GPU externa (en EC2) ha demostrado ser una alternativa viable gracias a tiempos de ejecución relativamente contenidos, aunque eso no haya sido suficiente en este caso para compensar su mayor coste, razón por la que solo es recomendable a quienes busquen una solución en la nube con poca latencia pero de coste elevado.

En cuanto a los costes de los diferentes escenarios ejecutados en Lambda, el uso de GPUs externas, bien *on-premises* o bien en EC2, no parece ser una alternativa competitiva debido al sobrecoste que supone tener una máquina con GPU activa en EC2 y al escaso ancho de banda en las comunicaciones con una GPU alojada en una infraestructura local. Hay que tener en cuenta además que el esquema de precios de EC2 es diferente al de Lambda, lo que reduce en cierta medida las ventajas de adoptar un modelo serverless de pago por uso.

3.5 Valoración de los resultados

En esta sección se parte de los resultados obtenidos para hacer una valoración general del proyecto desde diferentes perspectivas: proceso de desarrollo, rendimiento de las soluciones y coste económico.

3.4.1 Coste económico

Evaluar las ventajas económicas que ofrece una solución de cómputo cloud sobre otras alternativas puede llegar a ser muy complicado, ya que entran en juego factores como los esquemas de precios, los requisitos de infraestructura, la duración y frecuencia de las ejecuciones, la disponibilidad deseada, etc. Aun así, se pueden intentar obtener conclusiones generales que sirvan para recomendar el servicio que, a priori, parezca más rentable para cada grupo de casos de uso. Para la aplicación tratada en este proyecto, se pueden considerar varias alternativas dentro de AWS, por ejemplo:

- Lambda. Tal y como se ha visto en la sección correspondiente a la fase 3, permite la ejecución serverless de funciones. Su esquema de precios depende de los recursos asignados a dichas funciones, el número de invocaciones y la duración de las mismas.
- ECS/EKS permiten ejecutar contenedores sobre EC2 sin coste adicional.
- Batch ejecuta trabajos de cómputo en entornos que escalan según las necesidades de dichos trabajos y pueden usar EC2 como *backend* sin coste adicional.

La clave para elegir entre los servicios anteriores es el tipo de carga de trabajo. Para ejecuciones frecuentes y cortas, especialmente teniendo en cuenta que impone un límite de 900s de procesamiento, Lambda es una opción recomendable tanto desde un punto de vista de complejidad operacional como económico. En cifras concretas, el portal de precios de Lambda [74] explica que tres millones de peticiones de un segundo de duración a una función con 512 MB de memoria se traducen en 18.74 $ con el esquema de precios actual (incluyendo el descuento de la capa gratuita de AWS). Partiendo de dicha información, es posible calcular que si las mismas funciones tuvieran una duración máxima igual al límite del servicio (900 segundos), el coste total ascendería a 22498,23 $, también incluyendo el descuento por la capa gratuita. Claramente, es mucho más difícil justificar el uso de Lambda a ese coste. Por el contrario, dado que ECS, EKS y Batch no suponen ningún coste añadido al de EC2 y este último se factura exclusivamente por el tiempo en el que la infraestructura permanece activa, es posible obtener facturas mensuales de 239,62 $ por una instancia t3.2xlarge (8 vCPU, 32 GB RAM), una cifra que queda a medio camino entre las dos mencionadas antes para Lambda, y que puede ser reducida considerablemente si se hace uso de un plan de pago o de instancias *spot*. La justificación para los cálculos anteriores se encuentra en el Anexo 2.

Es necesario mencionar que, independientemente del servicio elegido, es necesario tener una máquina con acceso a GPU para hacer de servidor de rCUDA. Si esta máquina se ejecuta sobre hardware *on-premises*, su coste será amortizado durante el periodo de uso. Alternativamente, se puede obtener una instancia bajo demanda P3 en EC2 con 8 vCPUs, 61 GB de RAM y una GPU Nvidia Tesla V100 (la GPU usada a lo largo de este proyecto) por 3.06 $/hora (2203.2 $/mes), igualmente reducibles mediante pago por adelantado o instancias *spot*.

3.4.2 Desarrollo

El uso de tecnologías modernas en este proyecto ha simplificado en parte el proceso de desarrollo. Sin embargo, durante dicho proceso también se ha experimentado la complejidad inherente a los sistemas concurrentes desplegados en la nube. Esta complejidad no solo existe al intentar integrar los distintos conceptos o herramientas del sistema, sino que también se

extiende a su despliegue y *testing*. Esto puede pasar desapercibido para aquellos que se acercan por primera vez al desarrollo de sistemas cloud. Esta complejidad acumulada requiere a veces un enfoque innovador con respecto a las técnicas tradicionales, más centrado en áreas como el aislamiento de recursos, la separación de funcionalidad, la monitorización o la seguridad. Muchas de las herramientas utilizadas en este proyecto ya hacen un énfasis en estas áreas, por ejemplo, AWS cuenta con registros extensivos de la actividad de los servicios, Go propone un modelo de desarrollo ágil con concurrencia y gestión de dependencias simplificadas, y SCAR/OSCAR admiten una configuración declarativa de las ejecuciones para reducir la complejidad y facilitar la integración de cambios. También es interesante observar las implicaciones del uso de tecnologías que trabajan a un alto nivel de abstracción, como es el caso de rCUDA, Kubernetes y, por supuesto, AWS Lambda. Estas abstracciones pueden suponer una reducción del coste operativo de las aplicaciones que se adapten a ellas, pero también suponen un esfuerzo de desarrollo adicional, ya que muchas veces restringen la monitorización y el acceso a las capas inferiores de la arquitectura y esto repercute negativamente en el proceso de depuración de errores.

3.4.3 Rendimiento

Como se comentó en la Parte 1, una gran proporción de los sistemas cloud que se desarrollan actualmente están limitados por la entrada/salida (*I/O bound*), y esto es especialmente cierto para este proyecto. Como se discutirá en la siguiente sección, AWS Lambda es más rentable para tareas de cómputo esporádicas y de corta duración. Es en ese tipo de tareas en las que la entrada/salida representa la mayor proporción del tiempo total de ejecución y por lo tanto, un ancho de banda reducido entre los distintos elementos tiene un mayor impacto en el rendimiento. A lo largo de la memoria se han comentado algunas posibles soluciones al problema del ancho de banda, por lo que estas conclusiones no son definitivas.

Dejando de lado el ancho de banda, las partes limitadas por la capacidad de procesamiento, como es el caso de la inferencia en la GPU remota no han supuesto ningún problema. La única Nvidia Tesla V100 utilizada para las pruebas ha demostrado ser más que capaz ejecutar los trabajos en un tiempo razonable, especialmente con el conjunto de pesos reducido usado en la fase 3 por las limitaciones de espacio de Lambda. Por otro lado, la ejecución de orchGPU también está completamente dominada por la entrada/salida, por lo que su rendimiento a nivel de procesamiento en CPU ha tenido poco impacto en los tiempos de ejecución. El modelo de concurrencia basado en *goroutines* de Go ha demostrado ser una manera sencilla de obtener dicha concurrencia en sistemas cloud, siempre que no sea necesario obtener el máximo rendimiento del programa en CPU.

Parte 4. Conclusiones

En esta última parte se evalúa el grado en el que se han alcanzado los objetivos propuestos y se sugieren algunas líneas de trabajo futuro.

4.1 Valoración de los objetivos

Aunque el proyecto se ha llevado a cabo con cierto éxito, conviene explorar en esta sección cuáles han sido los motivos y, especialmente, identificar los incidentes que han provocado desviaciones con respecto a la planificación inicial. A continuación, se recuperan los objetivos académicos y técnicos de la sección 1.2 y se discute el progreso en cada uno de ellos:

- **Objetivos académicos**
  - **Objetivo 1:** Como parte "Infraestructuras de cloud público" y asignaturas similares, se han obtenido conocimientos sobre distintos servicios cloud, además de sus posibilidades de integración y aplicaciones. A lo largo de este proyecto, se ha profundizado en los fundamentos del paradigma serverless, la virtualización de aplicaciones y varias técnicas de integración de sistemas cloud.
  - **Objetivo 2:** La implementación de sistemas cloud implica trabajar a un alto nivel de abstracción, por lo que este proyecto ha permitido poner en práctica habilidades trasversales como la resolución de problemas complejos y la elaboración de documentación que comunique de forma efectiva el complejo funcionamiento interno de estos sistemas. Relacionado con esto último, la comunicación frecuente con miembros de distintos equipos también ha permitido practicar la síntesis de conceptos y su comunicación efectiva para proponer ideas, y obtener asistencia y *feedback*.
  - **Objetivo 3:** Si bien el progreso de la investigación y la complejidad de las soluciones han estado limitadas por el alcance del proyecto, se ha podido obtener información útil sobre tecnologías recientes que permitirán seguir avanzando en el área de la computación serverless y GPU. Además, se ha proporcionado *feedback* de las herramientas usadas en el proyecto: SCAR, OSCAR y rCUDA.
  - **Objetivo 4:** Se han obtenido conocimientos sobre el funcionamiento de rCUDA y cómo integrarlo en una aplicación. Sin embargo, no ha profundizado en la implementación del *framework*, en parte por ser software propietario, y en parte porque no era necesario para usarlo en el contexto del proyecto.

- **Objetivos técnicos**
  - **Objetivo 1:** Sin duda, uno de los mayores beneficios de haber llevado a cabo este proyecto es haber tenido al posibilidad de adquirir una experiencia extensiva sobre AWS Lambda y su aplicación a una caso de uso actual como es el aprendizaje automático. Dicho esto, con el fin de contener el alcance del proyecto, se han dejado en un segundo plano algunos factores que hubieran contribuido a hacer una simulación más realista, como asegurar la tolerancia a fallos o buscar la máxima disminución en la latencia de procesamiento.
  - **Objetivo 2:** Se puede considerar que este proyecto ha probado la viabilidad de las soluciones basadas en GPU remota para arquitecturas serverless. Las limitaciones encontradas (ancho de banda, tamaño de las imágenes) deberían poder ser sorteadas en algunos casos de uso y así poder obtener soluciones listas para ser desplegadas en entornos reales de producción.
  - **Objetivo 3:** A lo largo del proyecto se ha obtenido una visión relativamente clara de las posibilidades que ofrecen las tecnologías utilizadas. Además, el haber tenido que explorar distintas alternativas para las soluciones ha proporcionado una visión más global del ecosistema serverless y una mayor confianza a la hora de tomar decisiones de diseño e implementación en este campo.

4.2 Trabajo futuro

A juzgar por lo discutido en la sección anterior, se puede concluir que los objetivos marcados se han cumplido con cierto grado de éxito. En esta sección se proponen líneas de trabajo adicionales que amplían la soluciones desarrolladas en el proyecto:

- **Despliegue serverless de orchGPU:** la arquitectura planteada en la fase 3 ejecuta el binario orchGPU en la máquina del usuario, junto a la herramienta CLI de SCAR y al binario SSGM. Dado el énfasis de este proyecto en el modelo de ejecución serverless, tiene sentido preguntarse si es posible desplegar alguno de estos elementos en Lambda y así conseguir una arquitectura menos dependiente de la instalación de software en la máquina local del usuario. Dado que SSGM y orchGPU se distribuyen como simples binarios y su ejecución es *stateless* (cada ejecución es independiente del estado de la anterior) parece técnicamente posible su despliegue en una función Lambda invocada desde la máquina del usuario mediante SCAR-CLI. A priori, no serían necesarios cambios internos en el código de ninguno de los elementos. En la imagen inferior se muestra el diagrama de la nueva arquitectura propuesta:

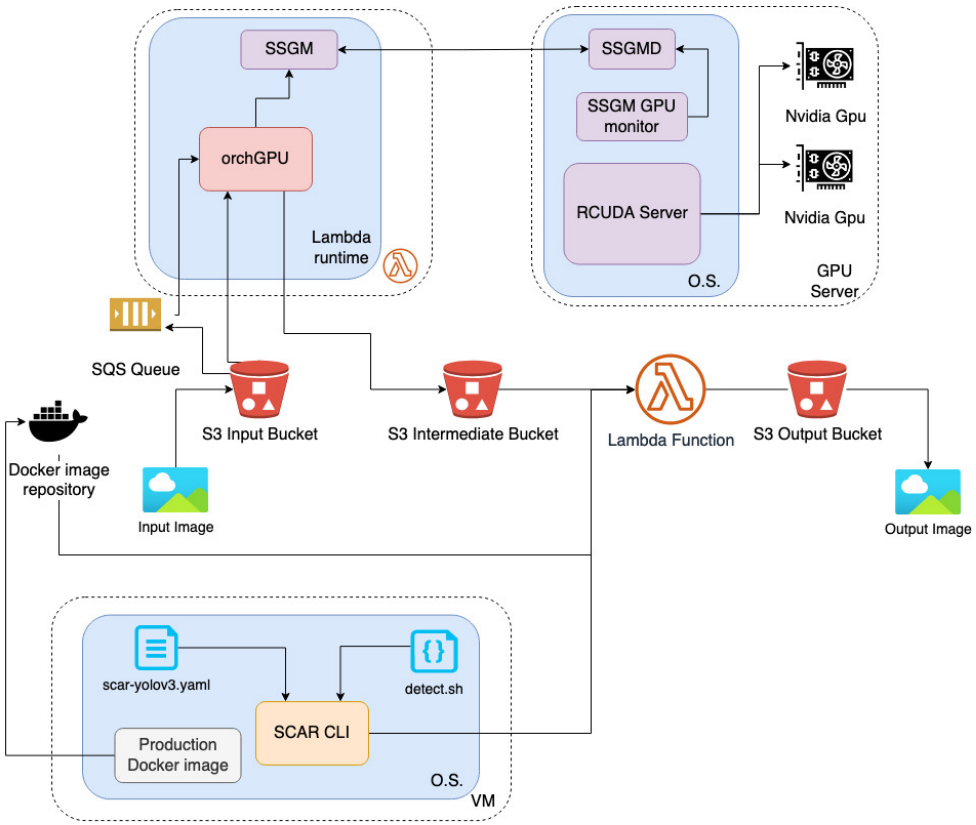

FIGURA 30. DIAGRAMA DEL DESPLIEGUE SERVERLESS DE ORCHGPU
FUENTE: PROPIA

- **Integración de nuevos casos de uso:** el código de orchGPU se ha desarrollado para ser mayormente agnóstico al formato de los archivos de entrada y al funcionamiento de las funciones SCAR que ejecute, por lo que con relativamente poco esfuerzo, se podrían integrar nuevas aplicaciones que hagan uso de GPUs remotas vía rCUDA y su scheduler. En principio, tan solo debería ser necesario escribir los nuevos archivos auxiliares para SCAR: Dockerfile, script de inicialización y definición YAML de la función.
- **Integración de una interfaz de usuario:** para conseguir una simulación más realista del caso de uso elegido, podría ser interesante simular la interacción con el usuario. Por ejemplo, en la fase 3, la ejecución comienza al cargar una nueva imagen en el bucket de entrada en Amazon S3. Sin embargo, los usuarios no suelen interactuar directamente con este servicio ya que forma parte del *backend* de las aplicaciones. Para hacer una simulación más realista, se podría desarrollar una interfaz web o móvil para permitir al usuario cargar una imagen en S3 directamente y descargar el resultado del bucket de salida a través de la misma interfaz. Esto no debería afectar al funcionamiento interno de la solución desarrollada hasta ahora, ya que su interacción con S3 quedaría inalterada.
- **Optimización del rendimiento:** Otra posible extensión del proyecto puede ser la realización de un estudio más detallado para localizar y resolver los posibles cuellos de botella de rendimiento. Como punto de inicio, se podrían tomar los resultados obtenidos en esta memoria, que ya identifican algunos de esos problemas de rendimiento. La

documentación de las tecnologías utilizadas también puede ser de utilidad. La de rCUDA, por ejemplo, propone soluciones para el limitado ancho de banda en las conexiones TCP/IP, como se comentó en la sección 3.1.2.

- **Refactorización:** con el fin de convertir orchGPU en un software listo para producción, se podría refactorizar o reimplementar con una arquitectura diferente (partiendo del diseño actual) que persiguiera una mayor tolerancia a fallos, complementando a los mecanismos ya utilizados en AWS. En este sentido, se podrían añadir mejoras como incluir mecanismos adicionales para evitar la pérdida de mensajes en caso de fallo (por ejemplo, que se produzca una excepción en orchGPU tras haber obtenido y eliminado un mensaje de la cola de SQS). Por otro lado, también se podría reducir el ancho de banda usado por orchGPU evitando que se hicieran peticiones de reservas de recursos al scheduler si se sabe que un job ya se está ejecutando y sus recursos no han sido liberados todavía.

Referencias

[1] AWS Lambda. https://aws.amazon.com/es/lambda/

[2] Datadog. *The State of Serverless*. https://www.datadoghq.com/state-of-serverless/

[3] Using container images to run PyTorch models in AWS Lambda. https://aws.amazon.com/es/blogs/machine-learning/using-container-images-to-run-pytorch-models-in-aws-lambda/

[4] Elastic Container Service. https://aws.amazon.com/es/ecs/

[5] Elastic Kubernetes Service. https://aws.amazon.com/es/eks/

[6] Sagemaker. https://aws.amazon.com/es/sagemaker/

[7] Web de GRyCAP. https://www.grycap.upv.es/

[8] rCUDA. rcuda.net

[9] orchGPU. https://github.com/grycap/orchgpu

[10] John McCarthy - Contributions to Computer Science. https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)#Contributions_in_computer_science

[11] Jeremy Daly. *Stop Calling Everything Serverless!* https://www.jeremydaly.com/stop-calling-everything-serverless/

[12] Lambda runtimes. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[13] Custom AWS Lambda runtimes. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html

[14] Event-drive Architecture. https://en.wikipedia.org/wiki/Event-driven_architecture

[15] Google Cloud Functions. https://cloud.google.com/functions

[16] Azure Functions. https://azure.microsoft.com/es-es/services/functions/#overview

[17] New for AWS Lambda – 1ms Billing Granularity Adds Cost Savings. https://aws.amazon.com/es/blogs/aws/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/

[18] Lambda Isolation Technologies. https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-isolation-technologies.html

[19] Configuring Lambda Function Options. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[20] News for Lambda - Container Image Support. https://aws.amazon.com/es/blogs/aws/new-for-aws-lambda-container-image-support/

[21] Pérez et al., 2019. *Serverless Computing for Container-based architectures*. https://doi.org/10.1016/j.future.2018.01.022

[22] SCAR. https://github.com/grycap/scar

[23] Amazon EC2. https://aws.amazon.com/es/ec2/

[24] AWS Fargate. https://aws.amazon.com/es/fargate/

[25] Web de Kubernetes. https://kubernetes.io/

[26] Amazon ECR. https://aws.amazon.com/es/ecr/

[27] AWS Batch. https://aws.amazon.com/es/batch/

[28] AWS CloudFormation. https://aws.amazon.com/es/cloudformation/

[29] AWS Elastic Beanstalk. https://aws.amazon.com/es/elasticbeanstalk/
[30] Amazon Lightsail. https://aws.amazon.com/es/lightsail/
[31] Container. https://en.wikipedia.org/wiki/OS-level_virtualization
[32] StackOverflow Survey 2021 - Most Popular Technologies - Other Tools. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-other-tools
[33] Web de Docker. https://www.docker.com
[34] Docker (Software). https://en.wikipedia.org/wiki/Docker_(software)
[35] Docker Hub. https://hub.docker.com
[36] Web de Kubernetes. https://kubernetes.io
[37] Kubernetes. https://en.wikipedia.org/wiki/Kubernetes
[38] Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/
[39] Don't Panic: Kubernetes and Docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/
[40] Red Hat. *Kubernetes adoption, security, and market trends 2021*. https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-2021-overview
[41] YAML. https://yaml.org/
[42] OSCAR. https://github.com/grycap/oscar
[43] EC3. http://www.grycap.upv.es/ec3
[44] IM. http://www.grycap.upv.es/im
[45] CLUES. https://www.grycap.upv.es/clues/eng/index.php
[46] MinIO. https://min.io/
[47] OpenFaaS. https://www.openfaas.com/
[48] Amazon S3. https://aws.amazon.com/es/s3/
[49] EGI Federated Cloud. https://www.egi.eu/federation/egi-federated-cloud/
[50] Naranjo et al., 2020. *Accelerated serverless computing based on GPU virtualization.* doi: 10.1016/j.jpdc.2020.01.004.
[51] General-purpose computing on graphics processing units. https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units#History
[52] Tensor Cores - Nvidia Developer. https://developer.nvidia.com/tensor-cores
[53] Nvidia CUDA. https://developer.nvidia.com/cuda-zone
[54] rCUDA User Guide. rcuda.net/pub/rCUDA_QSG.pdf
[55] Jorrit Sandbrink. *Searching the Clouds for Serverless GPU*. https://towardsdatascience.com/searching-the-clouds-for-serverless-gpu-597b34c59d55
[56] Yang et al., 2019. *PyPlover: A system for GPU-enabled Serverless Instances*. https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F19/projects/reports/project9_report_ver5.pdf
[57] Hendrickson et. al., *Serverless Computation with OpenLambda*. https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf
[58] Kuda. https://github.com/cyrildiagne/kuda

[59] Web de Knative. https://knative.dev/
[60] SCAR - Local testing. https://scar.readthedocs.io/en/latest/testing.html
[61] udocker. https://github.com/indigo-dc/udocker
[62] nvidia-docker. https://github.com/NVIDIA/nvidia-docker
[63] Use multi-stage builds. https://docs.docker.com/develop/develop-images/multistage-build/
[64] Web de Darknet. https://pjreddie.com/darknet/yolo/
[65] Minideb. https://github.com/bitnami/minideb
[66] Web de Infiniband Trade Association. https://www.infinibandta.org/
[67] OSCAR-CLI. https://github.com/grycap/oscar-cli/
[68] Caballer et. al., 2019. *Multi-elastic Datacernters: Auto-scaled Virtual Clusters on Enery-Awayer Physical Infrastructures*. https://doi.org/10.1007/s10723-018-9449-z
[69] Amazon SQS. https://aws.amazon.com/es/sqs/
[70] Golang. https://golang.org/
[71] Why goroutines instead of threads? https://golang.org/doc/faq#goroutines
[72] TOP500 - AWS Adds New Instances, Network Enhancements for HPC. https://www.top500.org/news/aws-adds-new-instances-network-enhancements-for-hpc/
[73] Web de GNU Time. https://www.gnu.org/software/time/
[74] Precios de AWS Lambda. https://aws.amazon.com/es/lambda/pricing/

Anexos

Anexo 1. Traza de ejecución de SCAR en AWS Lambda

En este anexo se muestra el contenido de los registros de CloudWatch para la inicialización y una ejecución de una función de SCAR, en los que es posible comprobar el orden de eventos de una ejecución sin errores, las características de la función o la duración total, entre otros datos.

Tabla 2. Traza de ejecución de SCAR

Timestamp	Mensaje
1627315016642	START RequestId: 709c0146-ec23-475b-b808-f1e09e93f4e7 Version: $LATEST
1627315016663	2021-07-26 15:56:56,663 - supervisor - INFO - Storage event found.
1627315016663	2021-07-26 15:56:56,663 - supervisor - INFO - S3 event created
1627315016664	2021-07-26 15:56:56,663 - supervisor - INFO - Reading storage configuration
1627315016725	2021-07-26 15:56:56,725 - supervisor - WARNING - There is no storage provider defined for this function.
1627315016725	2021-07-26 15:56:56,725 - supervisor - INFO - SUPERVISOR: Initializing AWS Lambda supervisor
1627315017205	2021-07-26 15:56:57,205 - supervisor - INFO - Found 'S3' input provider
1627315017205	2021-07-26 15:56:57,205 - supervisor - INFO - Downloading item from bucket 'alucloud-lambda' with key 'intermediate/0912cbae-6e07-46e7-be39-dbfae41775df.tar.gz'
1627315017335	2021-07-26 15:56:57,335 - supervisor - INFO - Successful download of file 'intermediate/0912cbae-6e07-46e7-be39-dbfae41775df.tar.gz' from bucket 'alucloud-lambda' in path '/tmp/tmpxl0fpnup/0912cbae-6e07-46e7-be39-dbfae41775df.tar.gz'
1627315017337	2021-07-26 15:56:57,337 - supervisor - INFO - INPUT_FILE_PATH variable set to '/tmp/tmpxl0fpnup/0912cbae-6e07-46e7-be39-dbfae41775df.tar.gz'
1627315017885	2021-07-26 15:56:57,885 - supervisor - INFO - Pulling container 'mrconrui/oscar-test:tiny' from Docker Hub
1627315023958	Downloading layer: sha256:c28dd9cbd8a0cd31f1a8c1340a81325370d4fe6aafddc0e3dd0f684e20ddcff6
1627315023958	Downloading layer: sha256:8429b238f806274a4f7beea1747771b7360cc4fd9db45d86b2db5f7dac81c636
1627315023958	Downloading layer: sha256:d1948a2355dfa9b123f20141231deb58dd43b81bf09f71a2b3b282b7a910cf1a
1627315023958	Downloading layer: sha256:edf6abdba515b65987c805ec90bf355701c1797fccc7ce9249e8482157b759ac
1627315023958	Downloading layer: sha256:08ae0fcb8c5d0d049a7887979545ca01ad38ee62ebf0c41cdeab9161b3790a91

Timestamp	Mensaje
1627315023958	Downloading layer: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
1627315023958	Downloading layer: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
1627315023958	Downloading layer: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
1627315023958	Downloading layer: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
1627315023958	Info: creating repo: /tmp/shared/udocker
1627315024426	2021-07-26 15:57:04,426 - supervisor - INFO - Creating container based on image 'mrconrui/oscar-test:tiny'.
1627315047704	Info: creating repo: /tmp/shared/udocker
1627315047704	c743e036-8949-31ea-8610-a34565325f94
1627315048590	Info: creating repo: /tmp/shared/udocker
1627315048671	2021-07-26 15:57:28,670 - supervisor - INFO - Executing udocker container. Timeout set to '857' seconds
1627315900422	2021-07-26 16:11:40,421 - supervisor - INFO - Searching for files to upload in folder '/tmp/tmphrije8ea'
1627315900422	2021-07-26 16:11:40,422 - supervisor - INFO - Checking files for uploading to 's3' on path: 'alucloud-lambda-out/24'
1627315900434	2021-07-26 16:11:40,434 - supervisor - INFO - Uploading file '24/predictions.png' to bucket 'alucloud-lambda-out'
1627315900614	2021-07-26 16:11:40,613 - supervisor - INFO - Uploading file '24/log.txt' to bucket 'alucloud-lambda-out'
1627315900709	2021-07-26 16:11:40,709 - supervisor - INFO - Creating response
1627315900712	END RequestId: 709c0146-ec23-475b-b808-f1e09e93f4e7
1627315900712	REPORT RequestId: 709c0146-ec23-475b-b808-f1e09e93f4e7 Duration: 884058.13 ms Billed Duration: 884059 ms Memory Size: 512 MB Max Memory Used: 512 MB Init Duration: 608.63 ms

Anexo 2. Cálculo del precio de una función Lambda

A partir del razonamiento explicado en los ejemplos del portal para precios de AWS Lambda, es posible calcular el coste en función del número de invocaciones, la duración y la elección de recursos para la función. Para el caso de la sección 4.1.2, en el que se tiene una función con 512 MB asignados a la que se hacen 3 millones de peticiones de 1s de duración y una tarifa de 0.00001667 $ por cada GB/s de ejecución y 0.20 $ por cada millón de llamadas, el coste total sería:

$$(\frac{p * d * m}{1024} - e_{grat}) * t_e + (p - p_{grat}) * t_m$$

donde:

p es el número de peticiones,
d es la duración de cada petición en segundos,
m es la memoria asignada a la función en MB,
e_{grat} son los GB/s de la capa gratuita (actualmente, 40000 GB/s),
t_e es la tarifa por cada GB/s de ejecución, en dólares,
p_{grat} es el número de peticiones de la capa gratuita (actualmente, un millón de pets.),
t_m es la tarifa por cada millón de peticiones,

y sustituyendo las cifras anteriores,

$$(\frac{3000000 * 900 * 512}{1024} - 400000) * 0.00001667 + (3000000 - 1000000) * 0.20 = 22498.23\$$$

En la sección 4.1.2 también se menciona el coste mensual de una instancia t3.2xlarge en AWS EC2. Dado el esquema de precios del servicio y el hecho de que esta instancia no está incluida en la capa gratuita, calcular su coste es tan sencillo como hacer el producto de la tarifa horaria por el número de horas en un mes. Si se supone que el coste por hora de una máquina t3.2xlarge es de unos 0.3328 $, se tiene un total de:

$$0{,}3328 * 24 * 30 = 239{,}62\$.$$

Este coste puede incrementarse si se hace uso de otras características de EC2 como la transferencia de datos o el balanceo de carga elástico.