Multi-tier GPU Virtualization for Deep Learning in Cloud-Edge Systems

Jason Kennedy, Vishal Sharma, Blesson Varghese, and Carlos Reaño

Abstract—Accelerator virtualization offers several advantages in the context of cloud-edge computing. Relatively weak user devices can enhance performance when running workloads by accessing virtualized accelerators available on other resources in the cloud-edge continuum. However, cloud-edge systems are heterogeneous, often leading to compatibility issues arising from various hardware and software stacks present in the system. One mechanism to alleviate this issue is using containers for deploying workloads. Containers isolate applications and their dependencies and store them as images that can run on any device. In addition, user devices may move during the course of application execution, and thus mechanisms such as container migration are required to move running workloads from one resource to another in the network. Furthermore, an optimal destination will need to be determined when migrating between virtual accelerators. Scheduling and placement strategies are incorporated to choose the best possible location depending on the workload requirements. This paper presents AVEC, a framework for accelerator virtualization in cloud-edge computing. The AVEC framework enables the offloading of deep learning workloads for inference from weak user devices to computationally more powerful devices in a cloud-edge network. AVEC incorporates a mechanism that efficiently manages and schedules the virtualization of accelerators. It also supports migration between accelerators to enable stateless container migration. The experimental analysis highlights that AVEC can achieve up to 7x speedup by offloading applications to remote resources. Furthermore, AVEC features a low migration downtime that is less than 5 seconds.

Index Terms—Edge Computing, Accelerators, Virtualization, Containers, Migration.

1 Introduction

The compute offerings at the edge of the network are emerging to complement resources provided by the cloud. While the cloud offers resources with significant computational power, the edge offers multiple tiers of relatively weaker resources. The edge is generally geographically closer to end-user devices and ensures data processing is carried out closer to where the data is generated. This in turn reduces traffic to the cloud and the overall communication latency of applications. In this paper, we propose our framework *AVEC* for enabling the virtualization of GPU accelerators within these networks. The AVEC framework incorporates mechanisms for edge virtualization, migration and workload placement considered in this article.

Computing at data centres or remote clouds allows a large number of users to access powerful resources for computation and storage. These advantages have led to large numbers of users accessing data centres for computing, which in turn leads to a large amount of data being generated and sent to the cloud. This computing model is challenged when the user base expands and application demands increase the scalability and resilience of the centralized cloud [1]. An evaluation of cloud and edge resources for completing latency-sensitive applications underpinned by deep learning that this paper considers was performed [2]. GPU tasks were offloaded to dedicated edge servers with reasonable computing power. It was noted that the edge is able to lower overall latency by reducing propagation delays between device and offloading, provided requirements such as bandwidth speed are met. Furthermore, existing literature shows that paradigms relying on cloud architectures are inadequate in areas like IoT computing. That is due to factors such as high bandwidth usage, unstable connections or high latency [3].

The paradigm we seek to deploy AVEC within is cloud-edge computing. In this paradigm, the network typically consists of three tiers: user level, edge level and cloud level. At each level in the network, devices with varying hardware capabilities are employed. Most importantly for AVEC, devices throughout the network will optionally have GPU accelerators with different capabilities. With AVEC, we seek to offload GPU-based computations from user devices and deploy them on nodes located in the edge or the cloud. To achieve this, we use docker containers, as this allows AVEC to easily and quickly deploy the offloaded GPU computations at any node in the network. To accommodate mobile users and to improve the robustness and availability of the edge, we enable stateless container migration of these offloaded workloads, deployed in containers, around the network.

It should be noted that remote accelerator virtualization used in cloud data centres cannot be directly applied to cloud-edge computing due to the differences with these paradigms. For example, cloud-edge computing is a distributed network, whereas data centres are centralized. This leads to differences in latency and bandwidth speeds, meaning that some applications will not be satisfied by the cloud [4]. As such, a method for appropriately placing workloads in the continuum is needed. Furthermore, edge nodes vary greatly in terms of computational power and

- *Jason Kennedy and Vishal Sharma are with Queen's University Belfast, United Kingdom. E-mail: {jkennedy49, V.Sharma}@qub.ac.uk.*
- *Blesson Varghese is with University of St Andrews, United Kingdom. E-mail: bv6@st-andrews.ac.uk.*
- *Carlos Reaño is with Universitat de València, Spain. E-mail: carlos.reano@uv.es.*

are generally much more resource constrained than in the cloud [5]. As such, a different method of virtualization is required. Generally, virtual machines are used to achieve this within frameworks designed for GPU virtualization in data centres. With AVEC, we utilize container virtualization, as this is a much more lightweight alternative. The aim is not only lightweight deployment, but also faster start up times.

Achieving accelerator virtualisation in cloud-edge networks is a significant technical challenge since accelerators are heterogeneous [6]; compatibility issues may arise from different hardware and software stacks running on different resources within the network. The two most common mechanisms used for achieving virtualisation and deployment are containers and virtual machines. Containers offer a more lightweight virtualisation solution because they do not virtualize the kernel space of the host and instead share it with the host [7]. This results in faster startup times as well as an increase in the portability of the application within the container. The most popular container virtualisation solution put forward is Docker [8]. Docker allows for software solutions to be packaged together and easily deployed at different locations, alongside its cache system allows for faster deployment.

Cloud-edge networks are dynamic [9], their traffic patterns and availability of resources change over time. If a resource is not available to service a user, then the workloads originally intended to execute on the node will need to be migrated elsewhere In the network. Therefore, a migration mechanism is required [10]. Migration of containers, in the context of accelerator virtualisation frameworks, is the process of moving a running workload from one resource to a destination resource. Container migration can be either stateless or stateful. With stateless migration, the data or state of the original container is not required to be moved to the new container to resume execution. In stateful migration, however, it is necessary to store the data or state of the original container, and it also needs to be moved to the new destination resource. Stateless migration is preferred as there is less data to transfer to the new resource and, therefore, less downtime [11]. Finally, with all systems, there is the possibility of failure at a particular node within the network. These are some of the problems investigated in this paper.

The nature of cloud-edge computing results in the potential for many devices with varying resources and capabilities to be present. As such, when the process of virtualizing a workload takes place, an appropriate remote node should be chosen in which the workload will be placed on. For example, certain workloads may require more computational power, specific hardware or be latency-sensitive. Therefore, these applications with particular requirements can only be placed at specific edge nodes. The problem of placing workloads at appropriate nodes is known as the workload or application placement problem [12]. As such, we propose to include in AVEC a solution to this problem when virtualizing and migrating workloads between nodes. Where to place the scheduler is another aspect to investigate. Other frameworks, such as RTEF [13] or ORCH [14], place the scheduler at the edge devices. In AVEC, we propose to place it on the user devices, as this saves resources on the edge devices.

This article makes the following contributions:

1. The development of an approach for adapting containers for accelerating workloads with virtual GPUs in cloud-edge systems. The proposed approach is implemented within the AVEC framework. Experimental results show that the use of containers using the proposed approach has low overheads and enable accelerator virtualization for resource constrained nodes.

2. A suitable approach to combine containers and stateless migration for inference-based deep learning workloads. The deployment of this approach within AVEC in cloud-edge environments provides live lightweight migration for mobile workloads accelerated by virtual GPUs.

3. Definition of placement heuristics to determine where to offload workloads in the cloud-edge when using frameworks such as AVEC. In addition to general case heuristics, we present a fine grained edge adaptive placement heuristics. This enables a wide range of applications access to capable edge nodes for the different applications specific needs.

4. General guidelines are provided on how overall performance is influenced by the proposed approach. This is achieved by an experimental evaluation that is carried out using AVEC across a range of applications with different requirements.

The remainder of this paper is organised as follows. Section 2 presents related work. Section 3 discusses the research questions. Section 4 presents the AVEC framework. Sections 5 and 6 discuss migration and scheduling within AVEC, respectively. Section 7 presents experimental studies. Finally, Section 8 concludes the paper.

2 Related Work

This section presents related work on accelerator virtualization, containerization, migration, and scheduling.

2.1 Accelerator Virtualization

There are four primary methods for virtualizing accelerators, namely API interception, pass-through, mediated pass-through and direct pass-through [15]. These methods are used for virtualizing accelerators in a wide range of areas, such as high-performance computing (HPC), edge computing or cloud computing.

Regarding HPC, general-purpose computing on graphics processing units (GPGPU) is a popular topic. Remote CUDA (rCUDA) [16] is one middleware-based solution that virtualizes remote GPUs using API interception. The middleware intercepts CUDA calls on a client node and forwards them to a remote node where the physical GPU is located. This allows for the GPU accelerator to be logically decoupled from the physical node, thereby allowing for other clients to access and share the same physical GPU on the server. VOCL [17] is a similar framework that is developed for OpenCL. vCUDA [18] is another CUDA based accelerator virtualization solution for HPC clusters. This solution incorporates remote procedure calls (RPC) as opposed to the middleware approach.

In the context of cloud computing, GVirtuS [19] is another accelerator virtualization solution. GVirtuS uses virtual machines and the TCP/IP model to enable remote accelerator virtualization in the cloud. GVirtuS is able to run on any kind of hypervisor. However, the choice of hypervisor affects the performance. GVirtuS employs API interception through the use of a CUDA wrapper library. qCUDA [20] is another cloud-focused accelerator virtualization solution. This solution is, however, hypervisor dependent and is designed for the QEMU-KVM hypervisor.

Although there is no related work that focuses specifically on edge computing, we find some works on accelerator virtualization using edge devices. In this sense, ARM-based single-board computers (SBC) are a promising avenue, as these lower-powered devices would be suitable for edge environments. For instance, GVirtuS incorporated accelerator virtualization for ARM (SBC) [21]. The framework offloads workloads from the ARM-based SBCs to remote accelerators. This work uses API interception to achieve accelerator virtualization. qCUDA-ARM [22] is another work. It is an extension of the qCUDA framework previously commented on and incorporates SBCs. In order to apply qCUDA to SBCs, the application source code has to be modified, as certain functions within CUDA are not supported in ARM devices. These works show that it would be feasible to use low powered SBCs in cloud-edge environments for accelerator virtualization.

In summary, we can conclude that existing accelerator virtualization solutions do not take into consideration important aspects of cloud-edge environments. There is significant research on virtualizing GPUs for paradigms such as high-performance computing (HPC) systems as discussed above. However, there is a lack of research in the context of cloud-edge computing. Remote accelerator virtualization in HPC cannot be directly applied to cloud-edge computing due to the differences in the natures of these paradigms. For example, cloud-edge computing is a distributed network, whereas HPC is centralized. This leads to differences in latency and bandwidth speeds which need to be accounted for in cloud-edge computing. Furthermore, the solutions discussed are designed for virtual machines, unlike lightweight deployments employed in cloud-edge, such as containers. Therefore, we propose to use as starting point our novel framework AVEC, which we use to explore the potential of introducing accelerator virtualisation within a cloud-edge environment. More specifically, this paper focuses on offloading deep learning kernels to perform the inference required by applications in remote virtual accelerators.

2.2 Containerisation

The two most commonly deployed methods for virtualization are containers and virtual machines. Existing literature presents a multitude of scenarios involving virtualization with containers, as well as comparisons between containers and virtual machines. Ramalho et al. [23] contribute a performance comparison between these two technologies. They observe that in all scenarios, Docker containers achieve almost native performance, whereas noticeable overheads are incurred when deploying KVMs. Xu et al. [24] examine the overheads incurred when deploying Docker containers to execute a variety of machine learning algorithms, as well as study areas similar to previous work. They report similar findings as to the previous work, and additionally, they note that overheads from using machine learning algorithms are also minimal (below 5%). Cloud-edge computing nodes are commonly deployed using SBCs such as Raspberry Pi[1]. These devices can be described as resource-constrained. Due to this, containers are a desirable virtualization solution for SBCs. Mendki [25] experiments with deploying containers on Raspberry Pi for deep learning analytics and reports that Docker containers do not provide additional overheads on the SBC. Nvidia-Docker[2] and Litener [26] are solutions being put forward to ease developers with harnessing the power of GPUs deployed in containers. The research indicates that containers are an optimal virtualization and deployment method for AVEC.

2.3 Migration

A key feature for containers in cloud-edge computing is the ability to migrate containers as they are running. The most common approach to implement this is through the checkpoint and restore in userspace (CRIU)[3]. Voyager [27] is one such framework that attempts to implement migration using CRIU. Voyager is a file system agnostic approach that leverages the data federation capabilities of union mounts to minimize migration downtime.

Puliafito et al. [11] categorize migration mechanisms into two groups: stateless and stateful. Stateful migration requires data transfer from the old container to the new container. Stateless migration means starting from scratch on the new container and, in some circumstances, is preferred as there will be less downtime because there will be fewer data transfers. In their experimental evaluation, the authors test both categories of migration and note that migration leads to non-negligible benefits for the application. However, there is also unavoidable downtime that will occur while the migration takes place. In addition, the potential for a loss of data, such as image frames during the migration, must be considered.

Due to the distributed nature of cloud-edge networks, migration of containers should attempt to limit how much data is transferred to lower bandwidth usage. One such attempt to improve migration from this perspective is the work produced by Ma et al. [28]. They leverage the layered nature of the storage system in Docker containers to remove redundant file transfers. Only the top layer of a Docker container can be modified, which means that all underlying layers remain constant. Dockers caching mechanism applies a unique ID to each layer, but the repeated layers will have different IDs, resulting in unnecessary transfers of layers. The authors address this issue and remove the redundant transfers. Additionally, they incorporate a mechanism for transferring the base memory image ahead of the transfer. Another approach for migrating containers is the work produced by Bellavista et al. [10]. They propose a solution that is application and device-aware. Using predictive modelling they are able to achieve migration speeds 50% faster with only minimal overhead. Deshpande et al. [29] propose an architecture for enabling container migration across embedded platforms on edge. They propose migrating Docker containers to prevent the clients' workload from failing in

1. https://www.raspberrypi.org/
2. https://github.com/NVIDIA/nvidia-docker
3. https://www.criu.org

the event of an edge node being unable to complete the request. The challenges they aim to overcome consist of eradicating downtime of containers, a fault-tolerant architecture, working in a dynamic environment like the edge and dealing with unbalanced load clusters.

In summary, we conclude again that containers are a potential solution for implementing the approach proposed in this paper. As mentioned before, this is due in part to the lightweight nature of containers and the tendency for edge nodes to be resource-constrained. Furthermore, in terms of migration, containers allow for stateless migration, which minimizes the overheads in the environment under analysis. Our approach combines containers and stateless migration for inference-based deep learning workloads. The deployment of this approach in cloud-edge environments provides low overhead and live migration for mobile workloads accelerated by virtual GPUs.

2.4 Scheduling

The problem of workload placement has been considered in the literature [12]. It is also referred to as the job dispatchment problem, where latency and the resources required are accounted for when scheduling a job [30]. Essentially, we must figure out which destination node is the best for a particular workload. Many works have investigated this in relation to edge networks. For instance, works such as [31] and [32] investigate application-aware task placement in edge networks. This is important within edge networks as applications may have particular requirements, such as latency ones. Thus, it is important to gather metrics pertaining to destination nodes, such as network bandwidth, latency, etc. As these works do not focus on workloads using accelerators, there are additional metrics pertaining to accelerators that must be considered, such as accelerator memory available.

In the context of scheduling or placement of workloads in an edge network with regards to accelerators, there are a few works that investigate this problem. Firstly, works such as [33] and [34] investigate Field Programmable Gate Arrays (FPGA). These works highlight the need for incorporating accelerators alongside CPU usage in order for these resource-constrained edge nodes to be as powerful as possible to meet user requirements. FPGAs are one possible accelerator that can be used at the edge due to their high energy efficiency. In this paper, however, we focus on GPU accelerators. Regarding workload placement for GPU accelerators in cloud-edge environments, there are few works available. Once such work provided by Zhang et al. [35] presents a platform built on Apache Storm[4] with a focus on minimizing latency. As Apache Storm built-in scheduling algorithms are not GPU aware, they propose their own heuristic with three main components: (1) estimate performance/requirements of task, (2) track available resources in the cloud-edge, and (3) a latency aware task scheduling algorithm. Regarding the latter, it is only based on the CPU utilization and the bandwidth of a node; summing these two values together gives the total latency value of a node. This solution is bound to Apache Storm, as it relies on using the topologies and bolts associated with Apache Storm. Bensalem et al. [36] investigate specifically placing deep neural network (DNN) inference models in an edge environment using a mathematical model. They also provide three heuristics to solve this: a general heuristic solution, a random placement algorithm and then an inference assignment algorithm. They focus on utilization cost and latency between nodes within these heuristics. They place emphasis on the benefits of sharing GPUs between workloads.

Our approach proposed in this work with regard to scheduling and placement of workloads advances this field of research as follows. In the first place, it offers lightweight virtual GPU scheduling. In addition, it takes into account the potential mobility of end-user devices and the nature of cloud-edge environments. This is achieved through locating the scheduler on the user device itself, as well as by considering metrics such as GPU memory, latency or bandwidth at the destination device.

3 Research Questions

This section describes the main motivations behind AVEC by discussing the research questions we seek to answer. Based on the above, the research reported in this paper investigates *the use of containers, stateless migration and scheduling to provide the flexibility required by applications accelerated by virtual GPUs in cloud-edge environments*. More specifically, the following research questions are addressed:

Q1: *How can accelerator virtualisation be offered at the edge given its limited compute resources?* To address this question, in this paper we present AVEC (accelerator virtualisation in cloud-edge computing). It is a framework that supports the use of virtual GPU accelerators in the cloud-edge continuum. AVEC provides transparent accelerator virtualisation through the use of API interception.

Q2: *Considering the heterogeneous nature of the edge, how can we deploy GPU accelerator virtualisation?* Containers offer OS level virtualization, which along with the benefit of being a more lightweight approach to alternative virtualization techniques, allows for containers using a base OS image to be run on hosts with different OS's. Furthermore, Docker has been shown to run on a wide range of CPU architectures such as ARM or x86 [25]. By preparing a container using images for x86 and also for ARM, we can choose at runtime which container is to be loaded depending on the host architecture. Nvidia container technology[5] allows for local physical GPUs on the host to be accessed by the container running on the host. We investigate addressing this challenge in the cloud-edge continuum to achieve the virtualisation of remote GPUs. We also investigate additional overheads incurred using this as opposed to a native domain.

Q3: *Considering the dynamic nature of the edge, how can we provide migration capabilities to applications accelerated by virtual GPUs?* To overcome some of the challenges associated with the edge, we explore stateless container migration. This helps with user mobility, fault tolerance, availability of network resources, etc. In addition, migration downtime is reduced compared with stateful container migration.

4. https://storm.apache.org/

5. https://developer.nvidia.com/nvidia-container-runtime

Q4: *Considering the requirements of different applications, where do we place each workload?* To answer this question, we implement a scheduling mechanism within the AVEC framework. It uses varying heuristics to place each application based on metrics such as latency, GPU memory available or network speed. Furthermore, resource monitoring of different nodes within the network allows for optimal placement.

Q5: *Can the AVEC approach be generalised across a range of applications?* We evaluate AVEC on three different applications with varying amounts of computational power requirements. We observe that applications with high computational power requirements benefit more from AVEC as opposed to more lightweight applications.

Q6: *What are the additional overheads incurred through deployment, migration and scheduling of GPU accelerator virtualisation in the edge?* In our experimental evaluation, we test the capabilities of the approach proposed and compare it with previous versions of the AVEC framework not supporting container migration. In this evaluation, we observe minimal additional overheads due to the lightweight nature of Docker alongside the implementation of stateless migration as opposed to stateful migration.

4 AVEC: Accelerator Virtualization in Cloud-Edge Computing

This section presents an overview of the AVEC framework. It introduces the design considerations and discusses the individual components that make up the framework. AVEC aims to provide remote GPU acceleration to inference based workloads. AVEC aims to reduce execution time of these workloads by leveraging more powerful hardware accelerators located in the cloud-edge network. The Caffe library is open source and offers deep learning algorithms and models for C++ and Python. These libraries usually make use of frameworks to leverage GPUs, such as CUDA[6]. AVEC executes Caffe kernels within cloud-edge computing using an accelerator virtualization approach. Currently, AVEC supports the Caffe library. It is expected that it will support additional libraries in the future.

We briefly consider the terms used in this article. “User device” (also referred to “host” and ”host device”) is the device in which the application is being executed. The “edge node” and the “cloud node” are the nodes in which the deep learning tasks will be offloaded to. The edge node is geographically closer to the user device than the cloud node, but the cloud node is computationally more powerful than the edge node. The terms “cloud-edge”, “cloud-edge node”, “remote node” or “remote host” are used to refer to both the “edge node” and the “cloud node”.

A GPU is normally accessed through calls to an API such as the one provided by CUDA. As the CPU on the host device attempts to access these APIs, it is possible to redirect or hook these calls and forward them to another library. While this technique may be used to access a GPU on the same physical host, it can also be used to provide access to GPUs installed in a remote host. Remote virtual GPUs use a front-end/back-end model in which the front-end intercepts all API calls (and their data) and transfers them to a selected back-end for execution [37]. The front-end data that has been intercepted includes data such as input parameters for CUDA functions. The output of these functions is returned in the same manner as the input data was sent. With the calculated output received from the remote GPU, the user device can continue execution of the application normally. This entire process, from the front-end intercepting data up until the output is sent back to the host device, is called API Interception.

With the creation of the programmable graphics pipeline, graphics hardware was capable of rendering graphics data from application data. Consequently, this pipeline was eventually adapted to execute non-graphics related data [38]. Using GPUs for tasks outside of graphics visualization is known as General-Purpose Graphics Processing Unit (GPGPU). GPUs, in contrast to CPUs, have many more cores. This allows for a higher level of parallelism, which results in certain tasks executing faster. GPGPU is seeing usage in areas such as machine learning, computer vision, scientific computing and modeling [39]. For the purposes of this paper, we focus solely on deep learning inference base workloads. In future our work will extend to other types of workloads that make use of GPUs.

AVEC is a middle-ware that intercepts calls made to a GPU-accelerated library (i.e. Caffe) and forwards the input parameters to a remote GPU in the edge or cloud for execution. The user device side instance of the middle-ware, where the application is executing, is responsible for intercepting and forwarding the library functions. AVEC features a scheduler or placement mechanism that is responsible for choosing which remote node to use. Additionally, this scheduler manages migration between cloud-edge nodes when required. Finally, the user device will, have its instance of the middle-ware communication module to communicate with remote nodes.

The destination side instance of the middle-ware, situated on the remote node receives the input parameters that are sent by the user device to be used in the execution of the inference function. The output of that execution is sent back to the original user device, where execution can continue using the received values. A server engine is running on the physical device to receive requests from user devices to facilitate this. This engine will then spawn and delete containers as needed. An architecture diagram in Figure 1 shows an overview of the previously explained modules. In order to aid with the deployment in the cloud-edge environment, AVEC uses Docker[7] containers. Docker has a built-in cache system that saves recently used containers. This cache system means that, as long as the container image has not been altered, the container does not need to be pulled again from the repository for each execution.

As mentioned, Figure 1 shows the primary components that make up the AVEC framework. A more in depth analysis of these components is as follows:

1: *Interception Library*. AVEC is designed to act as middle-ware. Application calls to the Caffe library are intercepted and redirected to the AVEC library. Within this library, the original Caffe library functions are present but have been

6. https://developer.nvidia.com/cuda-zone

7. https://www.docker.com/

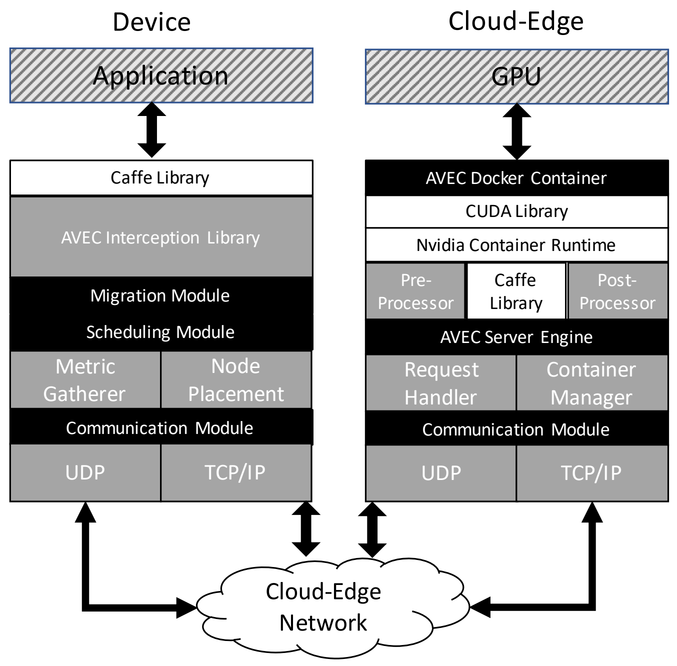


Fig. 1: Overview of the AVEC Framework

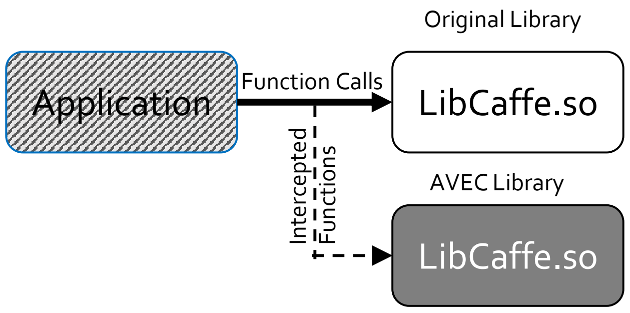


Fig. 2: The process of API Interception within the AVEC framework

modified to include the additional functionality required to send the necessary data for the Caffe function to the remote node. Within AVEC, we seek to enable this virtualization method in a transparent manner. Thus, the source code of the application requires no modification and additionally, the functionality takes place in the background, without input from the application or user. To do this, we use the Linux 'LD_PRELOAD' technique, which allows for the AVEC library to take precedence over the Caffe library at run-time. A diagram displaying this is shown in Figure 2.

2: *Communication Module*. The framework employs a communication module to transfer data between the host and destination nodes. The original application on the host continues executing using the output of the remote kernel. The communication between the host and destination is done via TCP/IP using Boost ASIO[8]. The host node in the cloud-edge environment may be a device or an edge node that does not have an accelerator or a sufficiently powerful accelerator. Once the interception takes place on the host node, the library creates a connection between the host and destination nodes. This module uses both User Datagram Protocol (UDP) and TCP/IP connections. The destination node hosts a physical GPU and the original Caffe library. Once the destination node receives the forwarded requests, it executes the kernels required by the host node on the physical GPU of the destination node. The output of these functions is sent back to the user node in the same manner as they were received.

8. https://www.boost.org

3: *Server Engine*. At each destination node, a system is required to enable a connection between the user device and the accelerator of the remote node. The server engine runs on the physical machine and is initially placed in a listening state, awaiting UDP messages. UDP is used in the discovery phase to identify nodes on the network using the UDP broadcast function. The server engine is responsible for sharing its metrics with user devices via these UDP messages. The scheduler will use the metrics to determine the appropriate node to offload to. UDP messages are also used to communicate to the server engine that a user device has chosen its node. If this happens, the server engine will spawn the container, which is saved in its cache. After that, it will notify the user device that the container is ready and the information needed to connect to the container. After this, communication between the user device and the container server is carried out over TCP/IP. At this point, the container serves that user device while the sever engine waits for other connections.

4: *Container*. In fulfilling these design considerations, challenges had to be overcome. The heterogeneous nature of the edge can potentially cause compatibility issues. Containers can help overcome these issues, given their operating system independence. Due to the architectural differences of devices, specific container images are required for each architecture. For example, an image for arm64 and a separate image for x86. Docker contains a registry where container images can be saved, allowing either of these images to be acquired. As the nodes at the edge are resource-constrained, the images themselves should be as lightweight as possible.

5: *Migration Module*. The migration module is responsible for monitoring application progress to determine if the user device requires its execution to be offloaded to a different node than the node it is currently being served by. For that purpose, the migration mechanism is able to halt the current progress of the execution of the application on the user device and then send the signal to the scheduler to locate a new remote node to offload. Given the nature of inference workloads, all that is required is the algorithm as well as the input data. More details on this mechanism are discussed in Section 5.

6: *Scheduling Module*. There will be multiple nodes within a cloud-edge network, with each node having differing computational requirements and abilities. As such, it is important that workloads be offloaded to nodes that are suitable for the workload. This is the responsibility of the scheduler. More details on the scheduler module are discussed in Section 6

To better illustrate the the different modules discussed, an activity diagram has been added to show how these modules work together to execute a virtualization workload. This is shown in Figure 3.

The activity diagram begins showing application execution on the host node. Upon a virtualization request, a

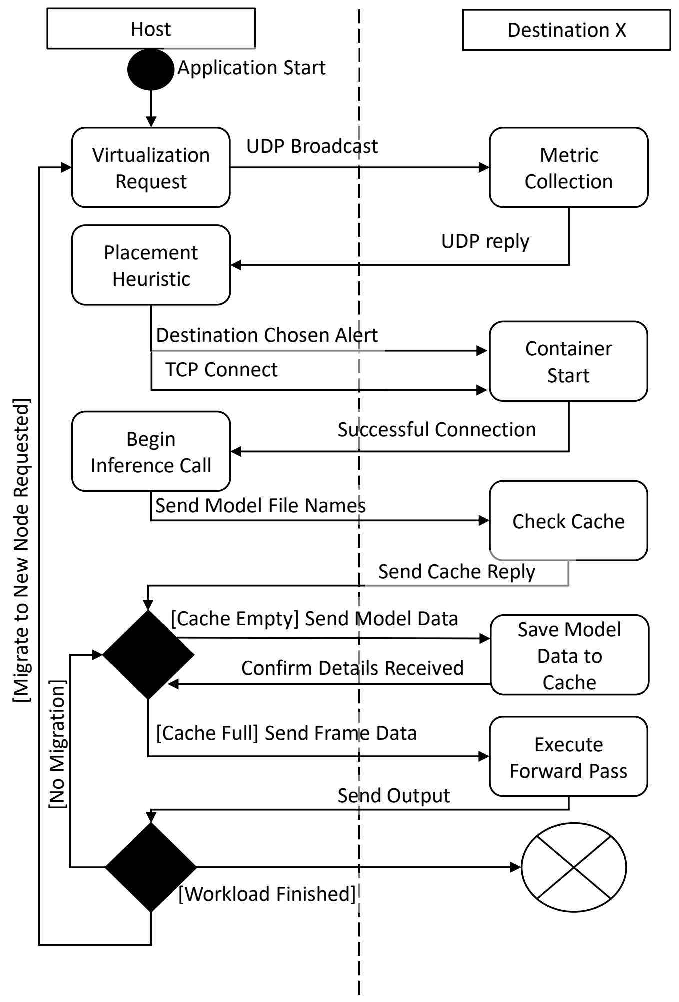


Fig. 3: Activity diagram of workload execution with AVEC.

broadcast signal is sent to remote nodes in the network. This signal is a template requesting metrics pertaining to the remote node, e.g. GPU specifications. The remote nodes fill this template and it is sent in response back to the host. A placement heuristic is carried out using the responses and the output of this heuristic is the chosen node. The chosen remote node will start up its container and a connection is established between the container in the chosen node and the host. Data pertaining to the models needed will be sent to check if the models are present on the remote node. If they are not present, they must be sent across. At this point, the inference can begin taking place with frame data being sent across and output data being sent back to the host. Migration checks are carried out after frames are completed. If a migration is needed, the process restarts with a new node being chosen. If the workload is finished, then the connection ends and the container is destroyed.

5 Container Migration

Live migration is the process of taking a running workload being executed by one device and moving it to another device seamlessly whilst continuing execution on the new device from the point at which migration occurred. Live migration has benefits such as improving fault tolerance in the network as well as the quality of service for mobile users [40].

For this reason, we propose employing stateless container migration within AVEC. To achieve this, we keep the state of the application execution stored locally on the user device, which leaves the user device responsible for tracking its own progress or execution state. By only virtualizing the GPU aspects of the application, we can achieve this. Furthermore, this has the added benefit of a lightweight migration, as we do not need to transfer data between edge nodes in the network.

The motivation for the migration of containers between remote nodes in the cloud-edge continuum is two-fold. Firstly, mobile users with latency sensitive applications. As users move, latency increases. As such, migrating to a closer edge node improves user experience, as it can reduce the latency [41]. Secondly, the availability of edge nodes may vary over time in a dynamic environment, as well as their resource constrained nature, leading to more chances of failure [42]. Therefore, a workload may require a new edge server to meet its requirements. Here migration enables moving the workload from one node to another, as it allows for workloads to be moved to other nodes, whilst continuing the applications execution.

The type of migration used with AVEC is stateless container migration. Stateless container migration is to be preferred when no state needs to be transferred from source to destination. With stateless migration, a new container is started from scratch on the destination node, and the old container is deleted from the source node [43]. This means that at the point of migration, we do not need to send data from the container on the original node to the container on the new migrated node. We are able to achieve this due to the type of workloads being executed, i.e. deep learning inference workloads. These workloads only require the machine learning model (in the case of Caffe, the prototxt file as well) and the associated frame data after it has been processed by the user devices' CPU. Frame data is removed after every frame is processed, so at migration time there will not be any frame data present. With AVEC it is possible to store machine learning models in the container itself, in cache memory of Docker. Due to the resource constrained nature of edge nodes, it may not always be desirable to save these files. If the machine learning model and associated files, they will need to be sent across at migration time, as presented in Figure 3. This will of course add overhead. The time taken to transfer these files depends on network speed. In our experiments, the files used are the following: OpenPose application prototxt (46.4KB) and model (210MB); MaskDetection application prototxt (34.3KB) and model (99.5MB); and Classification application prototxt (2.9KB) and model (243.9MB). The time taken to transfer the files of each application in our experimental setup was 1.814s, 0.861s and 2.103s, respectively.

In the AVEC framework, the host client is responsible for the decision to migrate when required (more details available in next section). When that decision is made, it will then terminate the connection with the current destination node. At this point, the execution of the application has paused on the client. The scheduling module, detailed in the next section, will be used for selecting a new node. The client will attempt to connect with a new chosen remote node. If this connection is successful, the container is then booted up, and the host device connects directly to this new container. Some initial data is sent to the new container to prepare it to continue execution, such as the machine learning model file name. From this point, the host device can continue with the execution of the application from the point of execution it was at when the migration was started.

By not sending data from the previous container to the new container, we are able to reduce the time taken to migrate as well, leading to a lightweight migration, as it will be shown in the experimental studies in Section 7.

6 Workload Scheduling

User device applications that are seeking to be offloaded through AVEC must be placed in a corresponding edge or cloud device. This is achieved through the scheduling module of AVEC. When a workload requests virtual accelerators or when it is being migrated, the scheduler is invoked.

Usually, a centralized approach is adopted for the purposes of scheduling or placement. Wherein, one node in the edge is responsible for retrieving metrics or information about other nodes present in the network and disseminating this information to the nodes when requested. With AVEC, we locate this mechanism on the user device so we can avoid having a centralized node. The benefit of our method is that all connections do not have to arrive at this centralized node, but the downside is that the scheduling may not be as accurate as it could be, i.e. the local device may not be aware of every device in the network due to the quick scheduling nature of UDP broadcast. Furthermore, edge devices are not as reliable as the cloud ones. Therefore, locating the scheduling mechanism on a singular edge node can be problematic. Whilst offloading to a remote node, there are a multitude of reasons why we would need to migrate the workload to another node. User devices may be mobile, as these devices move the latency requirements may become unsuitable for the currently accessed remote node. In situations like this, in which the workload being executed needs to be moved to another node, then having the user device responsible for scheduling has no detrimental effect.

Thus, in AVEC the scheduler is located on the user device itself, as opposed to being hosted in a node in the cloud-edge continuum. It contains a list of possible destination nodes as well as corresponding metrics pertaining to those destination nodes. For the user device to discover possible destination nodes, it uses UDP broadcasting. Thus, UDP messages are sent to all nodes on the local network. These messages are requests for information from each node. A template is filled out by destination nodes. It contains information on metrics such as available GPU memory, and further information such as IP address. Some nodes may not be on the local network, so an additional file manually populated with information about these nodes can be provided. The scheduler uses these metrics to decide which destination node is selected. To have updated data, every time the scheduler is invoked, the process of discovering devices is repeated.

Other works that explore application placement or scheduling in edge environments bring focus to metrics such as bandwidth capabilities at each node, or latency between each node and user device [31; 44]. With the added complexity of a GPU, additional metrics need to be considered, such as the type of GPU, platform capabilities of the GPU, GPU computing power, and available GPU memory. More specifically, the metrics we have chosen to focus on are: GPU memory available, number of GPU cores, latency between devices and bandwidth.

Based on these metrics, AVEC allows the following general case scheduling policies.

Policy 1 - First in, first out. This policy assigns a workload to the first available destination node. A user device will loop through the list of possible nodes that the user device has access to. For each destination node, it will first ensure that the node has the required GPU platform. The policy will then check if there is enough GPU memory available to run the workload. It should be noted that this checking of platform compatibility and GPU memory takes place across all policies.

Policy 2 - Random Placement. This policy initiates by generating a random number. This seed is used to then generate a value representing an index of a stored node in the list of destination nodes. That node will then have the platform and GPU memory checks. If it passes the checks, the node is selected. If not, a new random value (excluding the previous one) is generated, and the process repeats.

Policy 3 - Best bandwidth. The goal of this policy is to assign the workload to the node with the best possible bandwidth speed. Initially, the policy assumes that the first node in the list that meets the GPU memory and platform requirements is the best. Then, the policy loops through the rest of the nodes that meet the requirements by comparing their bandwidths. In the end, the node meeting the requirements with the best bandwidth is selected.

Policy 4 - Best latency. This policy aims to assign the workload to the node with the best possible latency. It is similar to the previous policy *best bandwidth*. The difference is that it checks for latency as opposed to bandwidth.

Policy 5 - Best Computing Power. This policy aims to assign the workload to the node with the best possible computing power. It is similar to the previous policies *best bandwidth* and *best latency* but checks for the largest number of GPU cores as opposed to bandwidth or latency.

Policy 6 - Spread. This policy focuses on spreading work across all available devices. For that purpose, the policy looks for the node with the highest amount of free GPU memory to place the workload in.

In addition to the previous general scheduling policies, we also provide an specific policy for edge computing called *GPU aware edge adaptive*, which is detailed next in Section 6.1.

An investigation into the performance of these policies was carried out. We timed them using the two actual testing nodes from our experimental setup: the edge and the cloud node, described in more detail in Section 7. In addition, we included 20 “dummy” nodes to simulate a larger scenario. Then, we measured the execution time for each policy.

All the policies performed similarly around 0.7 ms, except for the *random* policy, which took 3.1 ms. The increased time for this policy is likely due to the amount of “dummy”

α	*latency weight*	β	*bandwidth weight*
γ	*GPU memory weight*	δ	*GPU cores weight*
N	*current node under analysis*	N_t	*total number of nodes*
N_d	*current node disk space*	D	*disk space required*
N_p	*current node platform*	P	*required platform*
N_g	*current node GPU memory*	G	*GPU memory required*
N_l	*current node latency*	N_b	*current node bandwidth*
N_c	*current node GPU cores*	Y	*best nodes so far list*
N_ws	*current node weight sum*	Y_ws	*best weight score sofar*
N_gu	*GPU utilization current node*	Y_gu	*best GPU utilization*

TABLE 1: Notation used in scheduling policy Algorithm 1.

Algorithm 1 GPU aware Edge Adaptive Scheduling Policy

```
if type == heavyweight then
    $\alpha = 1, \beta = 1, \gamma = 10, \delta = 10$
else if type == middleweight then
    $\alpha = 5, \beta = 5, \gamma = 5, \delta = 5$
else type == lightweight
    $\alpha = 10, \beta = 10, \gamma = 1, \delta = 1$
end if
    let Y = $N_t(0)$    ▷ We assume a least 1 node is capable
for $\forall N \in N_t$ do
    if $N_p == P$ & $N_g \geq G$ & $N_d \geq D$ then
        $N_ws = (\alpha \times N_l) + (\beta \times N_b) + (\gamma \times N_g) + (\delta \times N_c)$
        if $N_ws > Y_ws$ then    ▷ n is now our best value
            clear the list of values in Y
            add n to the list of Y
            $Y = N$
        else if $N_ws == Y_ws$ then
            add N to the list of Y
        end if
    end if
end for
if $\mid Y \mid \geq 2$ then    ▷ >1 best weight, load balance
    for $\forall N \in Y$ do
        if $Y_gu > N_gu$ then
            remove $N_gu$ from Y
        end if
    end for
end if
return Y(0)    ▷ will only be 1 value in Y
```

nodes that do not actually exist. This policy selects one of these nodes, detects it does not work and then selects a different one. That explains why its time is significantly higher.

6.1 GPU Aware Edge Adaptive Scheduling

As commented, AVEC allows a more fine grained placement heuristic called *GPU aware edge adaptive algorithm*. This policy is specifically designed to satisfy differing applications and their requirements in the edge environment. It implements the algorithm shown in Algorithm 1 (the notation used in that algorithm is defined in Table 1). This scheduling policy takes into account the type of application that is being run, and use appropriate weights on the input metrics to reflect the needs of different applications.

Based on empirical experimentation, we divide our test applications into three categories, namely heavy-weight, middle-weight and light-weight. This categorisation is determined by considering metrics such as GPU cores, GPU memory, bandwidth and latency. Heavy-weight applications require more computing power, have high GPU memory usage, and are therefore less impacted by network bandwidth and latency, because the additional time to move data to the cloud-edge node will be offset by the large reduction in computation time. On the contrary, light-weight applications require less computing power, have a lower GPU memory usage, and are therefore more influenced by network bandwidth and latency. Finally, the requirements of a middle-weight application are positioned in between those of heavy-weight and light-weight and as such these applications are moderately affected by all metrics.

A weighting mechanism is used for the metrics considered: (i) GPU cores, (ii) GPU memory, (iii) bandwidth, and (iv) latency. Weights are in the range 1 to 10, with 1 representing the lowest workload impact, and 10 representing the metric having a large impact upon the workload. In our experiments, GPU cores and GPU memory are weighted equally. We assume that heavy-weight applications require more GPU cores and GPU memory than light- and medium-weight applications. Note that as commented below, this can be configured by the user. These weights are assigned at run time. Currently, the weights are static for evaluation, but can be altered as needed. The weight values themselves are chosen to satisfy the three test applications that are used in Section 7. We assume that the only workloads running on the network are from AVEC. We also assume that at least one node in the network has the capability of meeting the requirements.

The algorithm firstly converts all metrics gathered into appropriate integers, i.e. converting string values returned from the remote node that represents free GPU memory. Then, the weighting parameters α, β, γ and δ are configured. Note that these parameters can be configured by the user. They represent how important each metric is for the user. The α parameter is used to weight the latency, β the bandwidth, γ the GPU memory and δ the number of GPU cores. The weights range from 1 to 10, where 1 indicates that the metric has the lowest importance, and 10 the highest. This allows the algorithm to be fine tuned depending on the user's needs. In this case, we have configured the parameters based on the workload type. Thus, for the heavy-weight application, GPU cores and GPU memory have the highest weight. On the contrary, for the light-weight application, bandwidth and latency have the highest weight. For the medium-weight application, all the metrics have a medium weight.

After configuring the weighting parameters, we begin by choosing the first node in the list as the chosen node, and adding it to the list of potential nodes. For every node in the list of all nodes, we check if they meet the hardware requirements (e.g. CUDA capable), that they have enough GPU memory available, and that they have enough disk space available. Every node that meets these requirements will then have a calculated total weight sum. This will be the total value of the latency metric, bandwidth metric, number of GPU cores and GPU utilization metric multiplied by their corresponding weights. The higher the weight sum value, the better the node. After this calculation, the weight sum of the node being tested is compared to the chosen node weight sum. If it has a better value of weight sum, we

	User Device	Edge Node	Cloud Node
GPU Resource	Jetson Nano	RTX 2060	GTX 1080
Flops (GPU)	235 GFLOPS	6.5 TFLOPS	9 TFLOPS
GPU Memory	4 GB LPDDR4 (Shared)	6 GB GDDR6	8 GB GDDR5X
CPU Memory		16 GB DDR 4	32 GB DDR4
GPU Cores	128	1920	2560
CPU Cores	4	8	8

TABLE 2: Devices used in the experiments.

remove the previous chosen node from the list of potential nodes, add the new node to the potential list, and set the new node as the current chosen node. In the event that the new node has the same weight sum as the previous chosen node, we add both of them to the list of potential nodes. After every node has been checked in this manner, we know that the potential nodes list will contain either only the best node, or a list of nodes that all have the same best weight sum. In the event of this tie with multiple nodes with the highest weight sum, we call the *Spread* policy previously explained, which will result in balancing the load in the different nodes.

7 Experimental Studies

This section presents experimental studies implementing the proposed solution in the AVEC accelerator virtualization framework.

7.1 Experimental Setup

For the experimental setup, we use a three-tier cloud-edge network. At the edge and the cloud level, we use GPU accelerators of similar capabilities. The user device is a computationally much weaker embedded device. Details of these devices can be found in Table 2. The edge device, a desktop PC featuring an Nvidia RTX 2060 GPU, is connected to the user device, a Jetson Nano[9], through an Ethernet 1 Gbps connection. Access to the cloud device, fitted with an Nvidia GTX 1080 GPU, is done through broadband. The edge device is kept local with the user device in Carrickfergus, Northern Ireland. The cloud device is located 15 miles away at the university facilities in Belfast, Northern Ireland. Both the edge node and the cloud node are similar in compute capabilities, specifically with their GPU accelerators. It is the connection speeds and types between these devices and the user device what differs. Given the wireless nature of this connection with the cloud device, there is a level of variance due to factors such as bandwidth speeds alternating. This variance is as high as 8%. Subsequent tests that investigate the migration will alternate the user devices connection between both devices, splitting the workload accordingly. As the computation capabilities of both edge and cloud are similar, we predict that the edge device will perform substantially better due to the connection speed.

As commented, in our testing environment the user device is computationally much weaker than the edge and cloud devices. As such, in this scenario the user device seeks to offload the inference workload to a remote node with a much more powerful GPU accelerator. Both the edge node and the cloud node are similar in compute capabilities, specifically with their GPU accelerators. The difference between the edge node and the cloud node lies in the manner of connection to the network. The user device will offload to the edge device through a wired Ethernet connection with 1GBps connection speed, while it will offload to the cloud device through a wireless connection of 40Mbps.

9. https://developer.nvidia.com/embedded/jetson-nano-developer-kit

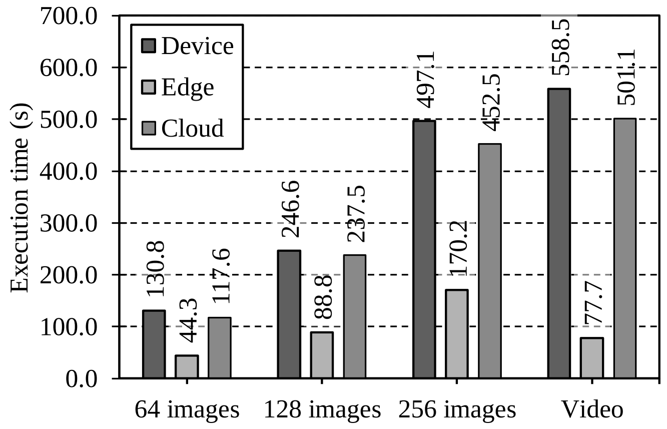


Fig. 4: OpenPose execution time for different image batches and video when using AVEC with both the edge device and the cloud device, as well as the base time without AVEC on the user device.

Tests are carried out using multiple test applications. Firstly, OpenPose [45], which is used across all tests. This application estimates the pose of a person in real-time. It works with both images and videos. For the images, we use the COCO data set [46]. The video is a sample video provided with the OpenPose application. The second application used is an image classification application provided with the Caffe [47] library. It takes a picture as input and makes a prediction of what is in the picture. In the experiments, we use as input data a selection of pictures from an Animal Dataset[10]. The last application is MaskDetection[11]. It takes images as input and detects if everyone in the image is wearing a face mask. This application uses images from a Face Mask Detection Dataset[12]. In the three applications, the images chosen from the data sets are randomly selected and set into batches of 64, 128, and 256 images, respectively. The same batches are reused in each test to keep results comparable.

An experiment was carried out to test the effects of using the VPN connection with the cloud device as opposed to the local connection with the edge device. Figure 4 shows the results of running the OpenPose application using the different workloads in each test scenario (i.e. no using AVEC on the user device, then offloading to either the edge of the cloud). As expected, the connection with the edge device performs substantially better than the connection with the cloud device due to the better connection speed. The cloud device performs slightly better than the user device.

10. https://www.kaggle.com/alessiocorrado99/animals10
11. https://github.com/didi/maskdetection
12. https://www.kaggle.com/andrewmvd/face-mask-detection

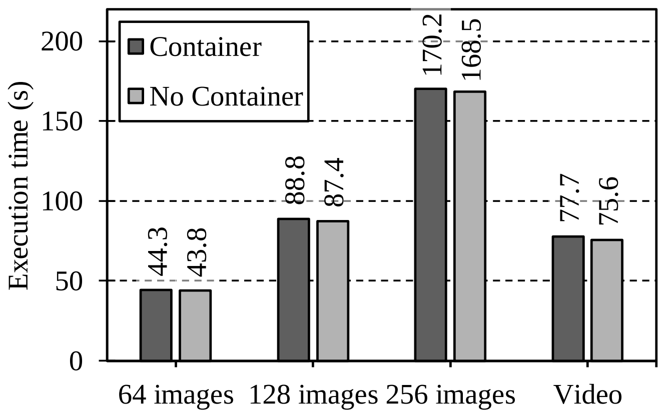


Fig. 5: OpenPose execution time for different image batches and video when using AVEC with and without containers, offloading computations to the edge device.

7.2 Container Deployment

As mentioned before, one of the key factors why we chose to use containers is faster startup times. To confirm it, we carried out experiments to determine if using containers brings any additional overhead to AVEC.

Two tests were carried out for each workload on each device. In the first test, we use AVEC without containers. In the second test, we deploy AVEC in a container in the destination node. Notice that this is not necessary for the host node, i.e. the user device, because the application will always run on the host, and the only part of the application which could potentially be migrated is the one offloaded to the destination devices.

Figure 5 shows the results of this experimentation when using the edge device as the destination node. Similar conclusions were drawn when using the cloud device as the destination node. As it can be seen, the startup time of the container is low, averaging 1.425 seconds (less than 0.4%) in these experiments.

7.3 Container Migration

The next set of tests is carried out to investigate the feasibility of migrating the containers that are executing the offloaded workloads between the cloud device and edge device. During the execution of these tests, we split the workloads into chunks depending on the number of migrations being tested. For example:

- 1 migration: the workload is divided into two chunks. The first workload chunk runs in the edge device. Then it is migrated to the cloud device, where the second workload chunk is executed.
- 3 migrations: the workload is divided into four chunks. The first workload chunk runs in the edge device. Then, it is migrated to the cloud device, where the second workload is executed. Migration then takes place back to the Edge device for the third chunk to be executed. Finally, it is migrated again, and the fourth workload chunk runs in the Cloud device again.
- 5 migrations: the workload is divided into six chunks. Migrations are carried out following a similar sequence as in previous examples.

As shown in the examples above, the offloading device used at the start of the execution is always the edge device. Then, the offloaded computation is migrated from the edge device to the cloud device and vice versa, depending on the number of migrations performed.

Across these tests, we chose to use an odd number of migrations, which enables an even number of work chunks. Therefore, the workload is split equally between the cloud and edge devices. Figure 6 shows the results of these tests. The figure also includes the downtime. It represents the amount of time in which the test application is not being executed because the workload is being migrated from the edge device to the cloud device or vice versa. Thus, in our case, downtime is the amount of time the migration takes. In the previous section, we evaluated the additional overhead of using containers with AVEC (1.425 seconds, on average). In this section, in addition to using containers, we are also migrating them. The downtime depends on the device we migrate to. Migrating to the cloud device takes approximately 3.8 seconds, whereas migrating to the edge takes around 2.3 seconds.

7.4 Scheduling

In this section, the scheduler component of AVEC is evaluated. It was necessary to expand the experimentation analysis to include two additional applications, bringing the total to three, namely OpenPose, MaskDetection and Classification. OpenPose will be carried out using the same data as before with 64, 128 and 256 Images, as well as the video. The Classification and MaskDetection applications will be examined using only the batches of 64, 128 and 256 images, as these applications do not support videos. The three applications require different computational power: the Classification application is lightweight, OpenPose is a compute-intensive or a heavy-weight application, and MaskDetection is a middle-weight application. Experiments were done considering the five scheduling policies discussed in Section 6. Due to lack of space, we only show results for the "Random placement" scheduling policy, as it is the worst-case scenario.

The first application to be tested is OpenPose. Results are shown in Figure 7. As expected, as the number of migrations increases, the total execution time also increases. The majority of the time is spent executing the application, as opposed to carrying out the migrations. If we compare these results to the ones in Figure 6, we can see that the downtime increases and thus the overall application execution time. This is because we have added a new component, i.e. the scheduler, on top of the components introduced in previous tests. More specifically, the amount of downtime taken per migration increases by approximately 1 second, which is the time that the scheduler requires to retrieve the necessary data from local network devices. Thus, we can conclude that the scheduler itself introduces minimal overhead. However, the more components we add, and the more migrations we do, results in a gradual increase in the amount of downtime. For example, if we consider Figure 7c,

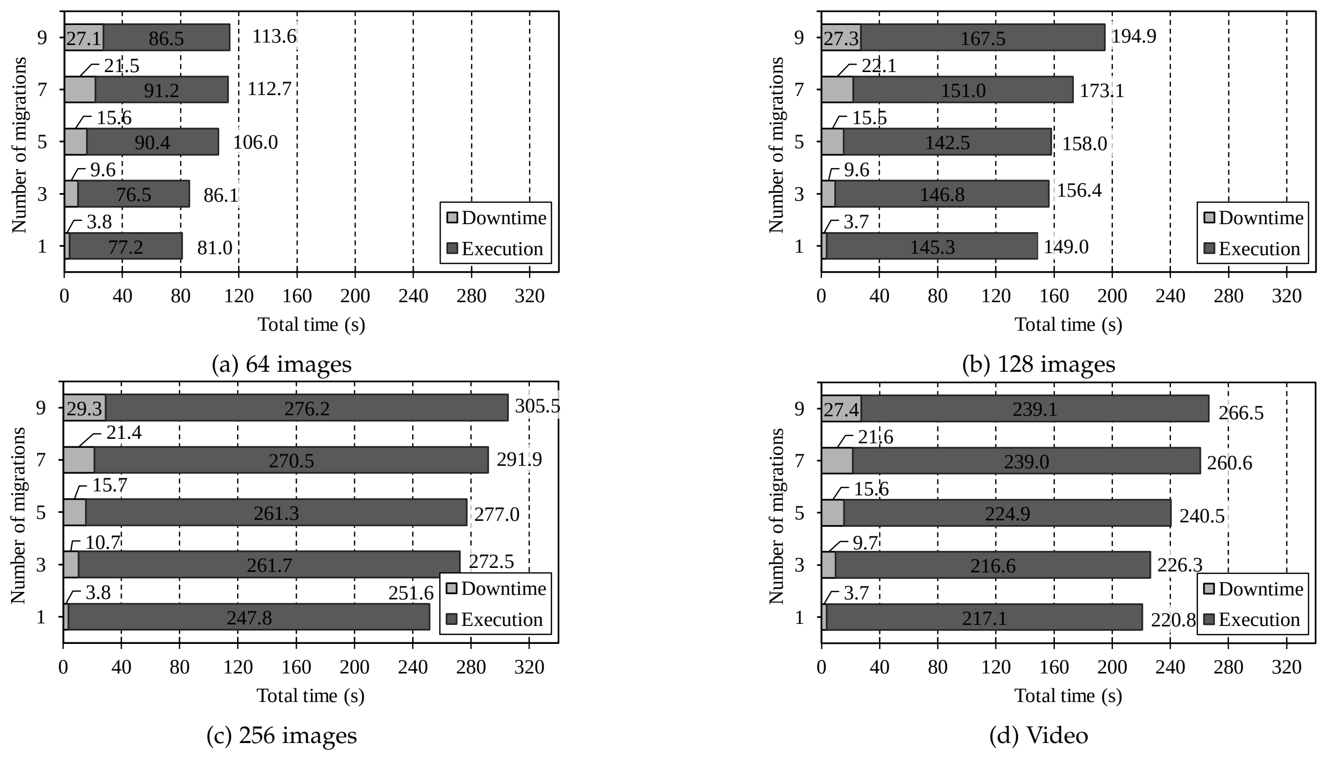


Fig. 6: OpenPose execution time for different image batches and video when migrating AVEC containers. 'Execution' refers to the amount of active time spent executing the OpenPose application. 'Downtime' refers to the amount of time spent in each migration step, including the startup of the new container.

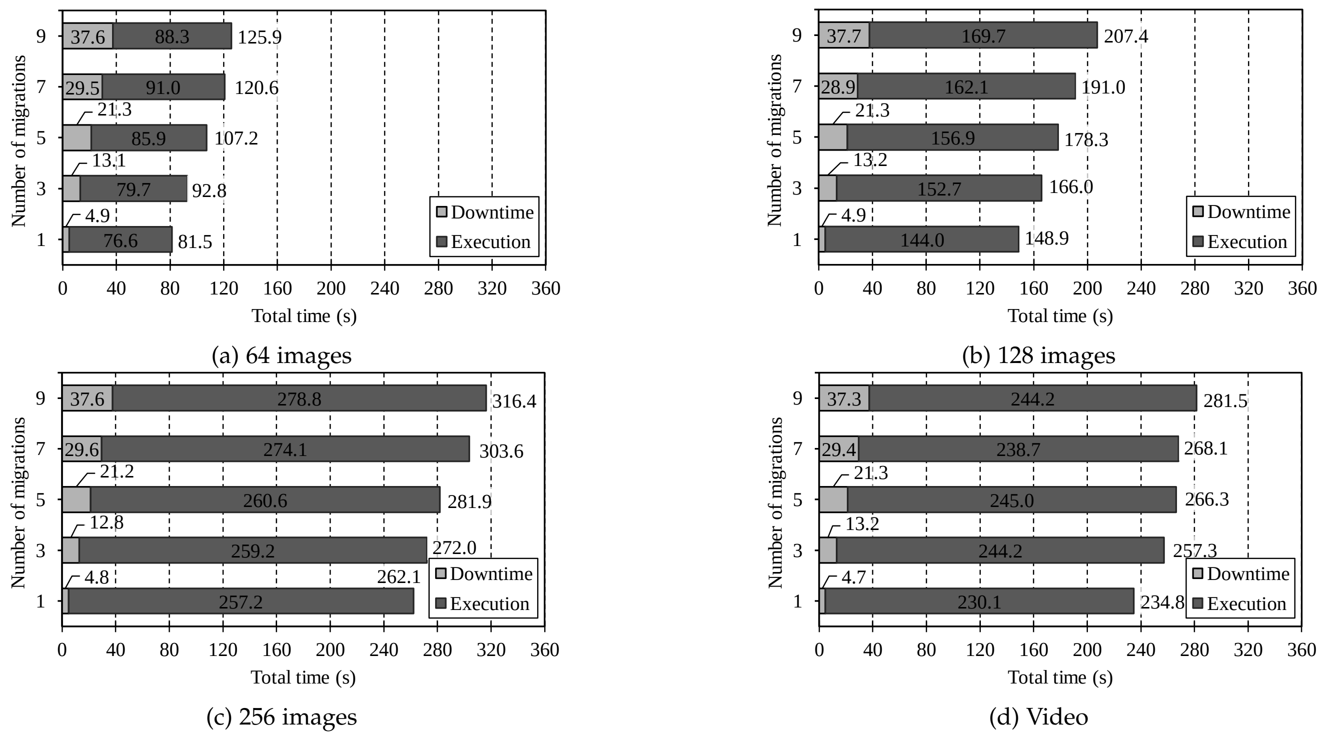
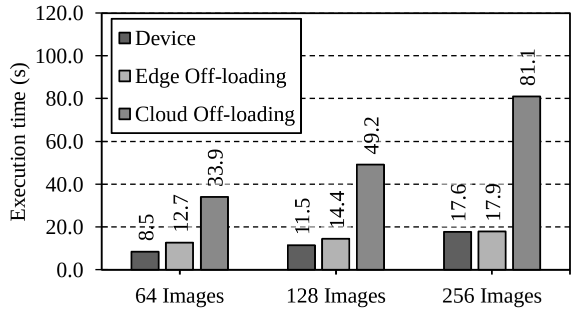
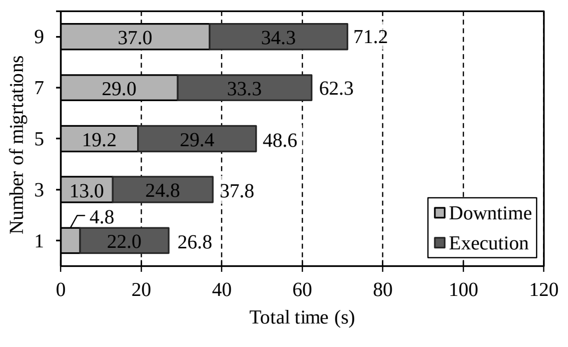


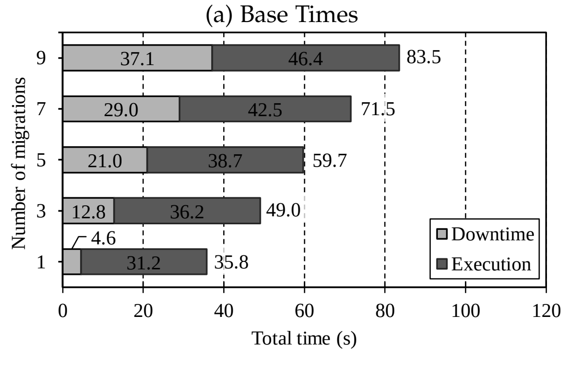
Fig. 7: OpenPose execution time for different image batches and video when migrating AVEC containers, with the added use of a scheduler dictating where to migrate. 'Execution' refers to the amount of active time spent executing the application. 'Downtime' refers to the amount of time spent in each migration step, including the startup of the new container.



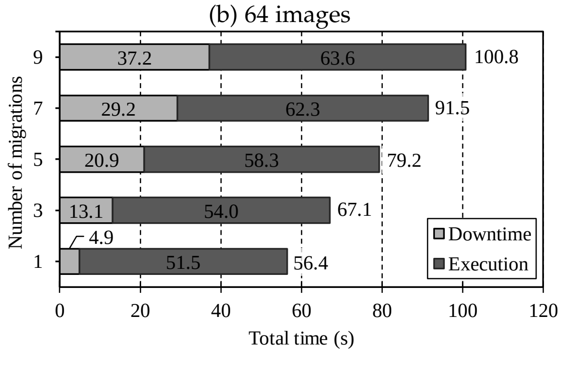
(a) Base Times



(b) 64 images



(c) 128 images



(d) 256 images

Fig. 8: Execution time of Classification application for different image batch sizes. Figure 8a displays base times without migration or scheduling for comparison purposes. Figures 8b, 8c, and 8d show execution time when migrating AVEC containers, with the added use of a scheduler dictating where to migrate. 'Execution' refers to the amount of active time spent executing the application. 'Downtime' refers to the amount of time spent in each migration step including the startup of the new container.

the total downtime for 9 migrations accounts for 11.9% of the total time, whereas for 64 images, it accounts for 29.8%. This is evidently quite detrimental to the performance of more lightweight workloads.

Figure 8 shows the results of testing the computationally lightest of the applications, Classification. In the first place, Figure 8a shows the performance of running this application on the user device, and then when computations are offloaded to the edge and cloud devices using AVEC. We can see that the performance decreases when we use AVEC. This is due to the lightweight nature of the application. The total execution time is low, and the time gained offloading computations to a more powerful device does not compensate for the overhead introduced by AVEC. This overhead accounts for factors such as container start-up time, scheduling, copying the GPU model to the device where computations are offloaded, etc. Considering the time per frame, the edge device is faster, as we can see from the device and edge execution times becoming closer with each increase in the workload. Further workload increases would result in the edge device eventually offering an improvement. The cloud device, however, seems will never offer an improvement for this application. The downtimes are consistent with the results obtained from the OpenPose experiment. This was expected because there are not much data sent between devices for a migration; therefore there is less room for variance. As the execution time for this application is significantly lower, the impact of continuous migrations and downtime is much more detrimental. For example, in Figure 8b we can see that the downtime accounts for over half of the total execution time when doing 9 migrations.

Results for the MaskDetection application are shown in Figure 9. Similar to the Classification application, we can see in Figure 9a that there is no performance improvement from cloud offloading, but unlike the previous application, there is an improvement when computations are offloaded to the edge device. This improvement continues to increase as more workload is added on. A similar trend is also seen regarding downtime and its impact on overall performance. However, in this case, this impact is not as great as for the Classification application due to the longer overall execution time of the MaskDetection application.

7.5 Detailed Overhead of the Proposed Framework

The graph in Figure 10 shows a breakdown of the overheads incurred when using AVEC to execute a 1-frame workload with the MaskDetection application, using the Jetson Nano user device and the edge device hosting the RTX 2060. The figure shows the best case scenario where container and machine learning model are already present on device. All values are rounded to one decimal place. The CPU aspect represents the time spent in the Jetson Nano executing tasks related to the application. Regardless of if AVEC is used or not, this time will remain constant. The GPU time is the most important aspect, this is what we focus on with AVEC. The time shown in this graph is the time spent executing the kernel for the frame on the GPU. The GPU

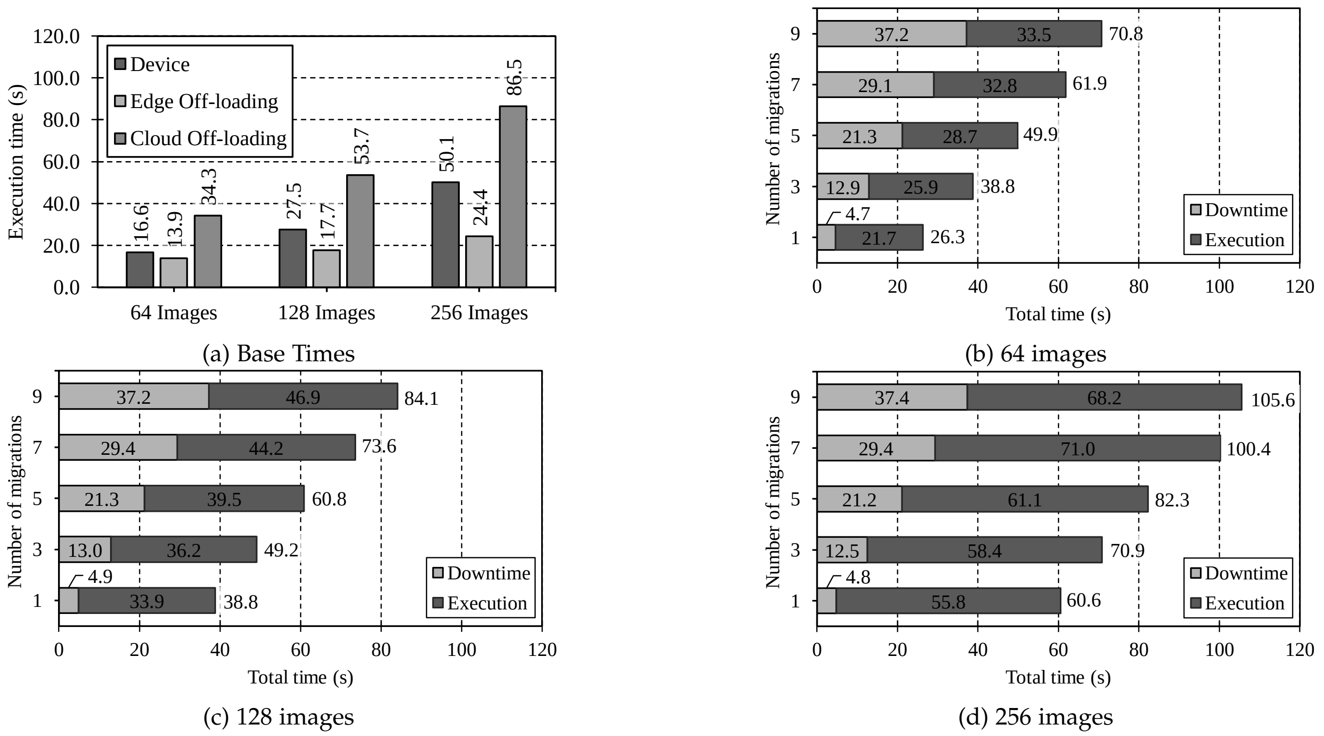


Fig. 9: Execution time of MaskDetection application for different image batch sizes. Figure 9a displays base times without migration or scheduling for comparison purposes. Figures 9b, 9c, and 9d show execution time when migrating AVEC containers, with the added use of a scheduler dictating where to migrate. 'Execution' refers to the amount of active time spent executing the application. 'Downtime' refers to the amount of time spent in each migration step including the startup of the new container.

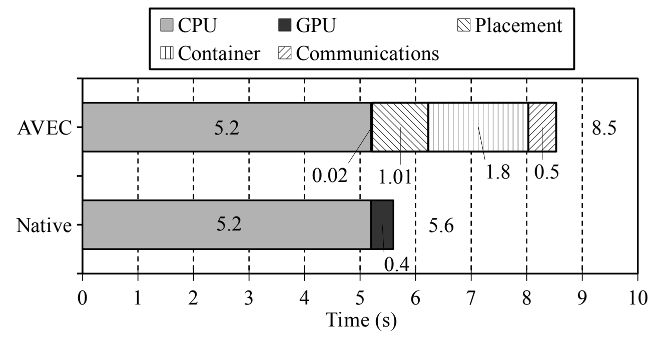


Fig. 10: Breakdown of the AVEC virtualization process compared with native execution.

time without AVEC is how long the kernel takes to execute locally on the Jetson Nano. GPU time with AVEC refers to executing that kernel on the remote node. Communication time represents the total amount of time spent sending data back and forth between the nodes and the processing or pre-processing that is required for it. There are a lot of different variables needed to be sent across throughout the application execution. The only time that is not accounted for in the timing of this communication is the time spent sending data back and forth in the placement recording, as communications time for the placement is taken into account within that reading. Placement timing represents the process of locating an appropriate node to offload the work to. The container represents the time taken to start the process of loading and connecting to the container.

As shown in Figure 10, although there is a significant speedup achieved with the GPU time going from 0.4 seconds to 0.02 seconds, the additional overhead incurred makes it a worse overall execution time. However, with subsequent frames, there will be less overhead, i.e. it is not necessary to start a new container, the placement has already taken place and not as much data will need to be sent. Due to this, the longer the workload goes on, the more overall improvement there will be.

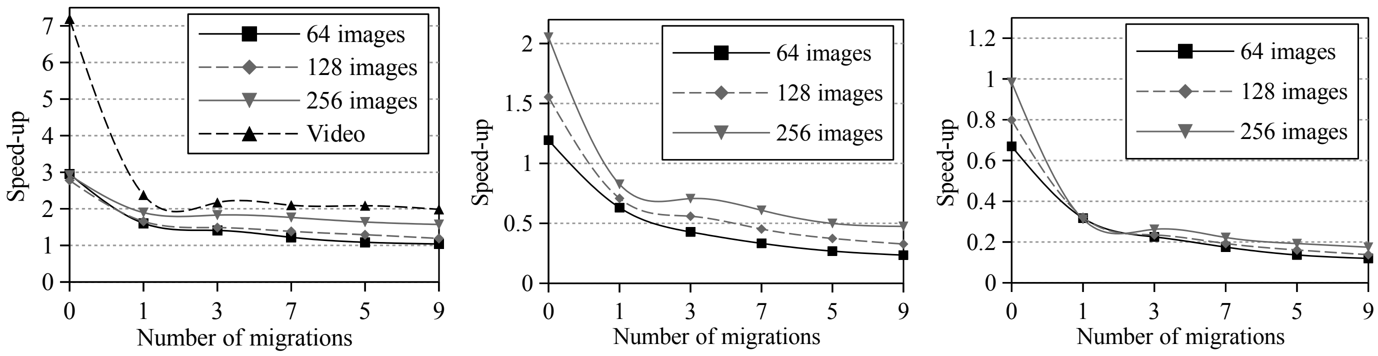
7.6 Model Based on the Number of Migrations

Before starting the experimental evaluation, we anticipated the migration time to follow the model shown in Equation 1.

$$E_T = (M \times M_T) + A_T \quad (1)$$

In this model, E_T represents overall execution time, M represents the number of migrations, M_T represents the time for a single migration and A_T represents the application execution time, i.e. the execution time for the application without the migration time. This model is based on the following assumptions: network connectivity speeds are the same between host node and destination node(s), each node contains the same hardware, with the same computational capabilities, and the network is a homogeneous environment.

However, after carrying out the experiments, we notice that the anticipated model differs from the experiments



(a) OpenPose (heavy-weight app.) (b) Mask detection (middle-weight app.) (c) Classification (light-weight app.)

Fig. 11: Speed-up obtained by AVEC for the applications tested varying the number of migrations from 0 to 9.

presented due to the assumptions provided. We can still conclude that the model is confirmed due to the following reasons. Migration time is consistent for each device. We have discussed in the findings that the amount of time that a migration requires is dependent on the device, but will be consistent, with a small amount of variance, for each migration. The main difference in the model and the experiments lies in the application execution time, as this is where the most amount of variance is seen due to differing bandwidth speeds, which our model assumes are equal.

7.7 Summary

The variances in these tests performed are minimal but exist. The variances arise mostly due to the method of connection between devices in the test scenario. The wireless connection between the user device and the cloud (GTX 1080 GPU device) can vary due to slight changes in bandwidth speed between the devices. As such, we have calculated the relative standard deviation (RSTDEV) in the proceeding tests. The maximum RSTDEV observed were 6.7, 4.0 and 3.6 for the Classification, MaskDetection and OpenPose applications, respectively.

In general, we can conclude from the experimentation analysis of these three applications that the more compute-intensive an application is, the more likely it is to benefit from AVEC. Certain applications may only benefit from using much more computationally powerful or higher-speed bandwidth nodes. Improvements could be made in the future to further improve the performance. However, there is likely to be lightweight applications in which it will not be feasible to use AVEC for performance improvement.

In particular, the key findings of the experimental evaluation are the following ones.

Total downtime. Total downtime incurred due to container virtualization, container migration and scheduling is consistent for all applications and devices. It differs for each device due to varying bandwidth, latency, and computational power. As highlighted in the experiments, the edge device incurs approximately 5 seconds of downtime, whereas the cloud is closer to 8 seconds.

Application speed-up. As shown in Figure 11, AVEC performs better with compute-intensive workloads that require more GPU usage. This is the case for the OpenPose application (Figure 11a), where up to 7x speedup is noted. The reason is that the longer time spent in the GPU outweighs the additional overheads incurred by the framework. For applications with moderate GPU usage, this speed-up is reduced to up to 2x (for example, the mask detection application, Figure 11b). For applications with low GPU usage, there is no performance gain (for example, the Classification application, Figure 11c).

Workloads placement and scheduling. The requirements for the workloads are different, and therefore a scheduling module is essential to ensure for efficient placement. The resource characteristics of the nodes and availability must be considered. Offloading the computations to nodes which will not be available during the whole execution of workloads will require to be migrated to other nodes. Experimental results (Figure 11) show that such migrations can considerably reduce speed-up; for the OpenPose application (Figure 11a), speed up may decline from 7x to 2x.

8 Conclusion

This paper presented new research in the cloud-edge continuum's accelerator virtualisation context. We investigated enabling the migration of containers for accelerator virtualization frameworks. More specifically, we studied the feasibility of deploying these frameworks through containers and migrating these containers between cloud-edge nodes. Furthermore, we investigated the use of scheduling policies to place these containerized workloads in the appropriate destination nodes. In particular, six research questions were addressed. To answer these questions, we presented the AVEC framework, an ongoing work regarding accelerator virtualization in cloud-edge computing.

As a result of this research, we concluded that the use of Docker containers in these frameworks allows for complete package and operating system level isolation. This enables accelerator virtualization frameworks to be deployed rapidly at a variety of different devices within a cloud-edge network. We highly recommend using the cache system employed in Docker to reduce start-up times. Although the use of this cache system will use up additional resources (i.e. storage memory), it is worth the trade-off to have substantially lower start-up times. Furthermore, implementation of the migration mechanism allows for a more complete and robust solution, as workloads can be moved to more optimal nodes during live execution. We proposed a scheduling system based on what we consider to be the most important metrics for GPUs in a cloud-edge environment: GPU platform, GPU memory, bandwidth and latency. An evaluation of these new mechanisms was carried out, and it was observed that overhead was acceptable.

In our experimental evaluation, we determined that the approach we have adopted within the AVEC framework is, in general, feasible to enable an improved quality of service for user devices. By evaluating three different applications, we concluded that not every application will benefit from this type of virtualization. Essentially, the speed-up gained from using more computationally powerful GPUs must compensate for the additional overheads incurred by the virtualization framework. Furthermore, only certain nodes will be able to meet the requirements of certain applications, which is why the inclusion of a scheduling system is paramount.

References

[1] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, "Edge cloud offloading algorithms: Issues, methods, and perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, feb 2019. [Online]. Available: https://doi.org/10.1145/3284387

[2] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and performance evaluation of edge cloud systems for latency constrained applications," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 286–299.

[3] E. H. Bourhim *et al.*, "Inter-container communication aware container placement in fog computing," in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–6.

[4] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *CoRR*, vol. abs/1808.05283, 2018. [Online]. Available: http://arxiv.org/abs/1808.05283

[5] Y. Kim and E.-n. Huh, "Edcrammer: An efficient caching rate-control algorithm for streaming data on resource-limited edge nodes," *Applied Sciences*, vol. 9, p. 2560, 06 2019.

[6] B. Varghese *et al.*, "Accelerator Virtualization in Fog Computing: Moving From the Cloud to the Edge," *IEEE Cloud Computing*, vol. 5, no. 6, pp. 28–37, 2018.

[7] S. Kim *et al.*, "An accelerated edge computing with a container and its orchestration," in *Internat. Conf. on Information and Communic. Technol. Convergence*, 2019.

[8] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[9] X. Wang *et al.*, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, p. 869–904, 2020.

[10] P. Bellavista *et al.*, "Differentiated service/data migration for edge services leveraging container characteristics," *IEEE Access*, vol. 7, pp. 139 746–139 758, 2019.

[11] C. Puliafito *et al.*, "The impact of container migration on fog services as perceived by mobile things," in *IEEE SMARTCOMP*, 2020, pp. 9–16.

[12] S. Wang *et al.*, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.

[13] V. Gezer and A. Wagner, "Real-time edge framework (rtef): task scheduling and realisation," *Journal of Intelligent Manufacturing*, vol. 32, pp. 1–17, 12 2021.

[14] K. Toczé *et al.*, "Orch: Distributed orchestration framework using mobile edge devices," in *Internat. Conf. on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–10.

[15] K. Tian *et al.*, "A full gpu virtualization solution with mediated pass-through," in *USENIX Conference on USENIX Annual Technical Conference*, 2014, p. 121–132.

[16] F. Silla *et al.*, "On the Benefits of the Remote GPU Virtualization Mechanism: The rCUDA Case," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 13, p. e4072, 2017.

[17] S. Xiao *et al.*, "VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units," in *Innovative Parallel Computing*, 2012, pp. 1–12.

[18] L. Shi *et al.*, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," *IEEE Transactions on Computers*, vol. 61, pp. 804–816, 2012.

[19] G. Giunta *et al.*, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *Euro-Par*, 2010, pp. 379–391.

[20] Y. Lin *et al.*, "qCUDA: GPGPU Virtualization for High Bandwidth Efficiency," in *IEEE International Conference on Cloud Computing (CloudCom)*, 2019, pp. 95–102.

[21] R. Montella *et al.*, "On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines," *Int. J. Parallel Program.*, vol. 45, no. 5, p. 1142–1163, 2017.

[22] B.-Y. Huang *et al.*, "qCUDA-ARM: Virtualization for Embedded GPU Architectures," in *Internet of Vehicles. Technologies and Services Toward Smart Cities*, 2020.

[23] F. Ramalho *et al.*, "Virtualization at the network edge: A performance comparison," in *Internat. Symp. on A World of Wireless, Mobile and Multimedia Networks*, 2016.

[24] P. Xu *et al.*, "Performance evaluation of deep learning tools in docker containers," 2017.

[25] P. Mendki, "Docker container based analytics at iot edge video analytics usecase," in *International Conference On Internet of Things: Smart Innovation Usages*, 2018.

[26] R. Dyson *et al.*, "Litener: An Accelerator-Enabled Lightweight Container for Edge Computing," in *IEEE IPDPS Workshops*, 2022, pp. 987–994.

[27] S. Nadgowda *et al.*, "Voyager: Complete container state migration," in *IEEE International Conference on Distributed Computing Systems*, 2017, pp. 2137–2142.

[28] L. Ma *et al.*, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, pp. 2020–2033, 2019.

[29] L. Deshpande *et al.*, "Edge computing embedded platform with container migration," in *IEEE SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*, 2017, pp. 1–6.

[30] H. Tan, Z. Han, X.-Y. Li, and F. C. Lau, "Online job dispatching and scheduling in edge-clouds," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[31] T. Oo *et al.*, "Application-aware task scheduling in

heterogeneous edge cloud," in *Internat. Conf. on Information and Communic. Technol. Convergence*, 2019.

[32] L. Lin *et al.*, "Distributed and application-aware task scheduling in edge-clouds," in *14th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, 2018.

[33] Y. Kim *et al.*, "Cpu-accelerator co-scheduling for cnn acceleration at the edge," *IEEE Access*, vol. 8, 2020.

[34] Z. Zhu *et al.*, "A hardware and software task-scheduling framework based on CPU+FPGA heterogeneous architecture in edge," *IEEE Access*, vol. 7, 2019.

[35] W. Zhang *et al.*, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM*, 2019, pp. 1270–1278.

[36] M. Bensalem *et al.*, "Dnn placement and inference in edge computing," in *International Convention on Information, Communication and Electronic Technology*, 2020.

[37] J. Walsh and J. Dukes, "Application support for virtual gpgpus in grid infrastructures," in *2015 IEEE 11th International Conference on e-Science*, 2015, pp. 67–77.

[38] G. A. Elliott and J. H. Anderson, "Globally Scheduled Real-Time Multiprocessor Systems with GPUs," in *18th International Conference on Real-Time and Network Systems*, Toulouse, France, Nov. 2010, pp. 197–206. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00546957

[39] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[40] L. Ma *et al.*, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, pp. 2020–2033, 2019.

[41] T. Kim, S. D. Sathyanarayana, S. Chen, Y. Im, X. Zhang, S. Ha, and C. Joe-Wong, "Modems: Optimizing edge computing migrations for user mobility," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1159–1168.

[42] M. Bukhsh, S. Abdullah, and I. S. Bajwa, "A decentralized edge computing latency-aware task management method with high availability for iot applications," *IEEE Access*, vol. 9, pp. 138 994–139 008, 2021.

[43] C. Puliafito *et al.*, "The impact of container migration on fog services as perceived by mobile things," in *IEEE International Conference on Smart Computing*, 2020.

[44] T. Bahreini and D. Grosu, "Efficient algorithms for multi-component application placement in mobile edge computing," *IEEE Trans. Cloud Comput.*, 2020.

[45] Z. Cao *et al.*, "OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, pp. 172–186, 2021.

[46] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," in *Computer Vision – ECCV*, 2014, pp. 740–755.

[47] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, p. 675–678.

Jason Kennedy received an MEng degree in Computer Science from Queen's University Belfast, United Kingdom, in 2019. He is currently a PhD candidate at the School of Electronics, Electrical Engineering and Computer Science of the same university, focusing his research on the virtualisation of accelerators in cloud-edge environments.

Vishal Sharma (Senior Member, IEEE) is working as a Lecturer in Computer Science at Queen's University Belfast, UK. He received a PhD in computer science and engineering from Thapar University in 2016. His research has been supported by ETRI-Korea, NRF-Korea, IITP-Korea, MSIT-Korea, AFOSR-USA, Innovate UK, PwC, VIAVI, UKRI-Horizon, DCMS, and DSTL. He has received four best paper awards and has authored/co-authored more than 100 journal/conference articles/book chapters and co-edited two books. His research interests are defence and security, distributed ledger technology and digital twins.

Blesson Varghese is a Reader in Computer Science at the University of St Andrews and an honorary faculty member of Queen's University Belfast. He directs the Edge Computing Hub (https://edgehub.co.uk) and was a Royal Society Short Industry Fellowship to British Telecommunications plc. He was the 2021 recipient of the IEEE Rising Star Award from the Technical Committee on the Internet for fundamental contributions to edge computing systems and applications. His interests include developing and analyzing novel parallel and distributed systems and applications that span the cloud-edge-device continuum. More information is available from www.blessonv.com.

Carlos Reaño (Member, IEEE) is Assistant Professor in Computer Architecture at University of Valencia in Spain, and Visiting Scholar at Queen's University Belfast in the UK, where he was Lecturer for 3 years. He received a PhD in Computer Science from Technical University of Valencia in 2017. He has authored over 40 publications. His research is mainly focused on the virtualization of GPU-accelerators in both HPC clusters and Cloud/Edge Computing systems. He is external collaborator of the rCUDA project http://rcuda.net, where he worked for 7 years. More information is available from www.carlosreano.com.