Andoni Salcedo-Navarro, Raúl Peña-Ortiz, José M. Claver, Miguel Garcia-Pineda, Juan Gutiérrez-Aguado, **Cloud-native GPU-enabled architecture for parallel video encoding**, In: *Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M. (eds) Euro-Par 2024: Parallel Processing. Euro-Par 2024. Lecture Notes in Computer Science, vol 14803, pp 327–341, Springer, Cham.*.

DOI: 10.1007/978-3-031-69583-4_23

Link to publication

Cloud-native GPU-enabled architecture for parallel video encoding

Andoni Salcedo-Navarro Raúl Peña-Ortiz José M. Claver
Miguel Garcia-Pineda Juan Gutiérrez-Aguado

Dept. of Computer Science. ETSE-UV. Universitat de València. Burjassot, Spain

Abstract

Multimedia streaming has become an essential aspect of contemporary life and the ever-growing demand for high-quality streaming has fostered the development of new video codecs and improvements in content delivery. Cloud computing, particularly cloud architectures, has played a pivotal role in this evolution, offering dynamic resource allocation, parallel execution, and automatic scaling—critical features for HTTP Adaptive Streaming applications. This paper presents two specialized containers designed for video encoding (using two implementations of H264: x264 that encodes in the CPU and H264 NVENC that also uses the GPU). These containers are deployed on a Kubernetes cluster with four GPUs. The experiments focus on the performance and resource consumption of the encoder containers under different Kubernetes cluster and replica configurations. The best setup shows a 12.7% reduction in encoding time for x264 and a 15.98% for H264 NVENC compared to the other configurations considered. Besides, the encoding time of H264 NVENC is reduced by a 3.29 factor compared to x264. To test the behavior in realistic scenarios, four videos were encoded at five different resolutions. The mean encoding time per segment is reduced by a 3.75 factor when using H264 NVENC compared to x264. These results hold significant implications for livestreaming applications, particularly for low-latency use cases.

Keywords: GPU, Cloud Computing, Kubernetes, Video Encoding, HTTP Adaptive Streaming.

1 Introduction

In the digital era, multimedia streaming has become a cornerstone technology, transforming how we consume and share content. It provides access to a wide array of audio and video content, revolutionizing entertainment, education, communication, and business. The rise of online streaming platforms has significantly challenged traditional broadcasting methods,

promoting on-demand viewing, binge-watching, and a greater variety of content [1]. The implementation of HTTP Adaptive Streaming (HAS) as the dominant video delivery method across the Internet underlines the critical role of efficient video encoding systems. These systems are essential for producing content in multiple resolutions and bitrates, aiming to improve the Quality of Experience (QoE) for users [2].

Moreover, the integration of multimedia streaming with cloud computing has played a vital role in this transformation. Cloud computing offers a scalable and robust infrastructure for global content encoding and delivery, facilitating efficient storage, processing, and distribution of multimedia content, marking a significant evolution in our access to multimedia today [3]. Cloud computing offers a cost-effective model due to its pay-as-you-use scheme, where charges align with resources utilized during operation. This model, coupled with the possibility to adapt the provisioned resources to the demand ensures optimal resource utilization, makes it well-suited for managing unpredictable workloads. Cloud providers handle server provisioning, maintenance, and scaling, reducing operational complexity and enabling developers to focus more on coding than infrastructure management. This accelerates application development and deployment cycles, leading to faster time-to-market. Cloud architectures enable efficient deployment of functionalities, particularly evident in HAS-based video coding systems for streaming, with support for dynamic resource allocation and parallel encoding execution.

Additionally, the use of Graphics Processing Units (GPUs) in the video encoding process significantly boosts performance and efficiency [4]. GPUs excel in handling the computationally demanding tasks of video encoding with their parallel processing capabilities, which can reduce encoding times. This is vital for real-time or nearly real-time video streaming, transcoding, and broadcasting of content with 4K or higher resolutions. GPUs also provide a cost-effective, high-performance option, and enhanced video encoding results through hardware-accelerated codecs [5].

Our research introduces a cloud-native GPU video coding system, deployed on top of Kubernetes on a resource-constrained infrastructure.

This type of restrictions can be encountered in Edge Clouds in contrast to the Core Cloud where much more computational resources are available. This system was evaluated in several scenarios to assess its performance in real-world settings, like a HAS environment, demonstrating the potential of cloud-based GPU video coding systems. The study's main contributions include an examination of replica behavior under different scenarios focusing on resource consumption and encoding times, and an investigation into the advantages of multi-resolution encoding in a single operation. It assesses various virtualization scenarios for cloud video encoding deployment, with and without GPUs, focusing on resource utilization, performance, and efficiency. An example of the deployed architecture is available on GitHub[1].

[1] https://github.com/cloudmedialab-uv/k8s-work-queue-video-coding

This paper is organized as follows. First, Section 2 shows similar works and analyze the main differences with our proposal. Afterwards, Section 3 presents the system architecture. Then, Section 4 provides a detailed evaluation of several scenarios and, finally Section 5 concludes this paper and shows some future work.

2 Related work

Proposed cloud-based solutions for distributed video coding and transcoding services have recently evolved to be adapted to the more demanding requirements and improve their performance [6]. Previous proposals for distributed video coding over the Cloud were mainly based on the MapReduce paradigm [7]. The need of dynamic instantiation of the number of encoders has been crucial for the new video coding services, characterized by high demand and adaptive streaming applications. For this purpose, new cloud-based solutions adapting the number of encoders to the needs of video streaming, such as high video resolutions with bandwidth constraints and quality requirements have been proposed. Thus, Gutiérrez-Aguado *et al.* [8] proposed a cloud-based distributed architecture tuned for video encoding (HEVC, VP9 and AV1 encoders). Based on an elastic pool of workers and media servers that provide fault tolerance, this solution allows the adaptation of the resources to the workload, offers high availability, and provides ubiquitous access.

Other proposals are based on distributed systems architecture, which present a novel approach to application development by decomposing a complex application into discrete, independent, and modular components known as microservices [9]. These microservices can be developed, deployed, and scaled autonomously. By integrating microservices, containers, and Kubernetes, it is possible to construct a scalable and resilient video encoding platform. Based on this premise, the authors of [10] introduce Morph, a cloud-based video transcoding system designed to optimize resource utilization. The method approaches task scheduling and resource provisioning as a stochastic optimization challenge across two time scales. The architecture comprises multiple modules, including a request management interface, homogeneous instance deployment for transcoding, a task scheduler implementing the Value-based Task Management (VBM) policy, and resource management strategies.

The proposal outlined in [11] introduces an architecture aimed at predicting optimal virtual resources for live video transcoding in the cloud. Transcoding tasks are executed through Docker containers within a Kubernetes platform. The authors have devised multiple models to forecast transcoding speed and CPU usage on resources. Through experimentation with different learning configurations, the developed prototype demonstrated that Random Forest (RF) regression outperformed Reinforcement Learning (RL) and Stochastic Gradient Descent (SGD) regression in terms of overall prediction accuracy for this specific case.

An additional option for multimedia content processing through a container-based cloud

encoding system is discussed in [12]. The architecture, adopting a server-worker model, consists of three layers: a) Front Layer: responsible for delivering the original video content from clients to the system; b) Service Layer: housing various internal modules such as transcoding, message queue, cache, etc.; and c) Control Layer: tasked with managing and overseeing the workflow between the preceding layers. The solution deployment involves Docker containers, with each container (worker) utilizing FFmpeg for video transcoding and multiple services operate in separate containers on a master node.

Video coding has also benefited from the use of GPUs to speed up the increasing complexity and high time cost of new encoders. Thus, the use of GPUs has been proposed to accelerate video coding in several formats[2]. Of the papers mentioned, only [12] indicates that its platform could use GPUs but does not show any tests or how it is implemented. The key difference of our proposal with those previous works is twofold: we present a real implementation of a cloud-native GPU video coding system; and we perform a detailed evaluation of the performance and resource consumption over several scenarios.

3 Proposed architecture

The system architecture designed for video encoding tasks on Kubernetes is shown in Figure 1. At the forefront, a custom proxy is in charge of intercepting incoming HTTP POST request and publishing them as messages. RabbitMQ was used as message broker, where a direct exchange with two queues have been created. Upon receiving an HTTP request, the custom proxy examines the *X-type* header, transforming the request into a RabbitMQ message, which is then stored in the appropriate queue.

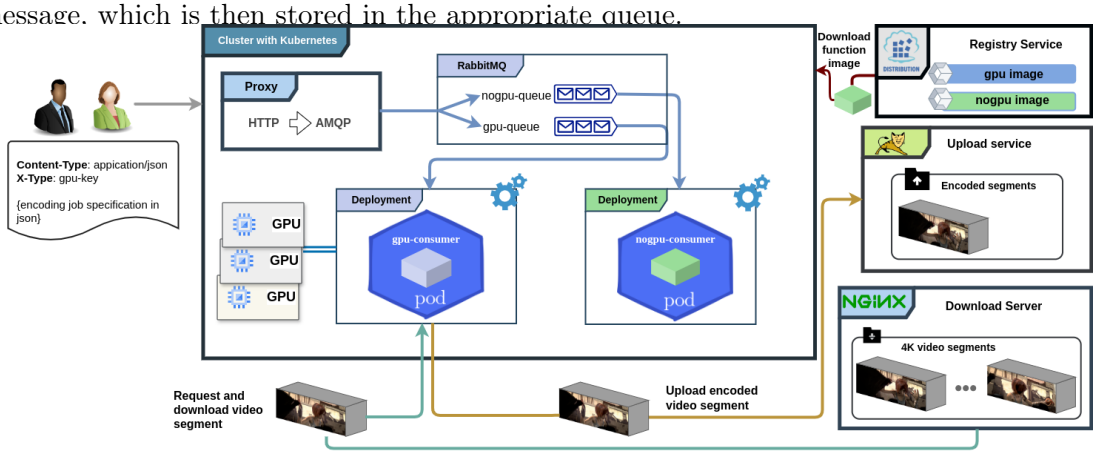


Figure 1: K8s-based architecture for parallel video encoding.

To efficiently manage processing demands, the architecture employs two distinct types of consumers. Each consumer type is configured to subscribe to one of the two queues created

[2]NVIDIA Video Codec SDK, available at: https://developer.nvidia.com/video-codec-sdk

in the message broker. One type is of consumer is configured for GPU-based video encoding tasks, utilizing H264 NVIDIA NVENC for encoding, ensuring high-performance computation suitable for intensive encoding operations. These GPU-based pods are deployed to handle video encoding tasks that require significant processing power.

On the other hand, video segments that do not request GPU acceleration are managed by a separate set of consumers. This second deployment includes pods that operate without GPU resources. By segregating tasks based on their processing requirements, the system ensures that each video segment is directed to the correct processing unit. This division enhances the overall efficiency of the video processing workflow allowing to use different hardware resources.

Concurrency within this architecture is managed through the *prefetch count* parameter. This parameter is configured via environment variables for each consumer replica, allowing the system to control the number of messages a consumer can handle simultaneously. By limiting the number of messages processed concurrently, the architecture allows for diverse configurations for encoding, enabling a flexible approach to resource utilization and workload distribution.

Storage and container image registry functionalities are seamlessly integrated within the architecture. A dedicated NGINX container operates as the download server, storing 4K video segments pending encoding. A private container image registry service is also deployed, maintaining the docker images for the encoding replicas, GPU image for GPU-dependent containers and no-GPU image for those without GPU requirements. Kubernetes interacts with this registry using appropriate credentials to pull images for pod instantiation. After a video segment has been encoded, the resulting videos are sent to an upload service, which is developed on Apache Tomcat. This service manages the subsequent storage or distribution of the encoded segments, thereby completing the encoding workflow.

4 Experiments and results

This section describes the testbed used for the experiments and the different scenarios considered: three to be used with x264 that do no utilize the GPU and another three with the H264 NVENC that do use the GPU. Then, we analyze the resource consumption in terms of RAM, CPU and GPU usage in all scenarios. Finally, the behavior when encoding multiple representations of each segment in a single call is analyzed.

4.1 Testbed and videos

The experimental setup involved five physical machines connected by a 10 Gbps switch. Three of them serve as commodity servers for distinct purposes: one as a private registry

for container images, another with a Nginx service for HTTP video segment downloads, and a third for HTTP-based video upload storage.

The Kubernetes cluster operates on the remaining two machines. The master node runs on a VM with 16 GB RAM and 10 vCPUs, while the other server, equipped with 128 GB RAM, 12 cores, and 4 NVIDIA Quadro M4000 GPUs (each with 8 GB RAM), hosts the Kubernetes worker VMs. The virtual infrastructure emphasizes performance, utilizing SR-IOV NICs for direct VM network access and GPUs assigned via PCI passthrough, ensuring high efficiency and fast data access through dedicated SSDs for each VM. Docker and Flannel were chosen for the Container Runtime and Network Interfaces, respectively. The deployment includes the NVIDIA Operator (v23.3.1) to ease the integration of GPUs on Kubernetes.

The 4K videos[3] used in the experiments are listed in Table 1. The videos have been downloaded, transcoded using x264 at very high quality (10 CRF), and finally split to obtain segments of 2 seconds. Table 1 shows the final sizes after that process of transcoding and splitting.

Table 1: Videos used to perform the experiments.

Video	Resolution	Duration (seconds)	chunks	Size (GB)
ANC	4K	188	94	2.9
ELD	4K	184	92	4.7
IND	4K	252	126	8.0
SKA	4K	276	138	7.7

4.2 Scenarios without and with GPUs

We have tested the performance on six different scenarios. In three of them, the replicas do not require GPU and use `libx264` to encode the segments. In the other three, the replicas request GPU and the encoder used is `h264_nvenc`.

In all the scenarios, the computational resources assigned to each experiment are the same: 8 vCPUs and 96 GiB RAM. When GPUs are considered, the total number of GPUs assigned will be the same. However, these computational resources will be distributed among the Kubernetes worker nodes differently in each scenario. The goal is to determine what scenario is the best (given these computational resources) in terms of video encoding time.

Scenario 1: four slim VMs with four slim pod replicas. In this configuration, we have four slim worker VMs with 2 vCPUs and 24 GiB RAM. On each VM, a *slim* replica, that requests 2 vCPUs and 24 GiB RAM, has been provisioned. Each replica is allowed to run only one concurrent encoding job.

[3] https://www.cablelabs.com/4k

Scenario 2: one fat VM with four slim pod replicas. This scenario employs a single fat worker VM with 8 vCPUs and 96 GiB RAM. This single VM allocates four slim replicas that are limited to 2 vCPUs and 24 GiB RAM per replica. Comparing this scenario with the previous one, the total amount of computational resources are the same, but their distribution is different. Each replica is allowed to run only one concurrent encoding job.

Scenario 3: one fat VM with one fat pod. This setup also uses a single fat worker VM with 8 vCPUs and 96 GiB of memory. However, instead of deploying four slim replicas, we have a single fat pod that requests 8 vCPUs and 96 GiB RAM. Each replica can run up to four concurrent encoding jobs.

Scenario 4: four slim VMs allocating four slim pod replicas with one GPU each. This scenario encompasses four slim worker VMs, with 2 vCPUs and 24 GiB RAM each. Besides, each VM has a GPU. The replicas in this scenario will request 2 vCPUs and 24 GiB RAM and will have access to the GPU of the underlying VM. Each replica is allowed to run only one concurrent encoding job.

Scenario 5: one fat VM allocating four slim pod replicas with one GPU each. A single fat worker VM with 8 vCPUs, 96 GiB of memory, and four GPUs allocates four slim replicas that are limited to 2 vCPUs, 24 GiB RAM, and one GPU. Each replica is allowed to run only one concurrent encoding job.

Scenario 6: one fat VM allocating one fat pod with four GPUs. In this setup, a single fat worker VM, with 8 vCPUs, 96 GiB of memory, and the four GPUs allocates a single fat pod. The pod requests 8 vCPUs, 96 GiB RAM, and the four GPUs. Each replica can run up to four concurrent encoding jobs.

4.3 Video encoding times and replica resource consumption

The video used for all the experiments in this section is SKA with 138 2-second segments, 7.7 GB of total size and a mean size per segment of 55.8 MB. For each scenario, the total time required to encode all the segments was computed. When collecting this data, no monitoring of resources was performed (as the monitoring itself consumes part of the resources).

The goal of this section is twofold. On the one hand, we want to compare the gain when using a GPU-enabled container orchestration platform with a baseline case, where GPUs are not available. On the other hand, we want to determine what is the best scenario, given the available resources, in terms of total time required to encode.

4.3.1 Encoding times and resource consumption in the scenarios without GPUs

As a baseline case, we present the data obtained when the encoding is performed in the scenarios that do not utilize GPUs. Listing 1 shows an illustrative example of the ffmpeg commands executed inside the replicas to encode using `libx264`. The output resolution is 4K and the bitrate is 11 Mb/s for `libx264`, which are the minimum recommended bitrates for these codecs[4].

Listing 1 Sample ffmpeg commands executed inside the replicas to encode a video chunk using x264.

```
ffmpeg -i chunk1.mp4 -c:v libx264 -b:v 11M -maxrate 11M -minrate 11M 1.mp4
```

Table 2 summarizes the times obtained for the encoder `libx264` on the three scenarios. Download, encoding, and upload columns show the mean time (in milliseconds) required to download, encode, and upload the 138 video segments. The best scenario is the scenario 2 (with one fat VM and four replicas of the function) and the worst setup is the scenario 1. The encoding time for x264 in the scenario 2 improves the encoding time by 12.17% respect to scenario 1.

Name	Dowload(ms)	Encoding(ms)	Upload(ms)	Total(s)
$<\text{x264}>^{e_1}_{\text{slim}}$	178	30812	85	1083.72
$<\text{x264}>^{e_2}_{\text{slim}}$	167	27468	101	968.74
$<\text{x264}>^{e_3}_{\text{fat}}$	169	28249	112	988.14

Table 2: Mean times in milliseconds (per segment) to download, encode, upload and the total seconds to encode all the chunks of the video without GPU.

The performance drop in scenario 1 is due to the Kubernetes service mesh consuming 24% of node available 2 vCPUs, resulting in almost 2 vCPUs used on the 4 worker nodes. In contrast, in scenarios with just one worker node, the service mesh's resource usage drops to 6% of the total 8 vCPUs available. The impact of this consumption on video encoding times is detailed in the Table 2.

Figure 2 shows the CPU consumption of each replica involved in the encoding process. Both encoders exhibited a similar behavior in all scenarios. Scenario 1 each encoding consumed nearly all the CPU resources of the underlying VM. In scenario 3 the same behavior is obtained. In scenario 2 each replica consumes one fourth of the available CPU resources, therefore the aggregated effect of the four replicas also saturates the CPU resources.

Finally, Figure 3 shows the memory consumption of each replica in scenario 1, with memory usage peaks at 1.9 GB per job, remains stable on different tasks. Scenario 2 shows higher peaks at 2.6 GB, with most jobs using around 2.3 GB, marking an increase of 0.4 GB per replica compared to scenario 1. This increase is attributed to the `ffmpeg` process

[4]See for instance https://support.google.com/youtube/answer/2853702

starting more threads due to the availability of more vCPUs in scenario 2, despite each function being limited to 2000 milli-cores. Specifically, in a slim VM setup with 2 vCPUs, `ffmpeg` starts two threads, whereas in a fat VM with 8 vCPUs, it initiates eight threads per replica. In scenario 3, memory consumption is significantly higher, with maximum usage four times that of scenario 2, attributed to continuous task concurrency and the constant activity of at least one `ffmpeg` process.

4.3.2 Encoding times and resource consumption in the scenarios with GPUs

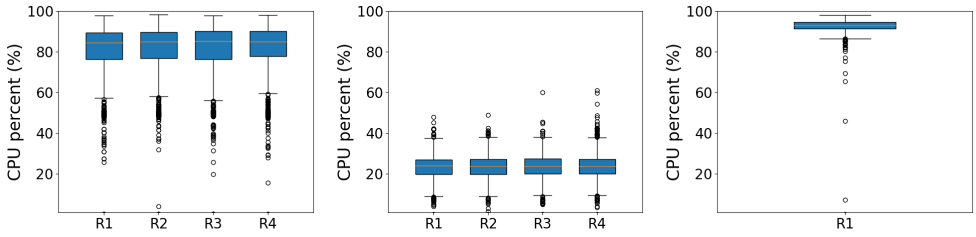
This subsection presents the data obtained when the encoding is performed in the three scenarios that utilize GPUs and compares these results with those presented in the previous section. Listing 2 presents an illustrative example of the ffmpeg command executed inside the replicas to encode one video chunk with `h264_nvenc` using GPU.

Listing 2 Sample ffmpeg commands executed inside the replicas to encode a video chunk with `h264_nvenc` using GPU.

```
ffmpeg -i chunk1.mp4 -c:v h264_nvenc -b:v 11M -maxrate 11M -minrate 11M -rc vbr 1.mp4
```

Table 3 presents the GPU encoder performance across three scenarios, focusing on the average times for downloading, encoding, and uploading 138 video segments. The findings are consistent with those from non-GPU scenarios, showing that the scenario has an impact on encoding times but not on download or upload times. The best performance is shown in scenario 5, with a remarkable 15.98% improvement in average encoding time per segment compared to the least efficient scenario, the scenario 4.

The total time (in seconds) required to encode all the segments of the video is shown in the last column of Table 3. Comparing the data for scenario 5 with the last column of scenario 2 in Table 2, we can see the benefits of using the GPU. Specifically, `h264_nvenc` is



(a) $<\text{x264}>^{e_1}_{\text{slim}}$ (b) $<\text{x264}>^{e_2}_{\text{slim}}$ (c) $<\text{x264}>^{e_3}_{\text{fat}}$

Figure 2: CPU consumption(in %) per scenario for each replica of the encoder containers for x264

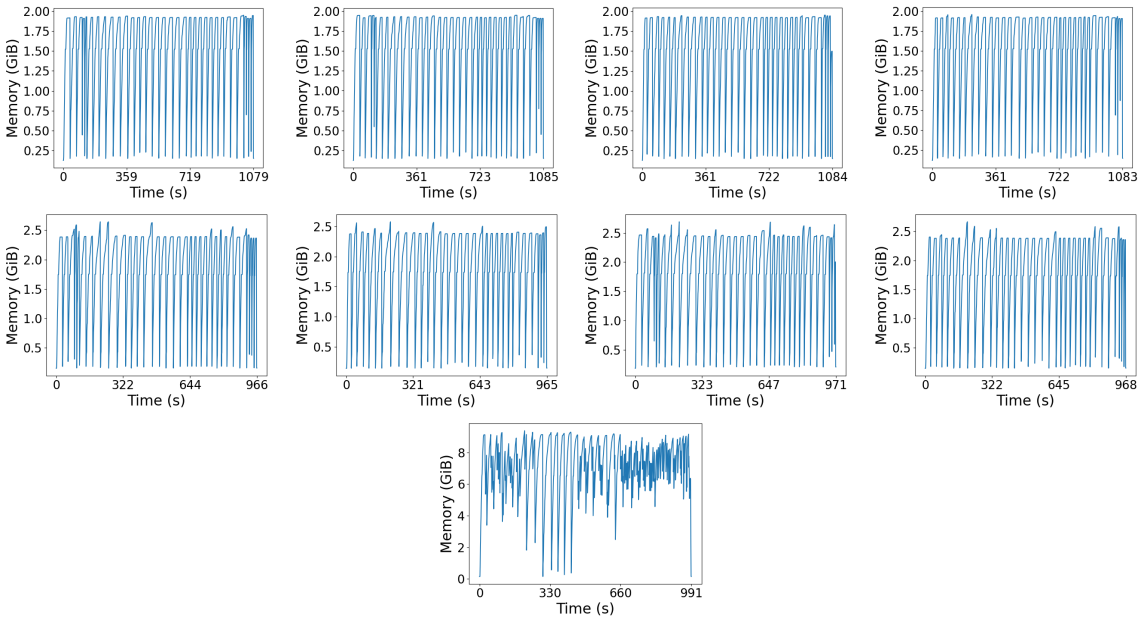


Figure 3: Memory consumption (in GiB) vs time (in seconds) per replica of the encoder function `libx264`. Top row shows resource consumption per replica for scenario 1, central row for scenario 2, and bottom row for scenario 3.

3.19 times faster than `libx264`.

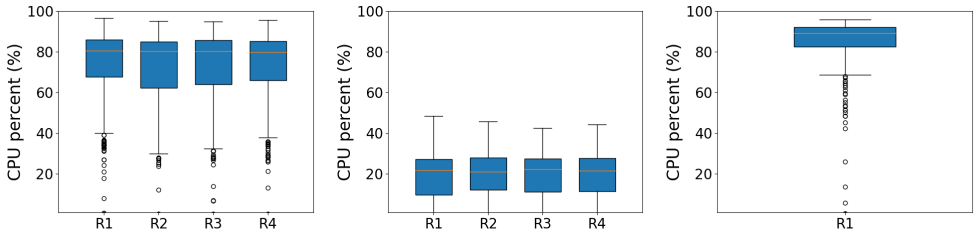
Name	Download(ms)	Encode(ms)	Upload(ms)	Total(s)
$<\text{h264\_nvenc}>^{e_4}_{\text{slim}}$	153	9371	73	335,69
$<\text{h264\_nvenc}>^{e_5}_{\text{slim}}$	160	8080	88	294,57
$<\text{h264\_nvenc}>^{e_6}_{\text{fat}}$	138	8288	71	299,26

Table 3: Mean times (per segment) to download, encode, upload and the total time to encode all the chunks of the video

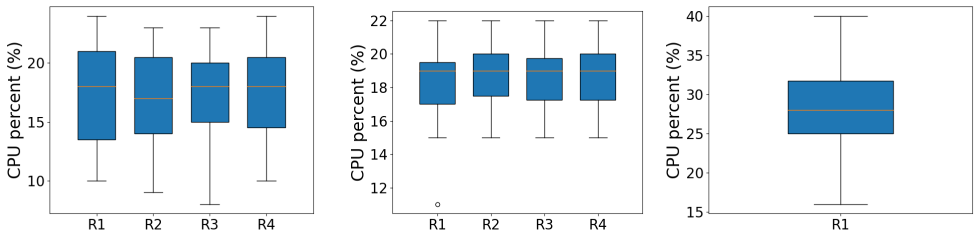
We examine the resource utilization of each GPU replica, considering resources from the VM (CPU and RAM) and resources from the GPU (GPU utilization and GPU memory usage) during the encoding process. Figure 4 shows how the based on GPU `h264_nvenc` codec influence CPU and GPU utilization, respectively, across different scenarios.

Comparing figure 2 with 4, we can observe a reduction in CPU consumption when GPUs are used. As an example, in scenario 4, using `h264_nvenc`, the mean CPU consumption is around 80%, while in scenario 1, using `libx264`, the mean is around 85%.

The RAM consumption for `h264_nvenc` is shown in Figure 5. Comparing this data with those of Figure 3, we observe the following reductions in RAM usage comparing `h264_nvenc`



(a) <h264_nvenc>$^{e_4}_{slim}$ (b) <h264_nvenc>$^{e_5}_{slim}$ (c) <h264_nvenc>$^{e_6}_{fat}$



(d) <h264_nvenc>$^{e_4}_{slim}$ (e) <h264_nvenc>$^{e_5}_{slim}$ (f) <h264_nvenc>$^{e_6}_{fat}$

Figure 4: GPU utilization (in %) (top) and CPU consumption (in %) (bottom) per scenario for each replica of the encoder containers using `h264_nvenc`

with `libx264`: 68.4% (scenarios 4 and 1); 70.2% (scenarios 5 and 2); and 72,5% (scenarios 6 and 3).

Finally, Figures 6 show the GPU memory usage for `h264_nvenc`. The maximum memory usage in the GPU for `h264_nvenc` is 0.43 GB for scenario 4, 0.43 GB for scenario 5, and 1.71 GB for scenario 6. Besides, adding the RAM consumption and the memory usage in the GPU, the total memory is lower than the RAM required for the encoders that do not use GPU.

Summarizing, there is a noticeable reduction in both CPU and RAM usage when GPUs are employed. This reduction is expected, as the GPU takes on the bulk of the computational load.

4.4 Function performance in multiresolution video encoding

From the data obtained in the previous subsections, it can be seen that scenarios 2 and 5 provide the best results in terms of coding time. In this section, we will select those scenarios to perform the encoding of all the segments of the four videos detailed in Table 1. Besides, each segment will be encoded in different representations in a single call so that each segment will be transferred to the GPU only once. The output of each function has five representations, with different resolution and bitrates. Listing 3 shows a sample command

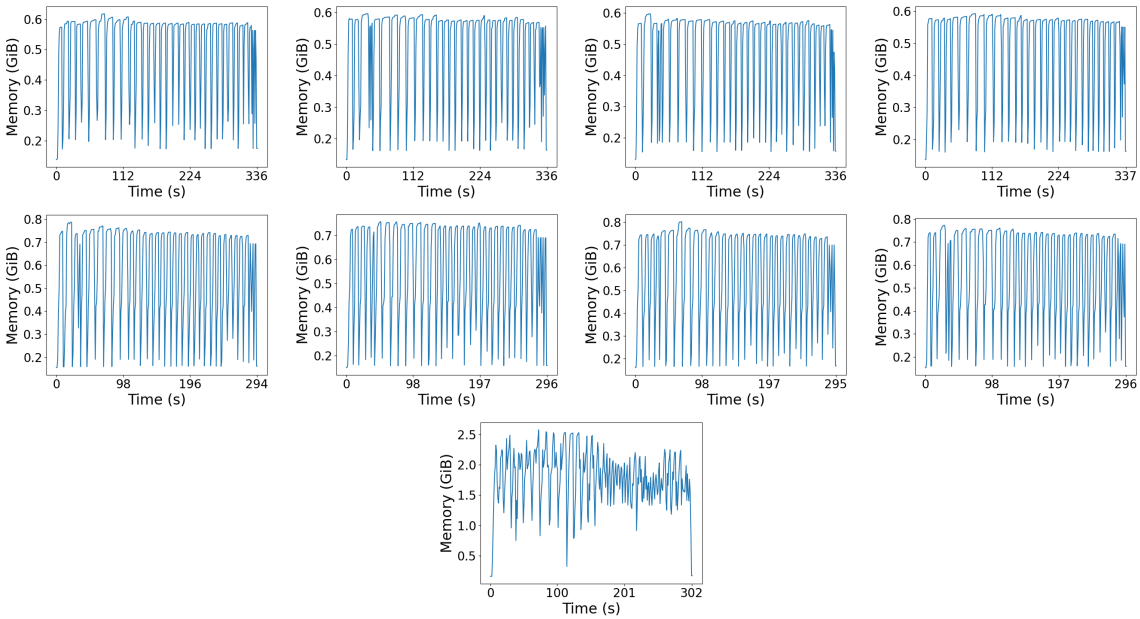


Figure 5: Memory consumption (in GiB) vs time (in seconds) per replica of the encoder function `h264_nvenc` for scenario 4 (top row), scenario 5 (middle row), and scenario 6 (bottom row)

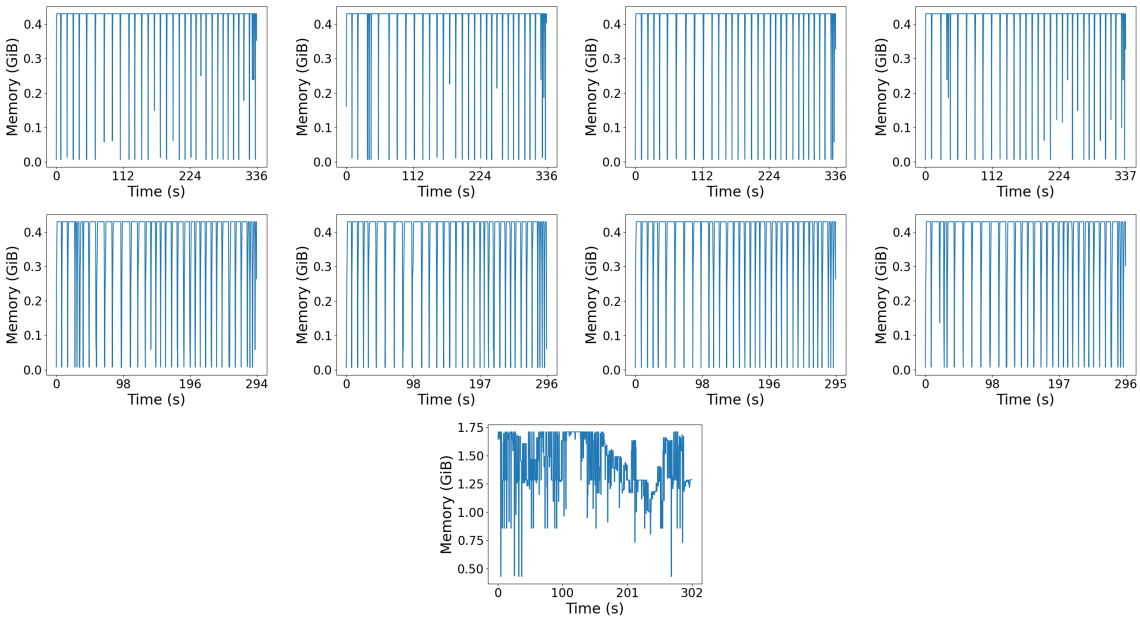


Figure 6: GPU memory usage (in MiB) vs time (in seconds) per replica of the encoder function `h264_nvenc` for scenario 4 (top row), scenario 5 (middle row), and scenario 6 (bottom row).

used to encode a segment.

Listing 3 Sample ffmpeg command executed in a replica to perform multiresolution encoding in a single call.

```
ffmpeg -i chunk1.mp4  -filter_complex '[0:v]yadif,split=5[out1][out2][out3][out4][out5]'
 -map '[out1]' -c:v <codec> -s 854x480 -b:v 1.35M -maxrate 1.5M -bufsize 3M 1_480p.mp4
 -map '[out2]' -c:v <codec> -s 1280x720 -b:v 2.75M -maxrate 4M -bufsize 8M 1_720p.mp4
 -map '[out3]' -c:v <codec> -s 1920x1080 -b:v 6M -maxrate 8M -bufsize 16M   1_1080p.mp4
 -map '[out4]' -c:v <codec> -s 2560x1440 -b:v 8M -maxrate 11M -bufsize 22M   1_1440p.mp4
 -map '[out5]' -c:v <codec> -s 3840x2160 -b:v 11M -maxrate 14M -bufsize 28M  1_2160p.mp4
```

Table 4 provides a summary of the encoding times for segments across four videos, including average times for downloading a segment, encoding a chunk, uploading the encoded chunk, and the total encoding time for all chunks.

Video	Download(ms)	Encode(ms)	Upload(ms)	Total(s)
Results for libx264				
ANC	95	48489	164	1163.48
ELD	161	46141	173	1084.57
IND	184	49782	178	1599.62
SKA	157	46088	167	1619.29
Results for h264_nvenc				
ANC	81	10654	171	262.57
ELD	138	12789	201	309.28
IND	165	13777	216	454.99
SKA	146	13557	193	485.50

Table 4: Mean times (per segment) to download, encode and upload. Last column shows the total time to encode all the chunks of the video.

For `libx264`, the mean encoding times are quite similar for all the videos. If we compare the average time to encode SKA with `libx264` in Table 4 and that of the scenario 2 in Table 2, we see that it takes around 46 seconds to encode the five representations in a single ffmpeg call and it took around 30.8 seconds to encode a single representation. The mean encoding time increments by a 1.5× factor, which is smaller than the factor of increment of the number of pixels to be encoded in the five representations (which is 1.84).

If we compare the mean time to encode SKA with `h264_nvenc` in Table 4 and that of the scenario 2 in Table 3, we see that it takes around 13.557 seconds to encode the five representations in a single ffmpeg call and it took around 8.08 seconds to encode a single representation. The mean encoding time increments by a 1.68× factor.

Taking the mean across all videos of the mean encoding times in Table 4, the mean time to encode a segment using `libx264` without GPU is 47.63 seconds, while the mean time using `h264_nvenc` with GPU is 12.69 seconds. Therefore, the mean time to encode a segment has been reduced by a factor of 3.75.

5 Conclusions and future work

Multimedia streaming, vital in today digital era, is significantly enhanced by cloud computing, especially in HAS applications. Cloud computing, offering dynamic resource allocation and automatic scaling, streamlines video coding systems by reducing encoding time and enhancing streaming efficiency.

This research implements two containers for video encoding and analyzes their performance on a platform managed by Kubernetes. By experimenting with various resource allocations and configurations, the study aimed to better understand their operational characteristics. The experimentation involved different combinations of resource assignation to worker nodes and pod replicas.

With the available resources, the most optimal configuration is a fat worker VM running four slim pod replicas. Conversely, the least favorable setup is characterized by four slim worker VMs each one allocating a slim pod replica. In terms of performance, when comparing `libx264` with `h264_nvenc` encoding, the latter is 3.29 times faster. This advantage was even more pronounced in multiresolution encoding, where `h264_nvenc` encoding reduced the mean time to encode a segment across five resolutions by a factor of 3.75.

As future work, we plan to explore the integration of GPUs in serverless platforms and analyze the behavior of other codecs. Additionally, our ongoing efforts are directed towards the development of a serverless platform designed for the field-of-view (FoV) based encoding of 360VR videos. This initiative aims to leverage serverless architecture to enhance the efficiency and adaptability of video encoding processes specifically targeted at immersive 360-degree virtual reality content. We seek to further advance the capabilities and versatility of our system, ensuring its relevance and applicability in emerging domains.

Acknowledgement

Grant by PID2021-126209OB-I00 funded by MICIU/AEI/ 10.13039/501100011033 and by "ERDF A way of making Europe", by the "European Union".

References

[1] S. Vijay, P. Mann, R. Chaudhary, and A. Rana, "Emerging Trends in Multimedia," in *Emerging Technologies in Data Mining and Information Security*, ser. Lecture Notes in Networks and Systems, vol. 491. Singapore: Springer, 2023, pp. 301—-311.

[2] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A survey on quality of experience of http adaptive streaming," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2015.

[3] X. Li, M. Darwich, M. A. Salehi, and M. Bayoumi, "Chapter four - a survey on cloud-based video streaming services," ser. Advances in Computers, A. R. Hurson, Ed. Elsevier, 2021, vol. 123, pp. 193–244.

[4] N.-M. Cheung, X. Fan, O. C. Au, and M.-C. Kung, "Video coding on multicore graphics processors," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 79–89, 2010.

[5] Y. Dong, L. Song, R. Xie, and W. Zhang, "Real-time uhd video super-resolution and transcoding on heterogeneous hardware," *Journal of Real-Time Image Processing*, vol. 17, no. 6, pp. 2029–2045, Dec 2020.

[6] W. Moina-Rivera, M. Garcia-Pineda, J. Gutiérrez-Aguado, and J. M. Alcaraz-Calero, "Cloud media video encoding: review and challenges," *Multimedia Tools and Applications*, Mar 2024.

[7] R. Pereira, M. Azambuja, K. Breitman, and M. Endler, "An Architecture for Distributed High Performance Video Processing in the Cloud," in *3rd International Conference on Cloud Computing*, ser. CLOUD'10. Miami, Florida, USA: IEEE, 7 2010, pp. 482–489.

[8] J. Gutiérrez-Aguado, R. Peña-Ortiz, M. García-Pineda, and J. M. Claver, "Cloud-based elastic architecture for distributed video encoding: Evaluating H.265, VP9, and AV1," *Journal of Network and Computer Applications*, vol. 171, p. 102782, 12 2020.

[9] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[10] Guanyu Gao, Y. Wen, and C. Westphal, "Resource provisioning and profit maximization for transcoding in Information Centric Networking," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, apr 2016, pp. 97–102.

[11] P. Pääkkönen, A. Heikkinen, and T. Aihkisalo, "Online architecture for predicting live video transcoding resources," *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–24, dec 2019.

[12] Y. Dong, X. Zhang, Y. Zhao, and L. Song, "A containerized media cloud for video transcoding service," in *International Conference on Consumer Electronics*, ser. ICCE'18. Las Vegas, Nevada, USA: IEEE, 1 2018, pp. 1–4.