Using Remote GPU Virtualization Techniques to Enhance Edge Computing Devices

José M. Cecilia[a], Juan Morales-García[b,*], Baldomero Imbernón[b], Javier Prades[a], Juan-Carlos Cano[a], Federico Silla[a]

[a]*Computer and Systems Informatics Department, Universitat Politècnica de València (UPV), Valencia, Spain*
[b]*Computer Science Department, Catholic University of Murcia (UCAM), Murcia, Spain*

Abstract

The Internet of Things (IoT) is driving the next economic revolution where the main actors are both data and immediacy. The IoT ecosystem is increasingly generating large amounts of data that are created but never analyzed. Efficient big data analysis in IoT infrastructures is becoming mandatory to transform this data deluge into meaningful information. Edge computing is proving to be a compelling alternative for enabling computing capabilities at the edge of the network. These computing capabilities could help in transforming the generated data into useful information. However, the edge computing platforms available on the market are low-power devices with limited computing horsepower. In this paper, we present a novel approach to providing computing resources to edge devices without penalizing their power consumption by using remotely virtualized GPUs. We evaluate this hardware environment by executing a computational-intensive clustering algorithm called Fuzzy Minimals (FM). Our results show that using a remotely virtualized GPU on the edge device provides a 3.2x speed-up factor compared to the local counterpart version. Moreover, we report up to 30% reduction in power consumption and up to 80% of energy savings at the edge device, delegating the GPU workload to the backend, transparently to the programmer.

Keywords:
Machine Learning, Clustering algorithms, Edge computing, Remote Virtualization, Virtualized GPUs, IoT.

1. Introduction

A key driver of the current digital revolution is the Internet of Things (IoT), where devices and humans are connected to the Internet, interacting with each other in real time. Two key factors that underpin this revolution are (1) data, which carries hidden patterns, correlations, and other valuable insights, and (2) real-time data analytics, required because knowledge is often time-sensitive and useful only within a specific time-frame [10]. The IoT generates zettabytes (10^{21} bytes) of "dark data" which is data never actually analyzed. In addition to the concern about the amount of data, the fast acceleration of data generation further complicates this scenario. Actually, 90% of the data stored in the world was just generated in the last two years [8]. Therefore, enabling efficient big data analytics within the IoT infrastructure is crucial to transform this data deluge into meaningful information.

Machine learning (ML) has become essential for the predictive analysis of huge amounts of data. Additionally, high-performance computing (HPC) plays an equally important role, particularly when the real-time response is crucial. The intersection between ML and HPC is mandatory to deal with large and complex datasets with real-time requirements. In this regard, using a cloud service approach, ML algorithms have been traditionally executed in supercomputers, where performance prevails over energy efficiency [17]. However, when performance is not the only concern, other approaches are feasible. For instance, edge/fog computing [20] is a recent trend towards decentralization, where initial computations on data are carried out close to (or actually at) the capture location. Processing in close proximity to mobile devices or sensors may provide energy savings, highly responsive web services for mobile computing, scalability, and privacy-policy enforcement for the IoT, as well as the ability to mask transient cloud outages. However, IoT devices have a limited power budget at this level of the network, as they typically rely on batteries or energy harvesters, leading to ultra-low power approaches. This limited power scenario translates into a major limitation for many components of the architecture, especially energy-intensive components such as wireless transmitters or even processing capabilities [2].

Edge devices used in IoT usually rely on ultra-low power solutions such as ARM-based CPUs or onboard microcontroller units. Microprocessor companies are releasing systems on chip (SoCs) that include higher com-

*Principal corresponding author
Email address: jmorales8@ucam.edu (Juan Morales-García)

puting capabilities by adding accelerators such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). Among them, we may highlight Nvidia's Jetson family [6], Intel's branch Movidius [14] or Google's Coral project [5]. Including these accelerators increases the power consumption of edge devices which is already a limiting factor. However, in terms of energy-efficient computing, accelerators substantially reduce application execution time, so that the increased power is amortized. An alternative or additional way to increase the computational capabilities of edge devices without paying for the power increase is the use of remote virtualization of accelerators [12]. This promising mechanism provides hardware resources (mainly GPUs) located on remote nodes to applications in a transparent way to programmers. Moreover, this technique increases GPU utilization thus reducing the waste of energy caused by the idle state of GPUs.

Several GPU virtualization solutions can be found in the literature. Based on CUDA [15], frameworks such as vCUDA [21], DS-CUDA [16], GViM [11], rCUDA [22], GVirtuS [9] or GridCuda [13] provide their own CUDA API implementations that are compatible with the original NVIDIA CUDA library. They are usually based on a client-server approach. The client is a library transparently linked to the CUDA application asking for GPU acceleration on the local node. The server is a daemon running in the remote node that owns the physical GPU. Every time the CUDA application performs a CUDA call, it is caught by the client and forwarded to the server to be executed on the physical GPU. Once the CUDA function is executed, results are returned to the client and forwarded to the CUDA application. It is important to note that the application is not aware of this virtualization process. Moreover, these solutions also allow remote GPUs to be concurrently shared among several applications. Table 1 summarizes the main features of GPU virtualization solutions.

Feature	Effect
GPU utilization	increased
Amount of required GPUs in the cluster	decreased
The total cost of GPUs in the cluster	decreased
Energy consumption	decreased
Execution time for a set of jobs	decreased

Table 1: Summary of the main features of GPU virtualization solutions.

In this work, we introduce a way to increase the computing capabilities of edge devices without increasing their power consumption. To that end, we leverage the rCUDA remote GPU virtualization middleware, using an Nvidia Jetson AGX Xavier as the client node. rCUDA is the best-maintained middleware among the aforementioned frameworks, also offering the best performance compared to other remote GPU virtualization solutions available as well as being available for all processor architectures [23]. The evaluation is carried out with the Fuzzy Minimals (FM) algorithm [3]: a challenging fuzzy clustering algorithm within the umbrella of ML. The main contributions of this paper are the following:

1. We provide a novel way to increase the computing power of edge devices without penalizing their power consumption. Our results show that a 3.2x speed-up factor can be achieved by reducing the power usage by up to 30% using a single remote GPU.
2. A multi-GPU implementation of the FM algorithm is provided to fully leverage multiple GPU instances. Different computational patterns are found in this algorithm, which affects the performance of the remote virtualization framework, concluding that CPU-GPU communication is a critical parameter in this environment.
3. An in-depth evaluation of Nvidia Jetson AGX Xavier is carried out, showing the benefits of introducing accelerators on edge devices for executing heavy workloads.
4. A thorough evaluation of the rCUDA middleware is performed as an alternative to provide the devices on the edge with increased computing capabilities. Several communication technologies are studied to assess the impact of communications within this framework.

The paper is organized as follows. We briefly introduce some preliminary concepts about the rCUDA middleware and the FM algorithm in Section 2. The parallelization in multi-GPU systems is introduced in Section 3. In Section 4 we describe the experimental results before we conclude the paper with a brief discussion and consideration of future work.

2. Background

2.1. The rCUDA middleware

The architecture of the rCUDA middleware, shown in Figure 1, follows a client-server distributed approach and works as described in the previous section (intercepting CUDA calls and forwarding them to the remote server). We refer the reader to a full description of rCUDA [18].

It is important to remark that rCUDA works in a completely transparent way to programmers, who do not have to change the source code of their applications in order to leverage this middleware. Actually, if the application is compiled to dynamically use the CUDA library, then the application does not even need to be recompiled to be executed with rCUDA. In this regard, rCUDA is binary compatible with CUDA 9.2 and implements the entire CUDA Runtime and Driver APIs (except for graphics

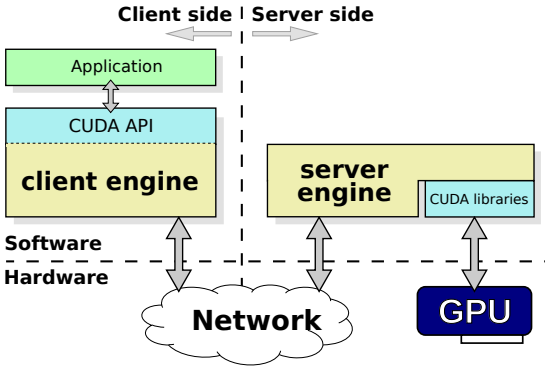


Figure 1: Architecture of the rCUDA middleware.

functions). It also provides support for the libraries included within CUDA (cuBLAS, cuFFT, etc). Additionally, it supports several underlying interconnection technologies by making use of a set of network-specific communication modules (currently TCP/IP, RoCE and InfiniBand). Independently of the exact network used, data exchange between rCUDA clients and servers is pipelined in order to attain high performance. Internal pipeline buffers within rCUDA use preallocated pinned memory, given the higher throughput of this type of memory [19].

rCUDA allows applications to make use of a technique known as multi-tenancy. This mechanism consists of providing several virtual instances of the same remote GPU to the same client application. That is, instead of providing the client application with several remote GPUs, all of them being different physical GPUs, with rCUDA it is possible to partition a physical GPU into several virtual instances and provide them to the same client application. In this way, the client application will have the illusion of being accessing several remote GPUs although all of those remote GPUs are finally mapped onto the same physical GPU. We will make use of multi-tenancy later in this paper.

It is important to notice that rCUDA guarantees the same isolation and security features among virtual GPU instances mapped onto the same physical GPU as the features guaranteed by CUDA. This is achieved by using a different GPU context for each of the virtual GPU instances being run at the same virtual GPU. Therefore, as far as CUDA guarantees isolation among GPU contexts when rCUDA is used, there cannot exist a data leak among applications concurrently using the same physical remote GPU.

2.2. The Fuzzy Minimals algorithm

Clustering analysis consists of assigning data points to clusters (or groups) so that points belonging to the same cluster are as similar as possible, whereas points belonging to another group are as different as possible to the former. This similarity process is based on the evaluation (i.e. minimization) of an objective function, which includes measures such as distance, connectivity, and/or intensity. Different objective functions may be chosen, depending on the dataset features or the application. Fuzzy clustering is a set of classification algorithms where each data point can belong to more than one cluster. Typical examples of these algorithms include the Fuzzy C-Means (FCM) and Fuzzy Minimals (FM) algorithms [1, 25]. The latter was initially proposed by Flores-Sintas *et al.* where authors demonstrate that it meets the expected characteristics of a classification algorithm; i.e. scalability, adaptability, self-driven, stability, and data-independent. In addition, the FM algorithm does not require setting the number of prototypes to be identified in the dataset, as it makes use of k-means or FCM algorithms. In what follows, a brief description of FM is provided. We refer the reader to [24, 25] for insights.

Let X be a set of n data points, such that

$$X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^F,$$

where F is the dimension of the vector space.

Algorithm 1 Fuzzy Minimals algorithm, where X is the input dataset to be classified, V is the algorithm output that contains the prototypes found by the clustering process. F is the dimension of the vector space.

1: Choose ε_1 and ε_2 standard parameters.
2: Initialize $V = \{\ \} \subset \mathbb{R}^F$.
3: $Load\_Dataset(X)$
4: $r = Calculate\_r\_Factor(X)$
5: $Calculate\_Prototypes(X, r, \varepsilon_1, \varepsilon_2, V)$

Algorithm 1 shows the sequential baseline of the FM algorithm. This algorithm has two main procedures: (1) the calculation of the factor r (line 4) and (2) the calculation of prototypes or centroids (lines 5) to complete the set V. The factor r is a parameter that measures the isotropy in the data set. The use of Euclidean distance implies that the homogeneity and isotropy of the feature space are assumed. Whenever homogeneity and isotropy are broken, clusters are created in the feature space.

$$\frac{\sqrt{|C^{-1}|}}{nr^F} \sum_{x \in X} \frac{1}{1 + r^2 d_{xm}^2} = 1. \quad (1)$$

The calculation of factor r is based on Equation 1. It is a non-linear expression, where $|C^{-1}|$ is the determinant of the inverse of the covariance matrix, m is the mean of the sample X, d_{xm} is the Euclidean distance between x and m, and n is the number of elements in the sample.

Once the factor r is calculated, the calculation of prototypes is executed to obtain the clustering result in V (see Algorithm 2). The objective function used by FM is given by Equation 2

Algorithm 2 *Calculate_Prototypes*() of fuzzy minimals algorithm.

for $k = 1; k < n; k = k + 1$ **do**
  $v_{(0)} = x_k$, $t = 0$, $E_{(0)} = 1$
  while $E_{(t)} \geq \varepsilon_1$ **do**
    $t = t + 1$
    $\mu_{xv} = \frac{1}{1+r^2 \cdot d_{xv}^2}$, using $v_{(t-1)}$
    $v_{(t)} = \frac{\sum_{x \in X} \left(\mu_{xv}^{(t)}\right)^2 \cdot x}{\left(\mu_{xv}^{(t)}\right)^2}$
    $E_{(t)} = \sum_{\alpha=1}^{F} \left(v_{(t)}^{\alpha} - v_{(t-1)}^{\alpha}\right)$
  end while
  if $\sum_{\alpha}^{F} (v^{\alpha} - w^{\alpha}) > \varepsilon_2, \forall w \in V$ **then**
    $V \equiv V + \{v\}$.
  end if
end for

$$J_{(v)} = \sum_{x \in X} \mu_{xv} \cdot d_{xv}^2, \tag{2}$$

where

$$\mu_{xv} = \frac{1}{1 + r^2 \cdot d_{xv}^2}, \tag{3}$$

Equation 3 measures the degree of membership for a given element x to the cluster where v is the prototype. The FM algorithm is an iterative procedure that aims to minimize the objective function through Equation 4, giving the prototypes represented by each cluster. Finally, ε_1 and ε_2 are input parameters that establish the error degree committed in the minimum estimation and show the difference between potential minimums respectively.

$$v = \frac{\sum_{x \in X} \mu_{xv}^2 \cdot x}{\sum_{x \in X} \mu_{xv}^2} \tag{4}$$

3. Parallelization strategies for the Fuzzy Minimals on multi-GPU systems

This section presents the CUDA parallelization of the FM algorithm in multi-GPU environments. Parallelization of the FM algorithm on a single GPU was initially proposed in [3]. However, the rCUDA middleware allows applications to use many GPUs on a single node in a shared memory fashion. Thus, the FM algorithm is redesigned to fully leverage this new computational landscape. Our multi-GPU parallelization approach is based on the distributed parallelization of the FM algorithm introduced in Timon et al. [25], called the Parallel Fuzzy Minimals (PFM); a parallel version of the FM algorithm to improve data-parallelism. Contrary to other proposals, the FM algorithm does not require cluster comparison to minimize the objective function. The PFM takes advantage of this feature to split the original data set into different subsets to which the FM is independently applied. The clustering procedure is not affected by this partition as the global properties of the original data set are not lost. Actually, each subset will be classified using an objective function that includes the factor r, which has information about the overall data structure. In fact, the factor r is a global parameter that contains information about the entire data set (see Figure 2).

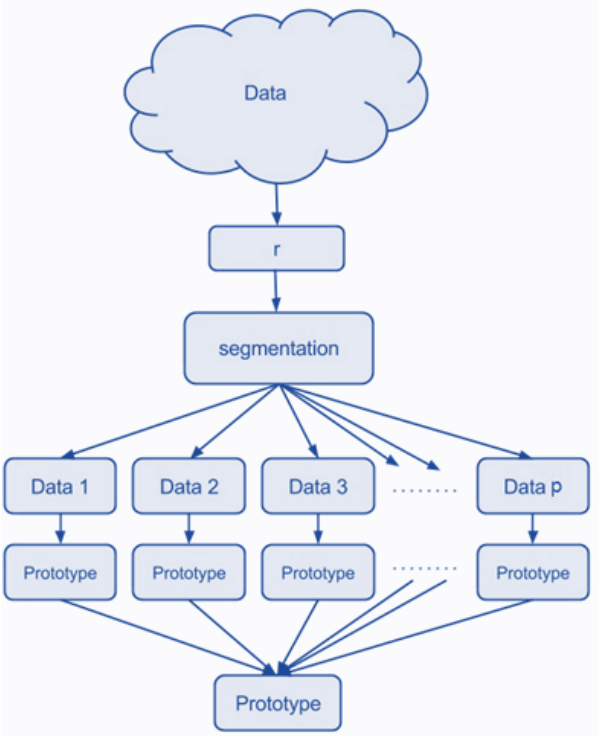


Figure 2: Parallel implementation of fuzzy minimals clustering algorithm based on [25].

In order to benefit from the PFM's intrinsic parallelism, the factor r must be first calculated using the whole dataset (see Figure 2). As previously explained, the factor r calculation is based on the resolution of a non-linear expression that is computationally intensive. Moreover, the factor r calculation grows exponentially with the number of variables to be clustered [4]. This may become a bottleneck. Therefore, the first goal is to leverage the available multi-GPU system for the factor r calculation. Algorithm 3 shows the multi-GPU implementation of computing factor r. The factor r procedure is based on two main tasks that are applied to each row in the dataset. They are (1) the calculation of the fuzzy covariance matrix, which is a square matrix ($columns \times columns$) and (2) the calculation of its determinant. It is an iterative procedure with as many iterations as rows (points in the datasets). These iterations can be carried out independently and thus they can be equally distributed among the number of GPUs available in the system. This is achieved by parallelizing the for loop with as many CPU

Algorithm 3 *Factor r calculation algorithm in GPU*

1: $\#pragma\ omp\ parallel\ for\ private\ (detvalue)\ schedule\ (dynamic)\ reduction\ (+:rfactor)$
2: **for** $i = 1; i < rows; i++$ **do**
3: $cudaSetDevice(omp\_get\_thread\_num())$
4: $covariance <<< bl, th, sh >>> (dataset, determ, row_i, rows, cols)$
5: $LU\_solver <<< bl, th, sh >>> (determ)$
6: $cudaMemcpy(detvalue, determ, sizeof(float), cudaMemcpyDeviceToHost)$
7: $rfactor += \frac{1}{\sqrt{detvalue}}$
8: **end for**

Algorithm 4 *Multi-GPU parallelization of the prototype calculation.*

1: $\#pragma\ omp\ parallel\ for\ private(jE\_total, jE\_reduction, mu\_total, mu\_reduction, mu, prot, prov\_prot, aux)$
2: **for** $i = 1; i < rows; i++$ **do**
3: $cudaSetDevice(omp\_get\_thread\_num())$
4: $cudaMemcpy(prot, (dataset + i * columns), columns * sizeof(FLOAT), cudaMemcpyDeviceToDevice);$
5: **while** $jE\_total > error$ **do**
6: $new\_prot <<< bl, th, sh >>> (dataset, mu, prot, prov\_prot);$
7: $reduction\_mu <<< 1, columns >>> (mu, mu\_reduction);$
8: $cudaMemcpy(mu\_total, mu\_reduction, sizeof(FLOAT), cudaMemcpyDeviceToDevice)$
9: $distance\_prots <<< 1, columns >>> (jE, prov\_prot, prot, mu\_total, columns);$
10: $reduction\_jE <<< 1, columns >>> (jE, jE\_reduction);$
11: $cudaMemcpy(jE\_total, jE\_reduction, sizeof(FLOAT), cudaMemcpyDeviceToDevice)$
12: $aux = prot$
13: $prot = prov\_prot$
14: $prov\_prot = aux$
15: **end while**
16: $cudaMemcpy(prot\_provisional + i * columns, prot, columns * sizeof(FLOAT), cudaMemcpyDeviceToHost);$
17: **end for**

threads as GPUs are available in the rCUDA client node (see line 1, algorithm 3). OpenMP is used to do this task and also to eventually reduce the factor r. Moreover, the performance is also affected by the number of columns (i.e. the dimension of points F). Actually, performance will be drastically affected by this last parameter, since the execution time of the determinant calculation, which is carried out in the GPU using the LU method, will increase exponentially with the size of the matrix. The fuzzy covariance matrix is then calculated to obtain factor r.

The multi-GPU parallelization of prototype calculation is shown in Algorithm 4. Here, the input dataset is equally divided into subgroups that are distributed among the GPUs available on the rCUDA client node. As with the factor r parallel design, this is achieved by parallelizing the outermost for loop (see Lines 1-3, Algorithm 4). Each GPU thread calculates the distance between each row of the data set and the different prototypes by calculating Equation 3 to obtain the probability of belonging to each subset. Finally, in each GPU, a block reduction must be performed to obtain the total degree of relevance for each point in the dataset. It is worth highlighting the large number of synchronizations and data transfers between CPU and GPU this function has.

4. Evaluation and discussion

4.1. Hardware setup and benchmarking

Our experimental environment includes nine nodes (see Figure 3). Seven out of these nine nodes will be used as GPU servers by leveraging the rCUDA middleware whereas the other two nodes will be used to execute the FM application. Regarding the GPU server nodes, six of them are equipped with processors belonging to different Intel Xeon generations. They comprise several Nvidia GPU architectures, ranging from old Kepler (2xK80 and 6xK20) to Pascal (3xP100) and Volta (1xV100) architectures. GPUs in these nodes are connected to the rest of the nodes by a PCI express v3.0 link (16 GB/sec). The seventh GPU server is an IBM POWER 9 system with two 16-core Power 9 processors at 2.7 GHz including two Nvidia V100 GPUs. These GPUs are connected to the processors and between them through an NVLink high-speed interconnect with a theoretical bandwidth equal to 300 GB/sec. All of these seven nodes in the cluster are connected among them by two network technologies: Gigabit Ethernet (1 Gbps) and EDR InfiniBand (100 Gbps).

In addition to the seven GPU-sever nodes, our testbench includes two additional nodes where the FM ap-

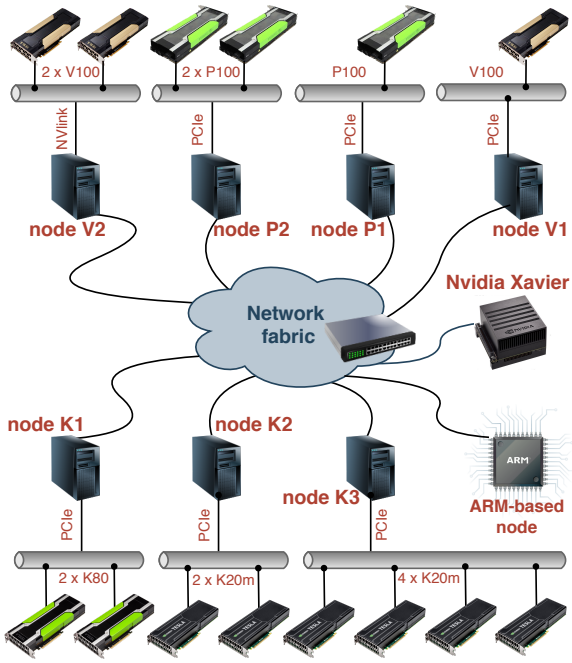


Figure 3: Hardware environment.

GPU ID	Model	Location	GPU ID	Model	Location
0	P100	node P1	**7**	K80	node K1
1	P100	node P2	**8**	K20m	node K2
2	P100	node P2	**9**	K20m	node K2
3	V100	node V1	**10**	K20m	node K3
4	V100	node V2	**12**	K20m	node K3
5	V100	node V2	**12**	K20m	node K3
6	K80	node K1	**13**	K20m	node K3

Table 2: GPU models and hardware location of the experimental environment.

Tag (amount of GPUs)	Devices used (GPU IDs)	Tag (amount of GPUs)	Devices used (GPU IDs)
1 GPU	0	**8 GPUs**	0,1,2,3,4,5,6,7
2 GPUs	0,1	**10 GPUs**	0,1,2,3,4,5,6,7 8,9
3 GPUs	0,1,2	**12 GPUs**	0,1,2,3,4,5,6,7 8,9,10,11
4 GPUs	0,1,2,3	**14 GPUs**	0,1,2,3,4,5,6,7 8,9,10,11,12,13
6 GPUs	0,1,2,3,4,5		

Table 3: Relationship between experiment tag and GPU ID shown in Table 2. Please refer to Table 2 in order to know the exact GPU location in the cluster.

plication will be executed along with the rCUDA client library. The first of these two nodes is an edge computing Nvidia AGX Jetson Xavier. This device is connected to the cluster through Gigabit Ethernet (1Gbps) and has an 8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3, 512-core Volta GPU with Tensor Cores, and 16GB 256-Bit LPDDR4x running at 137GB/sec. It is important to note that this is where the FM computation is performed using either the local GPU or the remote hardware resources provided by rCUDA.

The connection of the Nvidia AGX Jetson Xavier with the GPU servers is one of the main infrastructure bottlenecks. The remote GPU virtualization strategy to bring hardware resources from the GPU servers to the rCUDA client requires intensive communication between the rCUDA client and the remote server processes. Therefore, the technology used in this connection should be evaluated to identify the main bandwidth and latency requirements in this virtualized context. However, Nvidia Jetson Xavier has only an Ethernet interface, which limits this evaluation. Therefore, an additional ARM-based node with two different network interfaces, Gigabit Ethernet and EDR InfiniBand, has been considered. This node has two 28-core ARM processors ThunderX2 at 2GHz with 256 GB RAM and will be used as an rCUDA client in order to test the connection among the rCUDA clients and GPU servers.

Tables 2 and 3 summarize the hardware resources available for the evaluation and the keywords used for the ongoing experimental results.

An interesting discussion is how general is the hardware configuration leveraged in our experiments. In this regard, there are three considerations to be done: (1) is the NVIDIA Jetson Xavier node representative of actual edge nodes? (2) are the nodes hosting the real GPUs representative of current cluster nodes? (3) is the ThunderX2 node representative of current edge devices? Regarding (1), please notice that edge devices are based on ARM processors, as the NVIDIA Jetson Xavier is. However, the latter device comprises a powerful GPU, which could not be a common choice in edge deployments. In this regard, it is important to remark that, in our experiments, when we use remote GPUs according to our proposal, the GPU located within the NVIDIA Jetson Xavier node is not used. Actually, that GPU is only used for comparison purposes, and it is not part of our edge proposal. Therefore, using the NVIDIA Jetson Xavier device in our experiments is representative of current edge deployments. Regarding (2), the nodes we have used to host the real GPUs are a good example of what it could be found in current clusters. Thus, the combination of the NVIDIA Jetson Xavier and the cluster nodes used in this paper represents a general hardware solution (as far as the GPU used in the NVIDIA Jetson Xavier is not used). Finally, regarding (3), the big ThunderX2 ARM node used in this paper is not representative of the current edge devices. However, notice that we only use that node in order to deeper analyze the behavior of our proposal when a faster network is leveraged, as it could be the case when 5G or 6G infrastructure is widely available. In summary, the hardware configuration used in our experiments is general enough to get general conclusions from the experiments.

In order to calculate the energy consumption of our

proposal, we have measured, at intervals of one second, the power consumed by each of the devices used. The power consumed by Nvidia Jetson AGX Xavier has been measured using the Watts Up Pro power meter. Regarding the K20, K80, P100 and V100 GPUs, their power consumption has been measured using Nvidia Management Library (NVML).

Finally, the dataset to test our implementation is taken from the UCI Machine Learning Repository [7]. It is an input database with up to 100K points, belonging to different chemical sensors. These sensors have been provided with a set of parameters that measure the quality of the gas, as well as the temperature and humidity values of the environment.

4.2. Performance and energy evaluation

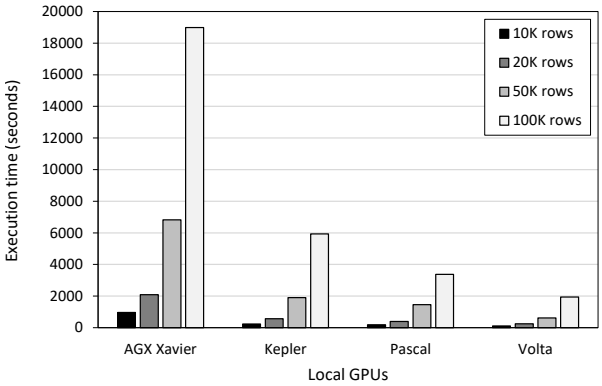


Figure 4: Execution time of FM algorithm using a local GPU for a different number of rows.

Figure 4 shows the execution time of the FM algorithm on different GPU architectures. These GPU architectures are those described in Section 4.1, which include the edge computing platform Xavier and Nvidia HPC GPU solutions K80 (Kepler), P100 (Pascal), and V100 (Volta). Only one local GPU has been used in each experiment. The difference in performance between Xavier and HPC GPUs is substantial as expected, reaching up to 13.2X speed-up factor. Computationally speaking, Xavier offers a low-power solution with some computational capabilities but there is still a big gap in performance between HPC GPUs and edge computing platforms and therefore cloud-based approaches are still mandatory to deploy high computational workloads such as those within the umbrella of ML.

Our main goal in this paper is to introduce the computational horsepower available in the cloud into the edge transparently to the programmer. In this way, programmers can develop edge computing-based applications with high-performance capabilities without penalizing power consumption. With this approach, programmers could also use already existing multi-GPU codes that were developed for shared memory multi-GPU nodes on edge computing platforms.

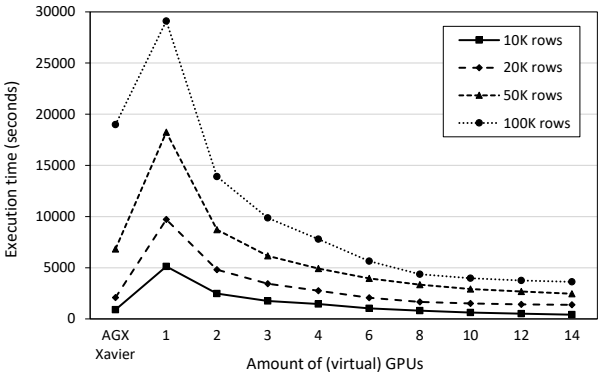
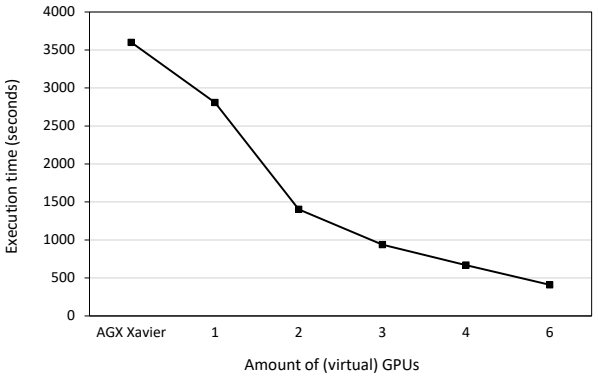


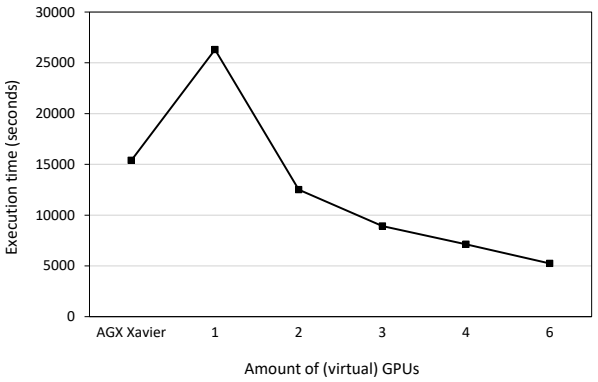
Figure 5: Execution time of FM algorithm on AGX Xavier, varying the number of virtual GPUs and also the number of rows. Execution times for the AGX Xavier node are used as a reference.

In the search for our goal, Figure 5 shows the FM execution times on the Xavier while varying the number of virtual GPUs available at the edge platform. In other words, the rCUDA client is executed at the AGX Xavier node where several rCUDA servers are running at different nodes where HPC GPUs are plugged into. It is worth highlighting that only one GPU process is running at each physical remote GPU, which means up to 14 physical GPUs are used in the cloud to provide these performance figures. Table 3 shows the exact GPUs used for each experiment, whereas Table 2 provides the location for each GPU. Figure 5 shows an important concern: while all virtualized GPUs have higher performance than the Xavier AGX (see Figure 4) the rCUDA middleware introduces some communication overhead, which can be perfectly seen in Figure 5 when only one virtual GPU is used, thus worsening the performance of the low-power GPU available on Xavier. However, as more virtualized GPUs are introduced into the edge, the overall application performance gets better. This is particularly true when the data set is large enough because the overhead introduced by rCUDA communications is hidden by the higher computational load assigned to each of the GPUs. This leads us to believe that for small problem sizes, GPUs are not busy enough and, therefore, the computation could be optimized by better using remote GPUs, as will be explained later in this section.

As explained in Section 2.2, the FM algorithm is divided into two main functions: *Calculate_r_Factor* and *Calculate_Prototypes*. They have different computational patterns that affect their performance in the virtualized environment targeted. Figure 6a shows the execution time of the *Calculate_r_Factor* function when this function addresses a large dataset composed of 100,000 rows and up to 11 different variables (columns), which is actually a very challenging scenario. The factor r calculation is made up of several GPU kernels that are executed independently. Actually, there is only a final reduction to get the final factor r. This means that there

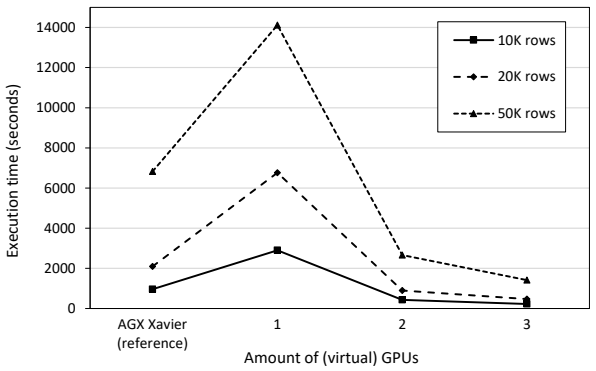


(a) Execution time of factor r

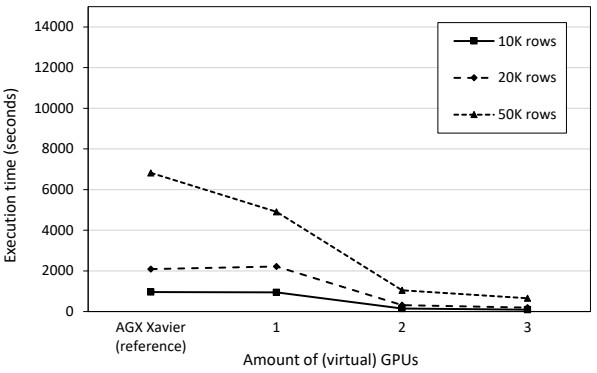


(b) Execution time of prototypes calculation

Figure 6: Execution time of factor r and prototypes calculation for 100,000 rows. The number of virtual GPUs are varied up to 6 virtual GPUs. The host node is the AGX Xavier that we also use as baseline.



(a) ARM-based node with Ethernet



(b) ARM-based node with InfiniBand

Figure 7: Execution time of FM algorithm, varying the number of virtual GPUs for different number of rows from ARM-based node with Ethernet and InfiniBand.

are few CPU-GPU or GPU-GPU communications in the loop which actually benefits the system's scalability. Remember that CPU-GPU communication in this virtualized environment means having communication over the network which penalizes the performance. Results in Figure 6a show the execution time of the factor r calculation in the local GPU included in the Jetson AGX Xavier and also using up to 6 remote GPUs. It can be seen in the figure that execution time for the factor r calculation progressively decreases along with the number of GPUs, showing almost linear scalability. On the other hand, Figure 6b shows the execution time of the prototype calculation. Here, the results are quite different because the CPU-GPU communication ratio is higher than in the factor r counterpart version. The set of prototypes obtained in each iteration by each GPU is shared with the rest of the GPUs. This means that the increment in network traffic must be compensated by adding more GPUs in the system that allow reducing computation time. This effect happens from 2 remote GPUs and beyond and this pattern is transferred to all the computations since the calculation of prototypes represents 95% of the total computation of the algorithm.

Summing up, there are two main conclusions from this initial analysis. The first conclusion is that remote virtualization of GPUs is drastically penalized by the CPU-GPU communication overhead. This makes sense since CPUs are connected to GPUs via low-latency and high bandwidth connections such as PCI Express 3.0 (approx. 16 GB/sec) or NVLINK (approx. 300 GB/sec). On the contrary, in rCUDA each CPU-GPU transfer goes through the network fabric, thus depending on the connectivity technology that is used in the cluster. The above results are based on Ethernet, whose bandwidth is around 100MB/sec. This is at least an order of magnitude less than the scenario within the node. This is actually shown in Figure 7 where the execution time of the FM algorithm is shown, varying the interconnection technology (i.e. Gigabit Ethernet in Figure 7a and InfiniBand in Figure 7b). It is important to note that experiments in these figures have been carried out in the ARM-based node instead of AGX Xavier as the latter does not have an InfiniBand interface. With the improvement in the interconnection technology, execution time using remote GPUs is dramatically reduced, reaching up to 3x speed-up factor.

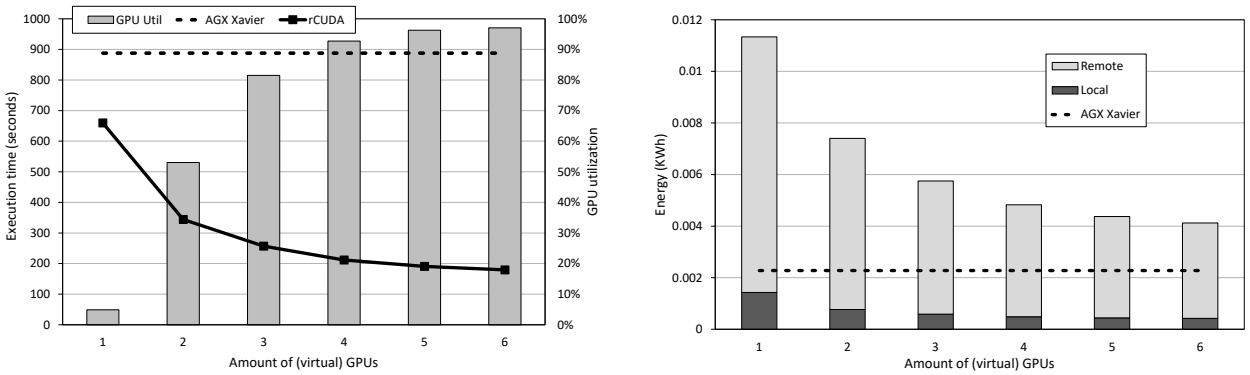
The second conclusion from this initial analysis is

that it looks like GPUs are not busy enough and therefore we may think about making better use of them. In this regard, in order to increase GPU utilization we may think about multi-tenancy. That is, instead of running one single GPU process per physical GPU as in the previous experiments, we may execute several GPU processes in the same physical GPU. This can be achieved by virtualizing multiple times a single physical GPU and providing multiple virtual instances of that GPU to the FM application.

This multi-tenancy approach is used in Figure 8, which shows performance and energy consumption when remote GPU virtualization is used in Nvidia Jetson Xavier but only one remote GPU (i.e. V100) is physically targeted although this GPU supports several GPU instances virtually. This increases GPU utilization as it is reflected by the bars of Figures 8a, 8c, and 8e. Almost 100% of the GPU utilization is achieved with 6 virtual GPUs running on the same V100. Indeed, this is translated into a performance gain as the continuous line shows (the dashed line shows the yield horizon set by Xavier's performance). Almost full utilization of the V100 resources (80%) is required to match Xavier's performance for the medium datasets (i.e. 50K). As for 10K and 100K datasets (Figures 8a and 8e), Xavier's performance is defeated with 1-2 Virtual GPUs, achieving a speed-up factor in the range of 3.5X to 4.5X with 6 virtual GPUs within the same physical V100. It is important to note that the performance using remote virtualization techniques does not entirely depend on the data size. Actually, there is another important factor that also penalizes performance as shown in Figures 7a and 7b; i.e. the number of CPU-GPU communications. This last factor depends on the number of synchronization, cudaMemcpy, CUDA kernel launches, and so on, carried out by the application, which in the case of FM, and clustering algorithms in general, depends on the convergence criterion as a function of the layout of the input data. The number of calls to *cudaMemcpyDeviceToHost* for 50K is $906,493$ and $892,587$ for 100K, which means less CPU-GPU communications for 100K case. On the right-hand side of Figure 8, the energy consumption of the FM algorithm is shown. In this case, the dashed line shows Xavier's energy consumption when running the FM algorithm locally; i.e. using its low-power GPU. It can be seen in Figures 8b, 8d and 8f that overall energy consumption is reduced as the multi-tenancy degree increases. Actually, with 6 virtual instances, overall energy consumption is less than one-half of the energy required with a single remote virtual instance. On the other hand, overall energy consumption is always larger than when the FM algorithm is executed locally in Nvidia Xavier edge node. However, it can be seen that the energy required by the edge node noticeably decreases when several virtual instances are employed. This is better analyzed in Figure 9 where the average power consumption of the edge computing device and NVIDIA V100, running the FM algorithm for 10K, 50K, and 100K is shown.

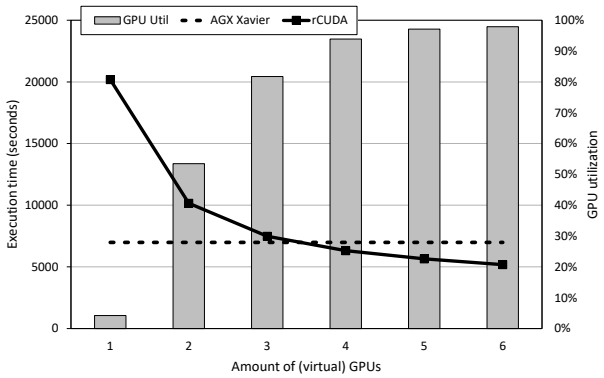
Figure 9 shows that the average power consumption of this edge device when the GPU is active is 10.15 Watts with 0.5 standard deviations. Furthermore, Xavier's power consumption is reduced by a factor of up to 30% when the execution is offloaded to a remote GPU. Xavier's local GPU can be switched off in this scenario, reducing its power budget in the range of 7-8 watts. As the number of virtual GPU instances increases, Xavier's CPU workload also increases, and this is translated into slightly higher power consumption. Actually, the power consumption for 1 virtual GPU is 7.77 watts and increases up to 8.3 for 6 virtual GPUs. But even in this worst-case scenario, our results show 21.15% power savings over using Xavier's local GPU. The server-side power consumption also depends on the workload supported by Nvidia V100 used in our experiments. It ranges from 54.54 watts for 1 GPU instance to 75.80 watts for 6 GPUs instances. The power consumption of V100 is higher than Xavier (up to 7.5x) but it offers better performance as expected (up to 3.1x speed-up factor using 6 GPU instances). A trade-off between power consumption and performance is clearly reported; the edge/fog computing devices such as Nvidia Jetson Xavier are low-power devices because they are designed to be in IoT infrastructures where power consumption could become a big issue. By delegating the heaviest workloads to the GPU servers, up to 30% of the power consumption could be transferred to a more suitable and supportive infrastructure. In addition, performance can be gained but taking into account that the total power consumption of the solution is penalized. Summing up, it is important to note that the execution time is reduced in exchange for increasing both the power and energy consumed in the IoT infrastructure as a whole (i.e. GPU servers and rCUDA client).

The most remarkable result to emerge from the data, however, is that the rCUDA client is where there are usually problems with energy and power supply in IoT infrastructures, and our approach reduces the execution time, power, and energy at this level of the architecture. Actually, Figure 10 clearly shows the energy savings achieved at the rCUDA client (Jetson AGX Xavier) thanks to the usage of a remote virtualized GPU providing multiple instances of virtual GPUs. In order to compute energy savings, the energy required by the AGX Xavier node when executing the FM algorithm with the local small GPU was used as the baseline. It can be seen in the figure that, the more virtual GPU instances are leveraged, the larger energy savings are attained. Energy savings, when 6 remote virtual GPU instances are used, ranging from 40% for 50K rows up to 81% for 10K rows. It is noteworthy that in the case of 100K rows energy savings are about 70%. These extraordinary energy savings could translate into much longer battery availability, for instance. On

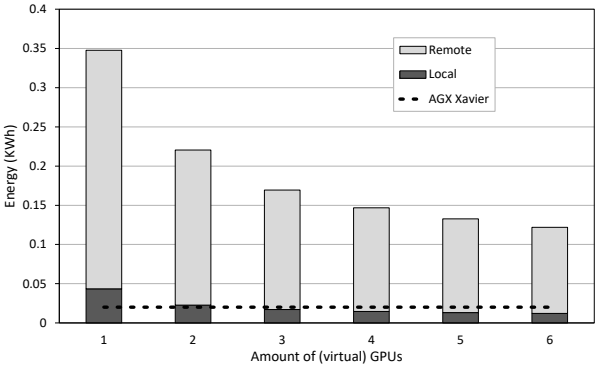


(a) Performance with 10K rows

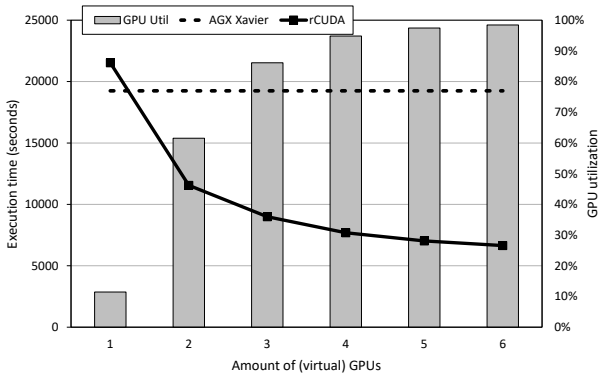
(b) Energy consumption with 10k rows



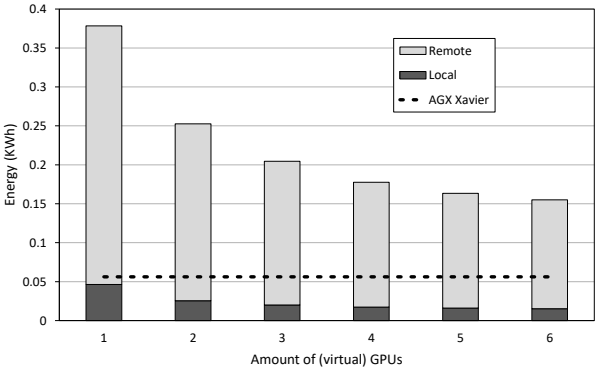
(c) Performance with 50K rows



(d) Energy consumption with 50k rows



(e) Performance with 100K rows



(f) Energy consumption with 100k rows

Figure 8: Execution time and energy consumption of the FM algorithm at local AGX Xavier and varying the number of virtual GPUs within the same physical GPU (V100)

the other hand, the previous discussions about the behavior of the FM algorithm for 50K rows also apply to this figure, where it can be seen that energy savings for this particular amount of rows are noticeably lower than for the other amounts of rows as the performance gains of using remote virtualization is lower than in the other case. But even with these lower performance benefits, the energy savings reach up to 40%.

5. Conclusions and future work

This paper evaluates the virtualization of remote GPUs in edge computing devices in order to provide them with computational horsepower without increasing the power consumption of these devices. A particular case study is developed with a fuzzy clustering algorithm (Fuzzy Minimals, FM) which is widely used for different data science applications. The FM's GPU version has two main kernels with different CPU-GPU communication patterns that strongly affect this kind of GPU virtualization. The factor r calculation shows little CPU-GPU

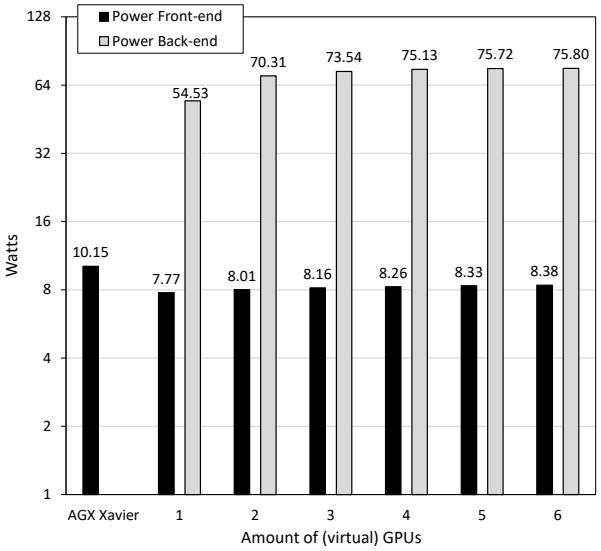


Figure 9: Average power consumption of Nvidia AGX Xavier (rCUDA client) and Nvidia V100 (GPU server), running 10K, 50K and 100K datasets and varying the number of virtual GPUs within the same physical GPU.

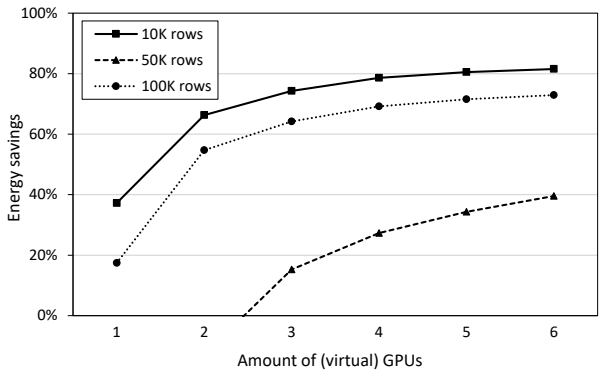


Figure 10: Energy savings achieved at the Jetson AGX Xavier node when using remote virtual GPUs with respect to local executions. 10K, 50K, and 100K datasets are run while varying the number of virtual GPUs within the same physical GPU.

communication while the calculation of the prototypes needs continuous global synchronization in each step of the algorithm. Our results show that new edge computing platforms, which include low-power GPUs such as Nvidia Jetson Xavier, provide an excellent framework for driving edge computing as a real alternative to smart applications. However, they have yet to further improve their performance to meet the challenge of the data deluge in IoT ecosystems. Remote virtualization of GPUs can be a good solution to increase computing power without increasing the power budget of edge devices. We report a performance gain of up to 3.1x speed-up factor by just using a single remote GPU running multiple virtual GPU instances on it. Moreover, the power consumption of the edge device is reduced by up to 30%, obtaining up to 80% of energy savings by delegating the GPU workload to the rCUDA servers as its GPU can remain off. However, the overall power consumption of the IoT infrastructure (rCUDA clients and remote GPU servers) is increased by a factor of 7x with the solution presented here. Anyway, notice that the GPU servers are not a dedicated infrastructure but they can provide service to multiple applications. Therefore, the increase in overall energy consumption is diluted, making the energy saving in the edge device more appealing.

The conjunction of virtualization and edge computing is still at a relatively early stage; we emphasize that we have only so far tested a relatively simple variant of this solution that is designed for supercomputer environments. We definitely think that designing low-power virtualization solutions can reduce overall power consumption by maintaining the performance gains for edge devices. Moreover, there are many other types of data science algorithms still to explore, and as such, it is a potentially fruitful area of research. However, the different kernels of the FM algorithm have different computational and communication patterns that provide interesting conclusions about the computational patterns that can benefit from this edge computing virtualized infrastructure. This and other algorithms could also be tested on frameworks other than the one tested in this paper in order to assess the impact of edge device characteristics on the performance and power consumption of these algorithms. We hope that this paper stimulates further discussion and work.

Finally, another direction for future work is to model performance vs. power consumption, varying the different hardware parameters used in this work, in order to provide the research community with a useful tool to evaluate the benefits of using remote GPU virtualization in the context of edge computing.

Declarations

Funding

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017861 and also by projects RTC2019-007159-5 and Ramon y Cajal Grant RYC2018-025580-I, funded by MCIN/AEI/10.13039/501100011033, "FSE invest in your future" and "ERDF A way of making Europe". The authors are grateful for the generous support provided by Mellanox Technologies Inc.

Authors' contributions

Conceptualization, J.M.C., F.S.; methodology, J.M.C., F.S., J.C.C.; software, B.I., J.P., J.M.G.; validation, J.M.C, F.S., J.C.C.; formal analysis, B.I., J.P., J.M.G.; investigation, J.M.C., F.S.; writing—original draft preparation, J.M.C., F.S.; writing—review and editing, J.M.C., F.S.; visualization, J.P.; supervision, J.M.C., F.S., J.C.C.; project administration, J.M.C., F.S., J.C.C.; funding acquisition, J.M.C., F.S., J.C.C..

Conflict of interest

The authors declare that they have no conflict of interest.

Consent to publish

All authors have read and agreed to the published version of the manuscript.

References

[1] Bezdek, J., Ehrlich, R. y Full, W., 1984. FCM: The Fuzzy C-Means clustering algorithm. Computers & Geosciences 10, 191–203.

[2] Capra, M., Peloso, R., Masera, G., Ruo Roch, M., Martina, M., 2019. Edge computing: A survey on the hardware requirements in the internet of things world. Future Internet 11, 100.

[3] Cebrian, J.M., Imbernón, B., Soto, J., García, J.M., Cecilia, J.M., 2020. High-throughput fuzzy clustering on heterogeneous architectures. Future Generation Computer Systems 106, 401–411.

[4] Cecilia, J.M., Timón, I., Soto, J., Santa, J., Pereñíguez, F., Muñoz, A., 2018. High-throughput infrastructure for advanced its services: A case study on air pollution monitoring. IEEE Transactions on Intelligent Transportation Systems 19, 2246–2257.

[5] Coral, . Coral: Build intelligent ideas with our platform for local ai. urlhttps://coral.withgoogle.com/.

[6] Ditty, M., Architecture, T., Montrym, J., Wittenbrink, C., 2014. Nvidia's tegra k1 system-on-chip, in: 2014 IEEE Hot Chips 26 Symposium (HCS), IEEE. pp. 1–26.

[7] Dua, D., Graff, C., 2017. UCI machine learning repository. URL: `https://archive.ics.uci.edu/ml/datasets/Gas+sensors+for+home+ activity+monitoring`.

[8] Duranton, M., De Bosschere, K., Gamrat, C., Maebe, J., Munk, H., Zendra, O., 2017. The hipeac vision 2017.

[9] Giunta, G., Montella, R., Agrillo, G., Coviello, G., 2010. A gpgpu transparent virtualization component for high performance computing clouds, in: European Conference on Parallel Processing, Springer. pp. 379–391.

[10] Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M., 2013. Internet of things (iot): A vision, architectural elements, and future directions. Future generation computer systems 29, 1645–1660.

[11] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P., 2009. Gvim: Gpu-accelerated virtual machines, in: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17–24.

[12] Imbernón, B., Prades, J., Gimenez, D., Cecilia, J.M., Silla, F., 2018. Enhancing large-scale docking simulation on heterogeneous systems: An mpi vs rcuda study. Future Generation Computer Systems 79, 26–37.

[13] Liang, T.Y., Chang, Y.W., 2011. Gridcuda: a grid-enabled cuda programming toolkit, in: 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications, IEEE. pp. 141–146.

[14] Movidius, . Movidius: On-device computer vision & ai. urlhttps://www.movidius.com/.

[15] Nvidia, C., 2019. Cuda c programming guide, version 10.1. NVIDIA Corp .

[16] Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., Narumi, T., 2012. Ds-cuda: a middleware to use many gpus in the cloud environment, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, IEEE. pp. 1207–1214.

[17] Papadokostaki, K., Mastorakis, G., Panagiotakis, S., Mavromoustakis, C.X., Dobre, C., Batalla, J.M., 2017. Handling big data in the era of internet of things (iot), in: Advances in Mobile Cloud Computing and Big Data in the 5G Era. Springer, pp. 3–22.

[18] Prades, J., Silla, F., 2019. GPU-Job Migration: The rCUDA Case. IEEE Transactions on Parallel and Distributed Systems 30, 2718–2729.

[19] Reaño, C., Silla, F., 2020. Redesigning the rCUDA Communication Layer for a Better Adaptation to the Underlying Hardware. Concurrency and Computation: Practice and Experience .

[20] Satyanarayanan, M., 2017. The emergence of edge computing. Computer 50, 30–39.

[21] Shi, L., Chen, H., Sun, J., Li, K., 2011. vcuda: Gpu-accelerated high-performance computing in virtual machines. IEEE Transactions on computers 61, 804–816.

[22] Silla, F., Iserte, S., Reaño, C., Prades, J., 2017. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. Concurrency and Computation: Practice and Experience 29.

[23] Silla, F., Prades, J., Baydal, E., Reaño, C., 2020. Improving the performance of physics applications in atom-based clusters with rCUDA. Journal of Parallel and Distributed Computing 137, 160 – 178.

[24] Soto, J., Flores-Sintas, A. y Palarea-Albaladejo, J., 2008. Improving probabilities in a fuzzy clustering partition. Fuzzy Sets and Systems 159, 406–421.

[25] Timón, I., Soto, J., Pérez-Sánchez, H., Cecilia, J.M., 2016. Parallel implementation of fuzzy minimals clustering algorithm. Expert Systems with Applications 48, 35–41.