GPU-Job Migration: the rCUDA Case

Javier Prades and Federico Silla

Abstract—Virtualization techniques have been shown to report benefits to data centers and other computing facilities. In this regard, not only virtual machines allow to reduce the size of the computing infrastructure while increasing overall resource utilization, but also virtualizing individual components of computers may provide significant benefits. This is the case, for instance, for the remote GPU virtualization technique, implemented in several frameworks during the recent years.
The large degree of flexibility provided by the remote GPU virtualization technique can be further increased by applying the migration mechanism to it, so that the GPU part of applications can be live-migrated to another GPU elsewhere in the cluster during execution time in a transparent way.
In this paper we present the implementation of the migration mechanism within the rCUDA remote GPU virtualization middleware. Furthermore, we present a thorough performance analysis of the implementation of the migration mechanism within rCUDA. To that end, we leverage both synthetic and real production applications as well as three different generations of NVIDIA GPUs. Additionally, two different versions of the InfiniBand interconnect are used in this study. Several use cases are provided in order to show the extraordinary benefits that the GPU-job migration mechanism can report to data centers.

Index Terms—CUDA, GPU, virtualization, migration, rCUDA.

1 Introduction

VIRTUALIZATION has become a very important mechanism to increase the efficiency of data centers. Virtualization allows acquisition costs to be better tailored to the real computing needs while reducing energy footprint by consolidating servers. The concept of virtualization can be applied at different levels, as exposed below.

Firstly, the virtualization mechanism can be applied at the computer level, leading to the well known and widely used virtual machine frameworks, which allow several virtual machines to be concurrently executed in a real computer, sharing its resources and hence increasing overall utilization. As a consequence of the widespread use of virtual machines, processor manufacturers incorporate an increasing virtualization support into their products [1].

In the context of virtual machine solutions, virtualization can also be applied at the device level in order to provide support to virtual machines. For instance, some network adapters include virtualization features [2] [3] which allow the adapter to be replicated, at the logical level, so that different replicas of the network card are assigned to different virtual machines. In a similar way, graphics processing units (GPUs) have recently included some virtualization support. For instance, the GRID GPU by NVIDIA [4] can be shared among virtual machines.

In addition to provide support to virtual machines, virtualization of individual devices may also be intended to provide an increased degree of flexibility at the cluster level. For example, networked disks enable sharing a file system across a cluster. In a similar way, the recent remote GPU virtualization technique, implemented in frameworks like rCUDA [5], GVirtuS [6], DS-CUDA [7], or FlexDirect by Bitfusion [8], allows GPUs to be logically detached from the node where they are installed thus creating a pool of GPUs that can be remotely accessed from any node in the cluster. This provides great flexibility when using GPUs.

The large degree of flexibility provided by the remote GPU virtualization technique can be further increased by allowing the GPUs assigned to a given application to move around in the cluster while the application is in execution. This movement means that the application is initially provided one or more GPUs in one or more nodes of the cluster but, during application execution, the GPU part of the application is transparently migrated to other GPU (or GPUs) elsewhere in the cluster. This migration of the GPU part of an application can provide many different benefits to data centers and other computing facilities.

Probably, the most immediate benefit of migrating the GPU part of an application is to support GPU server consolidation. In this regard, notice that resource utilization in data centers evolves over time, depending on the exact workload applied at every moment. Therefore, at some point in time, the utilization of the GPUs in the cluster may be uneven. That is, some nodes may present high GPU utilization whereas GPUs located in other nodes may be much less utilized. In this scenario it would be useful to consolidate GPU jobs into a smaller number of servers, so that nodes becoming free can be switched off.

Other benefit of GPU-job migration is related to the efficient management of different user priorities in a data center, as it will be shown later in Section 5. Carrying out GPU load balancing across the cluster is also possible.

In this paper we present the implementation of the GPU migration mechanism within the rCUDA remote GPU virtualization middleware. Up to our knowledge, this is the first proposal for a remote GPU virtualization middleware to include migration capabilities in the context of CUDA. Our proposal provides more flexibility to data centers than previous proposals, as revisited in Section 3. Additionally,

• *J.Prades and F.Silla are in Departament d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, Camino de Vera s/n 46020 Valencia, Spain*
E-mail: japraga@gap.upv.es and fsilla@disca.upv.es

we present a thorough performance evaluation of this implementation when applied to different real applications. Three generations of NVIDIA GPUs and two versions of the InfiniBand network fabric are used in this performance analysis. This analysis is the main contribution of this paper with respect to [9], which showed a non-mature yet implementation of the migration mechanism within rCUDA as well as a naive performance analysis. Notice that the proposal in this paper is not only useful for cloud infrastructures but it can also be applied to applications running in bare metal.

The paper is organized as follows. Section 2 presents a brief revision of the rCUDA middleware. Next, Section 3 provides a review about how the GPU migration mechanism has been implemented within different GPU virtualization frameworks whereas Section 4 presents how migration is implemented in the context of the rCUDA middleware. Section 5 presents a thorough performance analysis of using the migration mechanism within the rCUDA middleware. Finally, Section 6 concludes this work.

2 About Remote GPU Virtualization

Several software-based GPU sharing solutions have been developed in the context of CUDA during the recent years. All of them aim to offer the same API as the NVIDIA CUDA Runtime API does. Figure 1 depicts the architecture usually deployed by these GPU virtualization frameworks, which follow a distributed client-server approach. The client part is installed in the cluster node executing the accelerated application whereas the server side runs in the node owning the actual GPU. The architecture depicted in Figure 1 is used in the following way: the client middleware receives a CUDA request from the application and forwards it to the server middleware. In the server side, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client side, which forwards them to the accelerated application.

Among the several remote GPU virtualization solutions, we focus on rCUDA (remote CUDA), which supports version 9.2 of CUDA, being binary compatible with it, what means that CUDA programs do not need to be modified in order to use rCUDA. Furthermore, it implements the entire CUDA API (except for graphics functions and NVIDIA's Unified Virtual Memory (UVM), which is partially supported). rCUDA provides specific support for different interconnects. Currently, two modules are available: one intended for TCP/IP compatible networks, and another one specifically designed for the InfiniBand and RoCE interconnects, which make use of RDMA. Furthermore, security and isolation among applications sharing a given rCUDA server is achieved by creating a new GPU context for each of the client applications arriving at the server. In this way, different applications cannot see each other and, in case one of the clients die, the GPU contexts for the other clients can safely continue execution. Compared to other publicly available remote GPU virtualization frameworks developed in academia, the rCUDA middleware provides the best performance [10]. In this regard, the rCUDA middleware achieves near to native performance [11] [12] [13]. Also, contrary to commercial solutions such as FlexDirect, rCUDA provides support for a wide scope of applications.

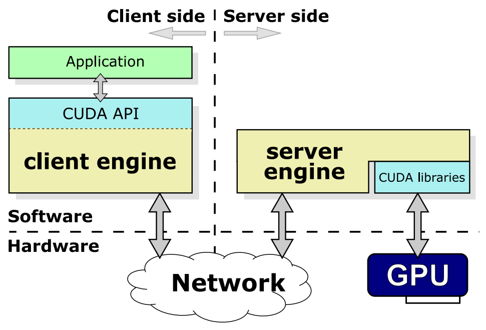


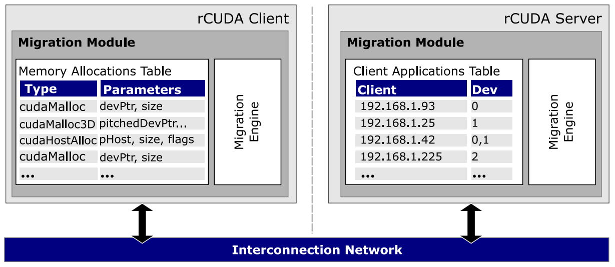
Fig. 1. General organization of remote GPU virtualization frameworks.

3 Related Work on GPU Migration

Migrating GPUs has been addressed in the past in very few works, although none of them was proposed in the context of CUDA. One of these works is presented in [14]. This proposal, intended for OpenCL instead of CUDA, is implemented within the VOCL remote GPU virtualization framework. In this proposal, every time a memory allocation OpenCL function is called, it is intercepted and all the necessary information about the reserved memory areas (starting address, length, etc) is recorded, so that it can be later used for migration purposes. Furthermore, this framework requires that kernels running in the GPU are completed before migration begins. Moreover, a couple of functions are provided in order to trigger migration from the executing application, thus requiring the source code of applications to be modified. On the contrary, in our proposal, migration is not triggered by the application (source code is not modified) but by an external signal. This signal, in the form of a TCP/IP connection to the rCUDA server, is originated at the job scheduler, for instance.

Recent implementations of GPU live migration can be found in NVIDIA's GRID [4] and Intel's GPUs [15], which allow the whole virtual machine (including both its CPU part as well as its GPU part) to be migrated between nodes in a cluster. However, contrary to our proposal, these technologies do not decouple GPUs from CPUs but they are tied together and must be migrated at the same time, thus not allowing the benefits provided by our proposal, such as GPU server consolidation, GPU load balancing, efficient management of user priorities, etc. Furthermore, these solutions require the usage of virtual machines to work whereas our proposal can migrate GPUs regardless of using virtual machines or bare metal.

The techniques to implement GPU migration and GPU checkpointing are similar. Thus, it is worth to also consider works on GPU checkpointing. In this regard, for instance, in [16] a prototype of a checkpointing framework, named CheCUDA, is presented. It only supports a fraction of the functions within the old CUDA 7.0. In order to know which are the GPU memory areas to be included in the checkpoint, CheCUDA provides a set of wrappers to some of the basic cuMemAlloc functions in the Driver and Runtime APIs.



Memory Allocations Table

Type	Parameters
cudaMalloc	devPtr, size
cudaMalloc3D	pitchedDevPtr...
cudaHostAlloc	pHost, size, flags
cudaMalloc	devPtr, size
...	...

Client Applications Table

Client	Dev
192.168.1.93	0
192.168.1.25	1
192.168.1.42	0,1
192.168.1.225	2
...	...

Fig. 2. Migration modules inside rCUDA client and server.

However, this solution does not support multi-threaded applications neither applications using several GPUs. Another proposal is described in [17], where a non-mature hybrid checkpointing technology intended to support checkpointing a running GPU kernel at any time during its execution is presented. The proposal is transparent to the programmer (no source code modification is required), although it is based on the debug interface of CUDA, therefore forcing kernels to run in synchronization mode and causing a large execution overhead. One more proposal, described in [18], supports UVM.

Finally, a proposal for checkpointing, named gHA, is described in [19] for Intel GPUs. gHA does not need any modification of the application source code. Also, no modification to the guest driver is required. Furthermore, gHA saves the Intel GPU registers during a kernel execution so that it does not have to wait for the running kernel to be completed.

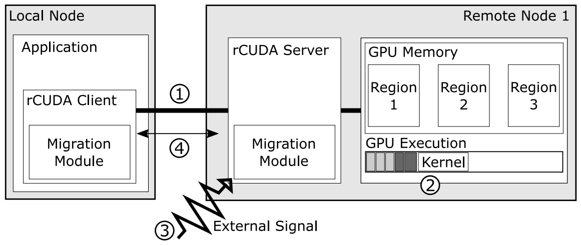
4 Implementing GPU Migration in rCUDA

In this section we present the main details of the implementation of the migration mechanism within the rCUDA middleware as well as its operation.

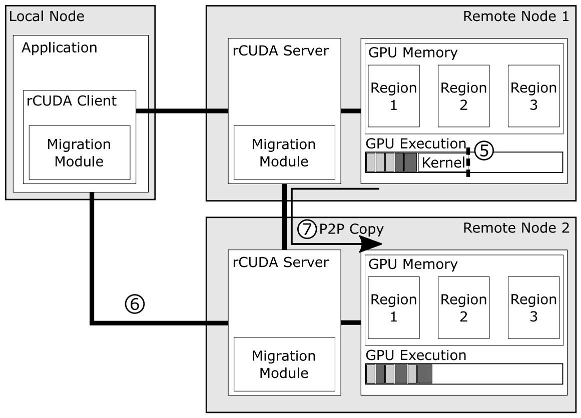
Figure 2 shows the migration module included both in the rCUDA client and in the rCUDA server. These modules comprise a migration engine where the logic that carries out the actual migration process is integrated.

The migration engines at the client and server sides are responsible for storing the necessary information to support migration. In the server side, the migration engine manages the information related to active client applications, which is stored in the Client Applications Table. The migration engine in the rCUDA server is also responsible for handling migration requests and coordinating them with the migration module in the corresponding client. In the client side, the migration engine tracks all memory allocation/deallocation functions. The Memory Allocations Table stores the GPU memory allocation information so that, whenever a migration between GPUs is requested, this information is used to recreate the memory allocations in the new GPU.

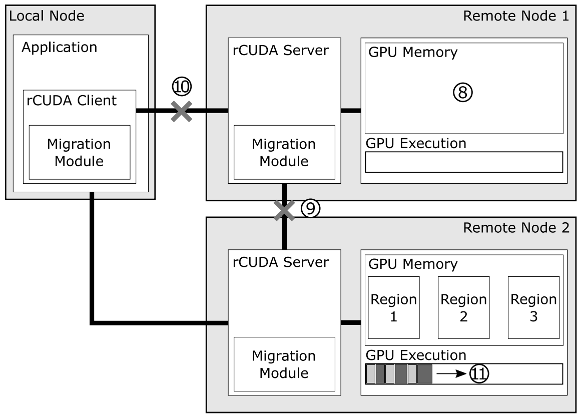
The operation of the migration module within rCUDA is shown in Figure 3. It can be seen in this figure how an accelerated application is migrated using the rCUDA middleware. At step 1 in Figure 3(a), the application starts execution and the connection between the rCUDA client and the rCUDA server is established. Once this initial connection is set up, the application continues its usual execution.



(a) The application starts execution and, at some point in time, the migration signal, triggered by the resource scheduler, arrives at the rCUDA server.



(b) The memory is copied from source GPU to destination GPU in another node of the cluster.



(c) Resources at the initial rCUDA server are released and execution continues in the new GPU.

Fig. 3. Complete operation of the migration module implemented within the rCUDA middleware.

In this particular example, as it can be seen in step 2, the application performs three memory allocations (light gray boxes in the ”GPU execution” queue) followed by 2 copies from host to device (dark gray boxes) in order to fill memory regions 1 and 2 previously allocated in the GPU memory. Finally, the application launches a kernel, which will operate with the data located in regions 1 and 2. This kernel will store the results in region 3. Notice that the information about these three memory allocations was stored in the Memory Allocations Table of the client migration module when the associated CUDA calls were intercepted at the client node. Some time later, during the execution of the aforementioned kernel, an external signal (coming from a resource scheduler, for instance) arrives at the server migration module, as shown in step 3. This external signal is a TCP connection and has associated the necessary information to carry out the migration: client identifier as well as source and destination GPUs. Finally, in step 4, the migration request will be communicated to the migration module in the corresponding client.

Figure 3(b) shows the core of the migration process. Once the migration request arrives at the rCUDA server and it is communicated to the client, a synchronization is performed in the source GPU, waiting for kernels to complete. In our example we can see in step 5 how the migration modules have to wait for the completion of the kernel being executed. Next, in step 6, a new connection between the rCUDA client and the new rCUDA server will be established. Once this connection is created, data must be copied between both GPUs (source and destination). To that end, for each of the regions stored in the Memory Allocations Table, a memory allocation will be performed in the destination GPU memory (light gray boxes) and afterwards the data for each of the regions is transferred directly from the memory of source GPU to the memory of the destination GPU by using the P2P copy module implemented in rCUDA [11] (dark gray boxes), as shown in step 7. This data copy is performed directly between the source and destination GPUs in case InfiniBand or RoCE are used (leveraging RDMA) whereas an intermediate copy involving the client node is required in case TCP/IP communications is used.

Figure 3(c) shows the final steps of the migration process. Memory regions in the original GPU are released in step 8. Then, the connection used for the P2P copies is destroyed (step 9) as well as the connection between the rCUDA client and the initial rCUDA server (step 10). Finally, the application continues execution in the new GPU (step 11).

There is an important final remark about migrating GPU jobs when different generations of GPUs are involved in the migration process. Notice that when using CUDA, there is no binary compatibility guarantee between GPU applications compiled for different generations of GPUs. That is, an application compiled for Kepler may not run on a Maxwell GPU and vice versa. Therefore, migrating a GPU application between different GPU generations may not be successful due to this lack of compatibility guarantee. Fortunately, the `nvcc` CUDA compiler provides options to generate binaries that can be run on different GPU generations. The `nvcc` compiler follows a compilation model based on two stages. In the first stage, an intermediate representation, called PTX, is generated. Later, in the second stage, it is used to generate the binary code for a specific GPU generation. This binary code can either be generated at compile time or at execution time by using JIT (Just-in-Time) compilation. Each of the options presents pros and cons. If it is generated at runtime, then it will perfectly match the requirements of the GPU that is going to be used. However, some overhead will be introduced by the compilation during the execution of the application. On the other hand, if the binary code is generated at compile time, `nvcc` allows the generation of multiple translations of the same source code targeted for multiple GPU generations. At run time, these multiple translations, which are organized in Fatbinaries, will allow the CUDA driver to select the appropriate binary code based on the actual GPU. In summary, if a GPU binary code can be executed with CUDA in a set composed of several GPU generations (either because it is using JIT or Fatbinaries), then it will be possible to migrate that code with rCUDA among that very same set of GPU generations. The use cases presented in next section are an example of this, given that applications are migrated between Kepler and Pascal GPUs.

5 Performance Evaluation of GPU Migration with rCUDA

This section presents a performance study of our implementation of the GPU-job migration mechanism within the rCUDA middleware. We consider three different scenarios for this performance evaluation. In the first scenario, addressed in Section 5.1, a synthetic application will be used. In the second scenario, thoroughly introduced in Section 5.2, we consider real applications for the migration experiments. Finally, in Section 5.3 we leverage a series of use cases in order to exemplify the usefulness of migrating GPU jobs among cluster nodes.

The testbed used in all these analyses consists of a cluster of 1027GR-TRF Supermicro nodes which include one FDR and one EDR InfiniBand network adapters, which provide 56 Gbps and 100 Gbps, respectively. Moreover, they include three different generations of GPUs: an NVIDIA K20 GPU, an NVIDIA K40 GPU and an NVIDIA P100 device. Using these three different GPU models will allow us to better exercise the migration mechanism in this section.

5.1 Synthetic Application

Migrating a job among two GPUs located in different cluster nodes requires two different types of actions, both of them contributing to the migration overhead. First, every memory region allocated by the application in the source GPU has to be copied to the destination GPU. Second, it is required to properly manage these copies.

Regarding the first type of actions, the movement of the data in each region from the source GPU to the destination device consumes most of the migration time. This time depends not only on the exact network fabric used but it also depends on the exact size of the memory region to be moved, given that the maximum bandwidth attained by a network fabric is only achieved for data transfers beyond a minimum threshold. For instance, in the case of copying data with rCUDA among GPUs located in different cluster nodes, the maximum performance is achieved when data transfers are larger than 10 MB [11].

On the other hand, the time required for managing the data copies cannot be neglected. In this regard, the connection between rCUDA servers must be first established in order to later use P2P copies. Afterwards, for every memory region to be copied from source to destination GPUs, a call to a CUDA memory allocation function has to be carried out in the destination GPU prior to copying the data of that region from the source GPU. Additionally, once the data of that region has been copied, a CUDA memory deallocation function has to be executed in the source GPU. Calls to CUDA memory allocation/deallocation functions require some time to be executed and, therefore, the more memory regions the application allocated in the source GPU, the longer it will take to manage the migration process, as it will be shown later.

In order to understand the impact on performance of each of the parameters involved in the migration of GPU jobs, in this section we leverage a synthetic application so that different parameters can be controlled in an isolated way. The synthetic application implemented for this study takes as input parameters the total amount of GPU memory regions and the size of each region. Then, by using these input parameters, the application allocates *n* equally sized memory regions in the GPU. Although this application is extremely simple, using it in this first scenario will allow us to understand the behavior of the migration process.

Regarding the network fabrics used in the experiments with the synthetic application, we have considered 6 different network throughputs in order to shed light to the performance results. First, we have leveraged FDR and EDR InfiniBand network fabrics, which make use of PCIe 3.0 x8 and PCIe 3.0 x16, respectively. Additionally, we have modified the PCIe settings in the testbed systems so that these network adapters were also used with PCIe 2.0 and PCIe 1.0 configurations. These additional configurations are intended to reduce network performance. The exact throughput of each of these configurations is shown in Figure 4 for transfer sizes ranging from 2 bytes up to 8 MB. It can be seen in this figure that we are considering effective transfer bandwidths ranging from 13.2 Gbps (FDR PCIe 1.0) up to 92 Gbps (EDR PCIe 3.0). Also, performance of EDR PCIe 1.0 and FDR PCIe 2.0 are almost identical.

Results obtained with the synthetic application are shown in Figure 5. For all the experiments depicted in this figure, a P100 GPU has been used. We have selected this GPU because it supports PCIe 3.0 x16, which provides a bandwidth equal to or larger than all the network configurations considered. In this way, the limiting factor in these experiments will be the exact network fabric configuration. Figure 5 also displays the performance of the migration process when the 1 Gbps Ethernet network is used. Notice that results for this network are presented only for comparison purposes, given that its low performance makes this network not to be an option for virtualizing GPUs among cluster nodes in production data centers.

It can be seen in Figure 5(a) that the amount of time required by the migration process directly depends on the amount of data to be migrated. In this figure, the synthetic application has been configured to allocate only one memory region. Therefore, only one call to the `cudaMalloc` function is carried out in the destination GPU. It can be seen in Figure 5(a) that total migration time has been split into copy time and "Other" time. Copy time refers to the time required to move the data from the original memory region in the source GPU to the newly allocated memory region in the destination GPU. "Other" time refers to the time required to manage the migration process (creation and destruction of the connection for P2P copies and calls to the `cudaMalloc` and `cudaFree` functions and other management tasks associated with the migration process in the particular implementation within rCUDA).

Figure 5(a) shows that the bandwidth attained by the underlying network directly impacts the performance of the migration process, as expected. It is worth noticing

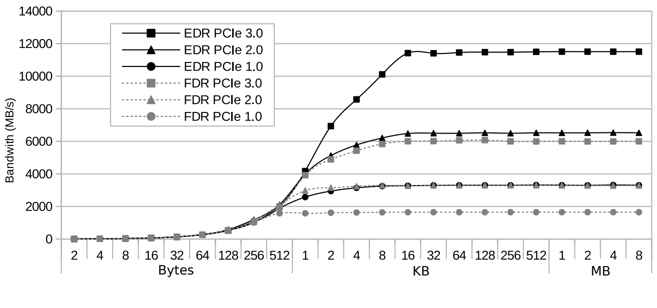
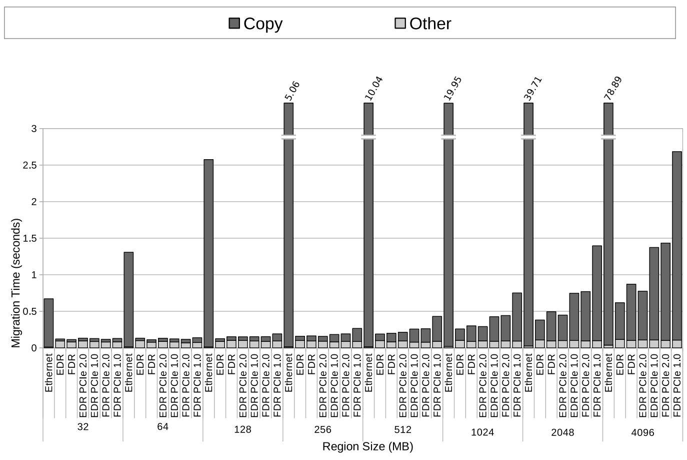
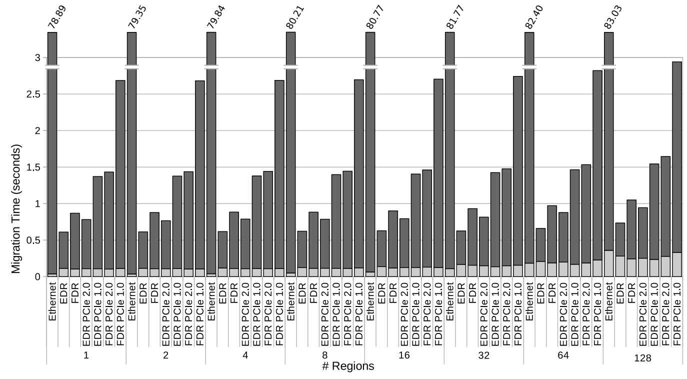


Fig. 4. Bandwidth attained for several network configurations using different transfer sizes.



(a) A single memory region is allocated in the GPU. Different region sizes are considered (from 32 MB up to 4 GB).



(b) Multiple memory regions are allocated in the GPU, accounting for a total of 4 GB GPU memory in all cases.

Fig. 5. Time required to migrate a job among two P100 GPUs located in different nodes. A synthetic application is leveraged. Several configurations of the FDR and EDR InfiniBand network adapters are used. Performance for the 1 Gbps Ethernet network is also displayed.

TABLE 1
Amount of seconds required for management tasks in Figure 5(b).

	1	2	4	8	16	32	64	128
Eth	0.04	0.04	0.04	0.05	0.06	0.11	0.18	0.36
FDR	0.10	0.11	0.11	0.11	0.12	0.16	0.19	0.24
EDR	0.11	0.11	0.12	0.12	0.13	0.17	0.20	0.28

TABLE 2
Characterization of the real applications used to analyze the migration mechanism.

Application	Execution Time (s)	rCUDA overhead (%)	# Kernels	Kernel Time (ms)			GPU Memory (Mbytes)			GPU Utilization (%)		
				Avg	Max	Min	Avg	Max	Min	Avg	Max	Min
GPUBLAST	134	2.14	3	14400	15600	12200	1207.4	1302	72	32.5	100	0
CUDASW++	15	-2.21	1	11500	11500	11500	762.5	931	72	70	100	0
TeaLeaf	156	9.81	1048557	0.03	0.11	0.0027	182.47	183	72	19	33	0
CUDA-MEME	213	7.4	2107	37.51	46. 29	22.38	157.63	162	72	38.4	69	0
CloverLeaf	271	3.35	405489	0.68	6.22	0.0027	1496.74	1502	72	97	99	0

that a speed up of about 128x is attained in the case of EDR InfiniBand with respect to 1 Gbps Ethernet although difference in maximum bandwidth among both network fabrics is only 92x (1 Gbps bandwidth in the case of Ethernet versus 92 Gbps of effective bandwidth in the case of EDR InfiniBand). In a similar way, in the case of FDR InfiniBand, a speed up of about 91x is achieved although the theoretical speed up should be about 48x (FDR InfiniBand provides 48 Gbps of effective bandwidth). The reason for achieving a speed up much larger than the theoretical one is that when using InfiniBand networks we can directly copy data from the source GPU to the destination GPU by making use of the RDMA features included in these adapters whereas data transfers using the 1 Gbps Ethernet network require an intermediate copy because the RDMA feature is not present in the Ethernet adapters.

On the other hand, it is interesting to notice that the time for "Other" is noticeably larger for InfiniBand than for 1 Gbps Ethernet. The reason for these larger times is that the time "Other" when using InfiniBand includes the time for creating and destroying the TCP connections to the remote servers required to control data movement using RDMA. These TCP connections are not needed for 1 Gbps Ethernet as the RDMA feature is not available.

Figure 5(b) shows the impact on performance when varying the amount of memory regions that hold the data of a fixed size 4 GB memory area to be migrated. It can be seen that, for each of the network configurations considered, copy time remains almost constant regardless of the amount of memory regions. The reason is that even for the smallest region size, which is 32 MB when there are 128 regions, attained data transfer bandwidth is the maximum one because the size of data to be transferred is larger than 10 MB. On the contrary, time required for the migration management purposes (bar section "Other") increases as the amount of memory regions to migrate increases.

Table 1 shows the exact values for "Other" for the three main network fabrics. It can be seen in the table that management time increases as the amount of memory regions increases. Management times for FDR and EDR InfiniBand networks are similar. It is also noteworthy the fact that management times for 1 Gbps Ethernet are lower than for InfiniBand (due to the creation and destruction of the TCP connections as described before). However, as the number of migrated memory regions increases, the time required for migration management purposes increases more significantly when using 1 Gbps Ethernet. This is due to the worst latency of this network. The larger the number of regions to be migrated, the higher the number of memory allocation/deallocation calls. These calls do not include too much data (they simply notify the remote GPU) so they are very sensitive to the latency features of the network.

5.2 Real Applications

In this section we perform a study of the migration mechanism when it is applied to five different real applications. The applications are GPUBLAST [20], CUDASW++ [21], CloverLeaf [22], TeaLeaf [23] and CUDA-MEME [24]. Table 2 characterizes these applications. Data in this table has been gathered during the execution of the applications when using a remote K20 GPU with rCUDA along with FDR InfiniBand. Table 2 shows that the GPUBLAST application requires up to 1302 MB of GPU memory during its execution, which lasts for almost 134 seconds. Additionally, this application consists of 3 long running kernels that make a full usage of the GPU resources while in execution (see Figure 8). Average GPU utilization for the GPUBLAST application is about 33%. Similar data is presented for the other applications considered in this section. Furthermore, Table 2 shows the overhead introduced by rCUDA with respect to the execution using a local K20 with CUDA.

Figure 6 presents additional information about the memory usage of these applications. In addition to show the GPU memory allocated by each of the applications, Figure 6 also displays the amount of memory regions allocated by each of them. In this regard, it can be seen that the GPUBLAST application allocates 8 different memory regions. This very same amount of regions is allocated by the CUDA-MEME application. On the contrary, CloverLeaf and TeaLeaf allocate a much larger number of memory regions. They allocate, respectively, 47 and 35 regions. Finally, the CUDASW++ application only allocates 3 memory regions.

Regarding the total amount of memory used by each of the applications, it can be seen, if comparing numbers in Figure 6 with numbers in Table 2, that values for memory

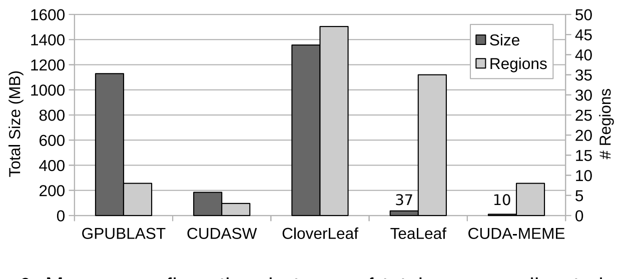


Fig. 6. Memory configuration, in terms of total memory allocated and number of memory regions, for each of the applications considered.

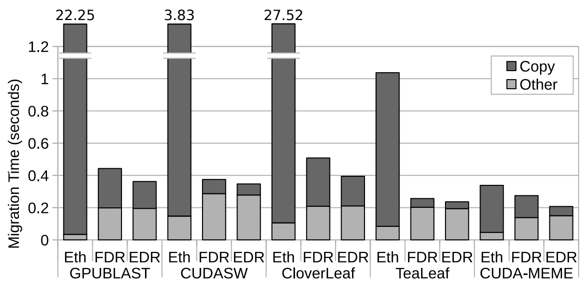


Fig. 7. Service downtime for each of the applications considered. Migration was triggered at 25% execution time for each of the applications.

usage seem not to match. The reason for the mismatch is that numbers in Figure 6 were gathered according to the information collected when intercepting the CUDA memory allocation calls with rCUDA. However, numbers in Table 2 were gathered by using the `nvidia-smi` application, which provides overall memory usage in the GPU, among many other parameters. In this regard, numbers in Figure 6 represent the exact amount of memory allocated by the application in the GPU. On the contrary, numbers in Table 2 represent total memory used in the GPU, which includes, for instance, the memory required to store the application context. Notice that this memory for the application context is allocated by the NVIDIA driver and not by the application. Therefore, when migrating the application, the memory used for the GPU context will not have to be moved to the destination GPU but a new context will be created in that GPU. After creating the new context in the destination GPU, all memory regions will be copied. In summary, memory sizes shown in Figure 6 can be seen as the amount of memory that has to be migrated among GPUs. That is, these memory regions, and memory sizes, are the only ones migrated in the experiments in this section, shown in Figure 7, for instance.

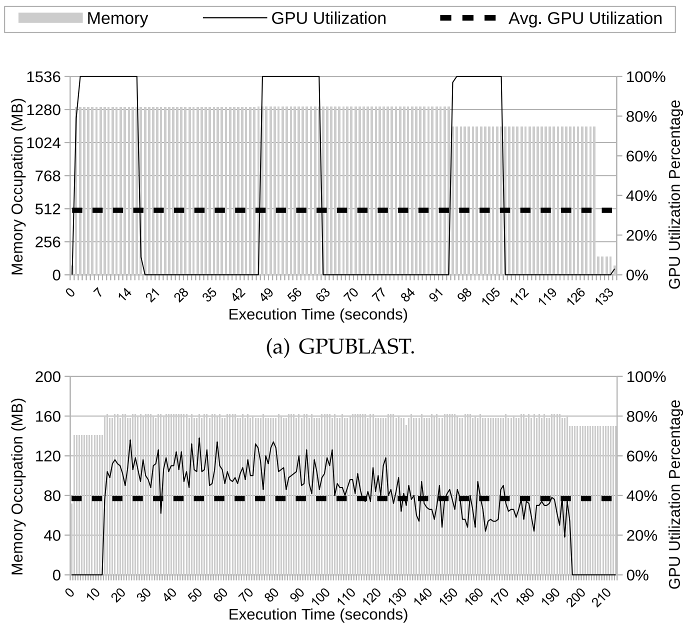
In order to measure migration time, an important concern is related to the exact moment when migration is triggered. Remember that after receiving the external signal triggering migration, kernels in execution in the GPU must be completed before beginning the migration process. In this manner, migrating an accelerated application among GPUs can be seen as a two step process where step 1 is just waiting for kernel completion and step 2 is moving data among GPUs. The first step has to do with kernels in execution at the time when the external signal triggering migration arrives whereas the second step has to do with the memory allocated in the GPU by the application.

It is important to notice that the time required for step 2 (moving data among GPUs) only depends on data size, amount of memory regions and underlying network fabric, as analyzed in previous section. However, the time required for step 1 (waiting for kernel completion) depends on the exact state of the execution of the application when the external signal arrives. As a consequence, we can differentiate among "total migration time" and "service downtime". The latter refers to how much time the GPU is out-of-service once migration begins after kernel completion. The former refers to the time required to restart the execution of the application in the target GPU since the arrival of the migration signal. Obviously, service downtime will always be less than or equal to total migration time given that total migration time includes service downtime plus the time waiting for kernel completion in the source GPU.

Regarding service downtime, notice that this amount of time is the overhead that the migrated application suffers due to the migration itself and it is independent of the execution time of kernels in the GPU. On the other hand, total migration time is the amount of time observed by the job scheduler when it triggers the signal to migrate the GPU part of an application.

In order to perform a thorough analysis of service downtime for the applications under consideration, we triggered migration at three different points for each of the applications. These points were 25%, 50% and 75% of their execution time. Furthermore, as execution time of applications using remote GPUs depends on the exact network fabric used, these three points in time will thus depend on which network was leveraged for executing the application. Therefore, in order to analyze migration time for each application, we performed 9 experiments: migrating the application at 25%, 50% and 75% execution time when FDR InfiniBand and K20 GPU were used, migrating the application at 25%, 50% and 75% execution time when EDR InfiniBand and K40 GPU were used and, finally, migrating the application at the aforementioned execution time percentages when 1 Gbps Ethernet and K20 GPU were used. Notice that the exact points in time for each of the execution percentages vary depending on network fabric and GPU used.

Figure 7 shows the service downtime for each of the applications considered in our study. The three main network fabrics previously used in Section 5.1 are also employed in this figure. Remember that times in Figure 7 are the overhead experienced by the migrated applications. In order to



(a) GPUBLAST.

(b) CUDA-MEME.

Fig. 8. Evolution of memory occupancy and GPU utilization during execution time of two of the applications considered in this study. Average GPU utilization for each of the applications is also shown.

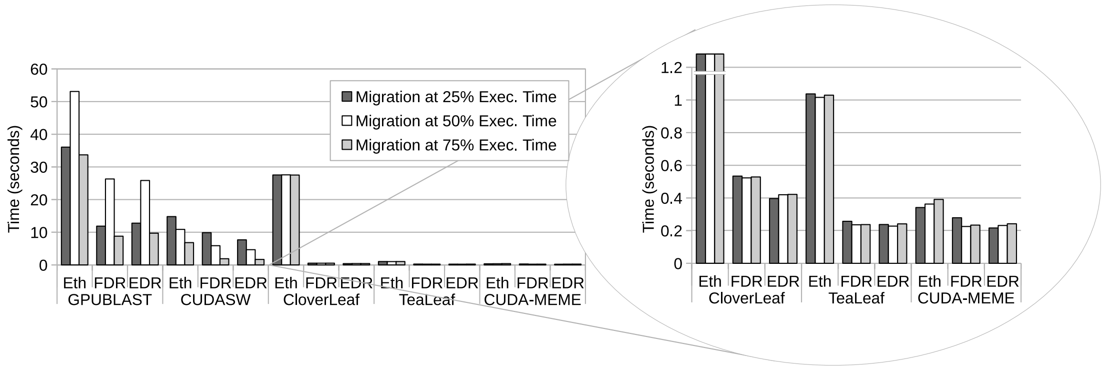


Fig. 9. Total migration time for the five applications considered in this study. Time is measured since the arrival of the external signal triggering migration until the application resumes execution in the destination GPU.

gather the numbers in Figure 7, migration was triggered at 25% execution time. Results when migration was triggered at 50% and 75% execution times were almost the same. This fact points out that these applications have allocated very similar memory regions in the three points mentioned above, as it is shown in Figure 8 for two of the applications.

Regarding the results displayed in Figure 7, it can be clearly seen the impact on service downtime of the amount of data to migrate (shown in Figure 6). In this regard, the GPUBLAST, CUDASW++ and CloverLeaf applications present very different service downtimes depending on the exact network fabric used: the low performance of 1 Gbps Ethernet causes that service downtime is much larger than when InfiniBand is used. It can also be clearly seen that EDR InfiniBand reports smaller service downtime than FDR InfiniBand due to its much larger bandwidth. Nevertheless, it is important to remark that service downtime is very small (less than 0.5 seconds) when InfiniBand is used regardless of the exact version of this interconnect. On the other hand, when the amount of data to be migrated is very small, as it is the case for the CUDA-MEME application, service downtime is similar for both 1 Gbps Ethernet and InfiniBand.

Regarding the results for the CUDA-MEME application shown in Figure 7, there is an interesting issue regarding copy time. If transfer time is carefully analyzed (0.0921 seconds for FDR InfiniBand and 0.0563 seconds for EDR InfiniBand), it can be derived that transfer time is much larger than it should be, according to the bandwidth available in these networks. The reason for this higher transfer time is that this application allocates memory by using CUDA array memory instead of the regular memory. Transferring data allocated as a CUDA array with rCUDA is not as optimized as transferring regular data due to the geometry of the allocation. Therefore, a lower bandwidth is attained for these copies.

Figure 7 also shows the time required for managing the migration (bar section "Other"). For the CUDASW++ application, which only allocates 3 memory regions, management time is larger than for other applications with a much larger amount of regions, such as CloverLeaf or TeaLeaf. In order to explain this result, we analyzed the source code of the CUDASW++ application and found that this application makes use of host page-locked memory regions allocated with `cudaMallocHost` or `cudaHostAlloc` functions (in addition to the GPU memory regions). This type of regions need a special management given that not only GPU memory has to be migrated but also some host memory. The time required for managing these regions is accounted within the time required for managing the migration.

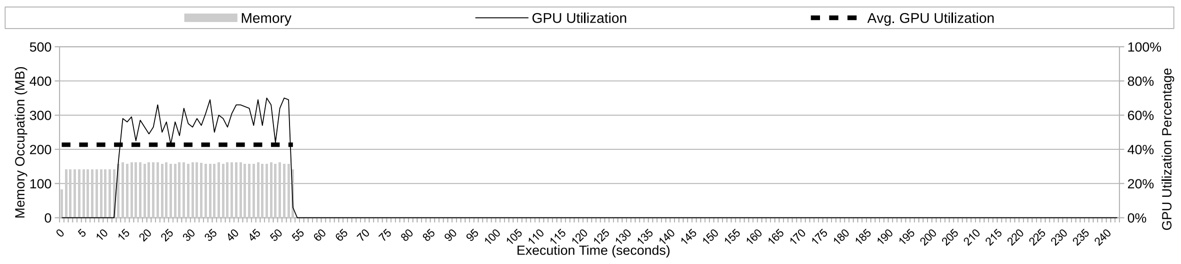
Finally, Figure 9 shows the delay between the arrival of the signal triggering migration until the application resumes execution in the destination GPU. This is total migration time. In this way, times displayed in Figure 9 include the waiting time until kernels in execution when the migration signal arrives are completed as well as the time to transfer the data from the source to the destination GPU. The figure displays the total migration times when migrating the applications at 25%, 50% and 75% of their execution times.

Two main conclusions can be derived from Figure 9. The first one is that total migration time greatly depends on the exact state of the application when the migration signal arrives. This can be clearly seen for the GPUBLAST application. The second conclusion that can be derived from Figure 9 is that when an application executes a large amount of small kernels (as it was shown in Table 2 for the CloverLeaf, TeaLeaf and CUDA-MEME applications) then the waiting time for kernel completion is noticeably reduced and thus total migration time is decreased, as it is shown in the right side of the figure.

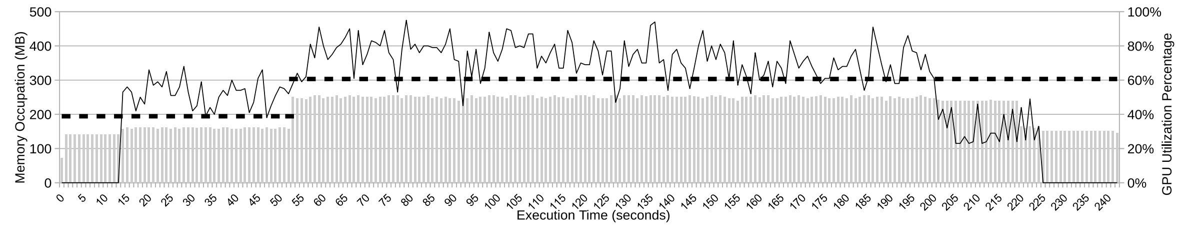
5.3 Use Cases for GPU-Job Migration with rCUDA

In the previous sections the performance of GPU migration was analyzed by using both synthetic and real applications. In this section we provide several use cases that show the usefulness of the GPU-job migration mechanism. Real applications will be used in this section.

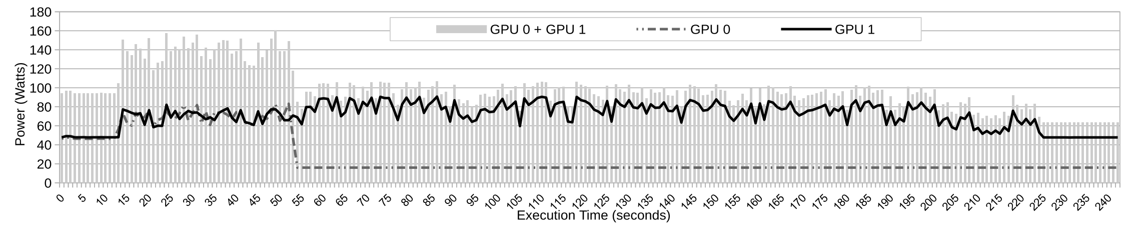
In order to provide the reader with the right context for these use cases, it is important to understand that we envision GPU-job migration as a powerful tool that can be used by job schedulers to improve different metrics in the cluster. One of these metrics could be minimizing overall energy consumption, for instance. Another metric could be reducing application execution time. Furthermore, the job scheduler could deal with different user priorities. In this way, higher priority users should be provided better service whereas lower priority users may experience some delays depending on workload evolution. At the bottom stage of the priority stack, users with the lowest priority could just



(a) GPU memory occupancy and GPU utilization in a server running the CUDA-MEME application. At time 55 seconds the GPU job is migrated to another server, shown in Figure 10(b), and thus the GPU is emptied.



(b) Server running the CUDA-MEME application. At time 55 seconds the migrated application from Figure 10(a) enters the GPU in this server. From that moment, the GPU in this server executes both applications concurrently.



(c) Power consumption of the two GPUs involved in the consolidation process. "GPU 0" is the source GPU whereas "GPU 1" is the destination GPU of the migration. Additionally, "GPU 1" is the GPU where both GPU-jobs are consolidated.

Fig. 10. GPU migration used to consolidate servers. Two instances of the CUDA-MEME application are being executed in two servers and, at some point in time, the job scheduler decides to migrate one of the jobs to the other server. The emptied server can be later switched off if required.

benefit from spare GPU cycles. That is, their jobs would execute as far as no other jobs belonging to higher priority users are present in the system. As soon as higher priority jobs enter the system, the lowest priority jobs should be preempted if required.

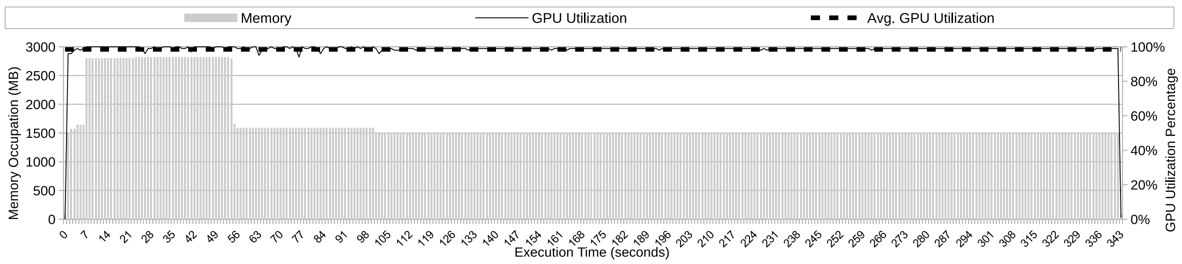
5.3.1 *GPU Server Consolidation*

The first of the examples about the usefulness of the GPU-job migration mechanism is devoted to server consolidation. The consolidation technique is specially appealing when several GPU servers in the cluster present low to medium GPU utilization. In this scenario, GPU utilization in those servers could be increased by aggregating jobs from GPUs in different servers into a single GPU. By increasing GPU utilization, a better usage of energy is made. Additionally, if the servers that are emptied are later switched off, then energy efficiency is noticeably increased. It is important to remind that only the GPU part of applications is migrated.

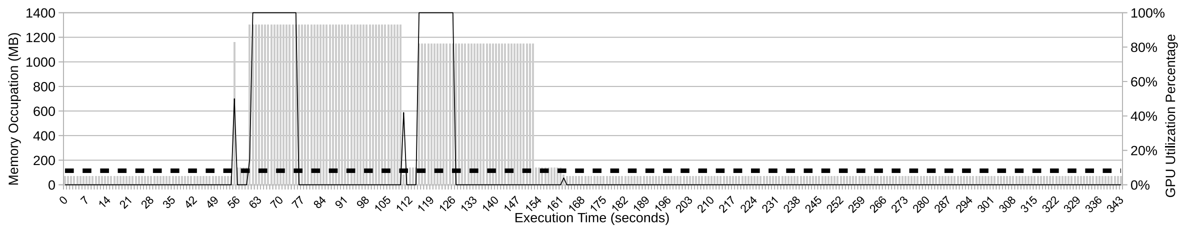
In order to implement this idea, the logic to decide whether to consolidate servers and which should be the GPU-jobs to migrate could be placed into the job scheduler. Additionally, the job scheduler should be enriched in order to gather information about the utilization of the GPUs in the cluster. In this way, once the job scheduler finds out that some of the GPUs in the cluster present a utilization under a given threshold, it could decide to consolidate the GPU-jobs from several servers into a single node, thus making a more efficient use of resources and saving energy.

Figure 10 presents an example of this idea. Two servers (Figures 10(a) and 10(b)) are executing, each of them, an instance of the CUDA-MEME application. The two nodes in Figure 10 are connected by the FDR InfiniBand network and include, each of them, an NVIDIA K20 GPU. As can be seen in Figures 10(a) and 10(b), average GPU utilization in both servers is about 40%. At time 55 seconds, the job scheduler realizes that GPU utilization in both servers is lower than the threshold (the threshold should be decided by the system administrator and could even be a composition of several parameters such as amount of jobs sharing the GPU, historical data about GPU utilization, etc). At that point in time, the job scheduler performs several checks prior to carry out the migration, such as making sure that the candidate destination GPU has enough free memory for holding both applications. Also, the job scheduler could check that aggregated GPU utilization for both applications does not exceed 100%. Once the job scheduler has carried out all the required checks, it migrates the application from the server in Figure 10(a) to the server in Figure 10(b). From that point in time, the server in Figure 10(b) begins executing both applications concurrently.

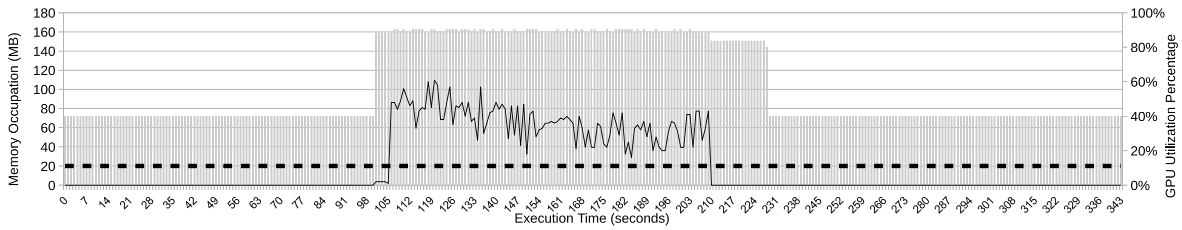
The effect of consolidating both GPU-jobs in the server can be seen in Figure 10(b). First, GPU utilization increases



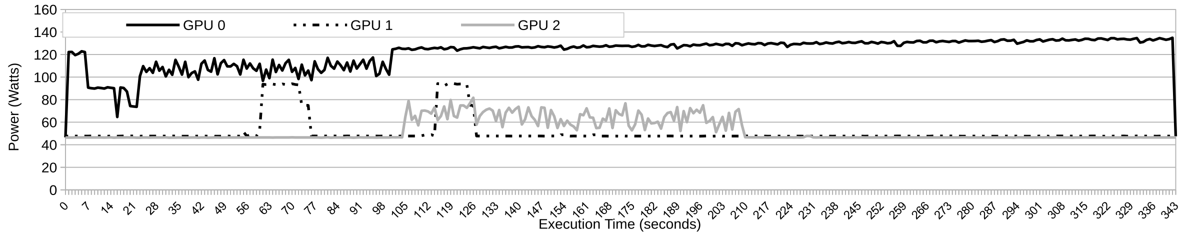
(a) ”GPU 0” is concurrently executing three applications: CloverLeaf, GPUBLAST and CUDA-MEME. At time 50 seconds the GPUBLAST application is migrated to GPU 1 (Figure 11(b)). At time 100 seconds the CUDA-MEME application is migrated to GPU 2 (Figure 11(c)). The CloverLeaf application completes execution in this GPU.



(b) ”GPU 1” receives the GPUBLAST application after migration at time 50 seconds.



(c) ”GPU 2” receives the CUDA-MEME application after migration at time 100 seconds.



(d) Evolution of the power consumption of the three GPUs involved in the load balancing example.

Fig. 11. Example of applying the GPU-job migration mechanism within rCUDA in order to balance the load among GPUs in the cluster.

from 40% up to 60%. Theoretically, it should have increased up to 80%. However, the dynamics of applications is not so straightforward. Second, the execution of both applications is slightly lengthened. In this regard, Table 2 showed that a single instance of CUDA-MEME lasts for about 210 seconds. However, due to server consolidation, the concurrent execution of both applications lasts for about 240 seconds.

An alternative point of view about this consolidation process is presented in Figure 10(c). This figure depicts the power consumed by each of the two GPUs during the execution of the applications. It can be seen in the figure that until second 55 both GPUs are active and consume between 60 and 80 watts. Gray bars display the aggregated power consumption of both GPUs, which can reach up to 140 watts. At second 55 migration occurs. At that point in time, the power consumption of GPU 1 (the server that receives the migrated job) slightly increases whereas the power consumption of the GPU sourcing the migration is noticeably reduced because it remains idle. The net result is a clear reduction in the power required to execute both applications, as can be seen by the gray bars in the figure. Furthermore, notice that Figure 10(c) only depicts power consumed by the GPUs. If we take into account the rest of the system, one can easily understand the large benefits that could be achieved if the node sourcing the migration is completely switched off.

5.3.2 GPU Load Balancing

As part of the natural evolution of the workload in a data center, it could happen that, at a given point in time, a GPU is noticeably overloaded whereas other GPUs in the cluster remain idle. This situation could be desirable if the policy in the data center is to consolidate servers as much as possible, as it was reviewed in the previous section. However, other policies are feasible. For instance, the system administrator could decide to balance load among servers as much as possible in order to provide customers with execution times as low as possible. Contrary to consolidation, load balancing may not save energy. But customers might be more satisfied.

With rCUDA it is possible to balance the load of the GPUs in the cluster thanks to its migration mechanism. Actually, when several applications share a given GPU in the cluster, the migration implementation carried out within the rCUDA middleware allows to migrate each of the GPU jobs of these applications to different destination GPUs. That is, it is not required that GPUs are migrated as a whole but individual GPU jobs can be managed independently from each other. This individual migration of GPU jobs allows that load is balanced across the GPUs in the cluster.

In this section we present an example of load balancing with rCUDA. Figure 11(a) shows a K20 GPU that is concurrently shared by three applications: CloverLeaf, GPUBLAST and CUDA-MEME. Sharing the GPU among these three applications means that all of them are executed slower than if they were executed in different GPUs. Let us assume that the policy in the cluster is to provide execution times as low as possible. Thus, the job scheduler would decide to look for idle GPUs across the cluster in order to balance load. We can see in Figures 11(a) and 11(b) that at time 50 seconds the job scheduler has found an idle GPU and thus it has migrated the GPUBLAST application to that GPU. Now in the original GPU there are only two applications: CloverLeaf and CUDA-MEME. Some time later, at second 100, a GPU in the cluster becomes idle and thus the job scheduler decides to migrate the GPU part of the CUDA-MEME application to that GPU. This can be seen in Figures 11(a) and 11(c). The overall result is that the load of the three GPUs has been balanced and the execution of the three applications has been perfectly adapted to the resources available at every moment. In this way, application execution time has been reduced as much as the circumstances allowed. Notice that in Figures 11(b) and 11(c) there is a memory occupancy of 70 MB by default. This memory occupancy is due to the CUDA context of the rCUDA daemon.

Figure 11(d) presents a similar point of view of the previous process although from a power consumption perspective. The evolution of the power consumption of the three GPUs is presented. ”GPU 0” refers to the initial GPU shared among the three applications. ”GPU 1” refers to the GPU where the GPUBLAST application is migrated to. Finally, ”GPU 2” refers to the GPU that receives the CUDA-MEME application. It can be seen in this figure that ”GPU 1” and ”GPU 2” remain idle until they receive the GPUBLAST and CUDA-MEME applications, respectively.

5.3.3 Improved Management of User Priorities

In the context of a cluster where a job scheduler deals with users having different priorities, a way to provide better service to higher priority users is to assign them the best GPUs in the cluster. However, due to the evolution of the cluster workload, it may be possible that by the time that a job from a high priority user must be placed into execution, all the powerful GPUs are already assigned to other high priority users. As a consequence, the job that is to be executed must finally use a regular GPU.

Once the job from that high priority user has entered execution in a regular GPU, it may eventually happen that some of the best GPUs become idle because they complete the execution of their jobs. At that moment, it could be possible to migrate the job that was in execution in a regular GPU so that it continues execution in one of the powerful GPUs in the cluster. This migration would satisfy the priority criteria of the cluster whereas execution time of that job would be reduced because of the better GPU.

The opposite scenario could also be possible. That is, an application is being executed in a powerful GPU but, during its execution, a higher priority user submits a job to the scheduler queues. As a consequence, the job scheduler looks for a suitable GPU to execute the higher priority job. However, it discovers that all the powerful GPUs are already in use. In this context, the job from the lower priority user can be moved out from the powerful GPU to a regular device in order to complete its execution. After moving the job out from the powerful GPU, the high priority job would enter the powerful GPU and start execution. The net result would be that the priority policy in the cluster is fulfilled at the cost of slowing down the lower priority job.

TABLE 3
Execution time of the CloverLeaf application in different GPUs.

Application	Execution Time (s)		Execution Time (s)	
	K20	P100	K20 to P100	P100 to K20
CloverLeaf	271	80	150	227

Table 3 presents the execution time of the CloverLeaf application in the previous scenarios. First, Table 3 shows that this application requires 271 seconds to be executed in a K20 GPU whereas it needs 80 seconds to be completed in a P100 GPU. On the other hand, when this application is migrated from a K20 GPU to a P100 one after 33% of its execution time, we obtain a total execution time of 150 seconds. In the opposite scenario (from P100 to K20) we obtain 227 seconds.

6 Conclusions

This paper has presented a thorough performance analysis of the migration support implemented within the rCUDA remote GPU virtualization middleware. Although providing this kind of support within GPU virtualization frameworks is not novel, the implementation carried out for the rCUDA middleware presents a better overall architecture, which is carefully devised to be integrated with job schedulers at different levels, as it has been widely shown in the performance evaluation section. In this regard, contrary to the rest of implementations of the GPU migration mechanism in other GPU virtualization frameworks, in the rCUDA implementation it is the job scheduler the one that triggers the migration process as well as the one that selects the destination GPU, according to the scheduling and energy efficiency policies decided by the system administrator. Additionally, the GPU migration implementation presented in this paper is the only one existing for GPU virtualization solutions supporting modern CUDA versions.

Performance results show that migration is feasible and its overhead is very low when the InfiniBand network is used in the cluster. Similar extraordinary performance results are expected for other network fabrics that also provide RDMA capabilities, such as the RoCE interconnect. Furthermore, the use cases shown in this paper clearly demonstrate that GPU-job migration is a powerful tool that can be used by the job scheduler in order to optimize the execution of accelerated applications in a cluster. In this regard, it is noteworthy that GPU-job migration provides job schedulers with an increased freedom degree when they carry out the scheduling process of accelerated applications. The reason is that, thanks to the GPU-job migration mechanism, job schedulers do not have to know, during the scheduling process, the exact amount of GPU memory used by the application being scheduled. In this way, job schedulers can assign GPUs to applications regardless of their GPU-memory footprint and, if they later experience GPU memory allocation problems due to lack of memory, the GPU jobs can be migrated to another GPU presenting more available memory. Furthermore, it could even be possible to store the GPU job in main memory in case no GPU is found with enough memory. This would stall the accelerated application until the required memory is available.

Acknowledgments

This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/77. Authors are grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] A. A. Semnanian, J. Pham, B. Englert, and X. Wu, "Virtualization technology and its impact on computer hardware architecture," in *2011 Eighth International Conference on Information Technology: New Generations*, 2011.

[2] Mellanox, "Connectx-3 vpi single and dual qsfp+ port adapter card user manual," http://www.mellanox.com/, 2013, accessed 27 September 2018.

[3] "Intel ethernet server adapter i350," http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-i350-server-adapter-brief.html, accessed 27 September 2018.

[4] "Nvidia grid accelerated virtual desktops and apps," http://images.nvidia.com/content/grid/pdf/188270-NVIDIA-GRID-Datasheet-NV-US-FNL-Web.pdf, accessed 27 Sept 2018.

[5] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and remote gpus perform similar with edr 100g infiniband," in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, 2015.

[6] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A gpgpu transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing*, 2010.

[7] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Ds-cuda: A middleware to use many gpus in the cloud environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[8] bitfusion, "The elastic ai infrastructure for multi-cloud," https://bitfusion.io/, 2019, accessed 27 March 2019.

[9] J. Prades and F. Silla, "Turning gpus into floating devices over the cluster: the beauty of gpu migration," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017.

[10] C. Reaño and F. Silla, "A performance comparison of cuda remote gpu virtualization frameworks," in *2015 IEEE International Conference on Cluster Computing*, 2015.

[11] ——, "On the support of inter-node p2p gpu memory copies in rcuda," *Journal of Parallel and Distributed Computing*, 2019.

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

[13] J. Prades, B. Varghese, C. Reaño, and F. Silla, "Multi-tenant virtual gpus for optimising performance of a financial risk application," *Journal of Parallel and Distributed Computing*, 2017.

[14] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng, "Transparent accelerator migration in a virtualized gpu environment," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012.

[15] J. Ma, X. Zheng, Y. Dong, W. Li, Z. Qi, B. He, and H. Guan, "gmig: Efficient gpu live migration optimized by software dirty page for full virtualization," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018.

[16] T. Suzuki, A. Nukada, and S. Matsuoka, "Transparent checkpoint and restart technology for cuda applications," in *GPU Technology Conference (GTC)*, 20156.

[17] L. Shi, H. Chen, and T. Li, "Hybrid cpu/gpu checkpoint for gpu-based heterogeneous systems," in *Parallel Computational Fluid Dynamics*, K. Li, Z. Xiao, Y. Wang, J. Du, and K. Li, Eds., 2014.

[18] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "Crum: Checkpoint-restart support for cuda's unified memory," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[19] Z. Zhang, X. Xu, M. Xue, J. Wang, Z. Qi, and Y. Dong, "gha: An efficient and iterative checkpointing mechanism for virtualized gpus," in *APSys*, 2016.

[20] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, 2010.

[21] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," *BMC Bioinformatics*, 2013.

[22] M. Martineau and S. McIntosh-Smith, "The arch project: physics mini-apps for algorithmic exploration and evaluating programming environments on hpc architectures," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[23] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, 2016.

[24] Y. Liu, B. Schmidt, W. Liu, and D. L. Maskell, "Cudameme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units," *Pattern Recognition Letters*, 2010.