On the Support of Inter-node P2P GPU Memory Copies in rCUDA

Carlos Reaño[a], Federico Silla[a]

[a] *Universitat Politècnica de València, Spain.*

Abstract

Although GPUs are being widely adopted in order to noticeably reduce the execution time of many applications, their use presents several side effects such as an increased acquisition cost of the cluster nodes or an increased overall energy consumption. To address these concerns, GPU virtualization frameworks could be used. These frameworks allow accelerated applications to transparently use GPUs located in cluster nodes other than the one executing the program. Furthermore, these frameworks aim to offer the same API as the NVIDIA CUDA Runtime API does, although different frameworks provide different degree of support. In general, and because of the complexity of implementing an efficient mechanism, none of the existing frameworks provides support for memory copies between remote GPUs located in different nodes.

In this paper we introduce an efficient mechanism devised for addressing the support for this kind of memory copies among GPUs located in different cluster nodes. Several options are explored and analyzed, such as the use of the GPUDirect RDMA mechanism. We focus our discussion on the rCUDA remote GPU virtualization framework. Results show that is possible to implement this kind of memory copies in such an efficient way that performance is even improved with respect to the original performance attained by CUDA when GPUs located in the same cluster node are leveraged.

Keywords: CUDA, Virtualization, GPUDirect RDMA

Email addresses: carregon@gap.upv.es (Carlos Reaño), fsilla@upv.es (Federico Silla)

1. Introduction

GPUs (Graphics Processing Units) are used in many data centers in order to accelerate the execution of applications from areas as different as Big Data [1], computational algebra [2], chemical physics [3], finance [4], image analysis [5], or biology [6], among others. The existence of libraries and programming models such as CUDA (Compute Unified Device Architecture) [7] has noticeably supported the introduction of GPUs into data centers. However, despite the remarkable reductions in application execution time enabled by GPUs, their usage is not exempt from several concerns. For instance, acquisition cost of cluster nodes containing GPUs is noticeably increased. Furthermore, this side effect is exacerbated by a usually low GPU utilization, which is the result of applications not being able to keep these accelerators busy all the time. Notice, however, that GPUs still consume energy even when idle, thus increasing overall power requirements.

In order to alleviate the concerns related to the use of GPUs, these devices might be virtualized so that they are seamlessly shared among several concurrent applications. Additionally, in order to provide a higher degree of flexibility, GPU virtualization frameworks may be designed in such a way that they provide applications access to GPUs located at nodes other than the one where the application is being executed. This is the case of remote GPU virtualization frameworks, which detach GPUs from nodes, in a logical way, thus creating a pool of GPUs that can be used from any node of the cluster, as shown in Figure 1. Nodes owning GPUs become GPU servers whereas nodes that execute applications become clients. This logical configuration is so flexible that it also allows to execute GPU-accelerated applications in nodes owning GPUs so that they become clients from other GPU servers. Frameworks such as DS-CUDA [8], gVirtuS [9], GViM [10], and rCUDA [11] implement this idea of remote GPU virtualization.

Remote GPU virtualization frameworks provide applications with transparent access to GPUs located at other cluster nodes. In this regard, these frame-

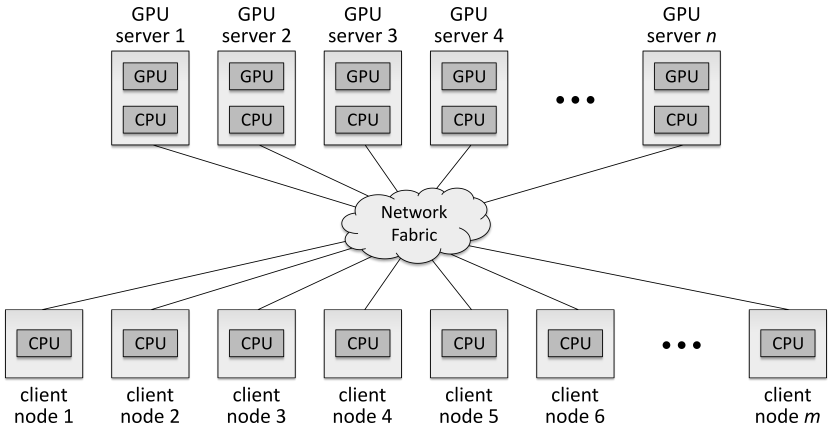
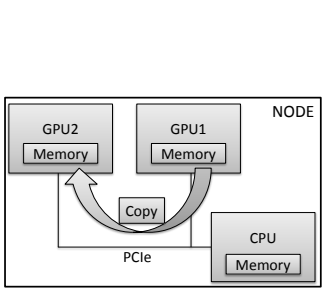


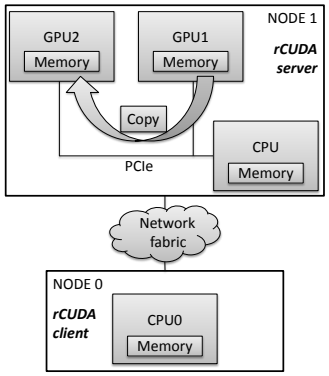
Figure 1: Logical configuration of a cluster making use of remote GPU virtualization.

works provide the same API (Application Programming Interface) and behavior than CUDA, despite of using a remote GPU instead of a local one. In particular, these frameworks are designed to offer the same API as the NVIDIA Runtime API [7] does. Furthermore, all remote GPU virtualization frameworks are still under construction and therefore their support for CUDA is improved over time. This is, for instance, the case for the rCUDA framework, which provides support for most of the CUDA functions in the Runtime and Driver APIs but still lacks support for peer-to-peer (P2P) memory copies between remote GPUs located in different cluster nodes. Notice that this support is also missing in the rest of GPU virtualization frameworks currently available (Section 3 provides a revision of the available frameworks), being the reason for this lack of support the complexity for addressing it in an efficient way.

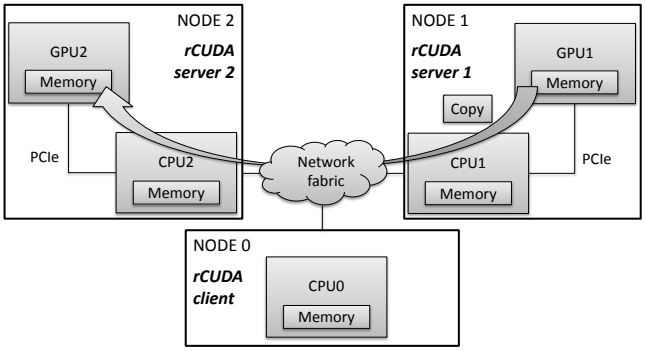
In order to better introduce the idea around P2P memory copies, Figure 2 presents the possible scenarios when carrying out this kind of memory copies with CUDA and rCUDA. As we can see, with CUDA there is only one possible scenario, depicted in Figure 2(a), where the GPUs are located in the same node and are interconnected by the PCIe link. On the contrary, when using rCUDA there are two possible scenarios for performing copies between remote GPUs: (i) the remote GPUs are located in the same remote node and are interconnected



(a) CUDA scenario.



(b) rCUDA scenario 1: remote GPUs in the same server node.



(c) rCUDA scenario 2: remote GPUs in different server nodes.

Figure 2: Possible scenarios when carrying out P2P memory copies with CUDA and rCUDA.

by the PCIe link as shown in Figure 2(b), and (ii) the remote GPUs are located in different remote nodes and therefore they are interconnected by the network fabric, as depicted in Figure 2(c).

Prior to this work, rCUDA already supported the first scenario exposed in Figure 2(b). In this manner, it was possible to carry out memory copies between remote GPUs located in the same server node. However, the second scenario presented in Figure 2(c) was not supported. In this regard, although the scenario depicted in Figure 2(c) may not be strictly required for many use cases, having the flexibility to use any of the GPUs in the cluster regardless of their exact

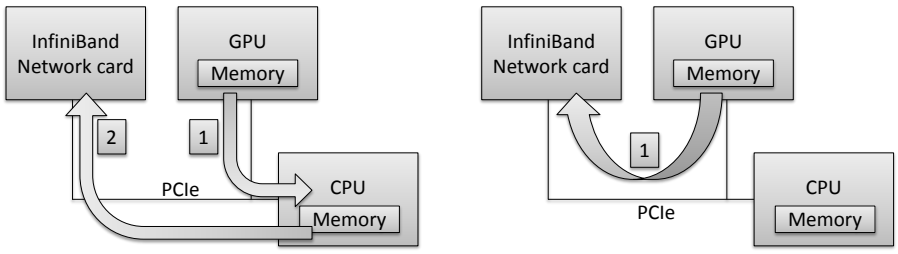
location provides a large improvement in overall performance, as shown in [12]. Additionally, remote GPU virtualization frameworks aim to provide the same semantics as CUDA does. Therefore, providing support within the virtualization framework for P2P memory copies between remote GPUs located in different nodes is mandatory.

In this paper we explore different options to implement memory copies between remote GPUs located in different nodes of the cluster. We use the rCUDA middleware as case study in this exploration because it is the most outperforming both in periodic updates and also regarding throughput [13], as it will be shown in Section 3. In this way, researching on different mechanisms to efficiently implement this support is the major contribution of this work, where we show how we have adapted rCUDA to accomplish this purpose. To the best of our knowledge, this is the first analysis on this kind of memory copies, given that none of the existing GPU virtualization frameworks provides support for P2P memory copies between remote GPUs located in different nodes.

The rest of the paper is organized as follows. Section 2 presents previous work, mainly related to the GPUDirect RDMA mechanism implemented by NVIDIA. Section 3 provides the reader with the required background on the GPU virtualization mechanism and frameworks. Later, Section 4 addresses the main goal of this work, where we explore different options to carry out memory copies between remote GPUs located in different cluster nodes. A performance comparison is carried out among the several implementation options considered in this study. A synthetic benchmark is leveraged to that end. Next, Section 5 provides a performance evaluation of the implemented P2P copies within the rCUDA middleware using a real application. Finally, Section 7 summarizes the main conclusions of this paper.

2. Related Work on P2P Memory Copies

In order to efficiently copy data between the memory of GPUs located in different nodes of the cluster, NVIDIA introduced GPUDirect RDMA [14] in



(a) Without NVIDIA GPUDirect RDMA. (b) With NVIDIA GPUDirect RDMA.

Figure 3: Scenarios with and without NVIDIA GPUDirect RDMA used with an InfiniBand network adapter.

2012. It is a technology that enables, by using standard features of the PCIe link, a direct path for data exchange between the GPU and a third-party peer device, such as an InfiniBand network adapter (see Figure 3). Support for GPUDirect RDMA was introduced by Mellanox into its InfiniBand network adapters [15] in order to provide high-speed InfiniBand networking for GPU-to-GPU communications.

Exploring the use of GPUDirect RDMA for InfiniBand networks has become very popular since it appeared. In this manner, a lot of researchers have attempted to use this technology for improving performance in different scenarios. One such example is the work by Hamidouche et al. in [16], which investigates the use of GPUDirect RDMA for improving the communication operations of OpenSHMEM, a Partitioned Global Address Space (PGAS) programming model. Another such example can be found in the work by Younge et al. in [17], where GPUDirect RDMA is one of the technologies used for running high performance molecular dynamics simulations in virtualized environments.

Other researchers have also presented improvements in the field of Message Passing Interface (MPI) communication libraries. For instance, Potluri et al. in [18] increase the efficiency of the inter-node MPI communication library MVAPICH2 [19] by using GPUDirect RDMA.

In addition to all the previous works referred above, an extensive perfor-

mance analysis of NVIDA GPUDirect RDMA over InfiniBand [20] shows the real capabilities of this technology when used in modern server platforms. Regarding bandwidth, the study concludes that GPUDirect RDMA is faster than a staging approach[1] for message sizes up to 400-500KB. For larger data sizes, staging to/from host memory and moving data via InfiniBand using regular RDMA probably provides better performance.

With respect to latency, according to the authors of the study, GPUDirect RDMA provides low latency, usually below 2μs, which is better than staging to/from host memory and moving data via InfiniBand using regular RDMA. In this regard, the authors state that using intermediate buffers requires either synchronous (`cudaMemcpy`) or asynchronous (`cudaMemcpyAsync`) memory copies, which take around 8μs and 9μs, respectively. Additionally, we have to add the InfiniBand host-to-host latency: 1.3μs. Thus, the expected GPU-to-GPU latency would be: 8μs (`cudaMemcpy` from GPU 1 to host 1) + 1.3μs (copy from host 1 to host 2) + 8μs (`cudaMemcpy` from host 2 to GPU 2). This is clearly larger than the latency provided by GPUDirect RDMA.

3. Background on GPU Virtualization

The remote GPU virtualization technique allows an accelerated application to be executed in cluster nodes which do not own a GPU. In this manner, GPUs located in other nodes of the cluster are assigned to the application. Note that the application is not aware of using a remote GPU. In fact, the application source code does not need to be modified because the GPU virtualization middleware transparently manages the access to remote GPUs. Additionally, remote GPUs can be concurrently shared among several accelerated applications.

During the last years, different remote GPU virtualization frameworks have

[1] A staging approach comprises copying from the source GPU memory to host memory; then copying from host memory to remote host memory using regular RDMA, instead of GPUDirect RDMA; and finally, in the remote node, copying from host memory to the destination GPU memory.

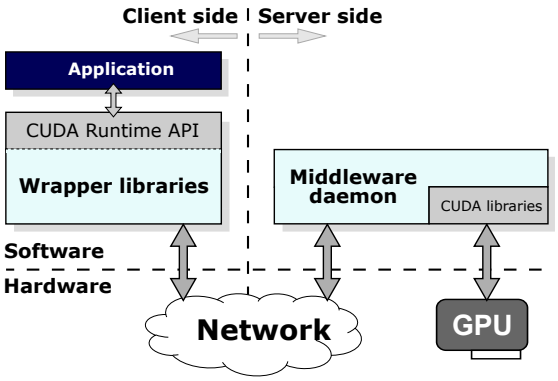


Figure 4: Client-server architecture typically used in remote GPU virtualization solutions.

been developed for CUDA, such as DS-CUDA [8], gVirtuS [9], GViM [10], or rCUDA [11]. All of these frameworks feature a distributed client-server architecture, as shown in Figure 4. The client node executes the accelerated application whereas the server node owns the GPU and provides GPU services to client nodes. A typical cluster configuration making use of the remote GPU virtualization technique may comprise several server nodes, each of them containing one or more GPUs. Nodes in the cluster not owning GPUs would become clients of those servers.

All the referred remote GPU virtualization solutions are based on forwarding the calls to CUDA functions to the remote server. This is why they are also known as API forwarding solutions. In these solutions, when the application performs a call to one of the CUDA functions, such function, along with its parameters, is intercepted and sent to the remote server owning the GPU. Once received at the server side, the call is executed in the GPU and the result is sent back to the client side of the middleware, which delivers it to the application.

Different remote GPU virtualization frameworks provide different degree of support. For example, DS-CUDA supports CUDA 4.1, gVirtuS supports CUDA 2.3, GVIM supports CUDA 1.1, and rCUDA supports CUDA 9.1. Additionally, only DS-CUDA and rCUDA feature an efficient communication layer among clients and servers based on the InfiniBand Verbs API [21]. This API provides

RDMA support for data transfers. Unfortunately, support for InfiniBand within DS-CUDA is quite limited, what constrains the usability of this framework. Finally, among the publicly available remote GPU virtualization solutions, rCUDA is the one that provides the best performance. The reader may refer to [13] for a thorough comparison among DS-CUDA, gVirtuS, and rCUDA.

In addition to API forwarding solutions such as rCUDA, other options also exist, like GPU full virtualization and GPU para virtualization. Contrary to API forwarding, these other options virtualize the GPU at the driver level. That is, a custom GPU driver must be provided to the application in order to make use fo the GPU. The main difference among these two virtualization mechanisms is that the para virtualization approach makes some changes to that custom driver so that performance is improved whereas the full virtualization option fully emulates the GPU thus not being necessary any modification to the driver. In the particular use case of GPUs, both full and para virtualization techniques face two major concerns. The first one is that information about native GPU drivers, as the one provided by NVIDIA, is not publicly available due to commercial issues. Therefore, creating a custom driver that provides the same features as the native one is extremely costly or even impossible due to the lack of documentation. Furthermore, GPU vendors keep including new features with every new GPU generation, what makes it extremely difficult to keep these custom drivers updated. As a consequence, existing solutions for GPU full and para virtualization provide custom drivers that implement a subset of the functions of the native driver [22]. The second major concern faced by these GPU virtualization approaches is performance. As shown in [22], even optimized versions of these mechanisms introduce a non-negligible overhead that hinders their usage in production data centers. The reason for these large overheads has to do with the fact that these GPU full and para virtualization mechanisms virtualize the GPU at a very low level (the driver), contrary to what the API forwarding techniques do, which virtualize GPUs at a very high abstraction level. Thus, for every API call performed by the application at the high level, typically many calls to the driver (at the low level) are carried

out. This increased amount of calls to the virtualization framework noticeably increases performance. Finally, notice that GPU full and para virtualization techniques could be enhanced to use remote GPUs, as it is the case for rCUDA. However, this would noticeably increase their overhead even more. This is why current solutions for GPU full and para virtualization do not consider the use of remote GPUs.

4. Implementing Efficient P2P Memory Copies within rCUDA

This section presents the main contribution of this work, which is the implementation of the memory copy mechanism devised for supporting memory copies between remote GPUs located in different nodes of the cluster. Several options are explored, such as the GPUDirect RDMA technique previously detailed.

Performance is evaluated and compared for each of the considered options. To that end, the setup used for the experiments reported in this section consists of three 1027GR-TRF Supermicro servers connected by an SX6025 InfiniBand switch (FDR). Each of the servers has two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz with 32 GB of DDR3 SDRAM memory at 1.6 GHz. They also include one Mellanox ConnectX-3 single-port InfiniBand adapter and one NVIDIA Tesla K20m GPU. The CentOS 6.4 operating system with Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools) and CUDA 7.5 with NVIDIA driver 352.39 are used.

The testbed servers used in our experiments are NUMA machines and therefore NUMA effects matter for the results shown in this paper. For this reason, both the NVIDIA GPU and the InfiniBand adapter are attached to the same processor socket (processor 0). Additionally, memory buffers and processes are bound to this processor in the experiments.

In addition, and for comparison purposes, when performing P2P memory copies with CUDA, executions have been carried out using a 7047GR-TRF Supermicro server, with similar characteristics to those of the 1027GR-TRF

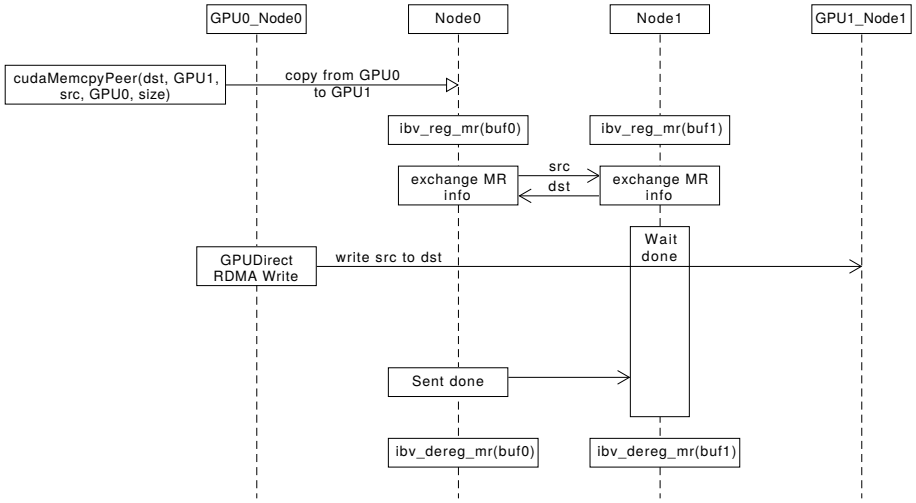


Figure 5: Sequence diagram of rCUDA version 1 for memory copies between different remote GPUs. This version uses GPUDirect RDMA to transfer the data.

servers mentioned before, but with two NVIDIA Tesla K20m GPUs (note that memory copies with CUDA must be carried out within the same node).

4.1. Version 1: Using GPUDirect RDMA

The first approach considered to copy data between remote GPUs located in different nodes of the cluster is based on the use of GPUDirect RDMA. Figure 5 presents a sequence diagram of the proposed solution. When a request for copying memory between GPUs is received (i.e., `cudaMemcpyPeer`), the following steps are followed:

1. InfiniBand memory regions (MRs) associated with the GPU memory addresses to be copied are registered in the nodes hosting the GPUs
2. Information related to the registered MRs is exchanged between nodes
3. Data is copied from the source GPU memory to the destination one
4. When the data copy has finished, the MRs are unregistered

Figure 6 presents the bandwidth obtained for different transfer sizes when using this version (labeled as *rCUDA P2Pv1*) for copying data between remote

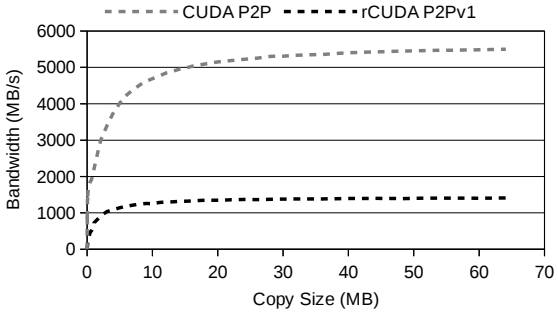


Figure 6: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA and version 1 of rCUDA are shown.

GPUs. For comparison purposes, the bandwidth obtained by CUDA when transferring data between local GPUs is also depicted (labeled as *CUDA*). As we can observe, the maximum bandwidth obtained with this version of rCUDA, over 1.40GB/s, is very low. The reason is that, for each copy, we must set-up the connection (i.e., register the MRs and exchange information with the remote node). This set-up introduces a high overhead, which turns into very low performance in this test.

4.2. Version 2: pre-allocating intermediate buffers

To improve version 1 previously explained, next we present a new version which pre-allocates intermediate buffers at initialization to avoid the overhead introduced by registering and exchanging MRs information for each data copy. Figure 7 presents a sequence diagram of this approach. When a request for copying memory between GPUs is received, the following steps are followed:

1. New GPU memory buffers are allocated and InfiniBand memory regions (MRs) are registered. Information related to the registered MRs is exchanged between nodes involved in the copy. This is done only once at initialization. These buffers will be reused until the application finishes
2. Data is copied from the source GPU memory to the local intermediate buffer just allocated in the GPU in the previous step

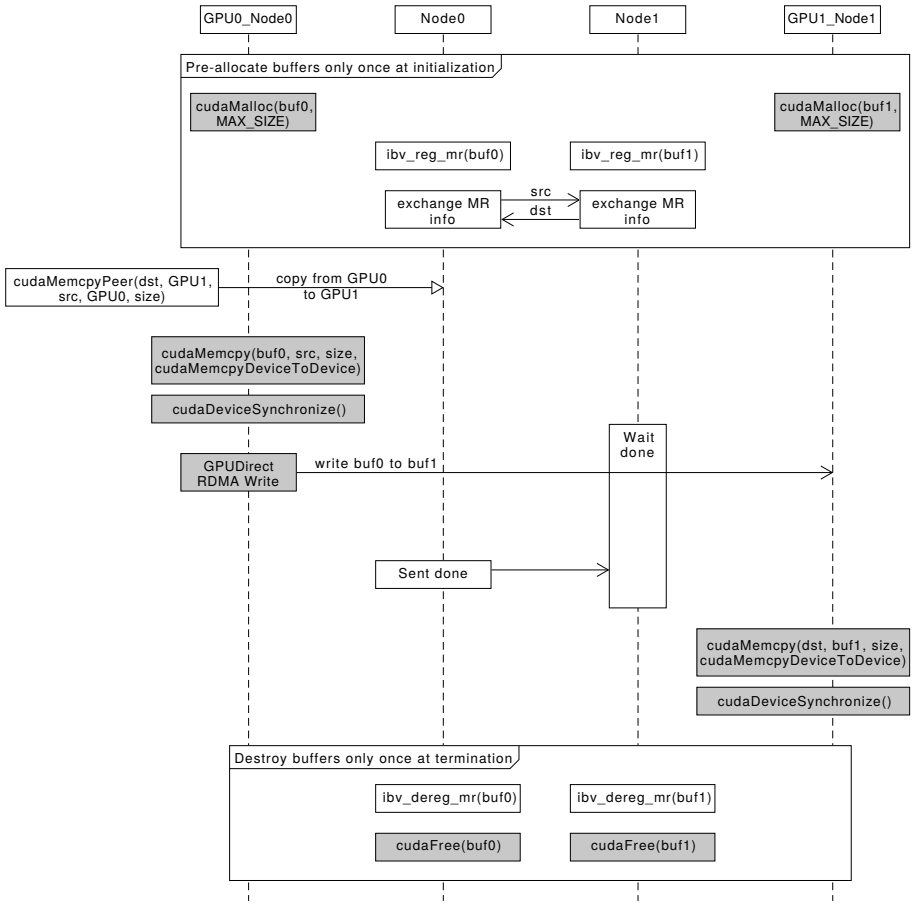


Figure 7: Sequence diagram of rCUDA version 2A for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2B in Figure 8. This version uses GPUDirect RDMA to transfer data, and pre-allocates intermediate buffers at initialization.

3. Data is copied from the local intermediate buffer to the remote intermediate buffer at the remote GPU by using GPUDirect RDMA
4. Data is copied from the remote intermediate buffer to the destination GPU memory

The question that arises now is the following: given that we are using intermediate buffers, and taking into account that RDMA transfers between host

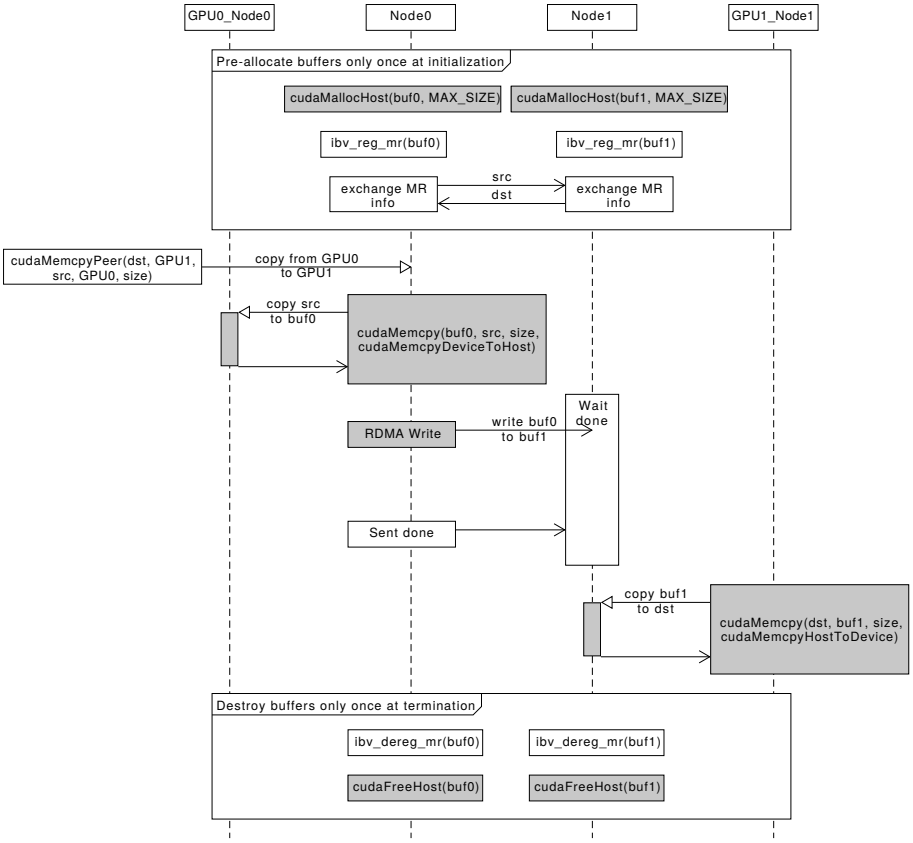


Figure 8: Sequence diagram of rCUDA version 2B for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2A in Figure 7. This version uses regular RDMA to transfer data and then copy it to GPU memory. Intermediate buffers are pre-allocated at initialization.

memory attains higher bandwidth than with GPU memory, would it be a better choice to use host intermediate buffers instead of the GPU intermediate buffers used in this version?

To answer this question we have implemented a new version, similar to the previous one, but using host intermediate buffers. From now on, versions using GPU intermediate buffers and GPUDirect RDMA will be labeled as the version number plus the letter A, while versions using host intermediate buffers (and

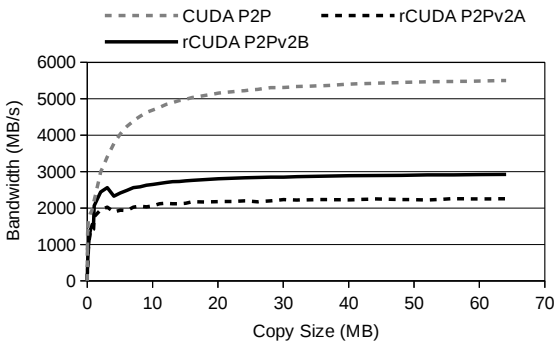


Figure 9: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA and versions 2A and 2B of rCUDA are shown.

not using GPUDirect RDMA) will be referred to as the version number followed by the letter *B*. This is why we referred to the version 2 previously explained as version *2A* in Figure 7.

Figure 8 presents a sequence diagram of the new proposed version 2B using host intermediate buffers. Differences with respect version 2A are highlighted in gray for clarity. As it can be observed, the two versions are similar, the only difference being the intermediate buffers: in version 2A they are allocated using GPU memory, while in version 2B they are allocated using host memory.

Figure 9 presents the bandwidth obtained when using the new versions. Bandwidth significantly improves, achieving a maximum of 2.25GB/s when GPU buffers are used, and a maximum of 2.90GB/s when using host buffers.

Using this approach improves performance. However, we must allocate one extra buffer, either in GPU memory or in host memory, in each node involved in the data copy. The size of this buffer must be the biggest possible copy size performed by the application. For instance, in our bandwidth test, the buffer had a size equal to 64MB, the maximum copy size in our test. Of course, this approach is only valid for testing purposes, and cannot be used with real applications because it would require knowing in advance the maximum data size transferred by the application. Additionally, even if this size were known, this approach requires to use a lot of memory for the intermediate buffers.

4.3. Version 3: using multiple intermediate buffers

In order to address the concerns commented in version 2, we next present a new version which, instead of using one large intermediate buffer in each node involved in the copy, uses multiple smaller intermediate buffers. Thus, the whole amount of data to be copied is split into several chunks of the size of the smaller intermediate buffers, and the copy is performed following a pipelined approach, overlapping copies in the different stages:

1. Data chunk *i-1* is copied from the source GPU memory to the local intermediate buffer *i-1*
2. Data chunk i is copied from the local intermediate buffer i to the remote intermediate buffer i
3. Data chunk *i+1* is copied from the remote intermediate buffer *i+1* to the destination GPU memory
4. Steps 1, 2 and 3 are overlapped and repeated until all the data has been copied

Figure 10 and Figure 11 present sequence diagrams of the new proposed versions 3A and 3B, respectively. As commented, version 3A uses GPU intermediate buffers, whereas version 3B uses host intermediate buffers. Again, differences between each version are highlighted in gray for clarity.

Figure 12 presents the bandwidth obtained when using these new versions (labeled as *rCUDA P2Pv3A* and *rCUDA P2Pv3B*, respectively). We can see that bandwidth has considerably increased in both versions, obtaining maximum values around 2.60GB/s and 5.40GB/s for versions 3A and 3B, respectively. In the latter case, version 3B, performance is almost the same as the one obtained by CUDA with local GPUs.

4.4. Version 4: adaptive intermediate buffer size

In the previous version we have used a fixed number of intermediate buffers, all of them with the same size. However, the optimal amount of buffers and the optimal buffer size probably depends on the actual transfer size. To analyze

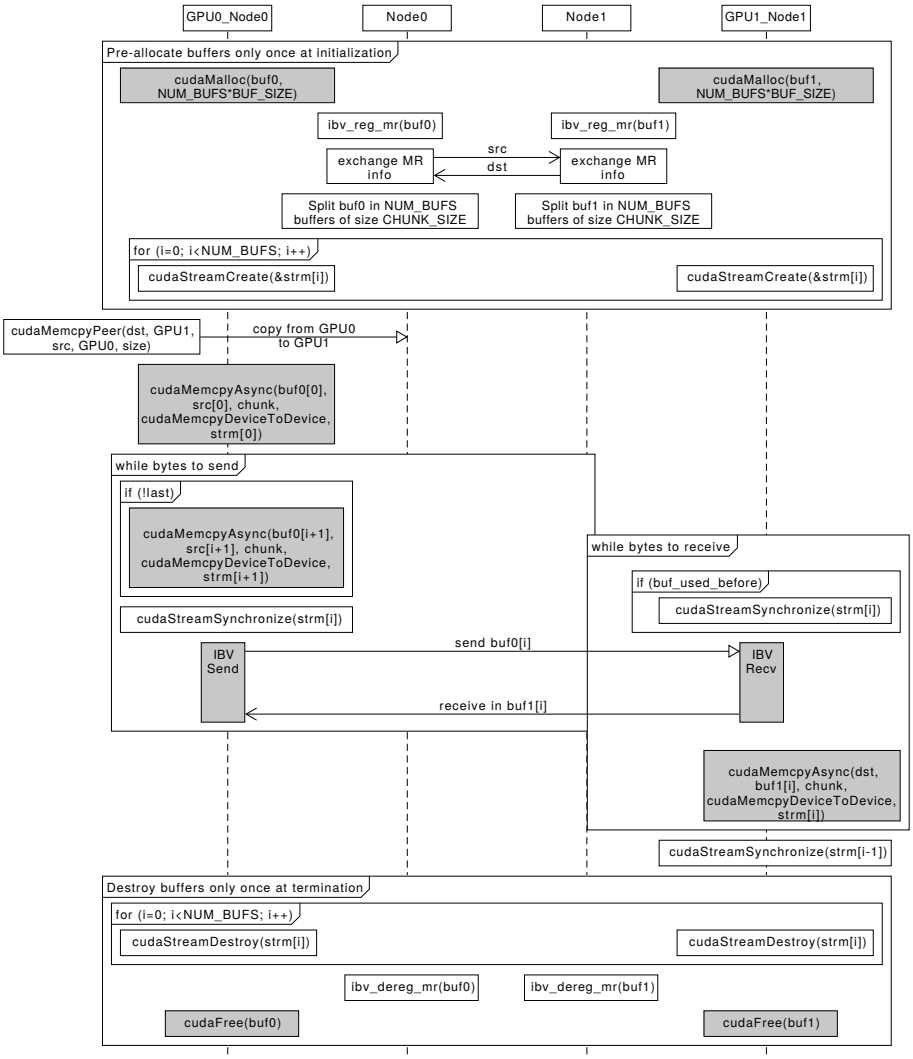


Figure 10: Sequence diagram of rCUDA version 3A for memory copies between different remote GPUs. Gray boxes depict differences with respect to version 3B shown in Figure 11. This version is similar to version 2A, but uses multiple intermediate buffers at the GPU.

the influence of these two factors we have run the same bandwidth test as in Figure 12 with versions 3A and 3B, but varying the number of intermediate buffers: 2, 4, 8, 16, 32 and 64. For each number of buffers, we have also run the test with different buffer sizes: 64KB, 128KB, 256KB, 512KB, 1MB, 2MB,

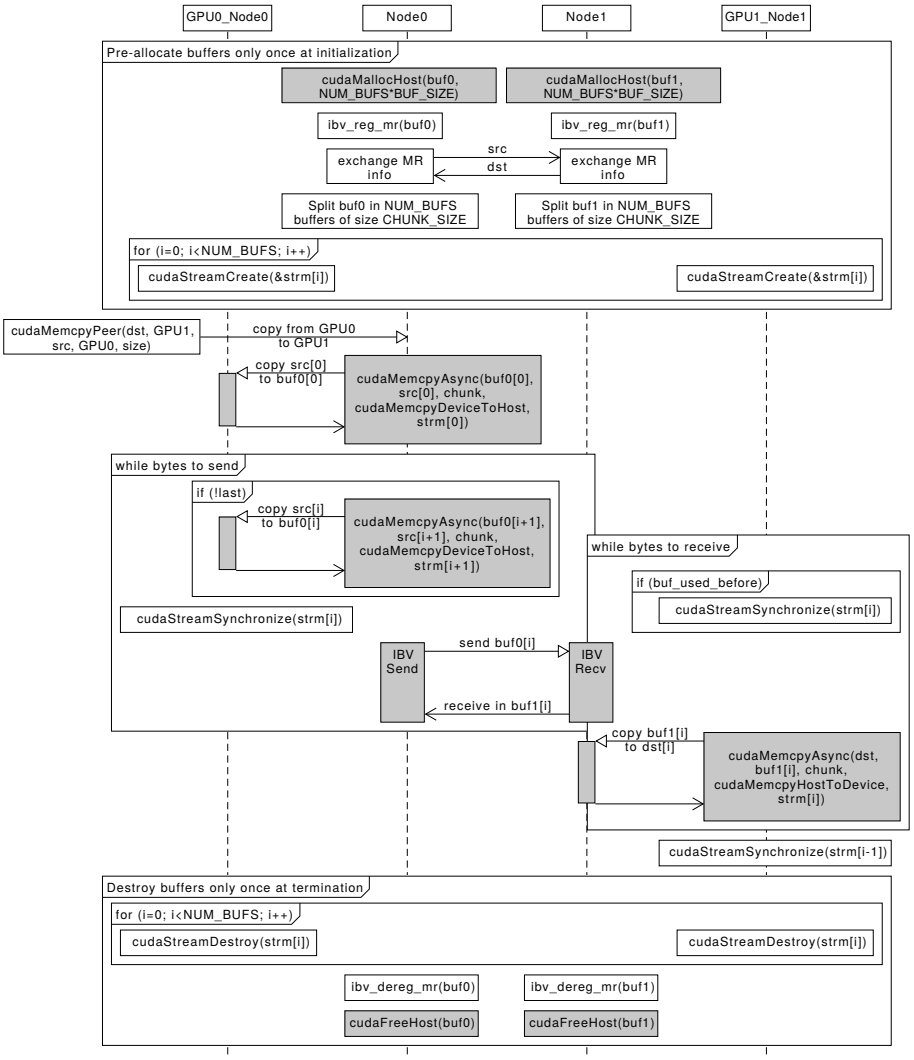


Figure 11: Sequence diagram of rCUDA version 3B for memory copies between different remote GPUs. Gray boxes refer to differences with respect to version to 3A shown in Figure 10. This version is similar to version 2B, but uses multiple intermediate buffers at host memory.

4MB, and 8MB.

In the case of using GPU intermediate buffers (version 3A), the results of the analysis point to select the following values as the optimal ones:

- Copy sizes over 4MB: the optimal number and size of intermediate buffers

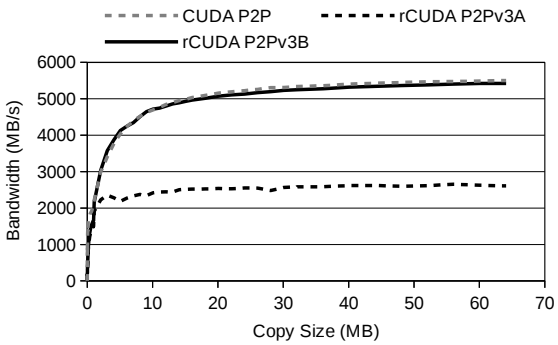


Figure 12: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA and versions 3A and 3B of rCUDA are shown.

is 16 buffers of 256KB

- Copy sizes between 700KB and 4MB: the optimal number and size of intermediate buffers is 2 buffers of 512KB
- Copy sizes below 700KB: the optimal number and size of intermediate buffers is 2 buffers of 1MB

In the case of using host intermediate buffers (version 3B), this analysis reveals the following results:

- Copy sizes over 14MB: the optimal number and size of intermediate buffers is 4 buffers of 1MB
- Copy sizes between 4MB and 14MB: the optimal number and size of intermediate buffers is 4 buffers of 512KB
- Copy sizes between 600KB and 4MB: the optimal number and size of intermediate buffers is 8 buffers of 256KB
- Copy sizes below 600KB: there is no apparent optimal value for the two factors under analysis. We decided to select as the optimal value 8 buffers of 128KB, following the trend of previous values

With the results of this analysis we have implemented a new version which automatically varies the number and size of intermediate buffers depending on

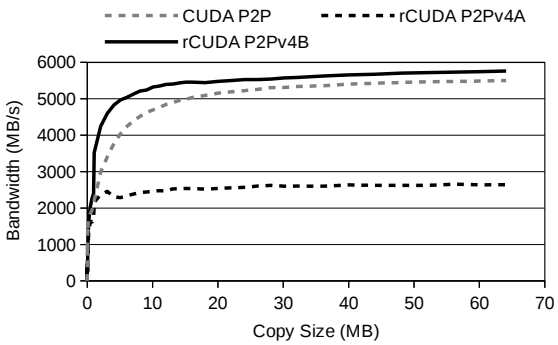


Figure 13: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA and versions 4A and 4B of rCUDA are shown.

the actual size of the data to be copied. Following our version nomenclature, we have named these new versions as 4A and 4B. Figure 13 shows the bandwidth results for these new versions, showing that version 4A improves over its predecessor, version 3A, for copy sizes up to 4MB. For larger sizes, the results are similar to those of the previous version. With regard to version 4B, we can see that it clearly outperforms version 3B, regardless of the data size of the copy.

It is also noteworthy that version 4B obtains, in general, better performance than CUDA. For copy sizes up to 300KB, CUDA achieves a higher bandwidth, but for larger copy sizes, rCUDA attains better results. The explanation for this higher bandwidth of rCUDA can be found in Figure 14. This figure shows the bandwidth attained by the different CUDA and InfiniBand memcpy functions involved in the analysis presented in this section. When using CUDA, a single call to `cudaMemcpyPeer` is done to copy the whole bunch of data between the two GPUs. The PCIe bus is used to move data from one GPU to the other. On the contrary, when using rCUDA, the data to be copied is split into smaller chunks of data and the movement of the several chunks is overlapped by using several simultaneous calls to: (1) `cudaMemcpyAsync` from GPU to host memory, (2) InfiniBand memory copy from local host memory to remote host memory, and (3) `cudaMemcpyAsync` from host to GPU memory. As we have previously commented, a pipelined approach is followed. Therefore, the maximum band-

width that can be achieved with this technique is the minimum bandwidth of each individual stage. In this case, the minimum bandwidth of the three mentioned calls is the one obtained by `cudaMemcpyAsync` from host to GPU memory (labeled as *cudaMemcpyAsyncToGPU* in Figure 14). As we can see, this bandwidth is larger than the one obtained by `cudaMemcpyPeer`, which explains why rCUDA is attaining more bandwidth than CUDA.

4.5. Overview of proposed versions for rCUDA

With the aim of providing a quick overview of all the approaches proposed and their performance, Figure 15 combines all the results previously presented in Figures 6, 9, 12 and 13.

4.6. Latency analysis

In previous sections we have thoroughly analyzed the bandwidth attained by each P2P version of rCUDA. Next, we analyze latency. As a summary,

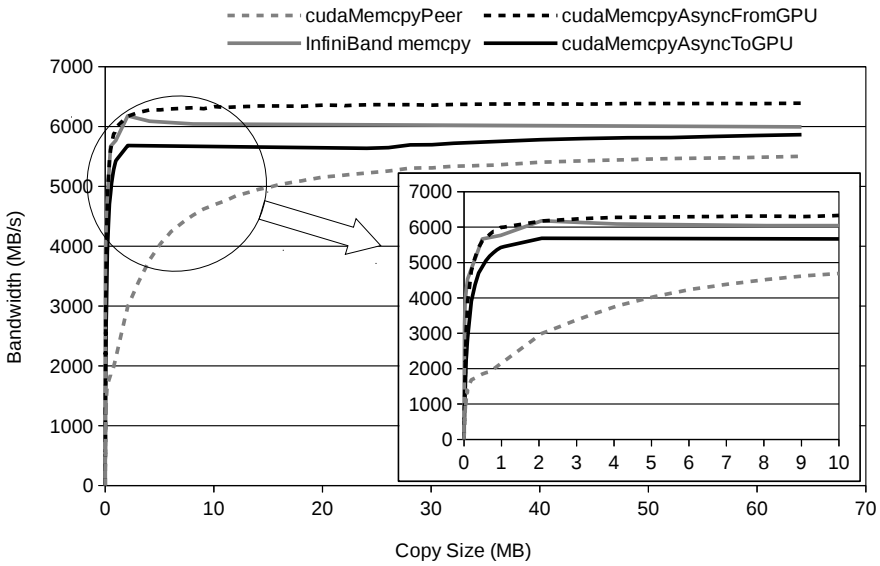


Figure 14: Bandwidth comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter.

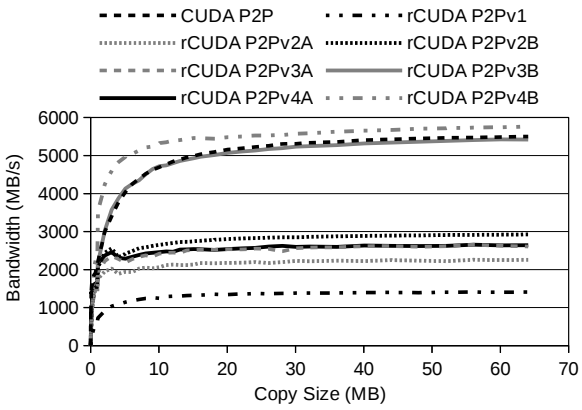


Figure 15: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results for the different versions of rCUDA previously discussed are shown. Results for CUDA with local GPUs are also depicted.

Table 1: Summary of the different rCUDA versions implemented for supporting memory copies between remote GPUs.

Feature	rCUDA version						
	1	2A	2B	3A	3B	4A	4B
GPUDirect RDMA	x	x		x		x	
Pre-allocated buffers		x	x	x	x	x	x
Multiple buffers				x	x	x	x
Adaptive buffer size						x	x

Table 1 presents the most important features of the different rCUDA versions implemented for supporting memory copies between remote GPUs.

Figure 16 presents the latency obtained when using CUDA and different versions of rCUDA to copy data between remote GPUs located in different nodes of the cluster. For clarity, results from versions 1 and 2 are not shown. It can be seen in the figure that the best results are obtained by CUDA, with a minimum latency of 22μs. Regarding the different versions of rCUDA, version 4A, using GPUDirect RDMA and all the improvements analyzed in previous

sections, seems to achieve the best results, with a minimum latency of 72μs. Version 4B, not using GPUDirect RDMA, presents a minimum latency of 78μs. Version 4A is more stable than version 4B and presents, in general, the best latency for copy sizes up to 200KB. Finally, versions 3A and 3B present a higher latency, both with minimum values of 131μs.

In the bandwidth analysis presented in previous sections we have seen that rCUDA obtained, in general, better results than CUDA, the only exception being copy sizes smaller than 300KB. The latency results match those conclusions, given that CUDA presents lower latency than rCUDA for that range of data copy sizes. In previous sections, we explained that rCUDA achieved, in general, higher bandwidth than CUDA because the internal functions involved in the memory copy were different and also presented different performance, in addition to follow a pipelined approach. This better performance was shown in Figure 14. Now the question is whether the reason for rCUDA presenting a higher latency than CUDA is also because of this.

To answer this question, Figure 17 presents a latency comparison of the different CUDA and InfiniBand memcpy functions involved in the memory copy between GPUs. It can be seen that the `cudaMemcpyPeer` function used by

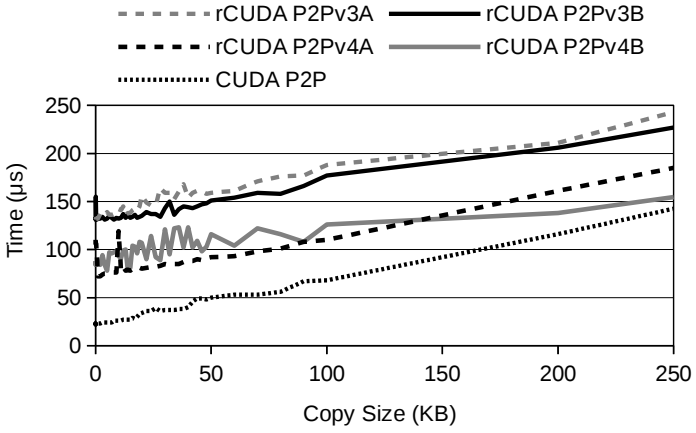


Figure 16: Latency obtained for transfer sizes up to 200KB when copying data between remote GPUs. Results from CUDA and different versions of rCUDA are shown.

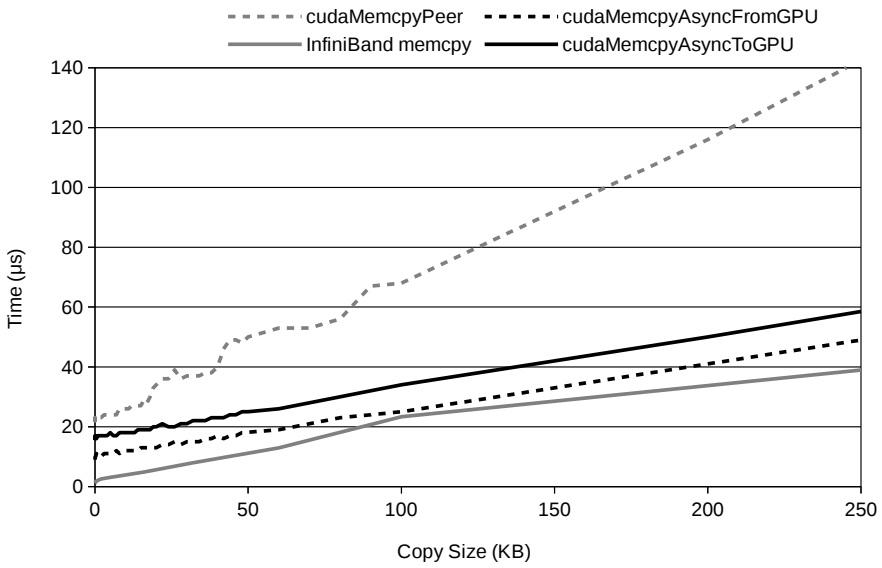


Figure 17: Latency comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter.

CUDA to copy data between local GPUs achieves the highest latency. In the case of copying 4 bytes of data, this latency is equal to 22μs. On the contrary, the operations involved in data copies with rCUDA present, individually, a smaller latency. For instance, in the case of copying 4 bytes of data, the `cudaMemcpyAsync` call to copy data from the GPU to host memory at the source node requires 9μs. At the destination node, the `cudaMemcpyAsync` call to copy data from host memory to GPU memory requires 17μs. Finally, the latency for moving those 4 bytes from one node to the other across the InfiniBand fabric is 1.3μs. Notice, however, that the data copy with rCUDA follows a pipelined approach. Therefore, the total latency of the P2P data copy is the result of the addition of the previous values, turning out a latency of 27.3μs.. This value is, however, smaller than the latency measurement provided in Table 2. The reason for the difference among both values is due to the management required to run our highly tunned pipeline, what seems to penalize our approach in terms of latency. In this regard, one could think about using more simple approaches for small copy sizes in order to improve latency. However, notice that we have

Table 2: Minimum latency achieved by the different rCUDA versions implemented for supporting memory copies between remote GPUs.

	rCUDA version						
	1	2A	2B	3A	3B	4A	4B
Latency (μs)	596	136	131	131	131	72	78

already tried simpler proposals, such as versions 1 and 2, and results are worse than the ones obtained with versions presenting higher complexity, such as versions 3 and 4 (see Table 2).

4.7. Final version: hybrid approach

After analyzing different versions in the previous sections, we conclude that the optimal version for the P2P implementation within rCUDA would be a hybrid approach combining versions 4A and 4B. Thus, for small copy sizes up to 200KB, the best results are achieved by version 4A, using GPUDirect RDMA. For larger copy sizes, the best results are obtained with version 4B, not using GPUDirect RDMA. We have implemented one last version featuring this hybrid approach. This new version will be the one used from now on in the rest of the paper. Bandwidth results for this new version are omitted for brevity, given that the results are very similar to the ones already shown: bandwidth of version 4B in Figure 13, and latency of version 4A in Figure 17.

It should be noted that, when executing an application with rCUDA using remote GPUs, the version here presented is only used when the GPUs are in different remote server nodes, scenario shown in Section 1, Figure 2(c). On the contrary, if the GPUs are in the same remote server node, scenario shown in Section 1, Figure 2(b), the CUDA call is forwarded to the CUDA driver in the remote node, which manages the copy. That is, if both remote GPUs are located in the same node, then a regular CUDA P2P data copy is carried out to transfer data among both GPUs. This behavior has been implemented within the rCUDA middleware as part of the work here presented and is transparent

to the user.

5. Experiments with a Real Application

In previous sections we have assessed the performance of the implemented P2P memory copy mechanism by using synthetic tests. This section presents the experiments carried out with a real application that makes use of the P2P copies provided by CUDA.

The selected application belongs to the area of network analysis, where the clustering coefficient and the transitivity ratio are concepts often used, creating the need for fast practical algorithms devoted to count triangles in large graphs. Furthermore, these algorithms can be programmed to be executed in GPUs so that total execution time is reduced. Therefore, for the performance evaluation in this section we have used an application for counting triangles in large graphs on GPUs [23]. From now on, we will refer to this application as TRICO (triangle count). TRICO is a CUDA implementation of a parallel algorithm for counting triangles (i.e. 3-cycles) in large graphs which additionally is able to take advantage of all the GPUs available in the node where it is being executed. Additionally, this application performs P2P data copies among the GPUs involved in its execution.

The setup used for the experiments reported in this section is the same as the one presented in Section 4. In the case of the experiments with CUDA, and given that the application can make use of several GPUs, the scenario used in the tests is the one depicted in Figure 2(a) of Section 1. In particular, two GPUs will be provided to the application. Regarding the experiments with rCUDA, the scenario is the one depicted in Figure 2(c) of Section 1. Therefore, the application is running in a node without GPUs, and it is using two remote GPUs located in two different server nodes.

Seven different graphs have been leveraged for the experiments in this section. These graphs are described in Table 3. The largest one, referred to as graph number 7, is a 180 million edge graph containing 8.8 thousand millions

Table 3: Graphs used in the experiments with the TRICO application.

Graph	Nodes	Edges	Triangles
1	65,535	4,912,142	118,811,321
2	131,068	10,227,970	287,593,439
3	540,486	30,491,458	444,095,058
4	434,102	32,073,440	872,040,567
5	524,287	43,561,574	1,625,559,121
6	1,048,576	89,238,804	3,803,609,518
7	2,097,152	182,081,864	8,815,649,682

of triangles. We have used this graph in order to characterize the TRICO application. Figure 18 shows the CPU and GPU utilization when executing the application with this largest graph in the CUDA scenario (local GPUs). It can be seen that the CPU presents a high utilization, almost 100% during all the execution of TRICO. In the case of the GPUs, GPU0 starts being used after 2 seconds of execution, while GPU1 starts working after almost 4 seconds. Then, both GPUs also present a high usage, close to the 100% until near the end of the execution of the application. In summary, Figure 18 shows that the TRICO application makes an intensive use of the available GPUs as well as the CPU.

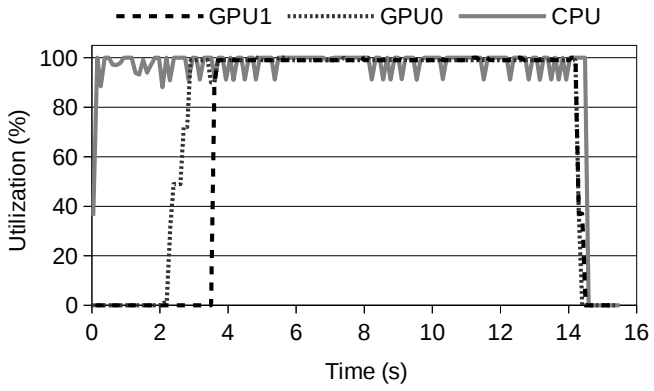


Figure 18: CPU and GPU utilization of TRICO when running the largest graph.

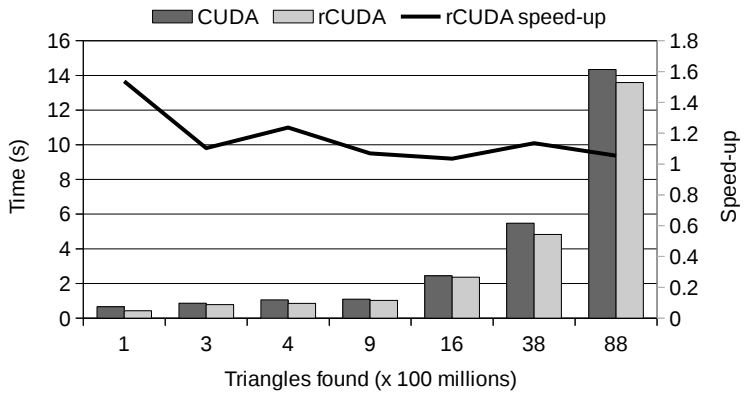


Figure 19: Primary Y-axis shows TRICO application execution time using CUDA and rCUDA. Secondary Y-axis presents the speed-up of rCUDA with respect to CUDA.

Figure 19 shows the execution time and speed-up when running the TRICO application using the different graphs described in Table 3. Results are the average of 10 repetitions. The primary Y-axis of the figure shows the execution time of the TRICO application when using CUDA and rCUDA. As we can observe, the application runs slightly faster with rCUDA than with CUDA for all the graphs evaluated. The secondary Y-axis shows the exact speed-up of rCUDA with respect to CUDA. On average, the speed-up is 1.13. These results showing that the application achieves better performance with rCUDA than with CUDA require a deeper analysis.

One possible explanation to the better results of rCUDA with respect to CUDA could be that rCUDA provides a better bandwidth when copying data between GPUs, as shown in Figure 13. Therefore, that higher bandwidth is causing the difference in the execution time with respect to CUDA. To find out whether this is the reason for the better results when using rCUDA, we have profiled the TRICO application in order to know the amount of calls to the functions that perform P2P data copies. Table 4 presents such profiling, showing for each CUDA function used in the application the total time employed by the application in that function as well as the number of calls done to it. The average, minimum and maximum time per call is also displayed. Surprisingly,

Table 4: Profiling of the TRICO application when running the largest graph with CUDA.

CUDA function	% Time	Time	Calls	Average	Min	Max
cudaDeviceSynchronize	87.05%	11.49s	8	1.433s	5.19us	11.42s
cudaMemcpy	4.41%	582.44ms	6	97.07ms	24.10us	531.04ms
cudaFree	4.39%	579.42ms	18	32.19ms	151.75us	219.06ms
cudaMemcpyAsync	3.81%	502.65ms	2	251.33ms	28.79us	502.63ms
cudaMemcpyPeer	0.27%	34.99ms	2	17.49ms	1.55ms	33.43ms
cudaMalloc	0.05%	6.13ms	16	383.17us	146.45us	1.19ms
cudaGetDeviceProperties	0.01%	1.95ms	4	489.61us	462.27us	543.63us
cuDeviceGetAttribute	0.01%	983.12us	166	5.92us	160ns	225.88us
cudaLaunch	0.01%	949.26us	65	14.60us	9.61us	61.66us
cudaFuncGetAttributes	0.00%	140.81us	41	3.43us	2.55us	13.43us
cuDeviceTotalMem	0.00%	109.11us	2	54.55us	53.71us	55.39us
cuDeviceGetName	0.00%	98.33us	2	49.16us	44.18us	54.15us
cudaSetupArgument	0.00%	58.70us	278	211ns	177ns	693ns
cudaGetDeviceCount	0.00%	38.47us	1	38.47us	38.47us	38.47us
cudaSetDevice	0.00%	36.23us	14	2.58us	833ns	11.89us
cudaConfigureCall	0.00%	24.50us	65	377ns	239ns	1.85us
cudaGetDevice	0.00%	18.25us	26	702ns	325ns	1.73us
cudaFuncSetCacheConfig	0.00%	3.77us	2	1.88us	1.46us	2.31us
cuDeviceGetCount	0.00%	3.61us	2	1.80us	447ns	3.16us
cudaDeviceGetAttribute	0.00%	3.25us	3	1.08us	701ns	1.69us
cuDeviceGet	0.00%	1.43us	4	357ns	202ns	498ns

it can be seen in Table 4 that there are only two calls to functions involving copies between GPUs (i.e., calls to the `cudaMemcpyPeer` function, highlighted in bold in Table 4). Furthermore, the time used by these calls is 34.990ms, what only accounts for 0.27% of the total execution time of the application. This small percentage of time does not seem to be the reason for the better results of rCUDA over CUDA. However, let us further examine these calls to the `cudaMemcpyPeer` function.

In order to continue with our analysis of the better execution times achieved by the TRICO application when using rCUDA instead of CUDA, we have next

Table 5: Time employed in the copies between GPUs by CUDA and rCUDA when running the TRICO application with the largest graph.

CUDA function	Size	Time	
		CUDA	rCUDA
1st cudaMemcpyPeer	695MB	33.433ms	125.225ms
2nd cudaMemcpyPeer	8MB	1.557ms	1.349ms

measured the time employed in the P2P copies between GPUs with rCUDA (notice that the times depicted in Table 4 were obtained with CUDA in the scenario shown in Figure 2(a)). In this regard, we have measured the time required to carry out these P2P copies with rCUDA in the scenario presented in Figure 2(c). Table 5 compares the time results obtained with CUDA and rCUDA. The table also shows the amount of megabytes copied during each call to the `cudaMemcpyPeer` function.

Interestingly, it can be seen that the first call to the `cudaMemcpyPeer` function takes more time with rCUDA than with CUDA, despite that the size of the copy, 695MB, is large enough to expose the better performance of rCUDA shown in the bandwidth experiments (see Figure 13). Additionally, the time required by rCUDA to perform the data copy is much larger than the time required by CUDA. This difference does not match the bandwidth results shown in previous sections. On the other hand, in the case of the second call to the `cudaMemcpyPeer` function, the time required to complete the copy with CUDA and rCUDA do not follow the trend of the first copy. On the contrary, in this second copy the time results in Table 5 match the trend observed in the experiments of previous sections where rCUDA presented a better performance than CUDA. Thus, the question at this point is what happened with the first P2P copy shown in Table 5.

After a deeper analysis to find out the reason for this behavior when rCUDA used, we found out that, in the first call to the `cudaMemcpyPeer` function,

rCUDA needs to carry out the necessary initialization of the P2P mechanism to copy data between the remote GPUs (i.e., creating the InfiniBand connections between the remote nodes where the remote GPUs are located and pre-allocating intermediate buffers for RDMA transfers). On the contrary, in the second call to the `cudaMemcpyPeer` function, this start-up is already done and therefore rCUDA achieves the higher bandwidth already shown in Figure 13.

Table 6: Time employed by CUDA and rCUDA in two consecutive copies of the same size between GPUs. A synthetic test is used

(a) Synchronization point introduced only after the second call.

CUDA function	Size	Time	
		CUDA	rCUDA
1st cudaMemcpyPeer	695MB	33.433ms	125.225ms
2nd cudaMemcpyPeer	695MB	123.758ms	116.951ms

(b) Synchronization points introduced after each call.

CUDA function	Size	Time	
		CUDA	rCUDA
1st cudaMemcpyPeer	695MB	123.869ms	125.336ms
2nd cudaMemcpyPeer	695MB	123.745ms	117.062ms

To show that this was the reason, a synthetic test performing two consecutive copies of the same size was carried out with CUDA and rCUDA. Table 6(a) presents the results of this experiment. Regarding rCUDA, it can be seen that, as in the previous case shown in Table 5, the first copy needs more time than the second one because it initializes the P2P memory copy mechanism between the remote GPUs. Additionally, when considering the second copy (notice that this time the second copy has the same size as the first one), it can be seen that it is completed 8ms earlier with rCUDA than with CUDA. This result matches the conclusions obtained in previous sections where rCUDA obtained better performance as shown in Figure 13.

Interestingly, when CUDA is used, it can be seen that the second copy takes longer to be completed than the first one, despite they transfer the same

amount of data. In particular, the second copy requires 92ms more than the first copy. This result was unexpected because it was assumed that both copies would last the same amount of time given that they are transferring the same amount of data. The explanation to this behavior was found after reviewing the CUDA documentation of the function `cudaMemcpyPeer` [7]. According to its documentation, this function is asynchronous with respect to the host. This means that, with CUDA, this call does not return the control to the application once the copy is completed but control is returned much earlier and the copy will be performed asynchronously with respect to the non-GPU part of the application. To ensure that the copy is finished, the application needs to add a subsequent synchronization point (such as `cudaDeviceSynchronize`).

In order to further analyze this behavior, we modified the synthetic test program used in Table 6(a) so that a call to `cudaDeviceSynchronize` is performed after each call to the `cudaMemcpyPeer` function. In this way we force that the first P2P copy is completed before the execution of the second P2P copy begins. Table 6(b) shows the new results. It can be seen that now both calls to the `cudaMemcpyPeer` function last the same time when CUDA is used. Additionally, the time required by each call when rCUDA is used is not changed. The reason why times are kept the same with rCUDA is that, as previously explained in Section 4.3, the P2P copy with rCUDA is performed following a pipelined approach, overlapping multiple smaller copies in the different stages of the pipeline. In order to smoothly run this pipeline, before starting the copy from the intermediate buffer to the remote GPU, rCUDA needs to make sure that the copy from the GPU memory to the intermediate buffer has finished. For that purpose, a synchronization point is added. That is the reason why both copies in Table 6(b) take the same time with rCUDA as in Table 6(a), because they already included the synchronization points.

As a summary, we started our analysis about the better performance of the TRICO application when it makes use of rCUDA instead of CUDA (shown in Figure 19) by presuming that one possible explanation to that behavior could be the better bandwidth achieved by rCUDA for the P2P copies (shown in

Table 7: Breakdown of the TRICO application when running the largest graph with CUDA and rCUDA. Stages marked with * include synchronization points.

Stage	Time (ms)		Difference
	CUDA	rCUDA	(CUDA-rCUDA)
Read file	853.0	862.5	-9.5
Pre-initialize context for all GPUs	365.0	277.5	87.5
Memcpy edges from host to GPUs*	479.0	469.0	10.0
Calculate number of vertices	12.0	10.0	2.0
Sort edges*	538.0	459.5	78.5
Calculate node array for 2-way zipped edges*	23.0	18.0	5.0
Remove backward edges*	185.0	179.2	5.8
Unzip edges*	27.0	24.83	2.2
Calculate node array for 1-way unzipped edges*	10.0	9.0	1.0
Calculate triangles on multi GPU**	11541.0	11033.8	507.2
TOTAL SUM	14033.0	13343.3	689.7

Figure 13). However, after our analysis we conclude that, not only this is not the explanation for the better performance, but our implementation of the P2P copies also shows a potential overhead of rCUDA with respect to CUDA due to the necessary initialization of the P2P mechanism to copy data between GPUs located in different server nodes. This initialization is performed at the first call to a CUDA function that performs P2P copies. However, the overhead associated with this initialization could be avoided if it were carried out during application start up. But in this case some memory could be wasted. We will elaborate on this in Section 7 as part of future work.

Let us come back to our original question: why does the TRICO application perform better with rCUDA than with CUDA? In this regard, Table 6(a) has shown that control after P2P copies is returned to the application earlier with CUDA than with rCUDA. On the other hand, kernels take the same time to be completed regardless of being executed in a local GPU with CUDA or in a remote GPU with rCUDA. As a consequence, the application should perform better with CUDA than with rCUDA, which is not the case. Therefore, where

during the execution of the application with CUDA, the time saved by the P2P copies is later lost?

In order to find the answer to this question, there must be in the application source code one or more CUDA functions that perform better with rCUDA than with CUDA. To see whether this is the case, Table 7 presents a breakdown of the TRICO application when running the largest graph with CUDA and with rCUDA. A detailed explanation of each stage can be found in [23]. As additional information, stages marked with the symbol * indicate that there is a synchronization point in that stage. In the case of the stage *"Calculate triangles on multi GPU"*, it includes two synchronization points, one per GPU. Time required by each stage to be completed is shown in the table.

It can be seen in the table that most stages are completed faster with rCUDA than with CUDA. Actually, those stages presenting significant improvement when using rCUDA include synchronization points[2]. Remember that in Table 4 we showed that the major part of the execution time of the TRICO application was spent in synchronization points (i.e., 87.05% of the time was used by 8 calls to function `cudaDeviceSynchronize`). Now Table 7 shows that the improvement of rCUDA with respect to CUDA in those stages containing those 8 synchronization points is 609ms. These savings, together with the time saved in the context initialization in stage *"Pre-initialize context for all GPUs"*, explains the total time saved when using rCUDA. The reason for this large time difference in the synchronization points lies in the internal algorithm used in the rCUDA middleware to determine the finalization of the CUDA tasks [24], which performs better than the method used within CUDA.

As a summary of this section, the TRICO application was considered as

[2] The only exception is the stage *"Pre-initialize context for all GPUs"*. In this case the improvement is due to the fact that the rCUDA server pre-initializes contexts on the GPUs at start-up. In this manner, the rCUDA server is waiting for requests from client applications with the context already initialized. On the contrary, applications running with CUDA must spend some time in the context pre-initialization.

use-case of an application using P2P copies. Execution times show that the application runs faster with rCUDA than with CUDA, being the most immediate reason the better bandwidth achieved by rCUDA for P2P copies, as shown in the previous section. However, although rCUDA attains more bandwidth than CUDA, finally the reason for the better performance of the TRICO application with rCUDA was the better management that this middleware makes for determining CUDA task finalization. In any case, the implementation presented in this paper for the P2P data copies does not penalize applications, although it could be still improved, as it will be explained in Section 7.

6. Performance in more recent platforms

Performance results shown for the P2P data copies in Sections 4 and 5 were gathered by using NVIDIA Tesla K20 GPUs along with CUDA 7.5. However, one may wonder whether the extraordinary bandwidth results reported by our proposal are stable across CUDA versions as well as across GPU architectures. In order to asses if the conclusions from the performance results presented in previous sections are also valid with more recent platforms, in this section we show experiments using newer hardware and software.

More precisely, we use a similar setup to the one of the previous section comprising three 1027GR-TRF Supermicro servers. The servers are now configured with CentOS 7.3 and CUDA 9.1 with NVIDIA driver 390.46. In addition, new hardware is also included in the experiments. Regarding the network connection, apart from the FDR one used in previous sections, the three server are now also connected by an SB7700 InfiniBand switch featuring EDR (100 Gbps). EDR InfiniBand cards are included in the servers. Regarding new GPU generations, both Tesla K40m and Tesla P100 GPUs are considered in this section.

We include experiments with the same Tesla K20m GPU used in previous sections, and also using Tesla K40m and Tesla P100 GPUs.

In first place, we have measured the bandwidth obtained for different transfer sizes when copying data between remote GPUs. Figure 20 shows the results

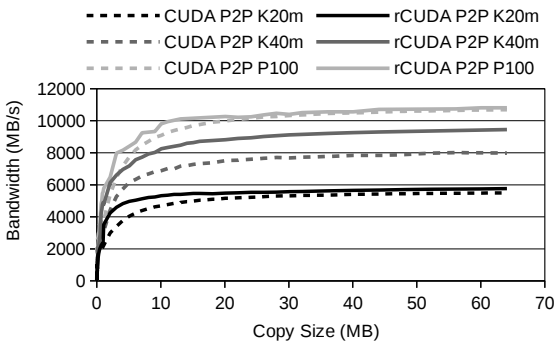


Figure 20: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA and rCUDA are shown using different configurations: K20m and FDR InfiniBand; K40m and EDR InfiniBand; P100 and EDR InfiniBand.

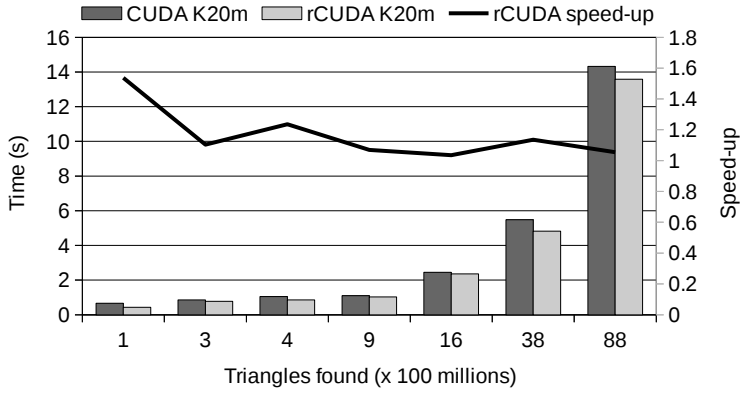
of these experiments. Notice that the figure also includes results for the K20m already presented in previous sections, but this time using the more recent operating system and CUDA version. In the case of the K20m over FDR InfiniBand, it can be seen that performance results are very similar to the ones presented in Section 4. This means that our designs are stable across CUDA versions. Furthermore, rCUDA results for the P2P data copies are very close of the maximum bandwidth achieved when copying data from host memory to GPU memory with CUDA (within the same node). This maximum bandwidth is 6 GB/s. The same happens when using the K40m GPU over EDR InfiniBand: on the one hand, it can be seen that performance with rCUDA for P2P data copies is larger than with CUDA; on the other hand, maximum bandwidth attained with rCUDA is close to the 10 GB/s achieved by the GPU in host to device memory copies with CUDA (within the same node). Moreover, performance of P2P data copies with rCUDA using the K40 GPU along with CUDA 7.5 presented similar results (not shown). These results point out that the performance provided by our proposal is stable across CUDA versions. Finally, rCUDA experiments using P100 GPUs over EDR InfiniBand also present better performance than when CUDA is used with two GPUs located in the same host. However, in this case performance is not limited by the GPU bandwidth (12 GB/s) but the limiting factor is the

EDR InfiniBand network bandwidth (11 GB/s). In summary, it can be seen that performance of our proposal is stable both across CUDA versions and also across GPU generations.

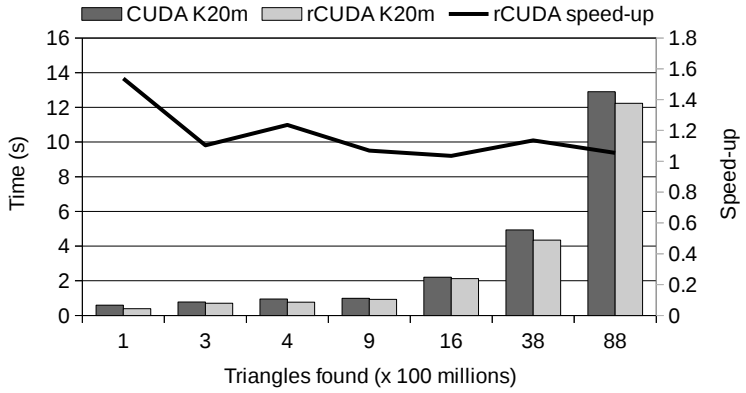
In addition to bandwidth tests, we have also run the same TRICO application used in previous sections. We have adapted and recompiled the application for CUDA 9.1. Figure 21 shows the results of executing the application over the new hardware and software test-bed system. Regarding the experiments using the Tesla K20m GPU over an FDR InfiniBand network, results with CUDA 9.1 shown in Figure 21(a) are very similar to the ones using CUDA 7.5 previously shown in Figure 19. On average, the variation of the results is in the range of $\pm 0.001\%$. With respect to the experiments using the Tesla K40m GPU over an EDR InfiniBand network, shown in Figure 21(b), it can be observed that the results are in line with the ones of the K20m over FDR. In this case, the GPU is more powerful and the network also provides more bandwidth, which translates into less execution time (an average reduction of 10%). Finally, the experiments with the Tesla P100 GPU over an EDR InfiniBand network, presented in Figure 21(c), show that the use of rCUDA in this scenario introduces some overhead. As shown in Figure 20, the bandwidth is not the cause, because rCUDA obtains higher bandwidth than CUDA. The reason is that in this case the new GPU, in addition to having higher computing capabilities, also presents a more modern GPU architecture. The Tesla K20m and K40m feature the Kepler architecture whereas the P100 GPU has the Pascal one. After performing a similar profiling than the one carried out in Section 5, this new GPU architecture seems to have improved the management for determining CUDA task finalization, probably being this improvement the reason for the better performance of the TRICO application with CUDA.

7. Conclusions and Future Work

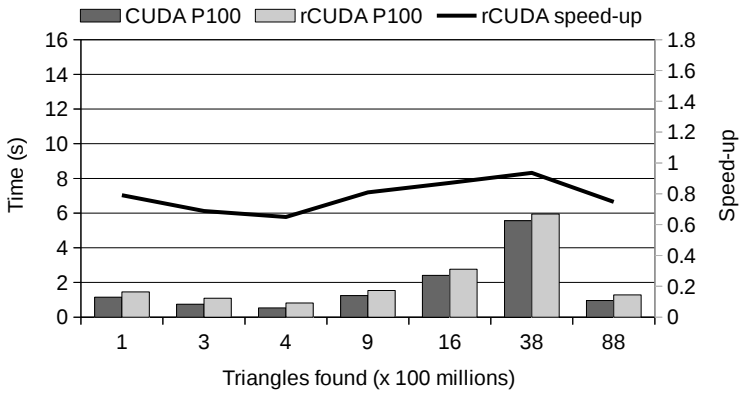
Remote GPU virtualization frameworks alleviate many of the concerns related to GPUs. As these frameworks aim to offer the same API as the NVIDIA



(a) Results using Tesla K20m GPU and FDR InfiniBand network.



(b) Results using Tesla K40m GPU and EDR InfiniBand network.



(c) Results using Tesla P100 GPU and EDR InfiniBand network.

Figure 21: Results of TRICO application using different GPUs and networks. Primary Y-axes show execution time using CUDA and rCUDA. Secondary Y-axes present the speed-up of rCUDA with respect to CUDA.

CUDA Runtime API does, they must also support the functions for memory copies between GPUs existing in such API. This support must also be satisfied when the peer GPUs involved in the data copy are located at different nodes of the cluster, a kind of situation which may occur when using remote GPUs.

This paper has presented an efficient memory copy mechanism devised for enhancing the rCUDA framework with support for memory copies between remote GPUs located in different nodes of the cluster. The finally proposed mechanism is the result of a thorough analysis of several implementation options, which have been explored and whose performance has been evaluated and compared with each other. The proposed mechanism consists in the use of multiple intermediate buffers following a pipelined approach, where several memory copies are overlapped in different stages. In this regard, we began our exploration with the GPUDirect RDMA technique proposed by NVIDIA (which initially was thought to the best implementation choice) but we have finally showed that a pipelined approach presents better performance. Notice that using a pipeline approach for communicating data is not new. Actually, this approach can be found in many areas such as computer networks, high performance on-chip and off-chip interconnects, etc. However, although the concept of the mechanism is basically the same in all these areas, the exact implementation is very different. In fact, in none of these areas the research emphasis is put on the pipeline mechanism but on its implementation.

Regarding the implementation of the P2P data copies proposed in this paper, it has been shown in the previous sections that one more refinement could be applied to it. In this regard, the initialization of the P2P mechanism within rCUDA is done at the first call to a P2P data copy function. Thus, this initialization reduces the performance of this first P2P copy. One possible alternative would be to carry out this initialization at application start-up. This would save the time associated with it because the initialization could be done within the rCUDA framework in parallel with the first instructions of the application. However, this would also mean that InfiniBand connections are created among all the GPUs involved in the execution of the application, thus increasing the

memory footprint of the rCUDA middleware. Notice that if the application later does not make use of any P2P copy, all those resources would have been wasted. This is an interesting research left for future work.

Additional future work would comprise the analysis of the proposed P2P copy implementation with other GPU models and InfiniBand versions. In this regard, the study presented in this paper was carried out with NVIDIA Tesla K20, K40 and P100 GPUs, using FDR and EDR InfiniBand network fabrics. Better versions of these technologies should be considered in future work, such as NVIDIA Tesla V100 GPUs and HDR InfiniBand fabrics providing 200 Gbps. The new Turing GPU should also be considered. Furthermore, the arrival of the NVLink intra-node connection as well as newer versions of the PCIe bus (PCIe v4 in the very near term and PCIe v5 in the mid term) will make necessary to review the conclusions from this study. Probably, changes in the proposed pipeline for P2P data copies should also be carried out.

Finally, notice that in this paper we have dealt only with the synchronous calls to P2P data copies within CUDA. However, one can find in CUDA the asynchronous versions of these functions, known as `cudaMemcpyPeerAsync`. Therefore, applying the proposals in this paper to the asynchronous calls is also a task to be carried out in the future.

References

[1] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, S. Yalamanchili, Red Fox: An execution environment for relational query processing on GPUs, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, ACM, 2014, pp. 44:44–44:54. doi:10.1145/2544137.2544166.
URL http://doi.acm.org/10.1145/2544137.2544166

[2] I. Yamazaki, T. Dong, R. Solc, S. Tomov, J. Dongarra, T. Schulthess, Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems, Concurrency and Compu-

tation: Practice and Experience 26 (16) (2014) 2652–2666. doi:10.1002/cpe.3152.
URL http://dx.doi.org/10.1002/cpe.3152

[3] D. P. Playne, K. A. Hawick, Data parallel three-dimensional Cahn-Hilliard field equation simulation on GPUs with CUDA, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA, 2009.
URL http://dblp.uni-trier.de/rec/bib/conf/pdpta/PlayneH09

[4] V. Surkov, Parallel option pricing with Fourier space time-stepping method on graphics processing units, Parallel Computing 36 (7) (2010) 372 – 380. doi:10.1016/j.parco.2010.02.006.
URL http://dx.doi.org/10.1016/j.parco.2010.02.006

[5] D. Yuancheng Luo, Canny edge detection on NVIDIA CUDA, in: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, IEEE, 2008, pp. 1–8. doi:10.1109/CVPRW.2008.4563088.
URL http://dx.doi.org/10.1109/CVPRW.2008.4563088

[6] P. K. Agarwal, S. Hampton, J. Poznanovic, A. Ramanthan, S. R. Alam, P. S. Crozier, Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures, Concurrency and Computation: Practice and Experience 25 (10) (2013) 1356–1375. doi:10.1002/cpe.2943.
URL http://dx.doi.org/10.1002/cpe.2943

[7] NVIDIA, CUDA API Reference Manual 7.5 (2015).
URL http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[8] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, T. Narumi, DS-CUDA: A middleware to use many GPUs in the cloud environment, in: Proceedings of the 2012 SC Companion: High Performance Computing,

Networking Storage and Analysis, SCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1207–1214. doi:10.1109/SC.Companion.2012.146.
URL http://dx.doi.org/10.1109/SC.Companion.2012.146

[9] G. Giunta, R. Montella, G. Agrillo, G. Coviello, A GPGPU transparent virtualization component for high performance computing clouds, in: Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I, EuroPar'10, Springer-Verlag, 2010, pp. 379–391. doi:10.1007/978-3-642-15277-1_37.
URL http://dx.doi.org/10.1007/978-3-642-15277-1_37

[10] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, P. Ranganathan, GViM: GPU-accelerated virtual machines, in: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt '09, ACM, 2009, pp. 17–24. doi:10.1145/1519138.1519141.
URL http://doi.acm.org/10.1145/1519138.1519141

[11] C. Reaño, F. Silla, G. Shainer, S. Schultz, Local and remote GPUs perform similar with EDR 100G InfiniBand, in: Proceedings of the Industrial Track of the 16th International Middleware Conference, Middleware Industry '15, 2015, pp. 4:1–4:7. doi:10.1145/2830013.2830015.
URL http://doi.acm.org/10.1145/2830013.2830015

[12] S. Iserte, J. Prades, C. Reaño, F. Silla, Increasing the performance of data centers by combining remote GPU virtualization with Slurm, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 98–101. doi:10.1109/CCGrid.2016.26.
URL http://dx.doi.org/10.1109/CCGrid.2016.26

[13] C. Reaño, F. Silla, A performance comparison of CUDA remote GPU virtualization frameworks, in: 2015 IEEE International Conference on Cluster

Computing, 2015, pp. 488–489. doi:10.1109/CLUSTER.2015.76.
URL http://dx.doi.org/10.1109/CLUSTER.2015.76

[14] NVIDIA, Developing a linux kernel module using GPUDirect RDMA (2015).
URL http://docs.nvidia.com/cuda/gpudirect-rdma/index.html#ixzz42PAc6GGX

[15] Mellanox, OFED GPUDirect RDMA product brief (2014).
URL http://www.mellanox.com/related-docs/prod_software/PB_GPUDirect_RDMA.PDF

[16] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C. Chu, D. K. Panda, Exploiting GPUDirect RDMA in designing high performance OpenSHMEM for NVIDIA GPU clusters, in: 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015, 2015, pp. 78–87. doi:10.1109/CLUSTER.2015.21.
URL http://dx.doi.org/10.1109/CLUSTER.2015.21

[17] A. J. Younge, J. P. Walters, S. P. Crago, G. C. Fox, Supporting high performance molecular dynamics in virtualized clusters using IOMMU, SR-IOV, and GPUDirect, in: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '15, ACM, New York, NY, USA, 2015, pp. 31–38. doi:10.1145/2731186.2731194.
URL http://doi.acm.org/10.1145/2731186.2731194

[18] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, D. K. Panda, Efficient inter-node MPI communication using GPUDirect RDMA for infiniband clusters with NVIDIA gpus, in: 42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013, 2013, pp. 80–89. doi:10.1109/ICPP.2013.17.

[19] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE.
URL http://mvapich.cse.ohio-state.edu/

[20] D. Rossetti, Benchmarking GPUDirect RDMA on modern server platforms (2014).
URL http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/

[21] Mellanox, RDMA aware networks programming user manual (2015).
URL http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

[22] Y. Suzuki, S. Kato, H. Yamada, K. Kono, Gpuvm: Why not virtualizing gpus at the hypervisor?, in: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, 2014, pp. 109–120.

[23] A. Polak, Counting triangles in large graphs on GPU, CoRR abs/1503.00576.
URL http://arxiv.org/abs/1503.00576

[24] C. Reaño, F. Silla, A. Castelló, A. J. Peña, R. Mayo, E. S. Quintana-Ortí, J. Duato, Improving the user experience of the rCUDA remote GPU virtualization framework, Concurrency and Computation: Practice and Experience 27 (14) (2015) 3746–3770. doi:10.1002/cpe.3409.
URL http://dx.doi.org/10.1002/cpe.3409