CrossMark

Exploring the interoperability of remote GPGPU virtualization using rCUDA and directive-based programming models

Adrián Castelló[1] · **Antonio J. Peña[2]** · **Rafael Mayo[1]** · **Judit Planas[3]** ·
Enrique S. Quintana-Ortí[1] · **Pavan Balaji[4]**

Abstract Directive-based programming models, such as OpenMP, OpenACC, and OmpSs, enable users to accelerate applications by using coprocessors with little effort. These devices offer significant computing power, but their use can introduce two problems: an increase in the total cost of ownership and their underutilization because not all codes match their architecture. Remote accelerator virtualization frameworks address those problems. In particular, rCUDA provides transparent access to any graphic processor unit installed in a cluster, reducing the number of accelerators and increasing their utilization ratio. Joining these two technologies, directive-based programming models and rCUDA, is thus highly appealing. In this work, we study the integration of OmpSs and OpenACC with rCUDA, describing and analyzing several

✉ Adrián Castelló
   adcastel@uji.es

   Antonio J. Peña
   antonio.pena@bsc.es

   Rafael Mayo
   mayo@uji.es

   Judit Planas
   judit.planas@epfl.ch

   Enrique S. Quintana-Ortí
   quintana@uji.es

   Pavan Balaji
   balaji@anl.gov

[1] Universitat Jaume I de Castelló, 12071 Castellón de la Plana, Spain

[2] Barcelona Supercomputing Center (BSC-CNS), 08034 Barcelona, Spain

[3] École Polytechnique Fédérale de Lausanne, 1202 Geneva, Switzerland

[4] Argonne National Laboratory, Lemont, IL 60439, USA

applications over three different hardware configurations that include two InfiniBand interconnections and three NVIDIA accelerators. Our evaluation reveals favorable performance results, showing low overhead and similar scaling factors when using remote accelerators instead of local devices.

Keywords GPUs · Directive-based programming models · OpenACC · OmpSs · Remote virtualization · rCUDA

1 Introduction

The use of coprocessors and hardware accelerators has increased continuously in the past decade. In the most recent TOP500 ranking [1], dated November 2015, 104 supercomputers were equipped with accelerators. These devices offer performance improvements over traditional processors for computational kernels that match their architecture; moreover, they are more energy efficient, delivering relatively large MFLOPS/Watt ratios.

The adoption of coprocessors to accelerate general-purpose codes became more popular with the release of the CUDA programming model for NVIDIA graphics processing units (GPUs) [2]. CUDA includes both high- and low-level application programming interfaces (APIs) to leverage NVIDIA devices as general-purpose accelerators. Prior to that, researchers matched GPU architectures to computational kernels, but these had to rely on complex graphics-oriented APIs such as OpenGL [3] or Cg [4]. Following CUDA, OpenCL [5] arose as an effort to offer a cross-vendor solution. The third-generation programming model for accelerators, based on compiler directives, is composed (among others) by OpenACC [6], OmpSs [7], and OpenMP 4 [8].

The traditional approach to furnish clusters with GPUs is to populate every compute node with one or more of these devices. However, these configurations present a low utilization rate of the computational resources available in the hardware accelerators, mostly because of mismatches between the application's type of parallelism and the coprocessor architecture. Remote virtualization has been recently proposed to overcome the underutilization problem. Among the virtualization frameworks, the most prominent is rCUDA [9], which enables cluster configurations to be built with fewer GPUs than nodes. With this approach, GPU-equipped nodes act as general-purpose GPU (GPGPU) servers for the rest of the compute nodes of the cluster, accommodating a CUDA-sharing solution that potentially achieves a higher overall GPU load in the system. Compared with other CUDA and OpenCL virtualization frameworks (e.g., DS-CUDA [10], vCUDA [11], VOCL [12], or SnuCL [13]), rCUDA is a mature, production-ready framework that offers support for the latest CUDA revisions and provides wide coverage of current GPGPU APIs.

The combination of both technologies to execute directive-based accelerated applications on remote coprocessors is obviously appealing. Nevertheless, two problems need to be tackled in this approach: the overhead due to the interconnect (in terms of increased latency and decremented transfer rate) and the particular GPU-usage pattern imposed by the programming model (which in many cases is suboptimal compared with fine-tuned coding directly using the underlying accelerators API). Since

remote accelerator virtualization techniques have been demonstrated to be reasonably efficient only in certain programming patterns, in this paper we address the open question of whether remote virtualization is suitable for applications accelerated via directives. Specifically, we analyze the performance of two popular directive-based programming models, OpenACC and OmpSs, on top of rCUDA, using several hardware configurations. For this purpose, we select two well-known applications for OmpSs and the EPCC benchmark application-level implementation and CloverLeaf OpenACC implementation for OpenACC. Our study reveals affordable overheads and fair scaling trends for directive-based applications using remote accelerators instead of local hardware.

In summary, the main contributions of this paper include (1) an analysis of the challenges involved in integrating the directive-based programming models for accelerators and the rCUDA remote GPU virtualization framework, and (2) a study of the performance impact of using directive-based acceleration on top of virtualized remote accelerator technologies.

The rest of the paper is structured as follows: Section 2 provides background information on the technologies explored in this paper. Section 3 reviews work related to our research. Section 4 covers the integration efforts of rCUDA and our target runtimes. Section 5 introduces our testbeds in terms of hardware, software, and test codes. Section 6 presents our experimental evaluation, and Sect. 7 closes the paper with a brief summary and discussion of future work.

2 Background

2.1 Directive-based programming models for accelerators

Directive-based programming models comprise a collection of compiler directives that the programmer employs to identify the pieces of code to be accelerated. These directives instruct the compiler to process certain parts of the code to map that computation to the available architecture and, if needed, to perform the required data movements between host and device memories. These high-level programming approaches are an appealing interface for improving an application's performance (including legacy codes) with relatively little effort and code intrusiveness.

For our study, we selected two popular directive-based programming models for accelerators, OpenACC and OmpSs, as representative of fine-grained and coarse-grained parallelism approaches, respectively.

2.1.1 OpenACC programming standard

OpenACC [6] is a standard API developed by PGI, Cray, and NVIDIA that enables C, C++, or Fortran programmers to easily leverage heterogeneous systems equipped with a general-purpose CPU plus a coprocessor.

Current compilers for directive-based accelerators still report inferior performance with respect to that obtained when directly leveraging GPGPU APIs. The development
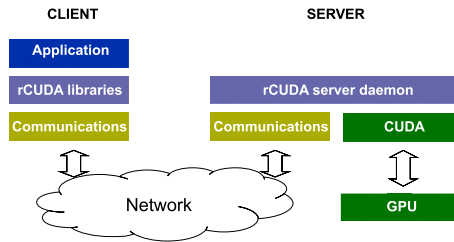
Fig. 1 rCUDA modular architecture

productivity of the former is clearly much higher, and both industry and research institutions are actively working to turn this into a broadly adopted solution.

2.1.2 OmpSs programming model

OmpSs [14], developed at the Barcelona Supercomputing Center, aims to provide an efficient programming model for heterogeneous and multicore architectures. Similarly to OpenMP 4.0 it embraces a task-oriented execution model.

OmpSs detects data dependencies between tasks at execution time, with the help of directionality clauses embedded in the code, and leverages this information to generate a task graph during the execution. This graph is then employed by the runtime to exploit the implicit task parallelism, via a dynamic out-of-order, dependency-aware schedule. This mechanism provides a means to enforce the task execution order without the need for explicit synchronization.

2.2 rCUDA framework

The rCUDA middleware [9] enables access to all GPUs installed in a cluster from all compute nodes. Figure 1 illustrates that rCUDA is structured following a client–server architecture. The client is installed in the compute nodes, and provides a transparent replacement for the native CUDA libraries. On the other hand, the server is executed in the nodes equipped with actual GPUs, providing remote GPGPU services. With this technology, the GPUs can be shared between nodes, and a single node can use all the graphic accelerators as if they were local. The aim is to achieve higher accelerator utilization while simultaneously reducing resource, space, and energy requirements [15,16].

The rCUDA client exposes the same interface as does NVIDIA CUDA [2], including the runtime and most of the driver APIs. Hence, applications are not aware that they are running on top of a virtualization layer. The middleware supports several communication technologies such as Ethernet [9] and InfiniBand [9].

3 Related work

Remote coprocessor virtualization technologies have been evaluated with various benchmarks and production applications in the past. The rCUDA prototype was pre-

sented along with early results for remote GPGPU acceleration from microbenchmarks based on the BLAS SGEMM and FFT kernels. Advanced features were introduced together with evaluations of several code samples from the NVIDIA SDK package. The literature containing these contributions is reviewed in [17]. More recently, the remote acceleration of production CUDA-enabled applications was evaluated with this framework, including LAMMPS and CUDASW++ [15]. Moreover, five applications from the CUDA Zone website were analyzed on top of the vCUDA framework [11]. DS-CUDA was presented for the REM-MD molecular dynamics simulator [10].

On the OpenCL side, the authors of the VOCL framework tested the SHOC benchmark suite and several application kernels, including the BLAS GEMM, matrix transpose, n-body, and Smith-Waterman [12]. Moreover, dOpenCL was released with the evaluation of a tomography [18] production application.

All prior evaluations were based on either traditional GPGPU APIs (CUDA and/or OpenCL) or custom high-level APIs specifically proposed for the framework in use. Many of these evaluations presented favorable results, reporting remote accelerations with low or negligible overheads. The studies, however, also identified codes that benefited from local acceleration but experienced unbearable overhead from remote acceleration because of the increased latency and possibly reduced bandwidth introduced by the interconnect.

In previous papers we reported a preliminary study of the interoperability between directive-based programming models for coprocessors and virtualized remote accelerators. In [19] we introduced a first study of simple OpenACC directives on top of rCUDA. In this paper we extend an in-depth analysis of the OmpSs programming model over virtualized remote GPUs [20] with an analogous study for the OpenACC programming model.

4 Integrating rCUDA with directive-based programming models

The rCUDA framework supports the whole CUDA runtime API and most of its driver API. However, supporting a high-level programming model may require some updates in the rCUDA framework because the different procedures adopted during code translation from a directive to CUDA code. Each scenario needs to be studied separately in order to avoid performance loss.

4.1 OmpSs and rCUDA

4.1.1 Necessary modifications

One of the necessary changes to rCUDA is the point where the CUDA kernels and functions are loaded. CUDA performs this initialization at the beginning of the application's execution, whereas in the OmpSs framework this event occurs the first time that a GPU task is issued for execution.

Because rCUDA was designed with CUDA in mind, its initialization procedure mimics that of CUDA; and this process occurs at the beginning of a CUDA application's execution. Nevertheless, when OmpSs uses CUDA, this mechanism is triggered

the first time a CUDA function is called; therefore, this behavior had to be modified in rCUDA. In particular, for each thread created on the client side, the first call to any kernel configuration routine triggers a load of the corresponding modules in the associated GPU server. From then on, any other routine configuration call directed to the same GPU server does not produce any effect. Several logical checks were added in order to ensure that all threads are able to execute the required functions. Once these checks are completed, the new loading mechanism avoids checking the function load status again.

4.1.2 Reducing the communication overhead

Current GPU boards from NVIDIA can promote idle GPUs from a high-performance state to an energy-saving passive one following the same approach that has been exploited in x86 architectures via the C-states defined in the ACPI standard [21]. When this occurs, the next CUDA call takes longer to start because the GPU driver first needs to reactivate the GPU and then execute the CUDA function. OmpSs favors high performance over energy efficiency, preventing GPUs from entering an energy-saving state via regular calls to the `cudaFree` function with no argument. This approach has no effect other than keeping the GPU active, even when there is no work to execute, and has the same effect as setting the GPU in persistence mode (e.g., by means of the nvidia-smi utility).

Fortunately, rCUDA does not need this mechanism, which would otherwise introduce certain communication overhead in the network by exchanging a number of short messages between the client and the remote GPU server. In particular the rCUDA server, which is a CUDA application per se, runs on the GPU server node, thus preventing the GPU from entering the passive state. We modified the rCUDA client middleware to intercept all `cudaFree` calls, forwarding them to the server only if they are true memory-free calls, or discarding the requests when they correspond to unnecessary activation commands. These commands are distinguished by the function argument: when it is a not NULL pointer; it is a real free call.

4.1.3 Dealing with synchronization

Task parallelism achieves high modularity by creating codes that can be totally different (e.g., I/O, computation, or communication) and executed in any order. Nonetheless, synchronization points are needed to maintain coherence and to control the execution flow. OmpSs uses the directive `#pragma omp taskwait` to ensure that all the previous tasks have been executed. However, adding synchronization points usually has a negative impact on the applications' performance, and the use of the named directive needs to be studied. The OmpSs framework translates this directive into a `cudaDeviceSynchronize` call, and the performance attained by CUDA and rCUDA in that scenario is crucial. rCUDA's synchronization mechanism outperforms that native in CUDA because the former accommodates a more aggressive implementation. Concretely, rCUDA's mechanism executes a nonblocking wait during a small period before the real synchronization call. If it is successful, the synchronization call is thus avoided. A detailed analysis can be found in [22].
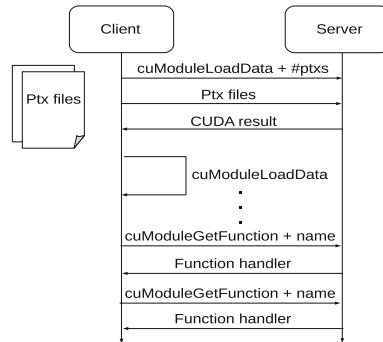
Fig. 2 Module load and function search in the new rCUDA module management

4.2 OpenACC and rCUDA

4.2.1 Necessary adaptations

We target a distributed environment where clients do not necessarily feature an actual GPU. Therefore, among the different options the PGI compiler offers to generate separate GPU modules, we choose to produce PTX files (`-keepptx`). This is a low-level source code that is compiled just in time on the GPGPU server to produce an executable optimized for the specific GPU architecture. The PGI compiler uses the following module management functions from the CUDA driver API, which the current rCUDA release does not support. `cuModuleLoadData`: This call loads an appropriate module for the target GPU architecture, comprising a set of GPU kernel functions, and makes it available for subsequent kernel executions. `cuModuleGetFunction`: This function searches the code implementing a given kernel name within the module loaded by the previous call and makes it available for subsequent use.

We implemented both functions in rCUDA and carefully tuned them for the distributed environment. Our `cuModuleLoadData` implementation allows the client to send all the GPU modules from the module repository of the executed application to the GPGPU server when the first call to this function is intercepted. Therefore, this mechanism is executed only once. These images are stored in contiguous memory addresses in the server to enhance efficiency in the `cuModuleGetFunction` call responses. This sequence of events is illustrated in Fig. 2.

4.2.2 Communication overhead

The PGI compiler performs data transfers between the host and the device, exploiting a double-buffer mechanism, as illustrated in Fig. 3a. This technique enables pipelined copies, overlapping transfers between the host memory and one of the buffers with those between the peer buffer and the accelerator memory. These intermediate buffers are registered as pinned memory, yielding faster transfers than if these were issued directly from pageable user memory. A similar mechanism is implemented by rCUDA
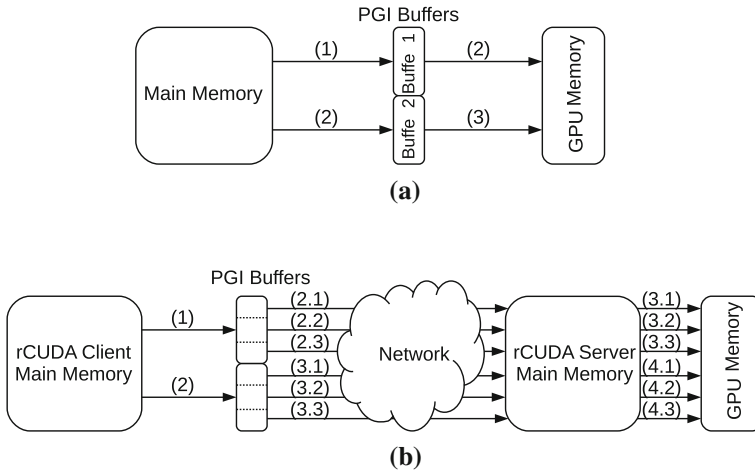
Fig. 3 PGI double-buffer mechanism. The *arrow* labels indicate the order of the corresponding transfer. **a** Using a local GPU (native). **b** Using a remote GPU. The example leverages PGI buffers three times bigger than those of rCUDA

when requested to perform transfers from pageable memory, but it then performs a direct transfer from the pinned buffers otherwise, avoiding the augmented latency and memory stress of an additional pipeline stage [9]. The data transfer process when using OpenACC on top of rCUDA is shown in Fig. 3b.

5 Experimental setup

We have selected three different systems:

– *Minotauro* is a cluster with 126 nodes, each with two Intel Xeon E5649 6-core processors, (2.53 GHz), and 24 GB of DDR3-1333 RAM, connected to two NVIDIA M2050 GPUs. The network is InfiniBand (IB) QDR.
– *Tintorrum* is composed of two compute nodes, each equipped with two Intel Xeon E5520 quadcore processors (2.27 GHz) and 24 GB of DDR3-1866 RAM memory. One of the nodes is connected to two NVIDIA C2050 GPUs; the remaining one is populated with four NVIDIA C2050. Internode communications are accomplished via an IB QDR fabric.
– *Argonne* consists of two compute nodes, each equipped with two Intel E5-2687W v2 8-core processors (3.40 GHz) and 64 GB of DDR3-1866 RAM. The GPGPU server is endowed with an NVIDIA Tesla K40m GPU. Both nodes are connected via an IB FDR interconnect.

In the experiments we employed rCUDA 5.0, with the modifications listed in previous sections, as the remote GPGPU virtualization framework. rCUDA servers are launched in each node where at least one GPU is physically installed exposing the real GPUs as virtualized remote accelerators. CUDA 6.5 was installed for the PGI compiler and for the OmpSs version 14.10. OmpSs was compiled using `g++ 4.4.7` in *Tintorrum* and `g++ 4.4.4` in *Minotauro*.

Support for OpenACC was obtained with the PGI (version 14.9) compiler from The Portland Group [23], since this is a complete commercial tool that implements most of the features in OpenACC 2.0.

5.1 Evaluation cases

The following OmpSs-enabled applications were selected for this study:

– *N-body* simulates a dynamical system of particles, under the influence of physical forces, and is widely used in physics and astronomy.
– *Cholesky factorization* is a crucial operation for solving dense systems of linear equations with a symmetric positive-definite coefficient matrix.

   For the OpenACC evaluation we used the following applications:

– *The EPCC OpenACC benchmark suite* [24] is divided into three levels focused on basic directives, well-known kernels, and applications.
– *CloverLeaf* [25] solves the compressible Euler equations on a Cartesian grid in two dimensions. Two CloverLeaf implementations based on either `kernels` or `loops` OpenACC directives were analyzed.

6 Experimental evaluation

All the results in the section are the average of 100 executions. The highest relative standard deviation observed in the experiments was around 12 %.

6.1 OmpSs

OmpSs implements workstealing between threads' workloads and uses device-to-device direct transfers via `cudaMemcpyPeer` functions that, unfortunately, are not supported by rCUDA. To obtain results for applications that include data movements between GPUs, we analyzed the overhead added by this type of data transfers. Two scenarios can be identified when remote GPUs are used. In the first case, remote GPUs are attached to the same server node (Fig. 4a), and the communication between them is done via the PCI-e bus. In the second case, GPUs are attached to different server nodes (Fig. 4b), and the communication is performed through the network interconnect.

6.1.1 N-body

The N-body test was executed in *Tintorrum* using 57,600 particles, corresponding to the largest volume that fits into the memory of a single GPU. This case study involves no GPU memory transfers, and thus we can compare directly the performance of the OmpSs-accelerated code executed on top of CUDA and rCUDA. The application runs in the 2-GPU node. Additional GPUs are "added" to the experiments by first populating the 4-GPU node with up to four server processes (one per remote GPU) and, from then on, mapping the additional servers to the 2-GPU node. The work is divided equally
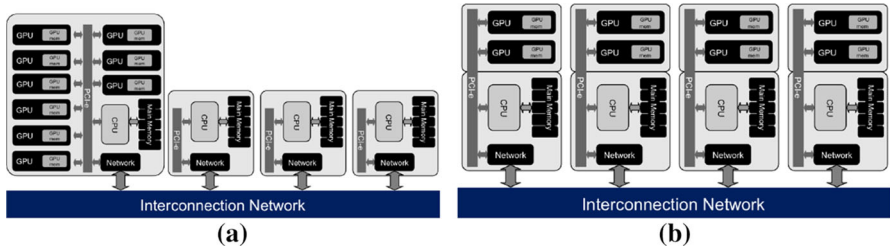
Fig. 4 rCUDA cluster configurations. **a** GPU pool configuration. **b** Several GPUs per node
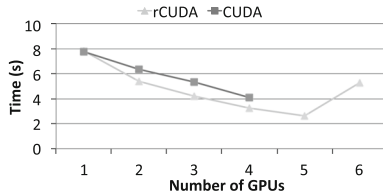

Fig. 5 Execution time for N-body using up to four local GPUs and up to six remote GPUs in *Tintorrum*
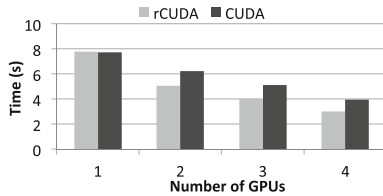

Fig. 6 Synchronization time for N-body using up to four local and remote GPUs in *Tintorrum*

among the GPUs, showing the following workload distribution: 100 % for remote GPUs when up to four are used; and 66–33 % for remote—local devices when all GPUs are employed. Figure 5 shows the outcome from this evaluation, showing a linear reduction of the execution time when up to five server processes/GPUs are employed. The speedups observed there demonstrate the scalability of this parallel application when combined with rCUDA to leverage a reduced number of GPUs. The same plot also exposes a notable increase in execution time when the sixth server process/GPU is included in the experiment. The reason is that, because of the architecture of the node with two GPUs, the transfers between this last GPU and the IB network occur through the QPI interconnect, which poses a considerable bottleneck for this application.

We also emphasize that the use of OmpSs on top of rCUDA clearly outperforms the alternative based on OmpSs on top of CUDA. Figure 6 demonstrates that the difference in favor of rCUDA is due to the overhead introduced with the synchronization that is enforced by OmpSs.

6.1.2 Cholesky factorization

The Cholesky factorization test was executed in *Minotauro* using a matrix of 45,056 × 45,056 single-precision elements. In this case, the GPUs performs direct

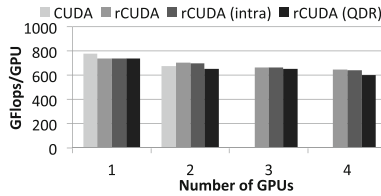**Fig. 7** Cholesky factorization using up to four GPUs in *Minotauro*
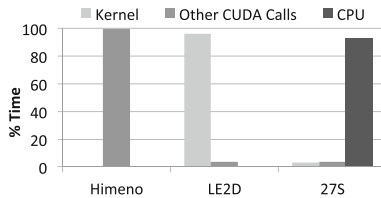

Fig. 8 Time distribution of the EPCC applications on *Argonne*

data transfers. Nevertheless, this feature is not yet supported in rCUDA. To overcome this deficiency, while still delivering a fair comparison between the scalability of OmpSs over rCUDA and over native CUDA, we determine the overhead introduced by a device-to-device communication for setup configurations corresponding to a pool of GPUs and several GPUs per node. This overhead helps us extrapolate the results by adding the network overhead to the real execution time.

In a preliminary evaluation, we detected that the optimal algorithmic block size for the Cholesky factorization using both CUDA and rCUDA is 2048, leading to data transfers of 16 MB. Next, we performed data transfers of 16 MB, using `cudaMemcpyPeer`, to simulate scenarios where both GPUs are in the same node, and using the `ib_send_bw` test included in the OFED-3.12 package to mimic a scenario where the GPUs are in different nodes. The results indicate that each data movement, respectively, added 0.192 and 5.283 ms to the rCUDA execution time. The study can be found in [20].

Using the native CUDA, we executed the code with one and two GPUs in a single node of the cluster. Using rCUDA, we employ up to four nodes with one GPU per node. Unfortunately, an internal problem in the current implementation of OmpSs when linked with NVIDIA's mathematical library CUBLAS impeded experimentation with a larger number of GPUs.

The three columns with the labels rCUDA, rCUDA (`intra`), and rCUDA (`QDR`) in Fig. 7 correspond to the execution of the task-parallel code, linked to rCUDA, with three different by-passes to deal with the lack of support for device-to-device copies by adding the overhead previously measured. In the first case, the copy is simply obviated so that this line reflects the peak performance that could be observed if the cost of this type of copies was negligible. In the second case, we assume that internode communications proceed at the rate of intranode copies. This result thus approximates the performance that could be observed in a configuration where all GPUs were in the same node. In the third case, all the device-to-device copies occur at internode speed.
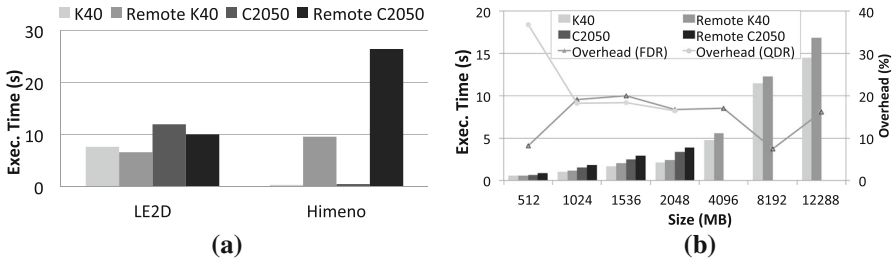
Fig. 9 Execution time for the EPCC application using local and remote GPUs. **a** LE2D and Himeno applications. **b** 27S application

The performance results for rCUDA in the chart clearly demonstrate the benefits of the remote virtualization approach for this application. We note that the lines there correspond to the GFLOPS per GPU and show a profile that is almost flat. Therefore, an increase in the number of GPUs by a factor of g implies an increase in the aggregated GFLOPS rate by almost the same proportion. The overhead of OmpSs over rCUDA with respect to OmpSs over CUDA, when executed using a single local GPU, is merely about 5 %.

Figure 7 also reveals a superlinear speed-up when just one GPU is used. The main reason is that before a CUDA application starts its execution, several steps need to be done (e.g., device initialization, libraries load, and context creation. As the rCUDA server is a CUDA application itself, all these procedures had been completed before the CUDA application's execution.

6.2 OpenACC

An early analysis involving a set of microbenchmarks that cover the basic OpenACC directives, including a scalability evaluation in terms of transfer sizes, was presented in [19]. In this section we explore the behavior of the applications.

6.2.1 EPCC applications

The tests in the EPCC benchmark collection are designed to enforce different accelerator usage patterns, representative of common scientific applications. Figure 8 shows different execution profiles of the benchmark examples. Himeno is a memory-bound application, where over 99 % of the execution time is spent in data transfers between the host and device memories. In contrast, LE2D spends more than 95 % of its time executing a computation directive. The 27S application spends roughly the same time in both types of tasks, but most of the time (over 90 %) is due to CPU utilization.

Figure 9a illustrates that an application such as LE2D clearly benefits from rCUDA thanks to the synchronization mechanism described in the previous section. On the other hand, the high volume of data transfers in the Himeno application (more than 117,000) reduces the performance of the rCUDA-based solution. Between these two cases, when the time of the data transfers is close to that of the kernel execution,
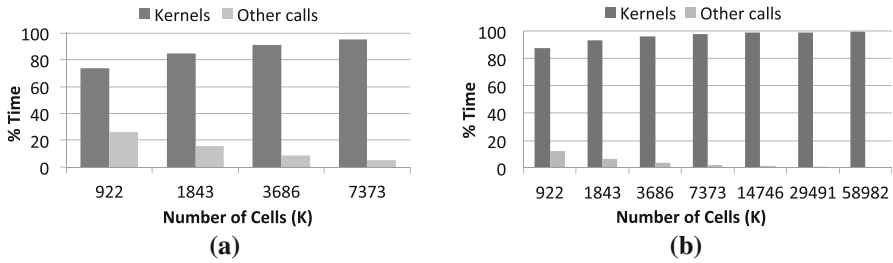
Fig. 10 Distribution of time for the CloverLeaf Kernels implementation. **a** System Tintorrum. **b** System Argonne
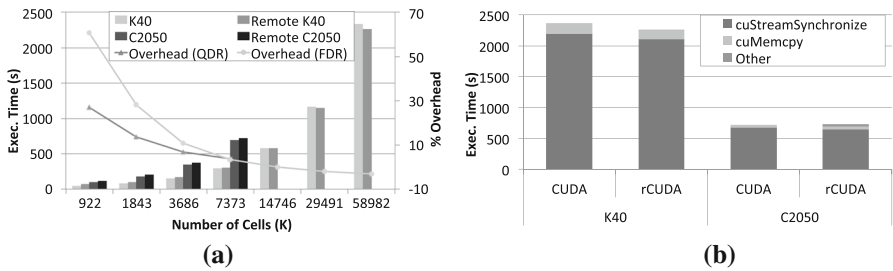

Fig. 11 CloverLeaf using local and remote GPUs. **a** Execution time with different data sizes. **b** Execution time of the main functions for maximum problem data sizes

the overhead added by the use of a remote GPU is visible but, as shown in Fig. 9b, generally small. In the 512 MB scenario, the overhead using the C2050 is close to 40 % because during the short execution time, the performance difference between the network and the PCIe bus cannot be compensated.

6.2.2 CloverLeaf

Figure 10 shows the distribution of execution time for the CloverLeaf miniapplication. "Kernels" represents the time spent executing computational directives, while "Other calls" represents the rest of the CUDA functions generated by the compiler, including CUDA initialization, data transfers, and synchronization mechanisms. This experiment shows that the portion of time that corresponds to the GPU computations increases with the problem size.

On the CloverLeaf Kernels implementation, while the execution time increases with the problem size, the overhead of using remote accelerators decreases from 60 (remote K40m) and 27 % (remote C2050) to 3 % for large sizes with the C2050 GPU and a negative overhead of −6 % for the K40m (Fig. 11a). Increasing the execution time compensates for the network usage by reducing the overhead.

The execution in the K40m GPU allows the evaluation of larger data sizes because of its larger memory. In this case the superlinear speedup brought by rCUDA becomes visible again. Figure 11b shows the difference between the CUDA and rCUDA synchronization mechanisms. For the K40m, the rCUDA implementation decreases the

execution time by 100 s, cancelling the network overhead. For the C2050 GPU, this reduction is not sufficient to compensate for the network overhead because the QDR interconnect is slower than the FDR network.

Similar conclusions are derived for the CloverLeaf LOOPS implementation, where we obtain results that are close to those in Fig. 11a: from 70 and 45 % to −2 and 6 % for the K40m and C2050, respectively.

7 Conclusions

Our main conclusion from this study is the viability of providing remote accelerator virtualization services on top of two directive-based programming models, OmpSs and OpenACC. The performance of using virtualized remote accelerators tightly depends on the interconnection network. The number of memory transfers, although not explicitly addressed by the programmers in directive-based programming models, is a key factor impacting the application performance. In our experiments, executions featuring a large number of data transfers suffered the most performance degradation in comparison with locally accelerated runs. Applications with high compute-per-data ratios benefit from GPU performances with respect to multicore CPU ones, hence favoring the remote acceleration solutions. In fact, we obtained the largest overheads for executions using fairly reduced dataset sizes. On the other hand, the use of synchronizations tends to reduce the overhead when using rCUDA with respect to locally-accelerated executions because it integrates a more aggressive polling implementation than the native CUDA library.

Because the characteristics of the rCUDA implementation, we have demonstrated that the overhead introduced by remote accelerator virtualization is more than compensated for a few relevant applications. These results indicate that considerable benefits are possible for production scenarios.

As part of future work, we plan to analyze multi-GPU and OpenACC-enabled applications, and to redesign rCUDA in order to accommodate device-to-device communications embedded for high performance in OmpSs. Moreover, we plan to analyze how the distribution of the accelerators in the cluster affects the behavior of these programming models on top of remote virtualization, as well as how to tackle this from the compiler perspective.

References

1. Strohmaier E, Dongarra J, Simon H, Meuer M (2015) TOP500 supercomputing sites. http://www.top500.org/lists/2015/11. Accessed Nov 2015
2. NVIDIA (2015) CUDA API reference, version 7.5
3. Shreiner D, Sellers G, Kessenich JM, Licea-Kane BM (2013) OpenGL programming guide: the official guide to learning OpenGL. Addison-Wesley Professional, Boston
4. Mark WR, Glanville RS, Akeley K, Kilgard MJ (2003) Cg: a system for programming graphics hardware in a C-like language. ACM Trans Graph (TOG) 22(3):896–907
5. Munshi A (2014)The OpenCL specification 2.0. 0.5em minus 0.4em Khronos OpenCL working group
6. OpenACC directives for accelerators (2015). http://www.openacc-standard.org. Accessed Dec 2015
7. OmpSs project home page. http://pm.bsc.es/ompss. Accessed Dec 2015
8. OpenMP application program interface 4.0 (2013). OpenMP Architecture Board
9. Peña AJ (2013) Virtualization of accelerators in high performance clusters. Ph.D. dissertation, Universitat Jaume I, Castellón
10. Kawai A, Yasuoka K, Yoshikawa K, Narumi T (2012) Distributed-shared CUDA: virtualization of large-scale GPU systems for programmability and reliability. In: International conference on future computational technologies and applications
11. Shi L, Chen H, Sun J, Li K (2012) vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans Comput 61(6):804–816
12. Xiao S, Balaji P, Zhu Q, Thakur R, Coghlan S, Lin H, Wen G, Hong J, Feng W (2012) VOCL: an optimized environment for transparent virtualization of graphics processing units. In: Innovative parallel computing. IEEE, New York
13. Kim J, Seo S, Lee J, Nah J, Jo G, Lee J (2012) SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: International conference on supercomputing
14. Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J (2011) OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21(02):173–193
15. Castelló A, Duato J, Mayo R, Peña AJ, Quintana-Ortí ES, Roca V, Silla F (2014) On the use of remote GPUs and low-power processors for the acceleration of scientific applications. In: The fourth international conference on smart grids, green communications and IT energy-aware technologies, pp 57–62
16. Iserte S, Castelló A, Mayo R, Quintana-Ortí ES, Reaño C, Prades J, Silla F, Duato J (2014) SLURM support for remote GPU virtualization: implementation and performance study. In: International symposium on computer architecture and high performance computing (SBAC-PAD)
17. Peña AJ, Reaño C, Silla F, Mayo R, Quintana-Ortí ES, Duato J (2014) A complete and efficient CUDA-sharing solution for HPC clusters. Parallel Comput 40(10):574–588
18. Kegel P, Steuwer M, Gorlatch S (2012) dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In: International parallel and distributed processing symposium workshops (IPDPSW)
19. Castelló A, Peña AJ, Mayo R, Balaji P, Quintana-Ortí ES (2015) Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. In: IEEE international conference on cluster computing
20. Castelló A, Mayo R, Planas J, Quintana-Ortí ES (2015) Exploiting task-parallelism on GPU clusters via OmpSs and rCUDA virtualization. In: IEEE international workshop on reengineering for parallelism in heterogeneous parallel platforms
21. HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd., Toshiba Corp. (2011) Advanced configuration and power interface specification, revision 5.0
22. Reaño C, Silla F, Castelló A, Peña AJ, Mayo R, Quintana-Ortí ES, Duato J (2014) Improving the user experience of the rCUDA remote GPU virtualization framework. Concurr Comput 27(14):3746–3770
23. PGI compilers and tools (2015) http://www.pgroup.com/. Accessed Dec 2015
24. Johnson N (2013) EPCC OpenACC benchmark suite. https://www.epcc.ed.ac.uk/. Accessed Dec 2015
25. Herdman J, Gaudin W, McIntosh-Smith S, Boulton M, Beckingsale D, Mallinson A, Jarvis SA (2012) Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In: SC companion: high performance computing, networking, storage and analysis