GPGPU Virtualisation with Multi-API Support Using Containers

John Walsh(✉) and Jonathan Dukes

School of Computer Science and Statistics, Trinity College Dublin,
The University of Dublin, College Green, Dublin 2, Ireland
{John.Walsh,Jonathan.Dukes}@scss.tcd.ie
http://www.scss.tcd.ie/

Abstract. Virtualisation of GPGPUs using PCI-Passthrough is limited to costly specialised hardware. API-Interception provides an alternative software-based approach to GPGPU virtualisation that has been shown to provide good performance and increased utilisation on High Performance and High Throughput Computing systems. Furthermore, user applications can transparently access many non-local GPGPU resources. However, current API-Interception implementations have either limited batch system support or none at all. This paper introduces a new multi-component system that supports multiple API-Interception implementations on several batch systems. The system consists of: (a) a factory component that produces lightweight Linux Containers using Docker, where each Container supports one or more API-Interception implementations and controls a single GPGPU; (b) a registry service that manages the Container resources; and, (c) a set of plugin scripts that bridge between the batch system and the registry. This paper also evaluates the performance of benchmarks on the prototype.

Keywords: Linux Container · Docker · Virtual GPGPU · API-Interception · Registry batch systems · Utilisation

1 Introduction

Over the last decade High Performance Computing (HPC) and High Throughput Computing (HTC) have seen a shift towards massively parallel computation using many-core accelerators, such as General Purpose Graphics Processing Units (GPGPUs) and Intel's Xeon Phi, as a way to boost application performance [6]. At the same time, Machine Virtualisation and Cloud Computing have led to a growth in loosely-coupled and highly-scalable parallel computing models such as Map/Reduce. Indeed other benefits of Cloud Computing include user-customisable execution environments and improved resource isolation. Although these features could be beneficial to HPC/HTC systems, HPC and Machine Virtualisation/Cloud Computing would appear to be somewhat juxtaposed – full Machine Virtualisation can decrease application performance [18]. Furthermore,

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 802–812, 2015.
DOI: 10.1007/978-3-319-27308-2_64

despite the benefits that Cloud can offer, it is not always practical or desirable to integrate cloud solutions (e.g. OpenStack) into existing HPC/HTC environments. However, an alternative approach using Containers (isolated process namespaces) looks promising [9,10,21].

This paper introduces a new lightweight system that adds a GPGPU virtualisation management layer on top of existing HPC infrastructures. This system virtualises GPGPUs (vGPGPU) and allows them be treated as *floating generic consumable resources* (FGCR), i.e. vGPGPU's are no longer logically bound to a machine. The extent to which a Local Resource Management System (LRMS), such as Torque [7]/MAUI [2], SLURM [5], can manage and schedule FGCR-based jobs is dependent on the choice of LRMS, so a new registry service has been developed to aide vGPGPU management. Two forms of GPGPU virtualisation are used in conjunction to create the vGPGPUs: lightweight machine virtualisation using Containers; and API-Interception. The former provides resource isolation whilst maintaining performance, and the latter allows one or more physical GPGPUs on remote machines to be seamlessly accessed as if they were local. Indeed, this form of combined GPGPU virtualisation may also help application developers avoid the need to use multiple APIs such as MPI. The proposed approach builds on existing virtualisation software to provide access to vGPGPUs. The contributions of this work are encapsulated in a system model consisting of: (a) a new vGPGPU Factory service to orchestrate the creation of vGPGPU VMs; (b) an external vGPGPU resource management service that enhances the existing LRMS and provides vGPGPU resource allocation where no such support exists in the LRMS; and (c) support for multiple API-Interception implementations and GPGPU hardware types within the one LRMS.

Section 2 looks at HPC and resource management, Cloud Computing for HPC, Virtualisation and GPGPU Virtualisation, and related work. This describes the influences and motivating factors behind the proposed model. The design and implementation is presented in Sect. 3, and two LRMS case-studies are discussed. Section 4 uses several benchmarks to examine the performance of the combined GPGPU vitualisation. Finally Sect. 5 summarises the objectives, how these have been achieved and the experimental findings before suggesting potential future work to improve and further evaluate the prototype.

2 Background and Related Work

HPC focuses on delivering the optimal computational power and capability possible, thereby allowing user jobs to execute as quickly as possible. HPC systems may be shared among hundreds or thousands of users from different scientific backgrounds and with different application needs. Such sharing requires management tools to optimise utilisation without overcommiting resources, and this is typically the responsibility of an LRMS (or *batch system*). The LRMS will queue user jobs until it determines that the specified job resources are available for it to execute. Typical resources include CPUs, but may also include network cards and GPGPUs that are implicitly bound to a machine called a worker-node (WN). A second type of resource, which can be accessed from any WN, is

called a *floating* resource. An example of a floating resource is a software licence. Non-floating resources can be configured either as a property of a WN, or they can be declared as a *generic consumable resource*. Properties are used to define the nature of a resource, whereas generic consumables are used to declare that the resource can be used concurrently on its associated WN a finite number of times. Floating Generic Consumable Resources (FGCR's) are used to declare that a set of resources can be used from any WN, but can only be used concurrently at most by the configured amount. FGCRs are often used to maintain a global count of how many times a finite resource is concurrently used. Support for micro-managing FGCRs may need to be provided by some system external to the LRMS. The level of support for integrating external resource management systems with an LRMS is not uniform.

Cloud Computing is geared towards loosely-coupled and on-demand computing tasks. It attempts to optimise utilisation of physical machine resources by allowing CPU, disk and other resources to be assigned to one or more Virtual Machines (VMs). VMs have their own independent machine environment, and they can be highly customised to execute specific user applications or services. Some Cloud Computing providers, such as Amazon, cater for HPC provisioning of hardware, including GPGPUs. Economic models have also shown the cost of running some HPC applications in a Cloud environment may be significantly cheaper than in a dedicated HPC environment [11]. However, the performance of HPC in Cloud is still an issue: namely, machine virtualisation has an impact on application performance.

These concerns can be alleviated by allowing the cloud management system to provision the physical machine (bare-metal provisioning), however this does not optimise resource utilisation. An alternative method using Containers avoids full virtualisation of the machine in software. Containers are restricted process namespaces executing on top of an existing operating system. They can have their own network address, and are used to allow processes to run as isolated micro-services. Executing processes in a Container has negligible impact on its performance. Furthermore, Containers can be configured to directly access individual hardware devices such as GPGPUs. Multiple solutions exist to build and deploy Containers [1,3]. Docker has received much attention because it allows container images to be layered upon one another, and it supports machine image templates. These facilitate rapid Container deployment.

GPGPU Virtualisation models can be classified into four categories: API-Interception, Kernel Device Passthrough, PCI-Passthrough, and PCI-Switching.

API-Interception is a software method for virtualising GPGPU resources. GPGPU hardware is normally accessed through calls to an API such as CUDA or OpenCL. These calls may be *intercepted* (or *hooked*) before being directed to a physical GPGPU. This technique is used to provide transparent access to remotely installed GPGPUs as if they were local. Remote virtual GPGPUs use a *frontend/backend* model in which the *frontend* intercepts all API calls (and their data) and transfers them over the network to a selected *backend* for execution. Several API-Interception implementations have been developed

for both CUDA (e.g. rCUDA [17], GridCUDA [15]) and OpenCL (VCL [8], dOpenCL [13], SnuCL [14]) runtime libraries. However, some not been actively developed in recent years and do not implement recent changes to their respective APIs.

Kernel Device Passthrough allows a GPGPU device (e.g. /dev/nvidia0) to be passed into a VM, and has the advantage of working with all GPGPUs. Impact on performance is negligible – 0 % for Containers [20] and 3 % for gVirtus [19].

PCI-Passthrough allows physical hosts to cede control of PCI-devices to a VM. This requires hardware support on the CPU, GPGPU, motherboard, and VM hypervisor. Relatively few GPGPUs can exploit this method. A recent study concluded that PCI-Passthrough has negligible performance impact [20].

PCI-Switching is a low-level technology that allows a physical machine equipped with additional specialist hardware to be assigned external PCI devices attached to the switch. Although this technology allows very flexible hardware (re)configurations, equipment cost is a significant disadvantage.

Some GPGPU virtualisation methods may be layered on top of each other, for example, a VM-encapsulated GPGPU using Kernel Device Passthrough, PCI-Passthrough or PCI-Switching can provide the first layer, with the API-Interception backend services providing the second layer. This combination allows the VM's GPGPU to be accessed remotely from a frontend node.

The prototype model presented in Sect. 3 is related to prior work that integrates rCUDA and VCL into SLURM. The weaknesses of these implementations are that: (i) they are SLURM specific; and (ii) if both systems are integrated with SLURM, both solutions attempt to manage the same vGPGPUs independently, so there are potential resource management conflicts. The prototype is designed to support several LRMSs and API-Interceptions technologies by using a single external vGPGPU management system. The use of LRMS prolog and epilogue scripts to extend the capabilities of the LRMS is inspired by ViBatch [16].

3 vGPGPUs as LRMS Resources

This paper proposes combining Kernel Device Passthrough (using Containers) with API-Interception, i.e. the Containers are assigned to a unique GPGPU, and one or more API-Interception software installed – these are *backend VMs*. Worker-nodes are configured at runtime as API-Interception *frontends*. The backend VMs are treated as a set of FGCRs, and are independent of the WNs. This section describes a model that facilitates API-Interception based vGPGPU resources usage on a range of different LRMSs. It consists of three new component parts: (a) the *vGPGPU Factory* subsystem creates backend VMs; (b) the Registry, which is used to aide both the installation and management of the backend VMs; and (c) a set of LRMS-specific script-based plugins that act as a bridge between the user's vGPGPU job, the LRMS, and the Registry.

These vGPGPUs should be easy to use, with much of the complexity hidden from the user. To aid this, a simple set of new key/value job attributes are

supported, namely: the number of nodes (CPU cores), the number of vGPGPUs per node, and the type of API-Interception used in the job.

3.1 The vGPGPU Factory

The vGPGPU Factory is a service that either creates new backend VMs or restarts existing ones. The service executes on nodes with physical GPGPU hardware, and starts by examining the hardware profile. Several properties (e.g. the GPGPU OS device name/number, the device vendor) are evaluated when a GPGPU is found. If a backend VM does not already exist, then a new one is constructed and labeled with a *Universally Unique Identifier* (UUID). The UUIDs are derived either directly from the GPGPU hardware (Nvidia), or constructed from a combination of the physical machine's hostname and the GPGPU's OS device name (AMD). The Nvidia and AMD hardware dependent variables (CUDA_VISIBLE_DEVICES or GPU_DEVICE_ORDINAL respectively) are set to the GPGPU's device number. Variables and GPGPU devices are passed into the VM at build time, helping to restrict access to the specified device. This construction method provides logical isolation of the VM's GPGPU. Finally, the GPGPU vendor value is used to determine which API-Interception software is installed and started on the VM. This is VCL for all Nvidia- and AMD-based VMs, and rCUDA for Nvidia-based VMs. The construction also ensures that multiple API-Interception virtualisation stacks are supported according to the hardware type. Docker [12] is used to build the VM and to install the rCUDA/VCL software. In addition, network bridging using Pipework [4] is used in preference to Docker's native Network Address Translation (NAT) solution because rCUDA did not function correctly under NAT, and because Pipework allows IP address assignment to the VM. In this way the VM's IP address can be managed through the Registry and assigned to the VM when it is initially created or instantiated.

3.2 vGPGPU Registry Service

The set of backend VMs form a pool of unmanaged resources. To add management capabilities, a new web-based service, the *vGPGPU Registry Service* (or *Registry*), has been developed. This helps manage two aspects of backend VMs: their life-cycle and their allocation to jobs. The Registry augments the resource management provided by the LRMS. The service implements a simple interface (Fig. 1), and the state of the backend VMs are maintained in a persistent database. The protocol and database schema are designed to be independent of LRMS implementations. The assignment of the individual backend VMs to a job is managed at runtime by requesting resources from the Registry. The request returns a list of backend VM IP addresses. In this prototype implementation the Registry interface is implemented using the HTTP protocol.

Method	Description
register	Registers a new vGPGPU container that is being added to the pool. **Input**: Container UUID, GPU Vendor, GPU Device Number; **Output**: IP address of Container
unregister	Unregisters a vGPGPU container that is being removed from the pool. **Input**:Container UUID; **Output**: None.
request	Request a vGPGPU allocation for a job. The Registry returns the information needed to construct the vGPGPU front-end. **Input**: JobID, Number of vGPGPUs, Node Number, API-Interception Method; **Output**: List of vGPGPU Container IP addresses.
release	Release a vGPGPU allocation when it is no longer required by a job. The released vGPGPUs become available for allocation to another job. **Input**: JobID; **Output**: None.
query	Return information such as the number of free vGPGPUs and the number of vGPGPUs supporting a specified API (e.g. CUDA, OpenCL). **Input**: QueryName, API-Interception Method; **Output**: Integer.

Fig. 1. vGPGPU Registry Service Interface

3.3 LRMS Integration

The LRMS plugin component is the only subsystem that requires specific customisation. Two LRMS use-cases demonstrate this integration: (i) SLURM; and (ii) Torque/MAUI. The key differences between the LRMSs affect how vGPGPU jobs are handled and scheduled – these include support for non-CPU resources (e.g. FGCRs), and how arbitrary job parameters are propagated into the job's execution environment. However, despite these differences, vGPGPU jobs depend upon three LRMS-independent factors: (i) the number of frontend nodes; (ii) the number of backend VMs required by each frontend; and, (iii) the API-Interception to be used. The prototype assumes a natural mapping between an LRMS node – which in practice is a CPU core – and a vGPGPU frontend node. This implies that the number of frontend nodes is specified by declaring the number of nodes (or cores) required. LRMS environment variables are used to define the number of backend VMs (*VirtualGPGPUPerNode*) and the API-Interception required (*VirtualGPGPUType*). These can be passed from the LRMS to the frontend WN, where they are used by job prolog/epilogue scripts. The scripts transparently hide the complexity of configuring the vGPGPU job environment and interact with the Registry to allocate backend VMs.

Use-case 1: SLURM

The flow of a SLURM-based vGPGPU job is illustrated in Fig. 2. In Step (1) the number of frontend nodes is specified by requesting normal SLURM nodes; backend VMs are specified by requesting one or more *vgpgpu* licences; and the VirtualGPGPUPerNode and VirtualGPGPUType variables are also *exported* to the WN environment. The job will remain in a waiting state until the specified number of nodes and vgpgpu licences are available – this is managed entirely by

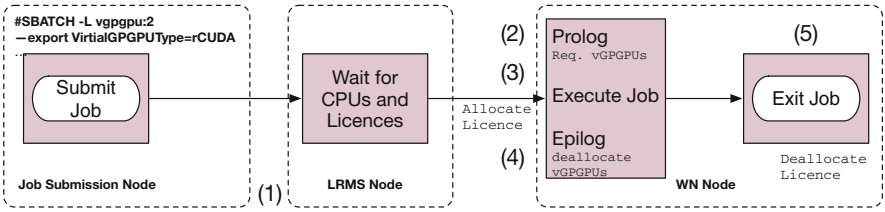

Fig. 2. The flow of a vGPGPU job through the SLURM LRMS

SLURM. During Step (2) a SLURM task prolog script will transparently execute. This checks that both VirtualGPGPUPerNode and VirtualGPGPUType are defined. If they are defined, then the prolog script requests a list of backend VMs from the Registry and then sets up the API-Interception execution environment. In Step (3) the GPGPU job will execute as normal; Finally, in Step (4), a SLURM task epilogue is transparently invoked to signal to the Registry that the backend VMs be made available for another job. However, the licence counter will not be incremented until the job exits in Step (5).

Use-case 2: Torque/MAUI

Torque/MAUI is a more complex use-case because it has limited support for generic consumable resources (implemented as a MAUI software patch), and limited support for FGCRs – it can only decrement a consumed resource by 1 at a time. To bypass these limitations, a new job pre-processing service and monitoring service have been developed. The flow of a Toruqe/MAUI vGPGPU job is illustrated in Fig. 3. In Step (1) the number of frontend nodes is defined to be the number of nodes; the VirtualGPGPUPerNode and VirtualGPGPUType are declared as variables, and these will be *exported* to the WN environment. In Step (2) a pre-processing filter examines the job definition; if the VirtualGPGPUPerNode and VirtualGPGPUType variables are set, an additional job directive is inserted to instruct Torque to place the job into a *holding* state; furthermore, the filter injects an additional call to a prolog script. The held job can only be released by an external *Monitor*. Once the job is released, Step (3), the prolog requests a list of backend VMs from the Registry and configures the job environment. Finally, in Step (4) the job is executed and the job exits. The Torque/MAUI implementation does not execute an epilogue script – backend VM recovery is left to the Monitor service.

The Monitor service continuously executes on the node where the LRMS runs. It has LRMS operator privileges, allowing it to unhold jobs. During each iteration, the Monitor implements garbage collection of completed vGPGPU job backend VMs and releases them for further use. The Monitor queries the number of free resources, and then iterates over the list of held jobs; if there are sufficient free backend VMs, then the Monitor requests that the required backend VMs are allocated to that job on its behalf, and the job is then released from its hold

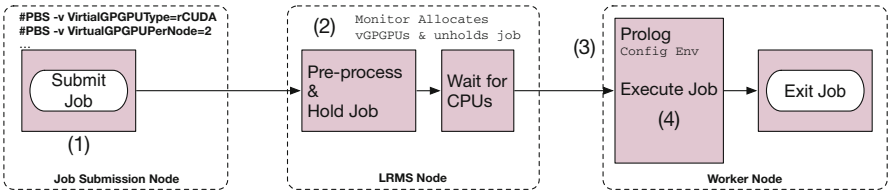

Fig. 3. The flow of a vGPGPU job through the Torque/MAUI LRMS

state. Unheld jobs must wait for available CPU cores. Only one job is released at a time, and this ensures there requests are free of race conditions.

4 Evaluation

To investigate whether the extra virtualisation layers (Docker and rCUDA/VCL) impact the performance and viability of the prototype, it is necessary to examine how applications behave. Two simple experiments were selected. The first examines the performance of a compute-intensive GPGPU application with minimal communication, while the second is a bandwidth intensive application that moves data from the WN to the GPGPU and back. Five scenarios were studied to help compare how each layer impacts performance, namely: (i) Native performance with direct access to the GPGPU (Local); (ii) WN access with rCUDA running locally on the WN (Local-rCUDA); (iii) WN access to a local GPGPU through rCUDA and a Docker container (Local-rCUDA/Docker); (iv) WN access to remote GPGPU using rCUDA only (Remote-rCUDA); and (v) WN access to a remote GPGPU using rCUDA and Docker (Remote-rCUDA/Docker). The network fabric uses 1-Gbit/Sec Cat5e Ethernet. The GPGPUs were Nvidia GTS 450s. The compute intensive application was executed 1,000 times for each scenario, while the bandwidth intensive application was executed 100 times.

Experiment 1:
The Black-Scholes application is provided by Nvidia to demonstrate how GPGPUs can be used to calculate the price of European financial market options. This application is distributed with the Nvidia CUDA Software Development Kit. Input values and initial conditions are hard-coded into the application, and a total of 8,000,000 options are calculated. There is minimal data transfer between the CPU and the GPGPU, so this application is a good indicator of how well an application will perform when it is not dependent on network I/O. The results in Table 1 show the total time taken to run each GPGPU scenario. The ratio between the time taken to run and the corresponding time taken on the local GPGPU is shown. The results consistently indicate that in both Local and Remote GPGPU cases, the combination of rCUDA and Docker has a negligible impact on the overall runtime in comparison to just using rCUDA alone.

Table 1. Execution data for Nvidia Black-Scholes application (1,000 invocations)

Access method	Total time (1000 jobs)	Relative performance	average per-job GPUTime (msec)	Average per-job memory bandwidth
Local	2773.52s	1.0	3.59	22.26 GB/s
Local-rCUDA	2810.99s	1.014	3.60	22.25 GB/s
Local-rCUDA/Docker	2831.12s	1.021	3.60	22.24 GB/s
Remote-rCUDA	3524.73s	1.271	3.60	22.24 GB/s
Remote-rCUDA/Docker	3524.69s	1.271	3.60	22.25 GB/s

Experiment 2:
The BandwidthTest application also comes from the Nvidia Software Development Kit. Its purpose is to measure the memcopy bandwidth of the GPGPU and memcpy bandwidth across the PCI-e bus. In the case of rCUDA and Docker, the application should generate significant network I/O that will have an impact on its performance. The results for these application runs are tabulated in Table 2.

Table 2. Execution data for Nvidia BandwidthTest application (100 invocations)

Access method	Total time (100 jobs)	Relative performance	Host to device bandwidth	Device to host bandwidth
Local	55.10s	1.0	3224.27 MB/s	3268.48 MB/s
Local-rCUDA	72.96s	1.32	2924.77 MB/s	1701.18 MB/s
Local-rCUDA/Docker	92.90s	1.69	1227.15 MB/s	1632.54 MB/s
Remote-rCUDA	665.05s	12.07	114.89 MB/s	120.61 MB/s
Remote-rCUDA/Docker	665.10s	12.07	114.87 MB/s	122.52 MB/s

The results show that even locally, rCUDA and Docker will have a noticeable impact on the application performance. The performance of remote GPGPUs under is very poor under 1-Gbit/Sec Cat5e Ethernet, with the bandwidth test taking over twelve times longer than running the same application locally.

5 Conclusions and Future Work

The prototype meets its core-objective to provide a lightweight model that integrates multiple API-Interception technologies into several batch systems. The experimental data shows that: (i) when local (i.e. contained to the same physical hardware) vGPGPU jobs execute computationally intensive applications, neither rCUDA nor Docker have an impact. There is a performance impact of

circa 25 % in both remote cases (where the frontend node is on separate hardware to the backend VM); and (ii) the bandwidth experiment results show that the performance degradation due to rCUDA and Docker is compounded at a local level, but Docker's impact is masked in the remote case.

Both results imply that the 1-Gbit/Sec Cat5e network infrastructure is problematic, and further tests are needed to see if any improvements can be made to the TCP/IP performance under both rCUDA and Docker. Further experiments also need to be carried out at a larger scale, and with low-latency networking such as Infiniband. The GPGPU locality results (Tables 1 and 2) indicate that if allocation preference were given to such local backends, then the job throughput may increase. This hypothesis has yet to be tested. Finally, the relationship between Nodes, CPU Cores and Cores per Node is more complex than that handled by the prototype, so further work is needed to accomodate a broader range of vGPGPU computing environment scenarios.

Acknowledgments. This work carried out on behalf of the Telecommunications Graduate Initiative (TGI) project. TGI is funded by the Higher Education Authority (HEA) of Ireland under the Programme for Research in Third-Level Institutions (PRTLI) Cycle 5 and co-funded under the European Regional Development Fund (ERDF).

References

1. LXC. https://linuxcontainers.org/
2. Maui. http://www.adaptivecomputing.com/resources/docs/maui/index.php
3. OpenVZ. http://openvz.org
4. Pipework. https://github.com/jpetazzo/pipework
5. SLURM. http://www.schedmd.com/slurmdocs/
6. Top 500 Supercomputers. http://www.top500.org/
7. Torque. http://www.adaptivecomputing.com/products/open-source/torque/
8. Barak, A., Shiloh, A.: The Virtual OpenCL (VCL) cluster platform. In: Proceedings of Intel European Research & Innovation Conference, p. 196 (2011)
9. Duran-Limon, H.A., Silva-Banuelos, L.A., Tellez-Valdez, V.H., Parlavantzas, N., Zhao, M.: Using lightweight virtual machines to run high performance computing applications: the case of the weather research and forecasting model. In: IEEE 4th International Conference on Utility and Cloud Computing, 2011, pp. 146–153 (2011). http://doi.ieeecomputersociety.org/10.1109/UCC.2011.29
10. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29–31, 2015, pp. 171–172 (2015). http://dx.doi.org/10.1109/ISPASS.2015.7095802
11. Gupta, A., Kalé, L.V., Gioachin, F., March, V., Suen, C.H., Lee, B., Faraboschi, P., Kaufmann, R., Milojicic, D.S.: The who, what, why, and how of high performance computing in the cloud. In: IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, pp. 306–314 (2013). http://dx.doi.org/10.1109/CloudCom.2013.47

12. Holla, S.: Orchestrating Docker. Packt Publishing, Birmingham (2015)
13. Kegel, P., Steuwer, M., Gorlatch, S.: dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems. J. Parallel Distrib. Comput. **73**(12), 1639–1648 (2013). http://dblp.uni-trier.de/db/journals/jpdc/jpdc73.html#KegelSG13
14. Ganesalingam, M.: Type. In: Ganesalingam, M. (ed.) The Language of Mathematics: A Linguistic and Philosophical Investigation. LNCS, vol. 7805, pp. 113–156. Springer, Heidelberg (2013)
15. Liang, T., Chang, Y.: Gridcuda: A grid-enabled CUDA programming toolkit. In: 25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2011, pp. 141–146 (2011). http://dx.doi.org/10.1109/WAINA.2011.82
16. Oberst, O., Hauth, T., Kernert, D., Riedel, S., Quast, G.: Dynamic extension of a virtualized cluster by using cloud resources. J. Phys. Conf. Ser. **396**(3), 032081 (2012). http://stacks.iop.org/1742-6596/396/i=3/a=032081
17. Peña, A.J., Reaño, C., Silla, F., Mayo, R., Quintana-Ortí, E.S., Duato, J.: A complete and efficient CUDA-sharing solution for HPC clusters. Parallel Comput. **40**(10), 574–588 (2014). http://dx.doi.org/10.1016/j.parco.2014.09.011
18. Regola, N., Ducom, J.C.: Recommendations for virtualization technologies in high performance computing. In: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 409–416. CLOUDCOM 2010 (2010). http://dx.doi.org/10.1109/CloudCom.2010.71
19. Vella, F., Cefala, R., Costantini, A., Gervasi, O., Tanci, C.: GPU computing in EGI environment using a cloud approach. In: International Conference on Computational Science and Its Applications (ICCSA) 2011, pp. 150–155 (2011)
20. Walters, J., Younge, A., Kang, D.I., Yao, K.T., Kang, M., Crago, S., Fox, G.: GPU passthrough performance: a comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications. In: IEEE International Conference on Cloud Computing, IEEE (2014)
21. Xavier, M.G., Neves, M.V., Rose, C.A.F.D.: A performance comparison of container-based virtualization systems for mapreduce clusters. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, pp. 299–306 (2014). http://dx.doi.org/10.1109/PDP.2014.78