# The High Performance Internet of Things: Using GVirtuS to Share High-End GPUs with ARM Based Cluster Computing Nodes

Giuliano Laccetti[1], Raffaele Montella[2(✉)], Carlo Palmieri[2], and Valentina Pelliccia[2]

[1] Department of Mathematics and Applications, University of Naples Federico II, Complesso Universitario Monte S. Angelo, Via Cintia, Naples, Italy
giuliano.laccetti@unina.it

[2] Department of Science and Technologies, Centro Direzionale di Napoli, Parthenope University of Napoli, Isola C4, 80143 Naples, Italy
{raffaele.montella,carlo.palmeiri,valentina.pelliccia}@uniparthenope.it

**Abstract.** The availability of computing resources and the need for high quality services are rapidly evolving the vision about the acceleration of knowledge development, improvement and dissemination. The Internet of Things is growing up. The high performance cloud computing is behind the scene powering the next big thing. In this paper, using the GVirtuS, general purpose virtualization service, we demonstrate the feasibility of accelerate inexpensive ARM based computing nodes with high-end GPUs hosted on x86_64 machines. We draw the vision of a possible next generation of low-cost, off the shelf, computing clusters we call Neowulf characterized by high heterogenic parallelism and expected as low electric power demanding and head producing.

**Keywords:** Hyerarchical parallelism · Hybrid algorithms · Adaptive algorithms · Multidimensional integration

## 1 Introduction

The Cloud Computing is an internet-based model in which virtualized and standard resource are provided as a service over the Internet. It provides a minimal management effort or service provider interaction and users interact with a virtual and dynamically scalable set of resources that can manage depending on their needs. Cloud Computing providers differ for the service provisioned and for the kind of the cloud architecture. The main consolidated service models are: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

The High Performance Computing (HPC) is one of the leading edge disciplines in information technology with a wide range of demanding applications in science [12,13], engineering, economy, medicine [1] and creative arts [7].

R. Wyrzykowski et al. (Eds.): PPAM 2013, Part I, LNCS 8384, pp. 734–744, 2014.
DOI: 10.1007/978-3-642-55224-3_69, 

The High Performance Cloud Computing (HPCC) model might offer a solution applying the elasticity concept of cloud computing to HPC resources, resulting in an IaaS delivery model. The cloud computing approach promises increased flexibility and efficiency in terms of cost, energy consumption and environmental friendliness [11] changing the point of view on performance contract systems [3].

Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for General-Purpose computing on the GPU). Especially in the field of parallel computing applications, virtual clusters instanced on cloud infrastructures suffers from the poorness of message passing performances between virtual machine instances running on the same real machine and also from the impossibility to access hardware specific accelerating devices as GPUs. Recently, scientific computing has experienced on general-purpose graphics processing units to accelerate data parallel computing tasks. Presently, virtualization allows a transparent use of accelerators as CUDA based GPUs, as virtual/real machines and guest/host real machines communication issues rise serious limitations to the overall potential performance of a cloud computing infrastructure based on elastically allocated resources using split-driver based components as GVirtuS [6].

The Internet of Things (IoT) services are build on the top of other services as a sort of construction game thanks to well documented public interfaces strongly leveraging on different web services technologies. IoT generally refers to uniquely identifiable objects and their virtual representations in an Internet-like structure. It is interesting consider a large number of this low power, low-performance processors teamed up to build a data center with similar processing power than regular CPUs, but smaller energy consumption. ARM processors, designed for the embedded mobile market, operate at about 1 GHz frequencies and consume just 0.25 W. There is already a significant trend towards using ARM processors in data servers and cloud computing environments. Those workloads are limited by the I/O and memory systems, not by the CPU performance. Recently, ARM processors are also taking significant steps towards increased double precision (DP) floating point (FP) performance, making them competitive with state-of-the-art server performance. The ARM Cortex-A15, targeted as the computing unit in the Barcelona Supercomputing Center Mont Blanc project, will increase super-scalar issue to two arithmetic instructions per cycle, and has a fully pipelined FMA unit, delivering 4 GFLOPS at 1 GHz, on potentially the same 0.25 W budget, achieving 16 GFLOPS/W. The new ARMv8 instruction set, which will be implemented in future generations of ARM cores, features a 64-bit address space, and adds DP to the NEON SIMD ISA1, allowing for 8 ops/cycle on an A15 pipeline: 8 GFLOPS at 1 GHz, for 32 GFLOPS/W.

In this paper we present our preliminary results in accelerating inexpensive HPC clusters, known as Beowulf clusters, made by off the shelf computing components using of low power ARM based computing nodes grouped in sub-clusters leveraging on one or more high-end GPGPU devices hosted on accelerator nodes. We perform some really promising experiments setting up a controlled testing environment imitating the core of a more complex architecture.

The rest of this paper is organized as follows: in the section two we draw out our vision of the next generation of really hybrid HPC clusters accelerated by Internet of Things based components and high-end GPUs; the third section deals with design and technical issues of the hybrid GPU/x86_64/ARM software architecture using GVirtuS as transparent bridge between the ARM living applications and the GPUs. The section number four is on implementation details, while in the one number five some tests and preliminary results are described and discussed. Finally, the last section, the sixth, is about the usual conclusions and future directions on those promising issues.

## 2 Vision and Contextualization

In the world of supercomputing the two top charts, Top 500 and Green 500, show, we have two trends: the number of core increases thanks the use of dedicated accelerators (GPUs, CPU array boards) and the compute/cost efficiency is increasing its important in the technology development, so, in the future the two charts will merge in just one considering the environmental (and economical) footprint of a HPC iron giant as a primary requirement. For many applications as operational computations [10] or for the cloud hosting providers the energy saving is no more a freak item but a mandatory issue. In the recent past a good amount of the world spread computing power has been achieved using the low/medium costs off the shelf Beowulf commodity clusters. A Beowulf is a cluster of machines interconnected by a high performance network employing the message-passing model for parallel computation. The key advantages of this approach are high performance for low price, system scalability and rapid adjustment to new technological advances. The latter point is the key for the next step of the Beowulf evolution in the vision described in this paper. As the now days CPU computing power increases, the need for electric power rises needing more cooling. The availability of Internet of Things derived ARM CPUs in their high performance incarnation (64 bit, multicore) lead the HPC world to ARM based clusters powered with on chip or on board GPUs. The idea we show here is dedicated to the low-end / middle-end in house solutions designing what could be defined as Neowulf the next generation of Beowulf clusters (Fig. 1).

The computing nodes of a regular old-style cluster behave as input/output nodes for ARM based inexpensive sub-clusters. In this way the amount of heat producers decrease while the high computing power demanding applications have to be refactored in order to fit this new heterogenic approach. Tanks to the software component we show in this paper, these devices are seen by each of the ARM based sub-cluster computing nodes as directly connected to them in a transparent way. This vision permits to gain more computing power reducing the expensive, power hungry and heat producer x86_64 based computing nodes, increase the parallelism at the sub-cluster level and, last but not the least, unchain the high-end GPGPU power to ARM based computing nodes.

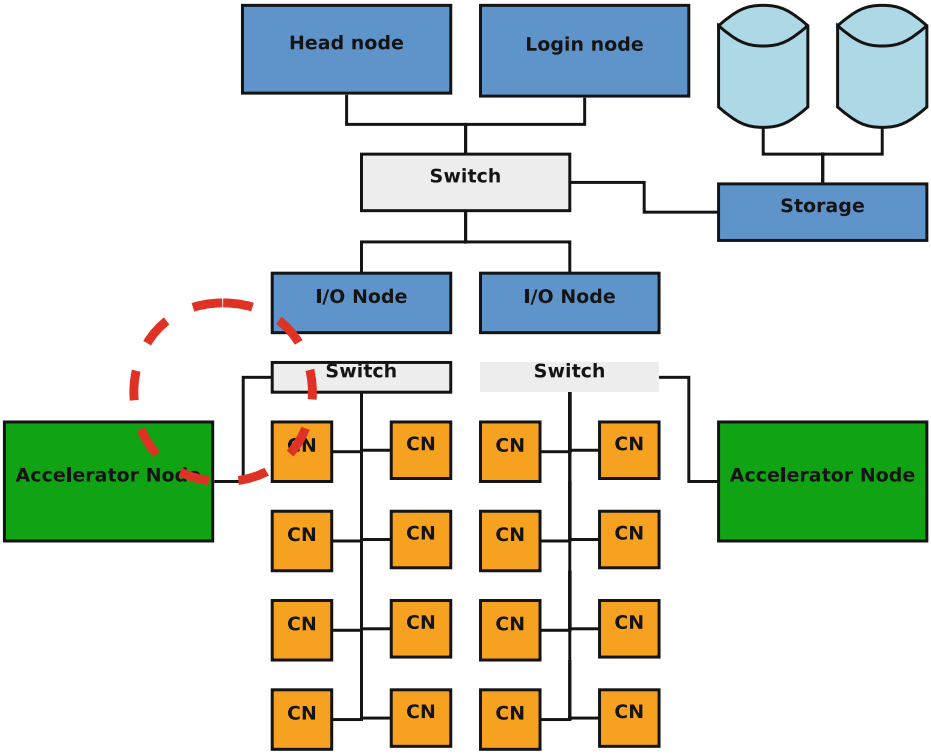


**Fig. 1.** The "Neowulf" big picture.

## 3 Design and Technical Issues

We use the GVirtuS framework model in order to design of our split driver implementation classically parted in front-end, communicator and back-end.

The front-end is a kernel module that uses the driver APIs supported by the platform. The interposer library provides the familiar driver API abstraction to the guest application. It collects the request parameters from the application and passes them to the back-end driver, converting the driver API call into a corresponding frontend driver call. When a callback is received from the frontend driver, it delivers the response messages to the application. In GVirtuS the front-end runs on the virtual machine instance and its implemented as a stub library.

The communicator maps the request parameters from the shared ring and converts them into driver calls to the underlying wrapper library. Once the driver call returns, the backend passes the response on the shared ring and notices the guest domains. The wrapper library converts the request parameters from the backend into actual driver API calls to be invoked on the hardware. It also relays the response messages back to the backend. The driver API is the vendor provided API for the device. The back-end is a component serving frontend

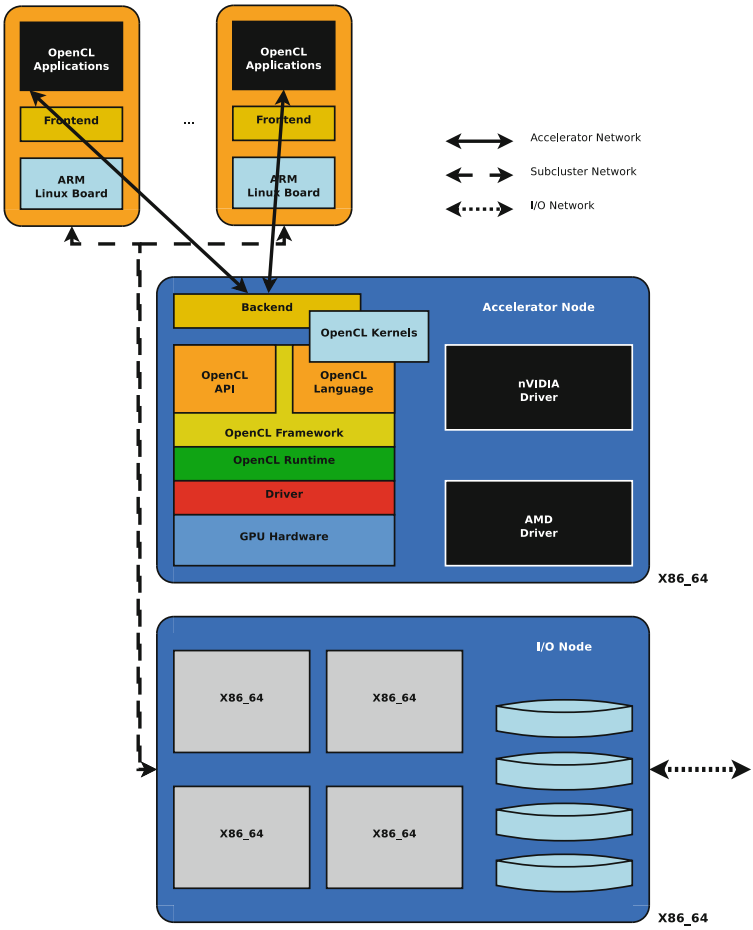


**Fig. 2.** The GVirtuS on ARM block diagram.

requests through the direct access to the driver of the physical device. This component is implemented as a server application waiting for connections and responding to the requests submitted by frontends. In an environment requiring shared resource the back-end must offer a form of resource multiplexing. Another source of complexity is the need to manage multithreading at the guest application level (Fig. 2).

### 3.1 GVirtuS on ARM

The GVirtuS porting on arm idea raised from different application fields such as High Performance Internet of Things (IPIoT) and HPC. In HPC infrastructures the ARM processors are used as computing nodes often provided by tiny GPU on chip or integrated on the CPU board. We developed the idea to share one or

more regular high-end GPU devices hosted on a small number of x86 machines with a good amount of low power/low cost ARM based computing sub-clusters better fitting into the HPC world.

From the architectural point of view this is a big challenge because involving word size, endianness and programming models. For our prototype we used the 32 bits ARMV6K processor supporting both big and little endian so we had to set the little endian mode in order to make data transfer between the ARM and the x86 full compliant. Due to the prototypal nature of the system all has been set to work using 32 bits. The solution is the full recompilation of the framework with a specific reconfiguration of the ARM based system. As we will migrate on 64 bits ARMs this point will be revise.

In a previous work we used GVirtuS as nVidia CUDA virtualization tool achieving good results in terms of performances and system transparency [5]. In order to fit the GPGPU/x86_64/ARM application into our generic virtualization system we mapped the back-end on the x86_64 machine directly connected to the GPU based accelerator device and the front-end on the ARM board(s) using the GVirtuS tcp/ip based communicator.

We chose to design and implement a GVirtuS plugin implementing OpenCL. This have been strongly motivated by several issues:

1. Since the CUDA version 4 the library design appears to be made not fitting with the split driver approach on which leverages GVirtuS and other similar products [];
2. The OpenCL is intrinsically open and all interfaces are public and well documented and, above all, work with nVidia devices, but is not limited to a particular vendor or architecture as GVirtuS itself;
3. OpenCL applications can be compiled directly on the ARM board without any installation of ad hoc libraries.

### 3.2 GVirtuS - OpenCL Plugin

OpenCL (Open Computing Language) is an open standard and royalty-free allowing to perform multi/single core generale purpose programming on highly heterogeneous systems. OpenCL allows developers to write their code once and run on CPUs and GPUs and different accelerator boards as mic based Intel Phi. In order to access a GPU in a virtual environment has been developed a wrapper for libOpencl.so. The virtualized library has the same interface of the original one and the independence from the communicator is guaranteed. The compatibility between the virtualized interface and libOpenCL.so allow the users to get a transparent virtualization system to run OpenCL applications. It is possible to run any of OpenCL applications without writing or recompile anything. Each GVirtuS OpenGL plugin components participate as follows:

Front-end side: For each OpenCL routine a stub method has been implemented with the same interface of the original one. All the stubs method have a common implementation consisting in the next five steps:

- Create a connection between back-end and front-end and flush all the buffers;
- Each parameters will be sent to the back-end through the input buffer;
- Request the execution of a routine using its name as parameter;
- Get and Use the exit code only if the execution is successful;
- Return the exit code the same one as the OpenCL routine.

Back-end side: Back-end has a stub method for each OpenCL routine in order to handle the frontend requests. All the handlers method have a common implementation consisting in the next five steps:

- Deserialize all the parameters from the input buffer;
- Execute the OpenCL routine and store the exit result;
- Insert the output parameters in a new buffer;
- Create an object Result containing the previous created buffer and the exit code;
- Exit and deliver the result to the frontend.

There are tree main input parameters types available:

- Host Pointer: back-end and front-end have different addressing space so a valid pointer on the front-end is invalid on the back-end and vice-versa. Aligning the addressed region makes the address translation.
- Device Pointer: the memory address is sent to the back-end or front-end. There is no need for translation because both, be and fe, refer to the device addressing space.
- Variables: It is really simple to add a scalar variable as a parameter.

In order to make the implementation effective and high performance, but with a good trade off in development straightforwardness we deeply used an OOP coding approach.

## 4 Implementation

The implementation, in C++ for all components, on the back-end side is related to an x86-based multi-core hardware platform with multiple accelerators attached via PCIe devices, running Linux as both host and guest operating system. In the font-end we used the same core running in a similar, but ARM based, Linux environment.

### 4.1 OpenCLFrontend

The OpenCLFrontend class establishes connections with the back-end and executes the OpenCL routine through the compiled library libGvirtus-frontend. The constructor method creates an object of the class Frontend from the libGvirtus-frontend library using the method GetFrontend using a factory/instance design pattern. All the stubs methods have a common schema. Every stub follows the

same interface of the handled OpenCL routine. The first step is to get the unique instance of the GVirtus Frontend class. This task is accomplished by the constructor method. The Prepare method reset the input buffer that will contain the parameters to send to the back-end. After that all the parameters are inserted in to the input buffer. The execute method forward the request for the routine using the name of the routine as parameter. If the method is successfully executed so we can get the output parameters. At last the method GetExitCode returns the exit code of the routine executed by the backend. The clGetDeviceIDs routine can be used to obtain the list of available devices on a platform. This simple explicative schema is common to all the stubs coded.

### 4.2 OpenCLBackend

The main task of GVirtuS back-end is to start a communication in server mode and waiting then accepting new incoming connections. It handles the loading of plugins previously installed. GVirtuS back-end invokes the GetHandler method in order to create a new instance of OpenclHandler class containing all the methods needed in order to serve the requests of OpenCL routine execution. In this class its possible to find all the methods to handle the execution of OpenCL routines. In the OpenclHandler class there is a table, mpsHandlers, associating function pointers to the name of the routines, so any routine can be handled in the right way. As in the front-end there is a stub method for each OpenCL method, in the back-end there is a function managing the execution of each method.

## 5 Evaluation

We set a prototypal hardware environment in order to evaluate the performance on ARM acceleration using external x86_64 GPUs, the GVirtuS overhead and the result reliability of a software testing suite. That evaluation process has two specific goals: (1) check the software stack accountability; (2) gather results on performance test. The OpenCL SDK provides a software suite which each component performs computations in bot CPU and GPU modes checking the result coherence and showing the brute performance results. All tests available on the standard OpenCL SDL have been successfully run using the GVrtuS-OpenCL SDK. We used a Raspberry Pi Mod.B rev.2 ARM 11 equipped with Wheezy Raspbian Linux as computing node and a Genensis GE-i940 Tesla powered by an i7-940 2.93 GHz fsb, Quad Core HT 8 Mb cache with one nVIDIA Qudro FX5800 4 Gb as GP device and two nVIDIA Tesla C1060 4 Gb as GPGPU device as accelerator node. For those tests no I/O node has been provided and the setup is related on a single node sub-cluster. In this context the GVirtuS fron-end was run on the ARM computing nodes while the back-end has been executed on the acceleration node. We used the OpenCL version of the testing software known as MatrixMul, DotProduct and Histogram (Fig. 3). ScalarProd computes k scalar products of two real vectors of length m. Notice that an OpenCL thread on

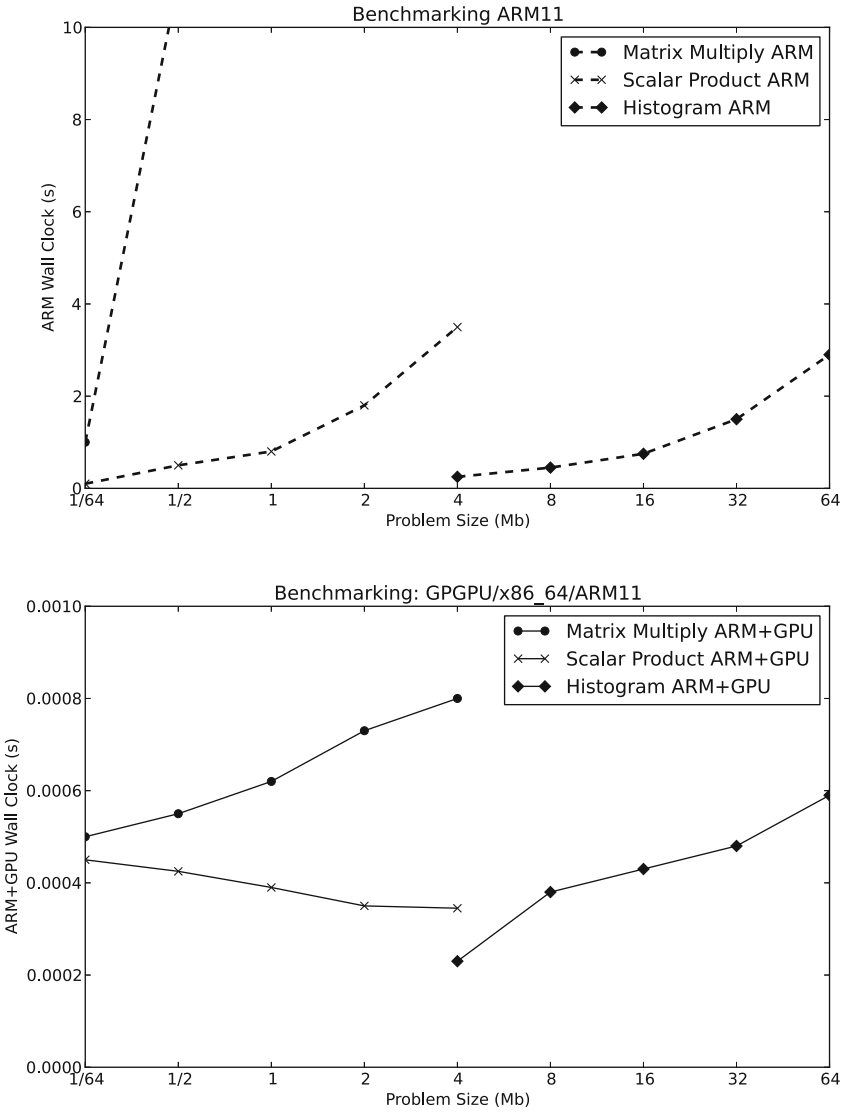


**Fig. 3.** ARM CPU without (up) and with (down) GPU acceleration.

the GPU executes each product so no synchronization is required. MatrixMul computes a matrix multiplication. The matrices are m n and n p, respectively. It partitions the input matrices in blocks and associates a OpenCL thread to each block. As in the previous case, there is no need of synchronization. Histogram returns the histogram of a set of m uniformly distributed real random numbers in 64 bins. The set is distributed among the OpenCL threads each computing a local histogram. The final result is obtained through synchronization and reduction

**Table 1.** Performance tests results.

| Test | Input size (MB) | Relative (%) |
|---|---|---|
| MatrixMul | 8 | 0.04 % |
| DotProduct | 16 | 0.27 % |
| Histogram | 64 | 0.65 % |

techniques. The Table 1 is a synthesis of the obtained results considering the regular ARMV6K as the reference:

During the DotProduct testing process we change the problem dimension from $2^{20}$ to $2^{22}$. The ARM performance are varying with the same problem dimension trend. The wall clock remains almost constant when is used the GPU acceleration. This demonstrates that the GVirtuS-OpenCL is fine working and the performances are not affected by the communication time. In the MatrixMul test the problem dimension has been varied in this steps $2^6 \times 2^9$, $2^9 \times 2^{12}$ and $2^{10} \times 2^{11}$. The performance results are pretty similar to the previous case with the GPU version having wall clock times almost unchanged. The Histogram has been used varying the problem size to $2^4$, $2^5$ and $2^6$. The results are trivially the same.

## 6 Conclusions and Future Directions

In this paper has been presented our preliminary results about the design and the implementation of an OpenCL wrapper library as GVirtuS framework plugin. The most challenging result achieved by our work is the implementation of a base tool unchaining the development of really distributed and heterogenic hardware architectures and software applications. The experiments we performed validate our promising vision. The incredible performance results we achieved, the wall clock using acceleration is less than the 1 % compared with the non-accelerated ARM board, have been affected by the computing power of the ARM side: they need for more investigation and developments. The next step will be setup a sub-cluster made by high performance ARM based boards provided by multicore ARM 64 bit CPUs and high bandwidth network interfaces. We expect some improvements from the ARM side, but even a better scalability because a more performing communication. In this scenario some other actors will get playing as the use of MPICH [2] for ARM to ARM and ARM to x86_64 message passing, the OpenMP for intra ARM board parallelism and, above all, one or more GPU devices hosted on the accelerator node have to be multiplexed by several ARM processes. As long range future directions we planned a complete reverse of the point of view has been planned: using GVirtuS components in order to abstract and virtualize the ARM HPC sub-cluster acting as an accelerator board for x86_64 machines and applications on instruments shared on the cloud [4,8].

## References


1. Boccia, V., D'Amore, L., Guarracino, M.R., Laccetti, G.: A grid enabled PSE for medical imaging: experiences on MedIGrid. In: Proceedings - IEEE Symposium on Computer-Based Medical Systems, pp. 529–536 (2005)
2. Gregoretti, F., Laccetti, G., Murli, A., Oliva, G., Scafuri, U.: MGF: a grid-enabled MPI library. Future Gener. Comput. Syst. **24**(2), 158–165 (2008)
3. Caruso, P., Laccetti, G., Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 982–992. Springer, Heidelberg (2005)
4. Di Lauro, R., Lucarelli, F., Montella, R.: SIaaS-sensing instrument as a service using cloud computing to turn physical instrument into ubiquitous service. IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 861–862. IEEE (2012)
5. Giunta, G., Montella, R., Laccetti, G., Isaila, F., Blas F.J.G.: A GPU accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory. In: Dr. Constantinescu, Z. (ed.) Advances in Grid Computing. ISBN: 978-953-307-301-9, InTech (2011)
6. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU transparent virtualization component for high performance computing clouds. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 379–391. Springer, Heidelberg (2010)
7. Maddalena, L., Petrosino, A., Laccetti, G.: A fusion-based approach to digital movie restoration. Pattern Recogn. **42**(7), 1485–1495 (2009)
8. Montella, R., Agrillo, G., Mastrangelo, D., Menna, M.: A globus toolkit 4 based instrument service for environmental data acquisition and distribution. In: Proceedings of the 3rd International Workshop on Use of P2P, Grid and Agents for the Development of Content Networks, pp. 21–28. ACM (2008)
9. Montella, R., Coviello, G., Giunta, G., Laccetti, G., Isaila, F., Blas, J.G.: A general-purpose virtualization service for HPC on cloud computing: an application to GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 740–749. Springer, Heidelberg (2012)
10. Montella, R., Giunta, G., Laccetti, G.: Multidimensional environmental data resource brokering on computational grids and scientific clouds. In: Furht, B., Escalante, A. (eds.) Handbook of Cloud Computing, pp. 475–492. Springer, New York (2010)
11. Montella, R., Foster, I.: Using hybrid grid/cloud computing technologies for environmental data elastic storage, processing, and provisioning. In: Furht, B., Escalante, A. (eds.) Handbook of Cloud Computing, pp. 595–618. Springer, New York (2010)
12. Murli, A., Boccia, V., Carracciuolo, L., D'Amore, L., Laccetti, G., Lapegna, M.: Monitoring and migration of a PETSc-based parallel application for medical imaging in a grid computing PSE. In: Gaffeny, P.W., Pool, J.C.T. (eds.) Grid-Based Problem Solving Environments. IFIP, vol. 239, pp. 421–432. Springer, Boston (2007)
13. Pham, Q., Malik, T., Foster, I., Di Lauro, R., Montella, R.: SOLE: linking research papers with science objects. In: Groth, P., Frew, J. (eds.) IPAW 2012. LNCS, vol. 7525, pp. 203–208. Springer, Heidelberg (2012)