Project Nº: 644312

D3.4 Specifications of Accelerator Infrastructures

September 30, 2015

Abstract:

This deliverable describes the main issues about RAPID and the cloud infrastructure for computing life support with particular regard to the computational resources granting the overall quality of service and the general purpose GPU virtualization and remoting components.

Document Manager	
Giulio Giunta	UNP

Document Id N°:	rapid_D3.4	**Version:**	1.9	**Date:**	2/11/2015

Filename:	rapid_D3.4_v1.9.docx

Confidentiality

This document contains proprietary material of certain RAPID contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information

The RAPID Consortium consists of the following partners:

Participant no.	Participant organisation names	short name	Country
1	Foundation of Research and Technology Hellas	FORTH	Greece
2	Sapienza University of Rome	UROME	Italy
3	Atos Spain S.A.	ATOS	Spain
4	Queen's University Belfast	QUB	United Kingdom
5	Herta Security S.L.	HERTA	Spain
6	SingularLogic S.A.	SILO	Greece
7	University of Naples "Parthenope"	UNP	Italy

Revision history

Version	Author	Notes	Date
0.1	Carlo Palmieri (UNP) Carmine Ferraro (UNP)	Table of Contents	08/09/2015
0.2	Giulio Giunta (UNP) Raffaele Montella (UNP)	Removed: Communication protocol specification Added: Executive Summary (draft)	14/09/2015
0.3	Giulio Giunta (UNP) Raffaele Montella (UNP)	Removed: Public Cloud Interaction Policy Added: Architecture, GVirtuS VM development Detailed: Introduction	14/09/2015
0.4	Giulio Giunta (UNP) Raffaele Montella (UNP)	Added chapter 4 on GVirtuS preliminary content	17/09/2015
0.5	Giulio Giunta (UNP) Raffaele Montella (UNP)	Chapter 4 revision and new content	18/09/2015
0.6	Giulio Giunta (UNP) Raffaele Montella (UNP)	Added preliminary introduction and conclusions	22/09/2015
0.7	Giulio Giunta (UNP) Raffaele Montella (UNP) Terpsi Velivassaki (SILO) Francisco Nieto (ATOS)	Integration with SILO and ATOS contribution	06/10/2015
0.8	Giulio Giunta (UNP) Raffaele Montella (UNP)	Added paragraph 4.2.2 Added chapter 4.3	15/10/2015
0.9	Giulio Giunta (UNP) Raffaele Montella (UNP)	Introduction and conclusions review. Final integration and last checking before partner review	16/10/2015
1.0	Giulio Giunta (UNP) Raffaele Montella (UNP)	Integration with comments from reviewers	22/10/2015
1.1	Giulio Giunta (UNP) Raffaele Montella (UNP)	RAPID deliverable from model Second iteration in comments integrations	26/10/2015

		Reordering the ToC Table of figures consistency check New introduction and conclusions.	
1.2	Terpsi Velivassaki (SILO)	Incorporating changes based on peer-review comments	27/10/2015
1.3	Francisco Nieto (ATOS)	Applied comments. Added new table 8 about hypervisors	28/10/2015
1.4	Giulio Giunta (UNP) Raffaele Montella (UNP)	Reference ordered by text	29/10/2015
1.5	Giulio Giunta (UNP) Raffaele Montella (UNP)	Pre-final version.	30/10/2015
1.6	Sokol Kosta (UROME)	Review of the pre-final version.	31/10/2015
1.7	Giulio Giunta (UNP) Raffaele Montella (UNP)	Incorporated comments of “pre-final”.	1/11/2015
1.8	Terpsi Velivassaki (SILO)	Incorporated comments from UROME.	2/11/2015
1.9	Sokol Kosta (UROME)	Final version.	2/11/2015

Contents

List of Figures

Executive Summary

This deliverable deals with the high-level technical aspects of the cloud to be used as utility and support infrastructure within the RAPID project. In particular, we compared and contrasted different and diverse open-source cloud infrastructure software. We describe the adopted solutions, which have been selected in order to match the requirements of the RAPID platform for providing the appropriate RAPID cloud services. This cloud service will be described by focusing on (but not limited to): support of heterogeneous execution; GVirtuS and ThinkAir compatibility; SLA-based mechanism required by the RAPID applications. The deliverable contains an evaluation of different hypervisors in order to choose the one achieving the best results with ThinkAir and GVirtuS software middleware.

1. Introduction

RAPID will develop an efficient heterogeneous Central Processing Unit (CPU) – Graphics Processing Unit (GPU) cloud computing infrastructure, which can be used to seamlessly offload CPU-based and GPU-based, using Compute Unified Device Architecture (CUDA), tasks of applications running on low-power devices such as smartphones, notebooks, tablets, portable/wearable devices, robots, and cars to more powerful devices over a heterogeneous network (HetNet).

Currently, the Top500 ranking[1] and its greener counterpart, the Green500 list[2], show an impressive 6 times improvement in the performance-power ratio of large-scale high performance computing (HPC) facilities over the last five years. In these two lists, it is clearly visible the trend of the adoption of hardware accelerators to attain unprecedented levels of raw performance with reasonable energy costs [1].

In order to design the more suitable cloud computing and acceleration infrastructures to the goals of RAPID, namely the deployment of the acceleration as a service, the basic points to focus on are: The Cloud Management Platform (CMP), the hypervisor, and the GPU virtualization tool. A CMP is an integrated product that provides for the management of public, private, and hybrid cloud environments. A hypervisor or Virtual Machine Monitor (VMM) is a piece of computer software, firmware, or hardware that creates and runs virtual machines. A GPU virtualization software, is a software layer that allows for General-Purpose GPU (GPGPU) computation on a Virtual Machine's (VM's) host server rather than on a physical endpoint device.

The first step towards building the complete RAPID architecture consists in carrying out the choice of the CMP and the hypervisor by means of an evaluation and comparison process.

In particular, the best CMP is selected amongst the considered CMPs on the basis of their ability to provide an effective hypervisor support, and to allow for an efficient GPU virtualization. Moreover, their policy for royalties, licensing, and pricing are also analyzed.

The choose amongst the considered hypervisors we have considered their integration model with the host hardware and operating system, and the type of virtualization that they can provide.

Furthermore, since the Acceleration Server (AS) on the cloud is hosted on a virtual machine, providing GPU acceleration through GPU Computing over virtual GPUs, the above analysis is also useful to identify the adequate hardware to compose the physical servers offering the required capabilities.

One of the most successful GPU based accelerating system is provided by NVIDIA and relies on the CUDA programming paradigm supporting high level languages tools [2]. Currently, virtualization issued by popular hypervisors (Xen [3], Linux KVM [4], Virtual Box [5]) does not allow a transparent use of accelerators as CUDA-based GPUs, as virtual/real machines and guest/host real machines communication issues raise serious limitations to the overall potential performance of a cloud computing infrastructure based on elastically allocated resources. RAPID's strategy consists in extending and adapting GVirtuS, a generic and open source virtualization service for GPU. The model that GVirtuS is based on, the communication interface, and the key software components are outlined and discussed.

[1] www.top500.org
[2] www.green500.org

In particular, this document is organized as follows: Section 2 is dedicated to the CMPs, i.e. on the private cloud infrastructure design; Section 3 presents the results of the comparison process of the principal hypervisors; Section 4 describes the GVirtuS tool; and Section 5 gives a concise view of the choices that have been carried out.

1.1. Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
AS	Acceleration Server
AWS	Amazon Web Services
CO	Confidential
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D	Deliverable
DMP	Data Management Plan
DoA	Description of the Action
EC	European Commission
EFI	Unified Extensible Firmware
EU	European Union
GA	Grant Agreement
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
HetNet	Heterogeneous Network
HPC	High Performance Computing
IaaS	Infrastructure-as-a-Service
IDE	Integrated Drive Electronics
KVM	Kernel-based Virtual Machine
PaaS	Platform-as-a-Service
OS	Operating System
PCI	Peripheral Component Interconnect
PU	Public
SCSI	Small Computer System Interface
SVN	Subversion
vCPU	Virtual CPU
vGPU	Virtual GPU
TCP/IP	Transmission Control Protocol/Internet Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine
WP	Work Package

2. Private Cloud Infrastructure Design

RAPID will utilize cloud infrastructures to test, deploy, and validate the RAPID framework during the development, integration, and verification phases of the project. Specifically, a private cloud installation will be realized in order to create the execution environment, capable of creating, suspending, resuming, destroying, or managing, Virtual Machines. In the following, a comparative analysis of existing cloud infrastructure software solutions, as well as most common hypervisors, is presented, concluding to the most applicable ones for the RAPID framework.

2.1. Private Cloud Software Evaluation and Comparison

Private cloud solutions are increasingly required to provide security, privacy, and flexibility on a single company/business scope. As a result, various cloud management platforms that provide basic virtualization functionality have emerged, employing different approaches and types of infrastructure support.

The most prominent private cloud management platforms are summarized in the following in Table 1.

Table 1. Most prominent private cloud management platforms.

Cloud Management Platform	Highlights	Pricing
Microsoft Cloud [6]	Microsoft's proprietary cloud management solution is based on Azure [7] infrastructure. Initially it offered its cloud management services in a public Platform-as-a-Service (PaaS) mode. Currently, Microsoft also allows private cloud configurations, based on its System Center (currently 2012 R2 [8]) product, which supports both Windows and Linux hosts. It supports natively Microsoft's Hyper-V virtualization manager, being able to integrate VMware vSphere and Citrix XenServer, offering support for Graphics Processing Unit (GPU) virtualization through Microsoft RemoteFX [9]. Microsoft Cloud supports both Infrastructure-as-a-Service (IaaS) and PaaS private cloud modes and can be easily configured to integrate with Azure public cloud, rendering the deployment of hybrid clouds easier.	Paid
VMWare vCloud [10]	VMWare's proprietary cloud management platform, vCloud, is arguably the most well-known private cloud management solution available. VMWare promotes the use of its own vSphere virtualization package, bundling VMWare's ESXi hypervisor. However, vCloud also offers support for Microsoft's Hyper-V and Linux KVM. vCloud provides native GPU passthrough support, i.e. direct connection of a dedicated GPU to a virtual machine, through vSphere, fully integrating vGPU services [11] and being compatible with NVIDIA GRID technologies. Moreover, similarly to Microsoft Cloud, VMWare vCloud can be easily integrated with public clouds towards the creation of hybrid ones.	Paid

Cloud Management Platform	Highlights	Pricing
Citrix XenCenter/XenServer [12]	The open source Xen [3] hypervisor was originally supported by XenSource Inc. before being acquired by Citrix in 2007. Since then, Citrix employs Xen (currently version 6.5 R2) to build its cloud software solutions (including IaaS and PaaS modes), including Server, Desktop, and Application virtualization via the XenServer, XenDesktop, and XenApp products, respectively. XenServer offers proven native NVIDIA GRID technologies support [13] and is able to host Windows, Linux, and Unix servers. However, the visualized management framework, XenCenter, runs only on top of Windows installations.	Free
OpenStack [14]	Initially a joint project of Rackspace and NASA (beginning in 2010), OpenStack is now led by the OpenStack Foundation, a non-profit corporate entity established in late 2012 and supported by more than 500 companies, including Cisco, Intel, IBM, Google, Hewlett-Packard, etc. [15]. OpenStack is a fully open source and supports a wide range of open source hypervisors such as Linux KVM and Xen, as well as proprietary ones including Microsoft Hyper-V and VMWare vSphere. Docker [16] containers support is also present [17]. Depending on the managing hypervisor, OpenStack can support GPU passthrough, enabling the exploitation of NVIDIA GRID capabilities. OpenStack can be installed on top of Linux, Unix and Windows physical machines and can host Linux, Unix and Windows virtual machines. It supports Amazon Web Services (AWS) APIs in order to create hybrid private-public clouds.	Free
Apache CloudStack [18]	Apache CloudStack is an open source cloud management platform managed by the Apache Software Foundation. It supports numerous hypervisors including XenServer/XCP, Linux KVM, Hyper-V, and VMware ESXi with vSphere. It can be installed on top of both Linux and Windows machines and can also host both types of virtual servers. Support for GPU acceleration through GPU passthrough and vGPU is inherently available, as is remote configuration and management through RESTful Application Programming Interfaces (APIs).	Free
Eucalyptus [19]	Eucalyptus (full name is "Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems") was initially released at Rice University in 2008, before being acquired by Hewlett-Packard in September 2014. As most cloud platforms, Eucalyptus offers management APIs and is inherently compatible with AWS, thus being able to form hybrid cloud infrastructures. It supports Linux KVM, Xen, and VMWare vSphere hypervisors. However, GPU virtualization is not inherently supported.	Free

Cloud Management Platform	Highlights	Pricing
Open Nebula [20]	OpenNebula is an open source cloud management platform released in 2008. OpenNebula is capable of managing Linux KVM, Xen, and VMWare ESXi. OpenNebula supports a wide range of cloud interfaces including Amazon EC2, OGF Open cloud computing interface, and vCloud. Recently, native support for GPU virtualization was announced, rendering it capable of properly handling NVIDIA GRID technologies.	Free

2.2. Hypervisors Support

In addition to the private cloud platforms, a hypervisor is required in order to allow physical devices to share their resources amongst virtual machines running as guests on top of the physical hardware. Table 2 presents the main hypervisors supported by the private cloud platforms management presented outlined in the previous section.

Table 2. Support for Hypervisors.

Cloud Management Platform	Linux KVM	Xen	XenServer	VMWare vSphere	Hyper-V
Microsoft Private Cloud	No	No	Yes	Yes	Yes
VMWare vCloud	No	No	No	Yes	No
Citrix XenCenter/XenServer	No	Yes	Yes	No	No
OpenStack	Yes	Yes	Yes	Yes	Yes
Apache Cloudstack	Yes	No	Yes	Yes	Yes
Eucalyptus	Yes	Yes	No	Yes	No
OpenNebula	Yes	Yes	No	Yes	No

Evidently, OpenStack, being supported by a very large industrial community, is the only platform inherently supporting all major hypervisors.

2.3. Support for GPU virtualization

The support of the private cloud management presented in the previous section with respect to GPU virtualization is tabulated in Table 3.

Table 3. Native support for GPU virtualization.

Cloud Management Platform	vGPU	GPU passthrough
Microsoft Private Cloud	Yes	Yes
VMWare vCloud	Yes	Yes
Citrix XenCenter/XenServer	Yes	Yes
OpenStack	No	Yes
Apache Cloudstack	Yes	Yes
Eucalyptus	Yes	Yes
OpenNebula	Yes	No

Microsoft Private Cloud, VMWare vCloud, Citrix XenServer, Apache CloudStack and Eucalyptus support both vGPU and GPU passthrough. OpenStack natively supports GPU passthrough (through proper configuration) only, whereas OpenNebula only supports vGPU technologies with no way to use the passthrough.

2.4. Royalties, licenses and pricing schemes

Table 4 tabulates the licensing, pricing, and accompanying royalties of the private cloud management platforms presented earlier.

Table 4. Royalties, licensing and pricing schemas

Cloud Management Platform	Open Source	License	Pricing
Microsoft Private Cloud	No	Proprietary	Paid
VMWare vCloud	No	Proprietary	Paid
Citrix XenCenter/XenServer	Yes	Proprietary	Paid
OpenStack	Yes	Apache License 2.0	Free
Apache Cloudstack	Yes	Apache License 2.0	Free
Eucalyptus	Yes	GPLv3 with Proprietary relicensing [21]	Free / Paid
OpenNebula	Yes	Apache License 2.0	Free

OpenStack, Apache CloudStack, and OpenNebula are accompanied by an Apache License 2.0 and hence, are more suitable for commercial exploitation, unlike other cloud management platforms that are either proprietary (Microsoft Private Cloud, VMWare vCloud, Citrix XenCenter/XenServer) or too restrictive (Eucalyptus), shown in Table 4.

2.5. Design and Deploy

As part of the accelerator infrastructures, private and public cloud resources will be utilized to host the Acceleration Server (AS). Each AS on the cloud side will be hosted on a Virtual Machine (VM), providing GPU acceleration through GPU Computing over virtual GPUs. In order to support the GPU Computing and virtualization functionality, the Consortium, and specifically SILO and UNP, will procure adequate hardware to compose the physical servers that offer such capabilities. These servers will be integrated first in the private and then in the public cloud, connected to the RAPID framework.

As the cloud infrastructure is intended to be used during development and integration activities, but also for testing, validation and evaluation purposes, an entry-level and a high-end server will be procured. They will cover the application and system requirements reported in D2.1 “Application Analysis and System Requirements”, referring to the acceleration or physical server. In brief, the servers to be procured will be the following:

- An entry-level server equipped with a mid-range GPU, which utilizes GVirtuS to achieve software-based GPU virtualization;
- A high-end server equipped with a high-end NVIDIA Grid GPU, which offers hardware GPU virtualization delivering GPU passthrough or virtual GPUs (vGPUs), which can be allocated per VM, based on specified GPU profiles.

The hardware procurement matrix in Table 5 lists the equipment planned to be acquired in order to build the RAPID cloud infrastructure. The table provides information about the equipment type, the tentative date of procurement, the responsible partner, the project relevant tasks, as well as the requirements (using the requirements' IDs as defined in D2.1, Section 5) covered by the specified hardware.

Table 5. Hardware procurement matrix.

Equipment	(Tentative) Procurement Date	Partner	Relevant Task	Requirements to be covered
Server	December 2015	SILO	T6.2, T6.3, T7.1, T7.2, T7.3, T7.4	BIOS_NET_07 BIOS_NET_08 HT3D_NET_09 HT3D_NET_10 HT3D_CPU_11 HT3D_CPU_12 HT3D_CPU_13
NVIDIA Grid GPU	January 2016	SILO	T7.3, T7.4	BIOS_CUDA_01 BIOS_CUDA_02 BIOS_CUDA_03 HT3D_CUDA_04 HT3D_CPU_11 HT3D_CPU_12
NVIDIA GeForce GTX970	December 2015	SILO	T6.2, T6.3, T7.1	BIOS_CUDA_01 BIOS_CUDA_02 BIOS_CUDA_03 HT3D_CUDA_04

Within RAPID we collected the set of application and system requirements as well as the software requirements that arise from background software, such as GVirtuS by UNP and ThinkAir by UROME [22], which provide the basis for RAPID development. Based on them, the server and GPU specifications were extracted and are summarized in Table 6 and Table 7, respectively. Table 7 lists common specifications for both lower-end and high-end GPUs.

Table 6. Server specifications.

Server specifications	
CPU	Intel Core i7, Multi-core, ideally > 4 cores
RAM	≥ 48 GB
HDD	x4 (1+1 for system and swap in RAID 1, 1+1 for storage in RAID 1)
RAID Configuration	Hardware RAID, e.g. 1
Host operating system	Linux
Network card	Ideally 1 Gbps or more

Table 7. GPU specifications.

GPU specifications	
Number of cores	≥ 768
Memory	≥ 4 GB
Technology Support	CUDA

Compute capabilities	3.0 or higher
CUDA version	6.5 or higher
Microarchitecture	Kepler or Maxwell (recommended)

Finally, Figure 1 depicts the time plan of procuring the two servers within task T6.1, as well as their planned use in subtasks of WP6 and WP7. The entry-level server will be used for all depicted subtasks, while the high-end server will be exploited during setting up the public cloud and evaluating the overall RAPID platform.

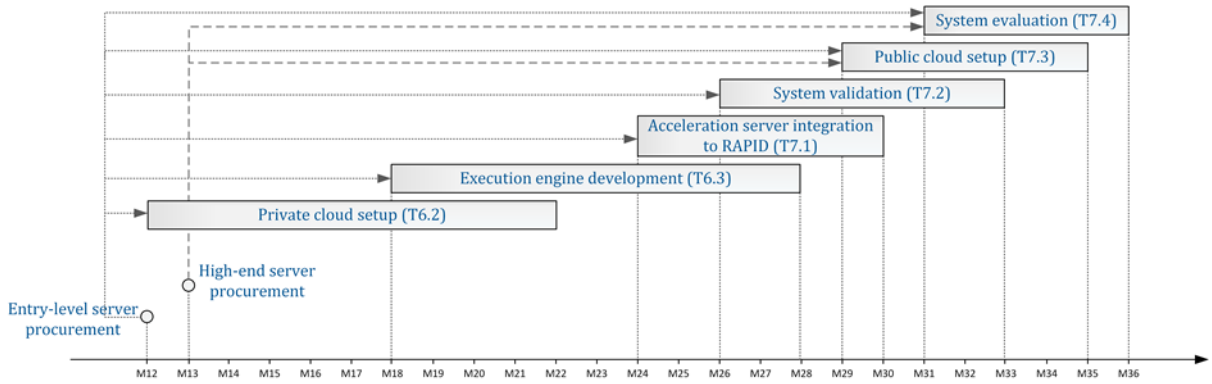

Figure 1. Planning of procurement, setup and use of cloud resources

2.6. Discussion

Based on the analysis and comparison presented in this section, the RAPID consortium has chosen to adopt OpenStack as the Cloud Management tool. OpenStack stands out of the rest of the available cloud management platforms as it is completely free and open source, has exploitation-friendly license and is supported by a broad industrial community, granting it with significant updates and functionality increments with each (6-month) release. It provides inherent support for all major hypervisors, including proprietary ones, and natively supports GPU passthrough. Moreover, OpenStack is able to successfully integrate GVirtuS [23] developed by UNP to provide CUDA access to KVM virtual machines [24], further validating its choice from the RAPID consortium.

3. Hypervisor Software Evaluation and Comparison

Virtualization is a key technology in Cloud environments, and made possible thanks to hypervisors. We have performed an analysis of the main hypervisors available, indicating the ones that are widely used in Cloud environments, as the base of the infrastructure management.

To better motivate and illustrate the different characteristics of the hypervisors, let us first present the various types of virtualization:

- **Full Virtualization:** The hypervisor emulates some hardware for the guest Operating System (OS), so it is not necessary to perform any modification and OSs run like in the non-virtualized environment. The hypervisor will need to manage dynamically the instructions received with binary translations.

- **Hardware-assisted Virtualization:** It is a full virtualization which takes advantage of the hardware instructions implemented in current processors. As a result, efficiency of the emulation increases, while reducing the overhead of instructions translation.

- **Paravirtualization:** In this case there is no hardware emulation and the hypervisor provides a special Application Program Interface (API) different from the one provided by the hardware. Therefore, the guest OS needs certain modifications for interacting with the hypervisor.

- **Operating-system-level Virtualization:** It is a quite new way of virtualization in which the guest OS shares the same running instance of the host OS. Somehow, they share the kernel of the OS, although each OS works as if it was running in an isolated environment. This is the technology used by 'containers', which would be the equivalent to typical VMs.

As paravirtualization requires some modification of the guest OS, in principle, it is more interesting to use full virtualization, especially if it is possible to use hardware-assisted virtualization. Moreover, there is an intermediate solution called **hybrid virtualization**, in which hardware-assisted virtualization is complemented with paravirtualized drivers for certain hardware (since this reduces VM traps, increasing efficiency).

Another aspect that affects hypervisors is their classification based on the type of their integration with the hardware and the host OS:

- **Type I hypervisors (bare-metal):** The hypervisor has either a microkernel or it is part of an OS kernel, so it has full privilege for interacting with the hardware and provides typical OS features at the same time (i.e. processes management, I/O management, etc.). Guest VMs are executed on top of the system, utilizing the hardware efficiently and maintaining good isolation between running VMs.

- **Type II hypervisors (hosted):** The hypervisor is installed in an OS as if it was another application. This means that the hypervisor that runs the VMs needs to forward system calls to the host OS, which has the appropriate privileges to interact with the hardware.

VMWare ESXi, Hyper-V, Xen, and KVM are type I hypervisors (although KVM is considered as type II by some people). VirtualBox and VMWare Workstation are type II hypervisors.

In the concrete case of the RAPID project, we are interested in determining the best choice for deploying a Cloud infrastructure to be used for trials. Although the project will not require to work in large datacentres, it is desirable that the selected combination of Cloud infrastructure and hypervisor will allow for scalability, while maintaining a good performance with respect to the deployment of applications directly in a server. This makes sense because we are expected to use relatively small VMs, enabling the optimization of resources usage thanks to the capability to run more than one VM in the same server. We expect small VMs with just a few pre-deployed components (i.e. the Java JDK), since the applications to be offloaded are Java-based applications from mobile devices, which do not use to be too complex and with a few dependencies. Lightweight guest OSs could be used for improving resources usage.

It is important to take into account the support for GPUs, a key element in RAPID. In fact, the pilot scenarios are expected to make use of the acceleration capabilities of GPUs for speeding up the execution of the deployed applications. Therefore, we focus on the GPU management access model of each hypervisor.

The next paragraphs summarize our findings about the comparative performance analysis of the hypervisors. In the analysis performed in [25], KVM, Xen, and ESXi were evaluated for a private CloudStack platform. They used Windows 2008 as the guest OS and executed some benchmarks (a simple Java application for retrieving CPU and memory information with SIGAR and a Passmark benchmark for simulating desktop-oriented workloads).

The performance of KVM, Xen, and ESXi has been also the focus of [26]. In this case, the guest OS was RedHat Enterprise Linux 6.2 and measurements of CPU and disk utilization were taken in the hypervisor itself. Performance test and live migration tests were done with different VM configurations and an increasing number of requests per second to the testing application (a large telecom application).

A very complete analysis was presented in [27] with several benchmarks (such as Bytemark, Ramspeed, Bonnie++ & FileBench, Netperf, application workloads with freebench and multi-tenant interference). In this case, Hyper-V, KVM, ESXi, and Xen were analysed and compared.

The GPU passthrough performance when using CUDA and OpenCL applications has been analysed in [28]. Thanks to GPU passthrough, the VMs are granted a thread on the physical GPU, being an alternative to other solutions such as GVirtuS. Two host architectures were employed in the analysis using several benchmarks, such as SHOC micro-benchmarks, LAMMPS for parallel molecular dynamics simulation, GPU-LIBSVM for machine learning classification, and LULESH for hydrodynamics. This analysis was performed using KVM, Xen, VMWare ESXi, and LXC.

Although the usage of Operating-system-level Virtualization (containers) is very interesting, the integration of this technology with current Cloud platforms is not mature enough yet. There are some approaches for integrating containers with Cloud platforms such as OpenStack and Open Nebula, but they represent an important risk for the project. Therefore, we have decided to not take them into account for this analysis.

The following table aims at summarizing the information gathered about the five hypervisors we have analysed.

Table 8. Hypervisors comparison

Hypervisor	Type	Virtualiz. Type	GPU Virt.	GPU Perf.	Scalab.	Max. Overhead	Bench. Ranking	Limits
KVM	I	Hardware-assisted	PCI passthrough	98-100 %	Good	15% (CPU) 24% (Mem) 38% (I/O)	1	OK
Xen	I	Hardware-assisted	PCI passthrough	96-99%	Medium	8% (CPU) 17% (Mem) 22% (I/O)	3	OK
ESXi	I	Hardware-assisted	PCI passthrough	96-99%	Good	4% (CPU) 3% (Mem) 7% (I/O)	2	OK
Hyper-v	I	Hardware-assisted	Remote FX Driver	?	?	?	4	OK
VirtualBox	II	Hardware-assisted	PCI passthrough	?	Good	?	5	OK

For the benchmarks ranking, we have considered the number of times each hypervisor obtained the best rank in each of the tests. In the case of Hyper-v and VirtualBox, since the amount of formal benchmarking data was too limited, the lack of information and comparisons penalized their proper ranking.

3.1. KVM

KVM is a type I hypervisor which uses hardware-assisted virtualization as its way to manage VMs (it also supports just full virtualization). There is some controversy, since many people consider KVM as a type II hypervisor arguing it as another component of the Linux kernel. But the Linux kernel just launches KVM and KVM takes over part of the hardware resources, using the hardware virtualization instructions for running VMs, as depicted in Figure 2.

It supports many guest OSs, including Linux, Windows, BSD, Solaris, etc. Moreover, it supports paravirtualization of some drivers in the most commonly used OSs, thanks to **virtio**[3], improving the performance of the hypervisor. KVM also supports live migration of VMs using the pre-copy technique for memory migration. This means that it copies the memory pages of the VM to be migrated and, just before suspending the original VM, any modified memory page is sent to the target VM, resuming it once all the memory pages and running processes are ready.

It is important to highlight some known limitations of KVM[4]. Although KVM supports memory overcommit (by using swapping), it cannot support CPU overcommit, since at most 10 virtual CPUs can be assigned per physical processor core. As for storage, Small Computer System Interface (SCSI) emulation is not supported (SCSI devices are disabled) and the maximum number of Integrated Drive Electronics (IDE) devices is four per guest. Peripheral Component Interconnect (PCI) devices are

[3] Virtio are a set of paravirtualized drivers for I/O operations, aiming at offering a common framework for hypervisors in I/O virtualization (http://www.linux-kvm.org/page/Virtio)

[4] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Virtualization/chap-Virtualization-Virtualization_limitations.html#sect-Virtualization-Virtualization_limitations-General_limitations_for_virtualization

limited as well, being 32 the maximum number of para-virtualized devices per guest using **virtio**. Live migration is only possible between CPUs of the same vendor (e.g., AMD to AMD, Intel to Intel).

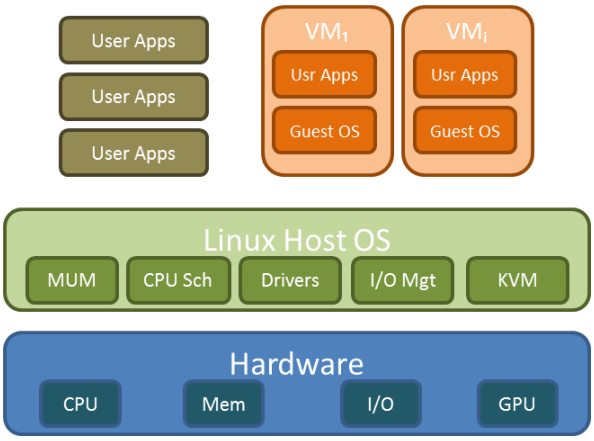

Figure 2. KVM Schema.

According to the analysis in [25], KVM exhibits a 4% overhead in the CPU with respect to a native OS (15% when using Passmark), a 24% overhead in memory usage, a 30% overhead when reading from disk, a 38% overhead when writing, a 21% overhead when sending data through the network and a 22% overhead when receiving data. In the comparison with Xen and ESXi, KVM has the worst performance.

In [26], KVM manifests a similar CPU usage as in a non-virtualized server, with a good (linear) scalability. For disk utilization, it increases linearly with the request per second and is always around 2000-5000 KB/s higher compared to the non-virtualized server. The response time of the application is higher than in the non-virtualized server (about 5ms more when requests per second were higher than 2000) but it exhibits a good scalability for a number of requests per second up to 4300, when the response time increases rapidly. As for the live migration, the downtime is only 0.7 seconds and resources usage is similar to VMware ESXi, behaving better than Xen.

In [27], KVM has a good performance according to most benchmark (except in Ramspeed, where its results were below other hypervisors), getting the first or second position in many experiments.

Finally, according to [28] about GPU passthrough performance, globally, KVM can achieve 98–100% of the base system's performance, being in the first or second position in most of the benchmarks.

3.2. Xen

This is a well-known type I hypervisor, based on a microkernel design on top of which different guest OSs run. Its initial design focused on paravirtualization, but today it claims to support in addition hardware-assisted virtualization (even with paravirtualized drivers).

Xen can be distributed with several host OSs (except for Windows, most of the OSs) and admits almost any guest OS (Windows and Unix-like systems). As in the case of KVM, it supports PCI and GPU passthrough, as well as live migration of VMs (using a similar technique).

The main difference between Xen and hypervisors such as KVM is the way they manage the VMs. Since Xen is running in a microkernel, it controls directly the access to the hardware. But it needs a host OS for some operations, especially related to I/O management (e.g., the messages, backend drivers). This is the so called Dom0 VM (as shown in Figure 3, in dark grey), which provides these functionalities,

especially for paravirtualization. The rest of VMs represent guest OSs (DomU) which will perform some operations through Dom0 through the frontend drivers (Figure 3).

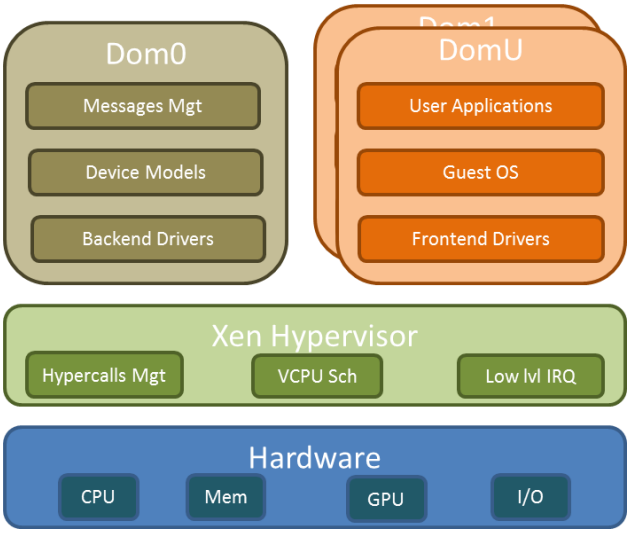

Figure 3. Xen Schema.

The limitations of Xen vary depending on whether it operates in paravirtualization or in a full virtualization mode (according to the same source as for KVM). In paravirtualization, guests may be limited to 16GB or 168GB of memory (if they are x86 or x86_64), 254 devices per guest and 15 network devices per guest. In full virtualization, x86 guests may be limited to 16GB of memory and 4 emulated IDE devices (increased when para-virtualized drivers are used). In other cases, Xen guests may work with 512GB to 1TB of memory and with 256 to 512 virtual CPUs (vCPUs) per guest.

According to the analysis in [25], Xen generates a 1% overhead in the CPU with respect to a native OS (8% when using Passmark), a 17% overhead in memory usage (1% in Passmark), a 12% overhead when reading from disk (4% according to Passmark), a 22% overhead when writing (6% for Passmark), a 2% overhead when sending data through the network and a 3% overhead when receiving data.

In [26], Xen's scalability performance is reported to be poor, since testing with more than 4300 requests per second made the system to fail. In the tests it did not fail, it exhibits 5% or higher of CPU usage compared to a non-virtualized server, with a good (linear) scalability (the worst in the tests). For disk utilization, it increases more or less linearly with the request per second and is approximately 2000-5000 KB/s higher compared to the non-virtualized server. The response time of the application is substantially higher than in the non-virtualized server (more than 8ms for a number of requests per second that is higher than 2000). There is a performance degradation in scalability when the number of requests per second reaches 3000, resulting in an exponential increase of the response time. As for the live migration, downtime is only 0.3 seconds and resources usage is at a similar level as in the case of VMware ESXi and KVM, although it is the highest one of the three hypervisors.

In the analysis performed in [27], Xen exhibits a relatively poor performance with respect to several benchmarks (especially with Netperf, Bonnie++, Filebench, and performance in multi-tenant interference), resulting the third or last ranked hypervisor. Xen reported a good performance when using 4 VCPUs with respect to Freebench, Ramspeed, and Bytemark.

Finally, according to [28], globally, Xen was able to achieve 96–99% of the base system's performance, with respect to GPU passthrough, behaving slower than other hypervisors in general, reaching the third or last position according to several benchmarks. It only got the first or second position in some SHOC microbenchmarks.

3.3. VMWare ESXi

The ESXi hypervisor is a bare-metal (type I) hypervisor, since it runs directly on the hardware. It has a light underlying operating system called VMkernel in charge of resources scheduling, processes management, I/O stacks and device drivers. ESXi provides full virtualization and, in its last versions, it is able to use hardware instructions for virtualization, so it supports hardware-assisted virtualization.

It is an easy-to-use hypervisor, thanks to the tools provided by the VMWare proprietary platform. As other hypervisors, ESXi supports multiple guest OSs, from Windows to other Unix-based OSs. It also supports live migration, through its vMotion technology, which allows VMs and virtual disks migration on the fly. Moreover, it supports PCI and GPU passthrough, for a better performance.

Like other hypervisors, ESXi has some limitations related to the maximum resources of the host, the resources of VMs (e.g., maximum RAM of 4TB, maximum number of VCPUS per physical CPU core of 32 and maximum number of processors per VM of 128), and the number of network interfaces, active VLANs. However these aspects do not impose any constraint in the RAPID infrastructure.

According to the analysis in [25], ESXi generates no overhead in the CPU with respect to a native OS (4% when using Passmark), a 3% overhead in memory usage (2% in Passmark), a 4% overhead when reading from disk (5% according to Passmark), a 7% overhead when writing, no overhead when sending data through the network and a 1% overhead when receiving data.

In [26], ESXi showed a similar CPU usage as in a non-virtualized server (sometimes even lower), with a very good (linear) scalability. For disk utilization, it increased almost linearly with the request per second and it was always around 7000-10000 KB/s higher compared to the non-virtualized server (around the double in some cases), being the worst hypervisor. The response time of the application was almost equal to the non-virtualized server and had a good scalability until requests per second reached 4300, when the response time increased with a higher slope (although it was still close to the original response time). As for the live migration, downtime was 3 seconds and resources usage is a bit better than in KVM, behaving much better than Xen.

In the analysis performed in [27], ESXi showed a very good performance in all the benchmarks, over performing in most situations and getting the first or second position in most of the experiments. Even if it was not so far from KVM, ESXi got better results.

Finally, according to [28] about GPU passthrough performance, globally, ESXi was able to achieve 96-99% of the base system's performance, behaving slower than other hypervisors in general. It only got the first or second position in some SHOC micro-benchmarks and in LAMMPS (getting similar results to Xen in most of the cases).

3.4. Microsoft Hyper-v

Hyper-v is a type I hypervisor, quite similar to Xen in its operation and architecture. Instead of referring to VMs as 'domains' (as Xen does), Microsoft uses the term 'partitions', e.g., having a parent partition and several child partitions. While the parent partition is a Windows based OS with some additional

components which support the hypervisor (device drivers, Virtualization Service Provider, other related processes, etc.), each child partition runs a guest OS which interacts with the hypervisor through a Virtualization Service Consumer. There is also a stand-alone version of the hypervisor, avoiding a full Windows OS as parent partition, although it has not exactly the same features as the one embedded in Windows.

It seems that Hyper-v is focused on paravirtualization, since it considers Hyper-v aware guest OSs, although it can also support full virtualization, by exploiting emulation and hardware-assisted virtualization. Guest Oss may be Windows and several Linux distributions, although the number of supported Oss is less extensive than in the case of other hypervisors (specifically, the list of supported OSs is shorter to those reported by KVM, Xen, or ESXi). Moreover, it also supports live migration of VMs.

The Hyper-v hypervisor is limited in several ways. The maximum number of vCPUs per VM is 64, the maximum amount of RAM per VM is 1TB and the maximum number of IDE devices is 4. Audio hardware is not virtualized in the VMs.

Hyper-V cannot perform PCI(e) (or GPU) devices passthrough but, instead, it provides RemoteFX, a driver (similar to GVirtuS), that sends the instructions to the hypervisor and the hypervisor forwards them to the GPU (or PCI device). However, it is not possible to compare its performance with other hypervisors, since the mechanism is very different and there are not formal benchmarks.

To the best of our knowledge, apart from [27], there is no other comparative performance analysis of Hyper-v. According to this work, Hyper-v had a relatively poor performance in Bytemark, being third in most cases (with one vCPU) or last (with 4 vCPUs). However, it did achieve a good performance in Ramspeed (especially in the case of 4 vCPUs). As for disk usage, while it scored well in Bonnie++ and Filebench with 1 vCPU, it did worse with 4 vCPUs. It exhibits a good network performance (with Netperf), but with Freebench it was scoring third and last in most of the tests (although it was not so far from other hypervisors). Finally, it obtained the third best performance with respect to the average response time.

3.5. VirtualBox

VirtualBox is a type II (hosted) hypervisor which may be installed in Linux, Windows, OS X, FreeBSD, and OpenSolaris host OSs. In its last version, it supports paravirtualization and hardware-assisted virtualization, thanks to new instruction set extensions. It is an easy-to-use hypervisor, which also supports live migration (known as 'VM teleportation' by VirtualBox).

It supports a wide range of guest OSs (Windows, Linux, BSD, Solaris, Haiku, etc.), although in some cases the usage of special packages is recommended (with device drivers and certain applications) for improving performance.

It seems VirtualBox limits the RAM of the guest OSs to 1TB, the number of CPUs per VM to 32 and the number of disks per VM to 4 IDE devices (there are no limits for other storage devices).[5] Moreover, since some of the new features are experimental, there might be some issues when using them with

[5] http://hyper-v-backup.backupchain.com/hyper-v-vmware-and-virtualbox-hypervisor-limitations/

certain guest OSs[6] (such as hardware 3D acceleration, PCI passthrough, Unified Extensible Firmware (EFI) firmware, etc.).

It is hard to find a recent work analysing VirtualBox performance and its comparison with other hypervisors. The latest benchmark we found is [29], where it is compared to VMware. This work uses LINPACK and Iperf as the benchmarks to measure performance of both hypervisors. While LINPACK is focused on basic linear algebra subprograms (in Fortran), Iperf analyses the network performance, measuring the bandwidth and the quality. The configuration of the host and guest VMs was the same for testing both hypervisors, being the guest OSs Windows XP and Ubuntu 10.10.

VirtualBox and VMWare had similar performance, even when increasing the problem sizes, showing good scalability. In fact, with the Ubuntu guest OSs, VirtualBox was a bit better. Regarding the network performance, VMware proved to be superior to VirtualBox in terms of bandwidth, without clear performance advantages with respect to the other network quality measurements (e.g., had lower jitter in Windows, while VMWare exhibited lower response times in all cases).

Although VirtualBox claims to support PCI and GPU passthrough, it was not possible to find any formal work performing benchmarks and comparing several hypervisors against its performance.

3.6. Discussion

According to the available information, it seems that, although there are three main hypervisors widely used (KVM, ESXi, and Xen), there are also two other options which are gaining momentum (Hyper-v and VirtualBox). All of them have similar features, being able to exploit hardware-assisted virtualization, supporting live migration and supporting the usage of several devices. The main difference here comes when looking for PCI passthrough capabilities, which are not available in all the hypervisors (e.g. Hyper-v doesn't support it).

About the performance, most of the comparisons between KVM, ESXi, and Xen show that KVM and ESXi are quite similar, while Xen is a bit behind in most of the cases. In an earlier comparison between VMWare and VirtualBox, VirtualBox was a bit behind VMWare solutions, especially in network performance, although this analysis may not really reflect the current status of these hypervisors.

Although these hypervisors have some limitations with respect to the amount of VMs which can be managed and the resources assigned to each guest OS, we do not expect that this will be an issue for RAPID, according to the VMs we expect to deploy during the project.

It is interesting to highlight that, from the user perspective, ESXi, Hyper-v, and VirtualBox seem to be easier to use and manage than Xen and KVM, thanks to the Graphical User Interface management tools that accompany them.

The support of libvirt[7] has not been analysed for each hypervisor. Libvirt is a tool for facilitating the use of the same API for performing VMs management operations with the underlying hypervisor, without the need of code customization for each specific case. Although libvirt supports all the analysed hypervisors, its integration with KVM is more mature and extends to a broader set of operations compared to the others.

[6] https://www.virtualbox.org/manual/ch14.html
[7] https://libvirt.org/hvsupport.html

Finally, it is important to mention that some open source cloud platforms can easily be integrated with KVM and Xen (mainly thanks to their strong community support), while Hyper-v and VMWare ESXi have been integrated mostly with proprietary solutions. It seems that VirtualBox is less attractive because of its condition of hosted hypervisor. Moreover, KVM is already deployed with modern Linux kernels, facilitating its installation.

Therefore, we believe that KVM is the best option for RAPID when providing a cloud infrastructure for the project. It has a well-known integration with platforms, such as OpenStack[8], a good performance, and supports GPU virtualization using PCI(e) passthrough.

[8] http://docs.openstack.org/developer/nova/support-matrix.html

4. VMs GPU Acceleration

Most recently, General-Purpose Graphics Processing Units (General-Purpose computation on Graphics Processing Units (GPGPUs) or GPUs) have become commonplace within high-performance supercomputers. Within HPC, there has also been a notable movement toward dedicated accelerator cards such as general purpose graphical processing units to enhance scientific computation problems by an upwards of two orders of magnitude. GvirtuS is a key component to extend the device acceleration support to the elastic computing power provided by the RAPID private cloud infrastructure.

4.1. GVirtuS overview

GVirtuS stands for Generic VIRTUalisation Service and could be considered as a generic framework for facilitating the development of split-drivers solutions. The split-driver model is a widely used approach to virtualize the hardware components of a computing system. It delegates the hardware access to a domain management specialist, which must be sufficiently privileged to be able to control the system's hardware interacting with legacy drivers **Error! Reference source not found.**.

GVirtuS extends and generalizes gVirtuS (with lower case g), a GPGPU transparent virtualization solution. The main motivation behind gVirtuS is to address the limitations of transparently employing accelerators such as CUDA-based GPUs in virtualization environments. Before gVirtuS, the use of the GPUs required explicit programming of the communication between virtual and physical machines and between guest and host operating systems. Furthermore, the solution had to deal with issues related to the vendor specific interfaces on the virtual machine side. These limitations drastically reduced the spread of virtualization solutions based on GPUs and hindered the employment of accelerators as on-demand resources in cloud computing infrastructures [23].

The generic virtualization service, GVirtuS (Generic Virtualization Service), is a framework for facilitating the development of split-drivers for virtualization solutions, as shown in Figure 4. As for gVirtuS, the brightest GVirtuS feature is the independence of all the involved technologies: the hypervisor, the communicator, and the target of the virtualization (general purpose graphics processing units for computing acceleration, high performance network interface cards, distributed parallel file systems, measurement instruments and data acquisition interfaces).

GVirtuS is different from his ancestor gVirtuS in the following ways: *i)* while gVirtuS proposes a virtualization solution for CUDA, GVirtuS uses a plugin-based architecture to offer virtualization support for generic libraries such as accelerator libraries (OpenCL, OpenGL, and CUDA as well), parallel file systems, communication libraries (MPI); *ii)* moreover, gVirtuS required explicit porting on virtualization technologies of front-ends, back-ends, and communicators, while GVirtuS fosters the simplification of the independence of virtualization technology by offering generic interfaces, which simplifies the porting of virtualization solutions across platforms.

In conclusion, GVirtuS is an abstraction layer for generic virtualization in HPC on cloud infrastructures.

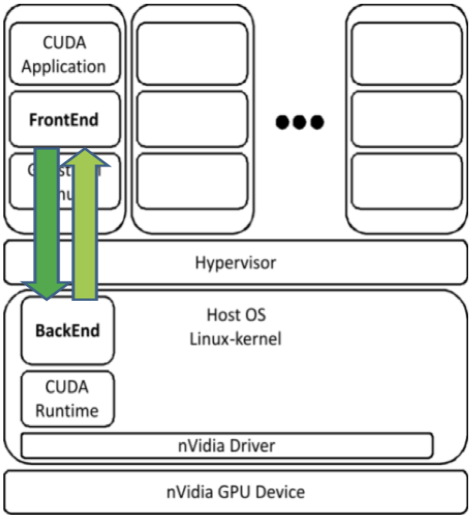

Figure 4. GVirtuS system architecture and design.

4.2. Split-Driver Model in GVirtuS

In GVirtuS the split-drivers are abstracted away, and provides the developers with abstractions of common mechanisms, which can be shared for implementing the desired functionality. In this way, the development of a new virtualization driver is simplified, as it is based on common utilities and communication abstractions. GVirtuS abstracts front-ends, back-ends, and communicators. The GVirtuS software stack is designed in a modular fashion: the frontend, the communicator, and the backend are implemented as plugins. For each virtualized device front-end and back-end are cooperating, while both of them are completely independent of the communicator. Developers can focus their efforts on virtual devices and resource implementation without taking care of the communication technology.

Focusing on GVirtuS, split-driver model is structured as follows:

The front-end driver is the component running on the guest machine (such as a virtual machine, physical machine, etc.), i.e. the machine running the CUDA application that may not have the GPU physical device. It uses the APIs (Application or Abstract Programming Interface) device driver support although the latter is not physically present on the machine. The system works by using a “wrapper library” that using the same programming interface library of the device drivers (specifically we use the CUDA library). This approach allows the binary compatibility (ABI) with the legacy driver (in this case, the CUDA drivers). The task of the wrapper library is intercepting the drivers’ function invocations mocking them with a same signature functions set. Once the program control has been captured in this way, it is possible to perform the invocation of back-end driver’s functions through the front-end driver interface. The wrapper library receives the result from the front-end driver and makes it available to the application.

The back-end driver is the component running on the host machine, typically the machine with the physical support to the GPU device. It receives the invocations by wrapper library through the front-end driver. Then it performs the functions invocation to the physical device with GPU support using the legacy APIs interface. Finally, the back-end driver returns the results to the front-end driver.

The software component used for communication between the back-end driver and the front-end driver is the "communicator".

4.3. Communication Interface in GVirtuS

A *communicator* is a key piece of software in the GVirtuS' stack because it connects the front-end and back-end operating systems. The communicators have strict high-performance requirements, as they are used in system-critical components such as split-drivers. Additionally, in a virtual environment the isolation between host and guest and among virtual machines is a design requirement. Consequently, the communicator's main goal is to provide secure high-performance direct communication mechanisms between guest and host operating systems.

In GVirtuS the communicators are independent from the hypervisor and from the virtualized technology. Additionally, novel communicator implementations can be provided independently from the cooperation protocols between front-end and back-end.

GVirtuS provides several communicator implementations, including also a Transmission Control Protocol/Internet Protocol (TCP/IP) communicator. The TCP/IP communicator is used for supporting distributed resources, however it can be used between virtualized resources if there are not more high performance communicators. In this way, a virtual machine running on a local host could access a virtual resource physically connected to a remote host in a transparent way. However, in some application scenarios the TCP/IP based communicator is not feasible because of the following limitations:

- The performance is strongly impacted by the protocol stack overhead.
- In a large size public or private cloud computing environment, the use of the network could be restricted for security and performance reasons.

For addressing these potential limitations, the GVirtuS plugins architecture allows for future protocols to be simply integrated into the architecture without any frontend or backend modification.

4.4. GVirtuS CUDArt Plugin

The GVirtuS CUDArt plugin provides a wrapped library that emulates the familiar driver API abstraction for the guest application. The stub library collects the request parameters from the application and passes them to the back-end driver, converting the driver API call into a corresponding front-end driver call (Figure 5).

When a callback is received from the front-end driver, it delivers the response messages to the application. In GVirtuS, the front-end runs on the virtual machine instance and it is implemented as a stub library. A stub library is a virtualization of the physical driver library on the guest operating system. The stub library implements the driver functionality in the guest operating system in cooperation with the back-end running on the host operating system.

Here we describe the implementation of the front-end plugins implementing CUDA version 6.5.

The *CudaRTFrontend* class establishes the connection with the back-end and executes the CUDA routine through the library libGvirtus-frontend. The constructor method creates an object of the class Frontend from the libGvirtus-frontend library using the method *GetFrontend*. This instance of the class will be alive through all the life cycle of the application and will be used any time we need a method

from the *CudaRTFrontend* class. The stub methods have a common schema. Every stub exposes the same interface of the handled CUDA routine. The first step is to get the instance of the GVirtuS Frontend class. This task is accomplished by the constructor method. The Prepare method resets the input buffer that will contain the parameters to send to the back-end. After that, all the parameters are inserted into the buffer, the Execute method forwards the request for the routine using the name of the routine as parameter. If the method is executed on the back-end, we can get the output parameters. Through the method *GetExitCode* we can get the exit code of the routine executed by the back-end. Using *cudaGetDeviceCount* and *cudaSetDevice* it is possible to get info about all the devices attached to the physical machine. This simple explicative schema is common to all the stubs coded.

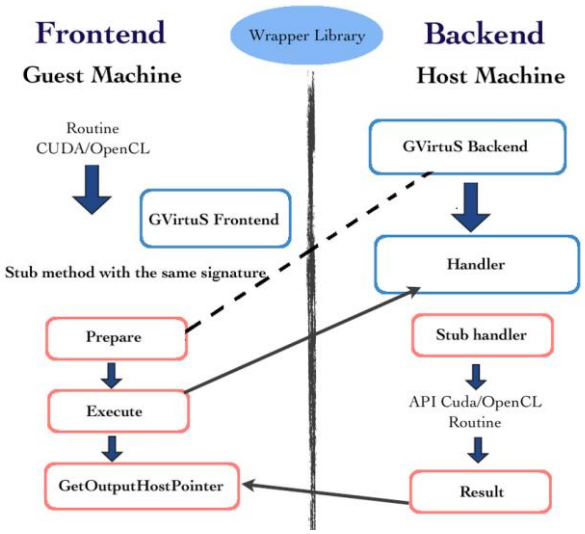

Figure 5. The GVirtuS approach to the split-driver model.

4.5. CUDA Kernel map and retrieve

The backend is a component that serves the frontend 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 that are submitted by front-ends. In an environment requiring shared resource access (as it is very common in cloud computing), the back-end must offer a form of resource multiplexing.

A *daemon* runs on the back-end in the user space or super user space, depending on the specific of application and security policy. The daemon implements the back-end functionality that deals with the physical device driver and performs the host-side virtualization.

GVirtuS can be configured by means of an ad-hoc file named gvirtus.properties. At the current stage of development, GVirtuS only handles two parameters, namely the communicator and the plugin. The first parameter selects the kind of communicator has to be used, choosing from a list of available communication mechanisms. The second one selects the plugins to be loaded. The main task of GvirtuS back-end is to start a communication in server mode, then accepting new incoming connections. It handles the loading of previously installed plugins.

As previously stated about the CUDA plug-in, GvirtuS back-end invokes the *GetHandler* method in order to create a new instance of *CudaRTHandler* class containing all the methods needed to serve the requests of CUDA routine execution. In this class, it is possible to find all the methods to handle the execution of CUDA routines. In the CudaHandler class there is a table, *mpsHandlers*, which associates the function pointers to the name of the routines, so that any routine can be handled in the right way. As in the front-end there is a stub method for each CUDA method, in the back-end there is a function that controls the execution of each method.

4.6. Device memory and CUDA map retrieve

Device memory (the memory allocated via *cudaMalloc* method family) and Host memory (the memory not allocated via CUDA methods) are treated differently.

As the device memory is concerned, the system does not track down the allocated pointers, which are stored client-side and passed back to the back-end when they are involved in a computation. For instance, this is the case of the arrays, *cudaArrays* and similar device specific structures.

In the CUDA API, there are structures that are allocated on the host memory and then bound to device specific structures. Such a behavior of that CUDA API must be treated differently in GVirtuS.

The back-end tracks down all the host pointers bound to the device memory by using a suitable map. When the execution requires the usage of one of those pointers, the back-end retrieves the associated structure before the kernel execution.

4.7. Simple CUDA life cycle application

In the following, we provide the description of a simple CUDA application's life cycle. The back-end is listening and ready to accept requests by front-ends, using one of the available communicators. The CUDA application is executed on the frontend by means of the code:

```
#include <stdio.h>
#include <cuda.h>

int main(void) {
    int n;
    cudaGetDeviceCount(&n);
    printf("Number of CUDA capable GPU(s): %d\n", n);
    return 0;
}
```

This application displays the numbers of the CUDA enabled devices. When the function *cudaGetDeviceCount* is invoked, the stub GVirtuS - frontend is activated. This stub is implemented in the following way:

```
extern "C" cudaError_t cudaGetDeviceCount(int *count) {
    Frontend *f = Frontend::GetFrontend();
    f->AddHostPointerForArguments(count);
    f->Execute("cudaGetDeviceCount");
    if(f->Success())
        *count = *(f->GetOutputHostPointer<int>());
    return f->GetExitCode();
}
```

In all the stubs, the first instruction obtains a reference of the front-end class. In the first invocation of a CUDA routine, the front-end object is not created, so the static method *GetFrontend* will create the new object of the Frontend class. The Frontend's private constructor opens a connection with the backend. The back-end instantiates a new *Process* to handle the execution requests from this application.

The *AddHostPointerForArguments* instruction adds the value of the object pointed by count in the input buffer. It will be sent to the back-end.

The *Execute* instruction sends the execution request and the associated buffer containing the serialized input parameters to the back-end. The Frontend's *Execute* method locks the execution, waiting for the back-end to run the requested routine.

The Backend instantiated process receives the execution request, and using the *Execute* method provided by the class *CudaRtHandler*, launches the associated handler. The code is the following:

```
Result * handleGetDeviceCount(CudaRtHandler * pThis, Buffer * input_buffer) {
    int *count = input_buffer->Assign<int>();

    cudaError_t exit_code = cudaGetDeviceCount(count);

    Buffer *out = new Buffer();
    out->Add(count);

    return new Result(exit_code, out);
}
```

The handler function obtains the input parameters, runs the requested CUDA routine, creates a new buffer containing the output parameters and then returns an object of *Result* type. This Result object contains the exit value of the CUDA routine and the Buffer object which contains the output parameters. The Process object will send to the front-end the Result object just created.

The front-end receives the exit value of the CUDA routine and the buffer containing the output parameters. The front-end's stub checks the correctness of the execution.

Finally, the stub process ends up returning the exit code from the CUDA routine. The application displays the number of the available devices and then terminates.

The destructor of the Frontend class closes the communication automatically when the application terminates.

5. Conclusions

In this document we focused on basic issues of the RAPID project, namely the private cloud infrastructure design, the choice of the hypervisor software, and the implementation of the GPU acceleration strategy.

We decided to utilize OpenStack as the Cloud Manager Platform tool, since it is able to successfully integrate all the requirements identified by the RAPID's applications, in particular the need for providing CUDA access to virtual machine. Moreover, the process of evaluation and comparison of several hypervisors software led to the choice of the KVM hypervisor, the de-facto standard for virtual machines in a Linux environment. GPU acceleration on the Virtual Machines will be provided by the GVirtuS framework, a generic virtualization tool for high performance computing, dedicated, but not limited to, transparent GPGPU virtualization with CUDA.

References

[1] Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared

[2] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: Gvim: Gpu-accelerated virtual machines. In: HPCVirt 2009: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17–24. ACM, New York (2009)

[3] http://www.xenproject.org/

[4] http://www.linux-kvm.org/

[5] https://www.virtualbox.org

[6] http://www.microsoft.com/en-us/server-cloud/solutions/virtualization-private-cloud.aspx

[7] http://azure.microsoft.com/

[8] http://www.microsoft.com/en-us/server-cloud/products/system-center-2012-r2/

[9] http://www.nvidia.com/object/microsoft.html

[10] http://www.vmware.com/products/vcloud-suite/

[11] https://www.vmware.com/products/horizon-view/features/vGPU.html

[12] https://www.citrix.com/products/xenserver/overview.html

[13] http://international.download.nvidia.com/pdf/grid/resources/NVIDIA-GRID-Case-Study-Peugeot-Citroen-OCT-2014.pdf

[14] https://www.openstack.org/

[15] https://www.openstack.org/foundation/companies/

[16] https://www.docker.com/

[17] https://wiki.openstack.org/wiki/Docker

[18] https://cloudstack.apache.org/

[19] http://www.eucalyptus.com/

[20] http://www.opennebula.org/

[21] https://www.eucalyptus.com/sites/all/files/HP-Eucalyptus-Master-Services-Agreement.pdf

[22] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," in INFOCOM, 2012 Proceedings IEEE, pp.945-953, 25-30 March 2012, doi: 10.1109/INFCOM.2012.6195845

[23] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," Book Chapter, Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, pp 379-391, Springer Berlin Heidelberg, 2010.

[24] https://wiki.openstack.org/wiki/HeterogeneousGpuAcceleratorSupport

[25] P. V.V.Reddy, L. Rajamani; *"Performance Evaluation of Hypervisors in the Private Cloud based on System Information using SIGAR Framework and for System Workloads using Passmark"*, International Journal of Advanced Science and Technology, Vol 70 (2014), pp.17-32. http://www.sersc.org/journals/IJAST/vol70/3.pdf

[26] S. Shirinbab, L. Lundberg, D. Ilie, *"Performance Comparison of KVM, VMware and XenServer using a Large Telecommunication Application"*; Published in the fifth International Conference on Cloud Computing, GRIDs, and Virtualization, 2014, pp 114-122.

[27] J. Hwang, S. Zeng, F. Wu, and T. Wood; *"A component-based performance comparison of four hypervisors".* In Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on, pages 269–276, May 2013

[28] Walters, J.P.; Younge, A.J.; Dong In Kang; Ke Thia Yao; Mikyung Kang; Crago, S.P.; Fox, G.C., *"GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications",* in Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on , vol., no., pp.636-643, June 27 2014-July 2 2014

[29] Vasudevan.M.S, Biju.R.Mohan and Deepak.K.Damodaran; *"Performance Measuring and Comparison of VirtualBox and VMware"*; 2012 International Conference on Information and Computer Networks (ICICN 2012); IPCSIT vol. 27 (2012); IACSIT Press, Singapore.

[30] Montella, Raffaele, Giuseppe Coviello, Giulio Giunta, Giuliano Laccetti, Florin Isaila, and Javier Garcia Blas. "A general-purpose virtualization service for HPC on cloud computing: an application to GPUs." In *Parallel Processing and Applied Mathematics*, pp. 740-749. Springer Berlin Heidelberg, 2012.