CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading

Raffaele Montella[1,2]-[1], Sokol Kosta[3], David Oro[4], Javier Vera[4], Carles Fernández[4], Carlo Palmieri[1], Diana Di Luccio[1,2], Giulio Giunta[1], Marco Lapegna[5] and Giuliano Laccetti[5]

[1]*Department of Science and Technologies, University of Napoli Parthenope, Italy*
[2]*Computation Institute, University of Chicago, USA*
[3]*CMI, Aalborg University Copenhagen, Denmark*
[4]*Herta Security, Spain*
[5]*Department of Mathematics and Applications, University of Naples Federico II, Italy*

SUMMARY

Low-power devices are usually highly constrained in terms of CPU computing power, memory, and GPGPU resources for real-time applications to run. In this paper we describe RAPID, a complete framework suite for computation offloading to help low-powered devices overcome these limitations. RAPID supports CPU and GPGPU computation offloading on Linux and Android devices. Moreover, the framework implements lightweight secure data transmission of the offloading operations. We present the architecture of the framework, showing the integration of the CPU and GPGPU offloading modules. We show by extensive experiments that the overhead introduced by the security layer is negligible. We present the first benchmark results showing that Java/Android GPGPU code offloading is possible. Finally, we show the adoption of the GPGPU offloading into *BioSurveillance*, a commercial real-time face recognition application. The results show that, thanks to RAPID, *BioSurveillance* is being successfully adapted to run on low-power devices. The proposed framework is highly modular and exposes a rich Application Programming Interface (API) to developers, making it highly versatile while hiding the complexity of the underlying networking layer.
Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Virtualization; GPGPU; Offloading; CUDA; Mobile Cloud Computing; Android

1. INTRODUCTION

Nowadays, we use mobile devices to write documents, browse the Internet, explore maps, watch or edit videos, play games, perform all the tasks we used to run on powerful computers, and many more. Users are so attached to their smart mobile devices, that researchers have found that they

[1]Correspondence to: Department of Science and Technologies, University of Napoli Parthenope, CDN Isola C4, Napoli, Italy. E-mail: raffaele.montella@uniparthenope.it

Copyright © 0000 John Wiley & Sons, Ltd.

would prefer to use their own devices even for working, a concept known as *Bring Your Own Device (BYOD)*[2].

To keep up with users' demands and applications' needs for ever-increasing computational resources, devices are being equipped with a myriad of additional hardware, sensors, and extra capabilities. Nevertheless, low-power devices do not possess such features, making newest applications difficult or even impossible to run. Furthermore, advances in battery technology have not been able to follow the fast smartphone race, making energy consumption a bottleneck for most users. To address these problems, developers and companies have engineered solutions that offload computational intensive operations from the low-power mobile devices to more resourceful remote machines, usually residing on the cloud [1], [2], [3], [4], [5], [6]. However, none of these works deals with securing the data during the offloading process [5]. Moreover, only recently researchers have proposed the need for offloading not only computationally-intensive CPU tasks but also operations performed by the GPU [7], [8], [9]. This mechanism can help the low-power devices overcome the limitations arising from the absence of a GPGPU component or from the small amount of resources available on such devices.

Focusing on CUDA GPGPU programming environment, the lack of offloading frameworks is due to NVIDIA's CUDA GPGPU proprietary platform, which collides with the open source philosophy [10]. This way, when it comes to Android devices, for example, CUDA availability is limited only to brand-specific Android products powered by the Tegra system on chip (SoC), severely hurting the mass adoption of GPU computation in the Android developers' community[3]. OpenCL represents a more open alternative to NVIDIA's CUDA, sharing the same programming model and achieving almost the same performance under selected benchmarks [11]. Nevertheless, Google has openly displayed its opposition to OpenCL, because mobile devices need performance portability between many different GPU and CPU vendors with very different architectures[4], motivating the support of parallel programming in the Renderscript model [12]. Although many vendors use OpenCL as Renderscript back-end, there is no standard OpenCL programming model for Android, because as for now it is impossible to write a unique kernel able to run acceptably on a range of devices. The offloading approach proposed in this paper overcomes these limitations, allowing CUDA or OpenCL code for GPGPU computation to be integrated in all Android devices, even on those without a GPU. Given that there is no significant difference in our solution with respect to CUDA or OpenCL, in this paper we present our contribution and results focusing only in the CUDA GPGPU approach. Moreover, not only Android devices but also low-powered Linux devices without a GPU can benefit from the GPGPU code offloading process, as we show in our real deployment of the *BioSurveillance* experiments.

With the emergence of deep neural networks (DNNs), there is a need to perform neural network inference for solving tasks such as object detection and recognition on legacy low-powered Linux and Android mobile devices. Highly-accurate proposed convolutional neural network (CNN) architectures such as ResNet [13] or Faster R-CNN [14] are typically computed on CUDA-enabled platforms, and require a high-end power-hungry discrete GPU for processing big pictures on a

[2]http://www.dell.com/en-uk/work/learn/mobility-byod
[3]https://developer.nvidia.com/codeworks-android
[4]http://stackoverflow.com/questions/14385843/why-did-google-choose-renderscript-instead-of-opencl

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

reasonable time. However, an increasing number of end-user mobile applications targeting low-power Android devices demand near real-time performance and high accuracy for object recognition tasks. We believe that bringing generic CUDA support to the Android world will open the door to transparently perform inference of such complex CNNs in an easy manner, and benefit from the existing deep learning GPU ecosystem.

In this paper, we discuss the design and implementation of RAPID, a complete offloading framework that enables remote offloading of CPU computations and resource-intensive GPGPU kernels on Android and Linux applications. RAPID comes in different flavours, so that it can support many low-power devices. Precisely, RAPID supports:

- CPU and GPGPU remote task offloading on Android devices;
- CPU and GPGPU remote task offloading on Java applications for Linux devices;
- GPGPU remote task offloading on C/C++ applications for Linux devices;

The CPU task offloading is based on ThinkAir [1], an offloading framework for Android applications. In this work we have extended ThinkAir to also support Java applications running on Linux devices. Moreover, we have added a security layer for encrypting the offloaded data to avoid eavesdropping of sensitive information by malicious users. As shown in [5], no other offloading framework offers this type of security. We show by extensive experiments that the data encryption process is quite lightweight and negligible in the case of non real-time applications.

The GPGPU task offloading is based on GVirtuS [15], an offloading framework for GPGPU task offloading on C/C++ Linux applications. In this work, we extend the existing GVirtuS with the design and the development of the completely brand new GVirtuS for Java and GVirtuS for Android frameworks. From the technical point of view, we improved the overall GPGPU remoting and virtualization availability, updating the support to the newest CUDA 8.0 APIs as compared to the previous implementation supporting CUDA 6.5, and enriching the set of supported functions with more advanced libraries, such as CUDA Fast Fourier Transform library (cuFFT), CUDA Basic Linear Algebra Subroutines (cuBLAS), and CUDA Deep Neural Network library (cuDNN).

Even though task offloading is a well known technique in the distributed computing landscape, it has not been adopted widely in the wild yet. One of the reasons for this, is that developers prefer to develop following the traditional approach of setting up web service infrastructures able to handle requests in a client-server fashion. To start developing applications using an offloading framework, a developer should first learn how to use the new tools, which, depending also on the clarity of the documentation, may be quite a hassle, compared to the traditional approach. Moreover, the recent trend of *cloud serverless computing*, such as Amazon AWS Lambda[5] or Google Cloud Function[6], may seem more attractive, given that they are supported by big technology drivers. The proposed approach enables the developer to implement offloadable CPU and GPU code avoiding any issue related to the infrastructure management. One of the most important contributions of the paper, is enabling developers to embed GPGPU code in Android/Java applications, which can later be distributed in regular ways (for example using the Google Play store) without the need to deal with infrastructure issues (technical details, virtual machine management, load balancing,

[5]https://aws.amazon.com/lambda/
[6]https://cloud.google.com/functions/

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

security, scalability, etc.). With the proposed approach, the typical Android development cycle is: i) developing the app code, ii) developing the GPGPU code using the regular CUDA toolchain, iii) embedding the GPGPU code in the application, iv) generating the signed APK, v) releasing the application on the Google Play store.

We perform different experiments and show that GPGPU offloading not only is possible, but is also convenient. We consider a matrix multiplication as a benchmark application to demonstrate the GPGPU offloading on Android devices, while on Linux devices we assess the effectiveness of the GPGPU offloading using the *BioSurveillance* application. In this paper, we omit the results about the benefits of the CPU task offloading, given that these results have been assessed extensively by many previous works, as described in Section 3, where in particular we compare and contrast different offloading approaches with the web service option considered as a common baseline. The proposed framework is subject to some limitations due to the task offloading itself. Some of those limitations could be addressed and mitigated as the overhead caused by the networking and the security management, others could be considered as a kick-start for future developments and could be turned into opportunities as the exploitation as a offloading task provider for public clouds entrepreneurs.

The rest of the paper is organized as follows: in Section 2 we give a short description of application scenarios; in Section 3 we position our paper with respect to the state of the art; in Section 4 we describe the high-level architecture of the RAPID framework, identifying its main components; in Section 4.2 we give an overview of the GPU virtualization technique we designed and used in our system; in Section 5 we present the design and implementation details of the Linux and Android GPGPU task offloading component; in Section 6 we show the results of an Android application exploiting GPGPU offloading and the results obtained by offloading parts of the real-time face recognition procedure in the *BioSurveillance* application; and finally, in Section 7 we conclude the paper with remarks on future directions.

2. REAL USE CASE SCENARIOS

We select two challenging compute-intensive application scenarios that extensively leverage CUDA on both low-power devices and HPC resources on which the contribution could impact: *i)* a real-time face recognition application and *ii)* an environmental model setup. Lately in this work, we use the first one as evaluation scenario.

The real-time face analytics application is being developed by Herta Security. The application relies extensively on CUDA kernels for performing the steps of face detection, facial template extraction, and matching over high-definition live video stream feeds. Performing these operations in real-time is already a difficult task. In the future, surveillance cameras are expected to provide even higher definition, 4K or even 8K [16], which means that performing real-time face recognition on crowded environments will become nearly impossible. In particular, performing CNN inference of complex models locally at such high resolutions on a low-power envelope (typically SoCs dissipating 3 W) is the biggest challenge. Moreover, the recent stagnation of Moore's Law, and projections reported recently in the latest International Technology Roadmap for Semiconductors (ITRS), point out that transistors will stop shrinking in 2021 [17], thus highlighting the necessity

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

of transparently offloading such compute-intensive GPU machine learning computations to remote GPU server clouds.

WaComM (Water quality Community Model) is a three-dimensional decision support model enabling the simulation and prediction of pollutant spills, its transportation, and dispersion in both inshore and offshore environments. Its development is contextualized in the effort for a robust, stable and consistent operational forecast system to protect mussel farms from pollution, which is one of the outstanding issues in many developing world economies. The numerical models[18], with a specific parametrization and spatial/temporal resolution, have been integrated in the Framework to Advance Climate, Economic, and Impact Investigations with Information Technology (Face-IT) work-flow system [19], providing support for experiment repeatability and interactive data publishing [20, 21]. WaComM uses a hybrid approach, based on Eulerian-Lagrangian models [22], implemented using a heterogeneous parallel approach[23]. WaComM requires high performance computing (HPC) capabilities to be able to provide near real-time results [24, 25], impacting the on-premise total cost of ownership [26]. The operational costs derived from such expensive infrastructures could be mitigated by the GPU offloader, running the CUDA applications in dedicated remote commodity GPU accelerator servers.

3. RELATED WORK

	CPU offloading	GPGPU offloading	Native (C/C++) offloading	Parallel. support	Plug-in communicator	Secure communicator	Mobile support	Linux support
Client-Server	/	/	/	/	/	/	/	/
ThinkAir [1]	✔			✔			✔	
MAUI [27]	✔						✔	
Cuckoo [28]	✔						✔	
Giurgiu [29]	✔						✔	
COMPSs [30]	✔			✔			✔	
POWER [31]	✔						✔	
GVirtuS [15]	✔				✔		✔	
rCUDA [32]		✔						✔
JCUDA [33]		✔						✔
This work	✔	✔	✔	✔	Only GPGPU	✔	✔	✔

Table I. Related work comparison

With the massive proliferation of mobile devices in recent years, researchers soon realized that the low computational capabilities and limited battery capacity constituted a bottleneck to end-users. In order to solve this challenge, different solutions have been proposed to boost the performance

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

of low-power devices by migrating resource-intensive computations to more powerful machines featuring a higher Thermal Design Point (TDP).

Table I synthesizes the contributions of the most prominent works related to this paper. As a reference, we also include the Client-Server baseline in the table. It is obvious that an experienced developer could implement the *offloading* characteristics manually, as it has been the case traditionally. In that case, the developer would implement a client and a server side for the application. The developer would take care of implementing the logic of deciding where to execute each heavy task, sending the task and the input data to the server, running the task on the server, and getting the result to/from the server. The drawback of this architecture is that the client and the server are tightly coupled with each other, making the developer responsible for implementing, debugging, and maintaining both sides of the application.

Contrary to the tightly coupled paradigm of the client-server architecture, offloading frameworks offer a higher degree of flexibility to developers, allowing them to focus only on the client side of the application. Using these frameworks, developers implement their applications as if everything were to be run locally on the client device. Indeed, the frameworks try to hide as much as possible the complexity of the offloading decision, the data transmission, and the remote execution from the developer. In this case, developers do not need to worry about the remote side at all, given that this is part of the offloading framework. When we compare the client-server architecture with the other offloading frameworks, we assume that nothing prevents an experienced developer to implement any of the features of any offloading framework. However, as mentioned earlier, this would require the developer to manually handle every aspect of computation remoting.

Based on their characteristics, we categorize the related works by eight aspects:

- **CPU offloading** indicates if the framework supports CPU computation offloading.
- **GPGPU offloading** indicates if the framework supports GPGPU computation offloading.
- **Native (C/C++) offloading** indicates if the framework supports native code offloading. This is an important feature, given that many applications embed native libraries that implement computationally intensive operations.
- **Parallelization support** indicates if the framework is able to handle automatic task parallelization on the remote side.
- **Plug-in communicator** indicates if the communication between the client side and the remote side is easily replaceable with another communication technology without interfering with the rest of the framework.
- **Secure communicator** indicates if the communication between the client side and the remote side is protected by means of encryption.
- **Mobile support** indicates if the framework supports offloading of mobile applications such as Android, iOS, or Windows.
- **Linux support** indicates if the framework supports offloading of Linux applications.

The rest of the section gives a more detailed description of the works shown in the table.

CloneCloud is one the first popular frameworks in this direction [34]. CloneCloud uses an offline static analysis of the apps' binary to determine the code sections that are better to offload to the cloud. The developer should then use the output of this analysis to build a database of offloading decisions, which are later loaded on the device. On runtime, when an offloadable portion of the

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

code has to be executed, the framework queries the database to decide where to execute the code. CloneCloud presents the clear benefit of working on the application binaries and not on the source code, being completely transparent to the developers. However, *training* the offloading decision to consider all possible combinations of parameters of influence, such as network type, latency, bandwidth, etc., is a tedious work to be performed for each application. MAUI [27] and ThinkAir [1] are two alternative popular solutions that try to mitigate the issues of CloneCloud of static offline partitioning. However, a drawback compared to CloneCloud is that these frameworks work at the source code level, requiring the developer to be aware of the offloading environment. MAUI is a method-level framework for Windows phones, which uses .NET language to achieve remote method offloading and invocation. ThinkAir is a method-based offloading framework for Android devices. The smartphone is associated with a virtual machine (VM) on a private or public cloud, and the offloadable methods are sent to the VM for remote execution. ThinkAir also supports dynamic resource allocation, exploiting the power of the cloud whenever the offloaded method can be executed in parallel on multiple VMs. Cuckoo [28] is another offloading framework for Android devices that works at the source code level, similar to MAUI and ThinkAir. Cuckoo also provides the developers with an Eclipse extension to facilitate the programming model. In [29], the authors present the implementation of a framework for Android that also considers the size of input data when partitioning applications for remote execution. POWER [31] is a recent offloading framework that is built on top of previous works. In this paper, the authors argue about the feasibility of offloading between devices of different architectures. COMPSs-Mobile [30] advances on previous works and proposes a parallelized version of the offloading paradigm for Android devices.

All the above-mentioned works strictly focus on offloading of CPU computations. Only recently researchers have started proposing the first ideas about GPGPU code offloading on mobile devices [7], [9]. The rest of this section focuses on the available GPU virtualization technologies and frameworks to better put our solution in context.

One of the most prominent solutions related to concurrent remote usage of CUDA-enabled devices in a transparent way is rCUDA [32]. Thanks to the split-driver approach, there is no need to modify and recompile the CUDA-enabled application in order to use it with rCUDA. Indeed, the framework takes care of all the necessary details in order to execute the CUDA kernels on a remote or local GPGPU [35]. The overhead introduced by using a remote GPU is evaluable as about less than 4% when a high performance network fabric is used [36]. At the time of writing, rCUDA delivers high performance CUDA virtualization and it is up to date supporting the latest CUDA 8.0 framework and its ancillary libraries exploiting the full advantage of using InfiniBand and the recently developed support for RoCE networks (RDMA over Converged Ethernet). The rCUDA binary distribution can be requested from the rCUDA web site[7] at no charge, but it is not freely available as open source.

JCUDA [33] is a Java framework that enables CUDA kernel invocation by delegating the responsibility of generating the Java-CUDA interface code and host-device data transfer calls to the compiler. The JCUDA implementation handles data transfers of primitives and multidimensional arrays of primitives between the host and device. The framework works thanks to the Java Native method invocation. Under this scheme, C functions are declared as native external methods dealing

[7] http://rcuda.net

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

with out-of-the-sandbox unmanaged memory. Additionally, it automates the process of CUDA kernel external compiling and JNI compliant interface code generation by offering a complete integrated device/host memory management.

Even though these related works share several similarities with our proposed GPU offloading mechanism, they all suffer from important drawbacks. For instance, while JCUDA offers a remarkable solution to the problem of Java-CUDA kernel development, it cannot be directly applied in the Android context, as the native CUDA compiler support is mandatory at runtime. rCUDA is an excellent GPGPU virtualization and remoting middleware characterized by notable performance and stability, sharing with GVirtuS many features and limitations, but unfortunately rCUDA is not open source.

Our conference paper [37], which represents the baseline for this work, already shows some preliminary results about the designing and the development of a framework supporting CUDA kernel offloading for regular Android applications have been demonstrated. The extended work presented in this paper differs from the conference paper in the following:

- The overall architecture is described with more details and it is contextualized in the RAPID framework, where GVirtuS (the GPGPU virtualization and remoting engine) [15] and ThinkAir (the offloading framework) are designed to be integrated and deeply cooperate in order to accelerate low-power devices like SoCs, wearable, and mobile.
- We have added a security layer to the offloading framework, used to protect the data transmitted during the offloading process from malicious users.
- This work focuses on the peculiar GVirtuS split-driver modular architecture used to virtualize and remote GPGPUs on both Linux and Android.
- A bunch of technical updating are given as, but not limited to, the GVirtuS version 8.0 supporting CUDA 8.0, the support to both PTX and *cubin* formats for Android CUDA kernel embedding.
- The new brand support for CUDA companion libraries such as cuFFT, cuBLAS, and cuDNN. A general refactoring has been performed on the Android front-end achieving better overall offloading performance.
- Even though the matrix product is still used in both papers as comparison tool for performance evaluation, in this paper we present the adoption of GVirtuS into a real-time face recognition *BioSurveillance* experimental application. The use of GVirtuS for GPGPU remoting is crucial for these kind of applications where GPGPU-capable SoCs are currently not powerful enough.

As demonstrated in this paper, the RAPID framework is in an active development state. Nevertheless, its CPU and GPGPU offloading technologies represent the current state of the art and are mature enough to be used in the wild for a selected application scenarios.

4. THE RAPID OFFLOADING SERVICE ARCHITECTURE

The proposed offloading framework is highly modular and it is composed of two main entities: the Acceleration Client (AC) and the Acceleration Server (AS). The AC is implemented in two versions, as a Java Linux and as an Android library, that specifies the Application Programming

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

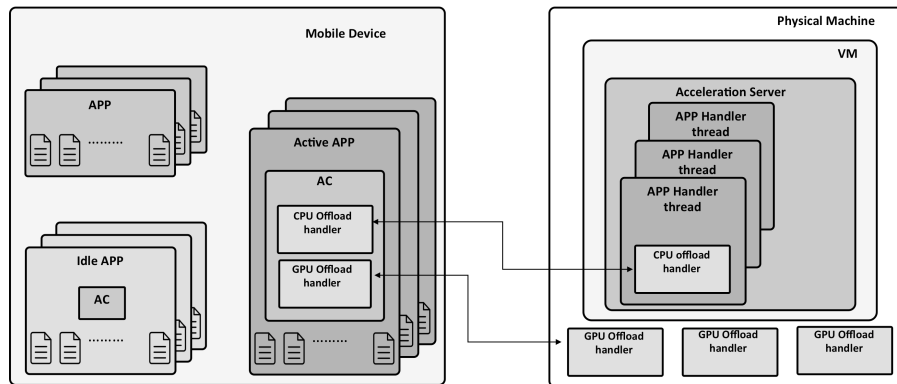

Figure 1. The architecture of the offloading framework, showing the AC, AS, CPU offload handler, and GPU offload handler.

Interface (API) that developers should use to implement method offloading. The AS is a server application that runs on a remote machine and receives offloaded methods from the AC for remote execution. On the other hand, the AC defines two API flavors, each one specialized to handle either CPU or GPU instructions. As such, the API dealing with CPU offloading is inspired by the ThinkAir framework [1], while the API dealing with GPU offloading is a novel implementation and Android-adapted version of the GVirtuS system.

The programming model for CPU task offloading is quite straightforward. The developer needs to simply tag the compute-intensive methods using *@Remote* Java annotations, where *Remote* is a tagging interface defined in RAPID. Then, the RAPID compiler converts the annotated code by adding the necessary code that will handle the offloading during runtime. When an annotated and modified method is called for execution, the AC takes the method and the state of the calling object and sends them to the AS. The AS then executes the method using Java Reflection and returns the result of the execution to the AC, which forwards it to the calling object. The CPU computation offloading is quite transparent to the developers, thanks to the fact that these methods are implemented in Java, which provides all the mechanisms needed for object serialization and remote execution via Reflection. When it comes to GPGPU code, however, the programming languages such as CUDA or OpenCL do not provide such features. In this case, the developer needs to utilize RAPID's API to be able to perform GPGPU computation offloading. The API we provide is composed of Java/Android method wrappers of the original CUDA functions. Obviously, the wrappers have the same signature as the respective CUDA functions, facilitating this way the adoption of our framework. Finally, the developer needs also to include the compiled CUDA functions as raw files in a specific folder in the application. During runtime, when a wrapper method of a CUDA function is called, the compiled CUDA code together with the input values will be transferred on the remote side, where the GPU offload handler will take care of executing it on the remote GPU. Then, the result of the invocation will be returned to the client.

Figure 1 shows a device on the left with multiple applications, some of which make use of the AC to migrate the heavy tasks. We can notice the two different modules implemented by the AC that are used to handle the CPU and GPU tasks offloading. On the right side of the figure, we can see the Acceleration Server running on a Virtual Machine (VM) with the same operating system as the device (e.g. Android OS). When an AC connects with the AS, the AS starts a new thread (*App*

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Handler) to serve the respective application, making it possible to support multiple applications on one VM. The CPU offload handler on the AS receives the migrated code, executes it, and then sends the result of the execution back to the AC. When the AC has to offload GPGPU operations, it connects to the GPU offload handler on the remote physical machine, which is part of the GPU virtualized framework. In the rest of the paper, we will provide a more in-depth overview of the offloading framework architecture by emphasizing the aspects of the GPU offloading component.

4.1. CPU Code Offloading

CPU offloading is an extension of ThinkAir [1], a method-level offloading framework for Android. ThinkAir is composed of three major components:

The Compiler is a source-to-source code converter, which helps the developers with the offloading process, facilitating their work by hiding the complexity of the offloading operations. This components comes together with a *programming model*, which allows developers to utilize Java annotations, such as @Remote, to *tag* the compute-intensive methods as offloadable. Then, the compiler converts the tagged methods to support offloading. In particular, the compiler injects portions of code that are able to handle computation offloading via the *Execution Controller*.

The Execution Controller is the component that handles computation offloading on the client side. Developers should embed it as a library in their application. The compiler will inject the needed code in the client's classes where @Remote tagged methods are found. The Execution Controller takes care of the communication with the Application Server, which the remote side of ThinkAir running on the VM and executing the offloaded code. Moreover, the Execution Controller includes a decision engine, which decides dynamically at runtime if a method should be offloaded or if it should be executed locally on the device. Offloading decisions are based on methods' resource utilization, which are monitored via ThinkAir's profilers: *i)* Hardware profiler, *ii)* Software profiler, and *iii)* Network profiler. Finally, the Execution Controller also includes an energy estimation model, which uses the profiled values to estimate the energy spent by each method.

The Application Server is the component that runs on the VM, responsible for executing the offloaded methods via Java Reflection. Moreover, if a method is able to exploit parallelization, this component will dynamically trigger a process for allocating multiple VMs and will distribute the offloaded code for distributed computation.

In this work, we have extended ThinkAir to support newer versions of Android and we have ported it to support Java Linux applications. Furthermore, we have added a security layer to protect the data transmitted from the Acceleration Controller (AC) to the Acceleration Server (AS) during the offloading process. Before the security extensions, the communication between the AC and the AS was performed using Java's *Socket*[8] and *ServerSocket*[9]. In the current implementation, the communication is implemented using the secured versions of these sockets, namely the *SSLSocket*[10]

[8] https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
[9] https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html
[10] https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSocket.html

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

and the *SSLServerSocket*[11]. These sockets offer an encryption layer, such as the Secure Socket Layer (SSL) [38] or the Transport Layer Security (TLS) [39], which protects the communication in three ways:

- **Integrity:** SSL/TLS protects messages from being modified by a malicious user that can stand in between the communication.
- **Authentication:** SSL/TLS offers the possibility for the client to authenticate the server, making sure that it is authentic. This way, the AC can be sure that the AS is not malicious and is executing the tasks correctly. The server can also authenticate the client, making sure that the AC asking for computation offloading is the one allocated to this AS. This way, the AS can be sure that is not running jobs for free-rider clients.
- **Confidentiality:** SSL/TLS supports data encryption, making sure that passive eavesdroppers standing in between the AC and AS would not be able to read the content of the offloaded data or the result of the offloaded computation.

Android replaces the default Java's SSL/TLS library with a customized version of *Bouncy Castle*[12], which provides a lighter implementation of the encryption algorithms. However, Android does not include all Bouncy Castle's algorithms, and that is one of the reasons why the *Spongy Castle* project was born[13]. In our implementation, we used the SSL/TLS sockets provided by java.net.ssl library with the Spongy Castle provider to create the secure communication channel between the entities. We used the default cipher suite choice of the library, namely SSL_RSA_WITH_RC_128_MD5, which means that it uses RSA as symmetric-key exchange algorithm [40], RC4 with 128 bits key as stream cipher, and MD5 as the hash function to provide data integrity.

SSL/TLS works thanks to cryptographic keys embedded in digital certificates, which are managed and distributed by a Public Key Infrastructure (PKI) [41]. The Certificate Authority (CA), is the main entity of the PKI, which uses its own private key to sign other certificates for the clients, the servers, and for other intermediate CAs. The certificates are then used by the clients and the servers for authenticating each-other and for the RSA symmetric-key exchange. In our current implementation, however, we do not implement a PKI. Instead, we manually generate the CA, the client, and the server certificates. Then, we sign clients' and servers' certificates using the CA certificate and embed them on the AC and the AS. Precisely, these are the main steps we performed:

- First, we create the Certificate Authority (CA) cryptographic keys (also known as root keys) using OpenSSL[14]. We use 4096 bits RSA for the root keys and protect the private key with AES 256.
- Second, we use the root keys to create the CA certificate.
- Third, we use OpenSSL to generate the keys for the clients and the servers. We use RSA with 2048 for these keys.
- Fourth, we create the client and server certificates, and sign them with the root CA.
- Finally, we embed the root CA in both AC and AS projects, and embed the client and server certificates in the AC and in the AS, respectively.

[11]https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLServerSocket.html
[12]https://www.bouncycastle.org/
[13]https://rtyley.github.io/spongycastle/
[14]https://www.openssl.org/

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

4.2. GPU Virtualization Architecture

The GPU virtualization architecture is inspired by our previous work done for GVirtuS, which is a generic framework for virtualization solutions based on a split-driver model [42]. This driver model, also known as driver paravirtualization, involves sharing a physical GPU. Hardware management is left to a privileged domain. A front-end driver runs in the unprivileged VM and forwards calls to the back-end driver in the privileged domain [43]. The back-end driver then takes care of sharing resources among virtual machines. This approach requires special drivers for the guest VM. The split driver model is currently the only GPU virtualization technique that effectively allows sharing the same GPU hardware between several VMs simultaneously [44]. This framework offers virtualization for generic GPU libraries on traditional x86 computers. At the current state, GVirtuS supports leading GPGPU programming models such as CUDA and OpenCL. It also enables platform independence from all the underlying involved technologies (i.e. hypervisor, communicator, and target of virtualization).

In particular, as opposite of OpenCL, CUDA is strictly proprietary and not open source, making the use of a virtualization/remoting layer non trivial. In our previous work about GVirtuS [45] we described the GVirtuS development status framed to the CUDA version 6.5. Because the nature of the transparent virtualization and remoting, GVirtuS strictly depends on CUDA APIs version and we motivated the use the 6.5 version by the following issues:

- We had to perform a serious refactoring in order to make GVirtuS compliant with the latest CUDA library design especially about the CUDA kernel management and invocation;
- Since the CUDA 6.5 release, both the compilers and required NVIDIA libraries are also available for ARM architectures (e.g. NVIDIA Jetson development boards);
- GVirtuS represents the core of the RAPID GPGPU accelerator service. In this context the target application requirements are CUDA 6.5 compliant.

As described in [46] with a remarkable level of details, GVirtuS leverages on the split-driver model: a *back-end*, a *communicator* and one or more *front-end*(s).

In the split-driver, the communication issue is critical, especially when the virtualized resources need to be thread-safe, as in case of GPUs providing CUDA support, and the one of the first middleware goals is achieving the best possible performance for the available communication technology (shared memory, TCP/IP, parallel streams, virtio, InfiniBand).

GVirtuS is composed of two main components: the *back-end*, installed on the GPU-capable remote device and the *front-end* installed on the client device. The data transport between the two components is managed by a *communicator*.

4.3. The back-end

In the GVirtuS framework, the back-end performs the actual remote CUDA function invocation. It is a server application running at user privilege with the only need of accessing to the actual CUDA drivers in order to control one or more GPGPU devices. If compared with other paravirtualization approaches, where the need for the super user space level is mandatory for software components designed for native hardware access, the GVirtuS back-end management is made straightforward

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

by the only constraint of dealing with the physical device driver. In a production environment the back-end can run as a daemon matching the system security policies.

4.4. The communicator

The communicator has the duty of high performance data transfer between the front-end and the back-end. In the GVirtuS architecture, the communicator is a pluggable module characterized by a public well defined API, which hides the underlying data transfer technology. This design allows the development of diverse and different communicators in order to match a specific use case or in order to exploit the best performance from a networking devices ecosystem. Each communicator implementation must fit the following strict requirements due to the fact that it is a system-critical component: *i)* security; *ii)* high-performance; *iii)* and liability. As such, the choice of the communicator deeply impacts the GPGPU virtualization and remoting performance.

To contribute in enforcing heterogeneous cloud computing environments [47], GVirtuS comes with several communicator implementations:

- The TCP/IP communicator has been implemented at the GVirtuS early development stages as testing playground [48], nevertheless due to the improvement of performance in host/guest machines communication using virtual networks, it became a first class component. This communicator is under a deep refactoring process in order to enable parallel data transfer for a better bandwidth utilization. The TCP/IP over InfiniBand is used for high performance GPGPU remoting with GVirtuS.
- The AfUnix communicator is used for supporting local uses of GVirtuS. Through this communicator a process can run via GVirtuS without the latency of the TCP/IP communicator. In this case, the back-end and the front-end must both be deployed on the same physical machine.
- VMSocket communicator aims to expose AfUnix socket between KVM virtual machines and the host. It provides a fast and reliable communication channel. VMSocket is implemented as a virtual PCI device inside QEMU, the emulator used by KVM.
- VMShm communicator enables QEMU virtual machines to access to POSIX shared memory objects created on the host OS. VMShm make it possible to an user space application running in a virtual machine to map up to 1M of a POSIX shared memory object from the host OS.

The TCP/IP communicator has been used in this work because it permits the communication between remote machines, so we can compare remote Android and Linux system performances.

4.5. The Linux transparent front-end

The front-end is transparent to the application developers, allowing applications to make CUDA or OpenCL calls without any adaption on the source code. This is achieved thanks to the imitation of the virtualised or remoted shared libraries by intercepting each function call. The front-end is incarnated as CUDA drivers, CUDA runtime, or other CUDA ancillary software components such as cuFFT, cuBLAS, cuDNN, replacing the regular shared CUDA libraries implementation. In this way, when a CUDA application invokes a CUDA function call or it launches a CUDA kernel, the GVirtuS front-end intercepts the call, performs some parameters packaging, taking care about the host and

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

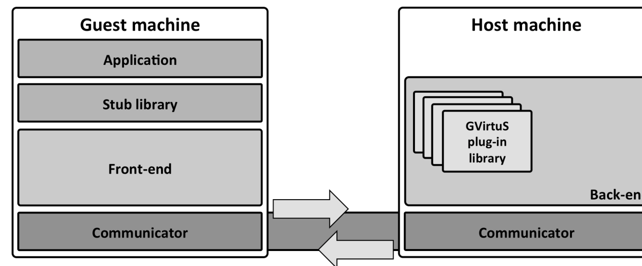

Figure 2. The GVirtuS approach to the split-driver model.

device pointers management, and sends what is needed to the back-end running on the remote machine hosting the physical GPGPU device. Then the front-end waits for the results produced by the back-end remote invocation, unpacks the returned parameters dealing with the pointers type and, finally, returns the result back to the caller program. The stub-library implemented by the front-end provides the APIs abstraction needed to make transparent the remote GPGPU function invocation allowing non-GPGPU-capable machines to be accelerated by CUDA or OpenCL kernels.

4.6. GVirtuS implementation details

GVirtuS is provided with a basic plug-in SDK enabling developers and advanced users to implement the NVIDIA CUDA stack split-driver in a relatively straightforward way. In this scenario, a developer has to subclass from Frontend, Backend and Handler classes. The Frontend class abstracts the communication buffer preparation, the input and output parameters management, function invocation and the gathering of the returned value. Thanks to the GVirtuS modularity and technology/architecture independence, many plug-ins are already available fully or partially implemented as OpenCL, cuFFT, cuBLAS, and cuDNN.

The CUDA-enabled application running on the virtual or remote machine requests GPGPU resources to the virtualized device using the stub-library. The CUDA application must be compiled choosing the use of the shared libraries, otherwise the GVirtuS front-end is not invoked. In general, on the back-end side, each function in the stub-library follows these steps:

- Obtains a reference to a Frontend instance from a FrontendFactory singleton class;
- Uses Frontend class methods for setting the parameters;
- Invokes the Frontend handler method specifying the remote procedure name;
- Checks the remote procedure call results and handles output data.

By design, the GVirtuS front-end does not implement data locality, meaning that a CUDA function will be executed every time that it is invoked, without taking into account if the function with the same parameters has been already invoked. This means that no local cache is available. We motivate this choice by the simple fact of avoiding data inconsistency in this early stage development. As for the current status, with the improvement of our software stability and with the aim of achieving the best peak performance, we have already designed and partially implemented two caching mechanisms:

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

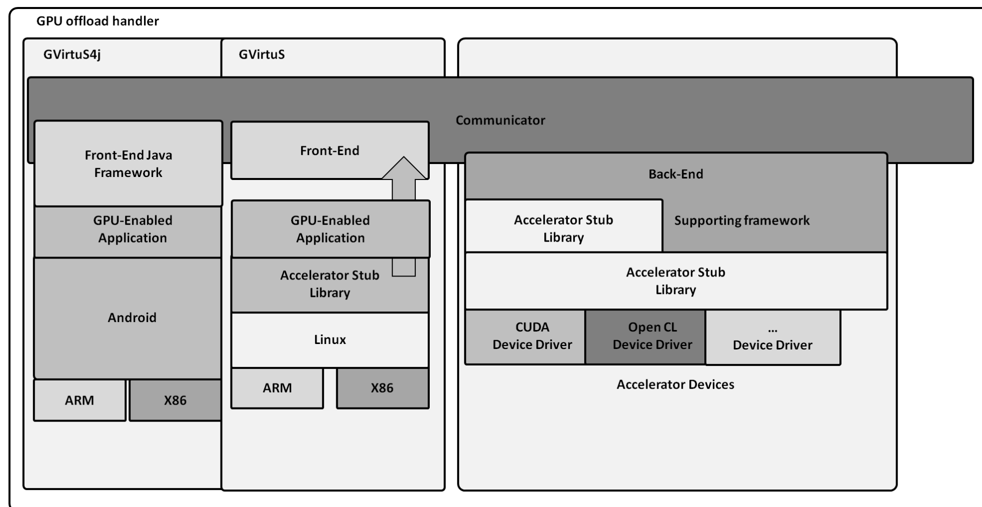

Figure 3. The GVirtuS-powered GPU offload handler architecture: GVirtuS (Linux) and GVirtuS4j (Java/Android) front-ends interact with the GVirtuS back-end leveraging on the communicator.

- **Local function caching.** Function signatures, parameters, and results are stored on the front-end side and managed by a front-end cache manager component in order to avoid remote useless calls.
- **Remote function caching.** Function signatures, parameters, and results are stored on the back-end side and managed by a back-end cache manager component in order to avoid useless GPGPU usage.

The first (local) is used mainly to reduce the need of a complete offload, but the cache size is limited by the device memory availability, while performance is limited by the local storage technology. The latter (remote) is used to reduce the GPGPU usage by making it available for calculations not performed yet thanks to a virtually unlimited cache size. The two function caching approaches are not mutually exclusive.

5. THE JAVA/ANDROID FRONT-END FRAMEWORK

In this section, we present the design of GVirtus4j, the Android GPU offloading framework, which is inspired by the architecture of GVirtuS described in Section 4.2. A CUDA program is composed of two different elements: *i)* CUDA application programming interface functions[15] such as memory management, and *ii)* the kernel functions[16], which are written in CUDA language. As such, we implement GVirtuS4J along two axes: *i)* Encapsulating the CUDA API functions and *ii)* handling the kernel functions.

In order to deal with the first problem, we build the front-end of GVirtuS4J as a Java/Android library that developers can include in their applications. We include the CUDA C++ methods with

[15]http://docs.nvidia.com/cuda/cuda-runtime-api/
[16]https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

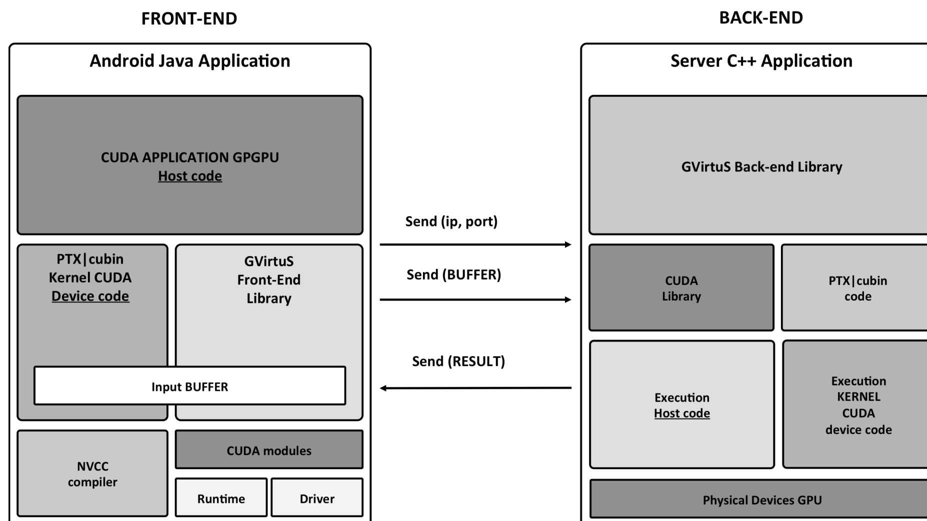

Figure 4. How GVirtuS4j face with the regular GVirtuS back-end: The GPGPU code is embedded in the Android application as a text PTX or binary *cubin*. The GVirtuS4j manages connection, the memory allocation, the context and the kernel invocation.

the same original signature whenever possible. Following an approach similar to JCUDA [33], Android developers can write Java code that directly calls CUDA kernels by using our API. As opposed to JCUDA, GVirtuS4J does not delegate the responsibility of generating the Java-CUDA bridge code and host-device data transfer calls to the compiler but it behaves as a local wrapper of the standard CUDA library. In the remote side, we use the regular GVirtuS back-end.

The communication between the Android GVirtuS4J front-end and the back-end is implemented using a customized TCP/IP communication protocol (see Figure 3). When a CUDA function is called in the Java/Android code, a socket connection with the back-end server is created, and a buffer for the data to be sent is allocated. Then, the CUDA function name together with the data (the parameters passed to the function) are serialized and sent to the back-end. The latter assigns them to the appropriate variables and invokes the corresponding CUDA function. The processed results are returned to the Java/Android front-end, which forwards them to the caller object. In order to add support for new API functions that may be included in future CUDA toolkit releases, we have implemented a series of support classes that simplify the communication and argument passing between C++ and Java. Frequently, argument passing is not strictly directed given the different nature of languages and architectures used.

In order to add support for CUDA kernel execution on Java/Android environments, and also to transparently execute them on a remote GPU server, it is necessary to send the kernel code to the back-end, as shown in Figure 4.

The kernel code has to be embedded in the application in PTX or *cubin* (CUDA binary):

- As it is well-known, the CUDA programming model stack relies on its proprietary pseudo-assembly intermediate language and split compilation architecture to ensure portability of GPU kernels across different NVIDIA GPU architectures. This intermediate language is called *Parallel Thread Execution* (PTX). Therefore, developers can write kernel code either using the low-level PTX or a high-level programming language such as C/C++. In both cases,

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

kernels must be compiled into binary code by the *NVIDIA CUDA Compiler* (NVCC) to effectively execute them on the device. NVCC compiles applications written in C/C++ and CUDA by splitting the code into two parts: the pure C/C++ part, which will be compiled by traditional C/C++ compilers such as GCC, and the CUDA part which is compiled by NVCC thus generating a PTX file with the GPU kernel code. The PTX source code file typically includes pseudo-assembly instructions, operands and other low-level operations responsible for encoding symbols and manage memory addressing. Unfortunately, current compilers are not able to generate binary code from CUDA kernels that are written in Java language.

- A CUDA binary (also referred to as *cubin*) file is an ELF-formatted file which consists of CUDA executable code sections as well as other sections containing symbols, relocators, debug info, etc. By default, the CUDA compiler driver NVCC embeds *cubin* files into the host executable file. In order to embed *cubin* files in a Java or Android application it must be generated separately by using the "-cubin" option of NVCC. Using GVirtuS4j, the *cubin* files sent on the remote host and the they are loaded at run time by the CUDA driver API. The input program is preprocessed for device compilation and is compiled to CUDA binary (*cubin*) which are placed in a fat binary.

GVirtuS4J solves this issue by mapping the entire PTX or *cubin* file into a buffer, encoding it as a string, and finally sending the string to the back-end to perform the execution of kernels on remote GPUs (see Figure 4).

However, the driver API needs an additional level of control by exposing lower-level concepts such as CUDA contexts and CUDA modules that are not implicit. Applications that use the driver API must compile code to separate files and explicitly load PTX code or *cubin* fat binaries. GVirtuS4J takes advantage of this procedure to mix host code written in Java (i.e. code that executes on the CPU), and then using the front-end library of GVirtuS-CUDA and device code (i.e. kernel code targeting the GPU) previously compiled using NVCC.

Under certain circumstances, GPGPU code offloading could have not been performed. A short and incomplete list of errors triggering this issue could be the following: *i)* unavailability of network connectivity (totally absent or too poor to offload the task), *ii)* the remote acceleration server could be overloaded or not correctly working, and *iii)* the application could embed one or more kernels that cannot be executed by the accelerator server. While GVirtuS4j offers a framework enabling CUDA kernel offloading, the local fallback management is under the developer's responsibility, as it is usually the case with GPGPU programming. If the code accelerated using GVirtuS4j cannot be executed for any reason, the developer has to provide a local alternative, making *de facto* the GPGPU code offloading optional:

- If there is no other way to perform the computation in an accelerated fashion, a local Java or native coded algorithm will be called, if provided by the developer.
- If the device has an on board GPGPU resource, as the NVIDIA Shield tablet for example, the computation could be performed locally, meaning that if the developer wants to support the fallback to the local GPU he/she has to provide that implementation as well[17]

[17]GVirtuS4j and the regular NVIDIA CUDA SDK for Android are similar, but they are not the same entity.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

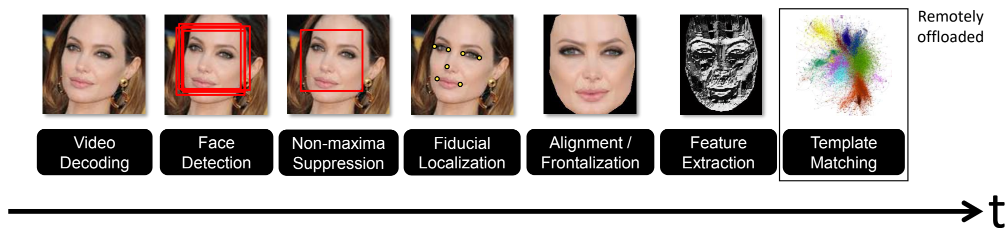

Figure 5. Face recognition pipeline used in *BioSurveillance*

If the GPGPU accelerated code is within a *remoteable* CPU code and the CPU code is executed remotely on the accelerator server, the issues about network connectivity and GPGPU code offloading are implicitly solved because the actual GPGPU device is proximal to the resource where the CPU code is executed.

6. PERFORMANCE EVALUATION

In this section, we show the obtained performance results for a real-wold face recognition application offloading CUDA kernels using GVirtuS on a Linux environment, and performance tests obtained by executing CUDA kernels remotely in Android.

6.1. Remote CUDA Kernel Execution in the BioSurveillance Application

BioSurveillance is a real-time highly-accurate face recognition software that extensively relies on GPU computing, and is specifically aimed for surveillance purposes in highly-crowded environments. It implements fine-grain parallelization using CUDA kernels for accelerating steps such as video decoding, face detection, template extraction and matching in databases populated with hundreds of thousands of suspects. Unfortunately, this application requires a single or several high-end NVIDIA discrete GPUs for concurrently analyzing faces on multiple 4K video feeds on a single machine. An approach for minimizing the total cost of ownership of the deployment of facial recognition systems is to split computations in a distributed manner. Typically, the most time-sensitive part of the face recognition pipeline are the video decoding and face detection substeps, as they are the input of the final face matching process. If video decoding and face detection do not meet the real-time deadline (i.e. 40 ms at 25 FPS), the system must discard video frames thus losing precious information while dramatically slowing down the end-user experience. Therefore, in order to avoid discarding frames, such steps should be implemented nearby the video capturing device or the surveillance camera. The approach we follow for adopting a client/server architecture in *BioSurveillance* is to deploy the NVIDIA Jetson TK/TX1 boards close to the cameras so they can perform the video decoding and execute the face detection/template extraction kernels on the CUDA-enabled on-die GPU of the Tegra SoC. Once facial templates have been extracted, the template matching kernel is then automatically offloaded to a remote high-performance discrete GPU by means of the GVirtuS framework (see Figure 5).

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

By following this architecture, a benchmarking experiment is conducted to study the feasibility of such approach, which substantially differs from other remote face recognition solutions such as the Amazon Rekognition API[18], Microsoft Cognitive Services API[19], or Google's Vision API[20]. These solutions require sending the whole image frames to their remote cloud before returning the results in JSON format. We believe our alternative is better suited for video workloads, as the input video feed resolution of surveillance cameras will keep improving to higher pixel densities (e.g. 4K or 8K resolutions). Moreover, we expect that future surveillance cameras to feature some kind of limited GPGPU capabilities, and to rely on low-power SoCs similar to the NVIDIA's Tegra platform for running image processing, and object detection kernels inside the camera.

In order to prove the feasibility of such distributed architecture, we benchmark the performance of the face template matching when performed on the NVIDIA Jetson TK1 board compared to offloading it using GVirtuS. The selected platform for issuing kernel launches is the NVIDIA Jetson TK1 board attached to a 1080p camera. The initial video decoding is performed by relying on the on-die H.264 video decoder by means of the *GStreamer* interface provided by NVIDIA. Decoded frames are then mapped to the face detector kernel, which determines face locations. These locations, are then used as an input of the facial template extraction kernel, which is also executed on the board. At this point, we distinguish the two different experiments by executing the template matching CUDA kernel *i)* on the local GPU of the Jetson TK1; and *ii)* on a remote GeForce Titan X thanks to GVirtuS.

Results of face template matching against a suspects database (remotely stored in the GeForce Titan X dedicated memory) are summarized below in Table II.

CUDA Kernel	Tegra K1 GPU (ms) [Local]	GeForce Titan X GPU (ms) [Remote - GVirtuS]
Template matching	290	5.2

Table II. Execution time of the face template matching CUDA kernel (local GPU vs. remote GPU with GVirtuS)

As the results depicted in Table II show, GVirtuS is a promising approach for transparently offloading and massively scaling template facial matching on low-power embedded GPGPU platforms. Even though the test was performed on a local area network (LAN) with a low average round-trip delay time of 1 ms, the $55\times$ obtained speed-up indicates great opportunities for deployments in networks with higher latencies. Additionally, it also opens the door for the execution of CUDA kernels matching against millions or records of suspects, which could be stored in a distributed manner over several powerful remote discrete GPUs equipped with high-bandwidth memories (HBM).

6.2. Executing CUDA Kernels in a Regular Android Application

In order to evaluate our prototype, we performed some tests which proved that *i)* GPU code offloading is feasible and *ii)* convenient under the right circumstances. Choosing a performance

[18]https://aws.amazon.com/rekognition/
[19]https://www.microsoft.com/cognitive-services/
[20]https://cloud.google.com/vision/

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

$A \times B$	CPU (PA64, s)	GPU* (PA64, s)	CPU (NEO, s)	GPU* (NEO, s)	CPU (S2T, s)	GPU* (S2T, s)
$A[192, 128] \times B[128, 128]$	0.3	4.9	3.2	5.9	0.01	1.1
$A[384, 256] \times B[256, 256]$	3.6	15.9	52.0	**11.2**	0.7	2.4
$A[768, 512] \times B[512, 512]$	60.7	**42.4**	505	**45.0**	15.6	**5.4**
$A[1152, 768] \times B[768, 768]$	223.6	**94.5**	1783	**94.7**	47.2	**11.9**

Table III. Results: Time in seconds, *GPU offload. PA64: PineA64+ developer board; NEO: UDOO Neo Internet of Things board; S2T: Samsung Galaxy S2 Tab (T-819), top-range Android mobile tablet. In **bold** when the offload GPU execution beats the local CPU performance.

testing suite accepted by the community was not possible, due to the lack of such suite, given that the GPU code offloading for Android devices is a novel approach. We chose one of the NVIDIA's CUDA SDK 8.0 samples, which are included with the standard CUDA Toolkit distribution[21]. Precisely, in this paper, we benchmarked *Matrix Multiplication*. The choice was motivated by its clarity of exposition on illustrating various CUDA programming principles, which makes it easy to clearly present the needed modifications for making it work with GVirtuS4J. Moreover, performing linear algebra operations is a common task assigned to GPGPUs [49], and it is becoming increasingly important in the context of CNNs, as they extensively rely on BLAS SGEMM operations for both training such networks and inferencing them. Preliminary tests have been conducted with a varying problem size as shown in Table III, where the size of matrix A is given by 4**block_size*, 6**block_size*, and the size of matrix B is given by 4**block_size*, 4*block_size (*block_size=32*). The Accelerator Server (Dual Intel Xeon 6-core E5-2609v3 1.9Ghz - 8GB DDR4) we used for this experiment is equipped with two NVIDIA Titan X CUDA-enabled devices. We also executed performance tests using a PineA64+ single board computer (ARM Cortex A53 64-bit Quad core), a UDOO Neo Internet of Things board (ARM Cortex-A9 32-bit Single Core, 1.0GHz, 1GB RAM) and a tablet Samsung S2 mod. T-819 (8 cores, 1.8GHz, 3GB RAM). The connection between the front-end and the back-end was made through a traditional IEEE 802.11 Wi-Fi infrastructure. The results include the network overhead.

This experiment demonstrated that remote GPU offloading is a promising approach to be considered as the problem size of a given workload is increased. The break point of offloaded GPU beating the local CPU is related to the device (single board computer and tablet behaves differently), the algorithm used, and the network condition.

These results demonstrate how the proposed framework enables CUDA kernel execution on devices that are not provided with GPUs with CUDA support. However, a more extensive performance test and evaluation suite is needed in order to define the range of problem size thus reflecting the feasibility of the proposed approach. The comparison with local CUDA execution on devices with CUDA GPUs, such as the NVIDIA Shield K1 Tablet, would be quite useful to show same results as we previously did in Section 6.1 with the NVIDIA Jetson TK1. The NVIDIA CUDA official support to Android is very limited to a short list of devices[22]; the NVIDIA CUDA toolchain differs from the one common in regular Android development and, last but not the least, the Android based products supporting NVIDIA CUDA represent a niche market not straightforwardly available

[21] http://docs.nvidia.com/cuda/cuda-samples/
[22] https://developer.nvidia.com/codeworks-android

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

off the shelf. However, we plan on extending our performance test application enabling the local CUDA support and designing a three-step fallback system: remote GPU, local GPU, local CPU.

6.3. Overhead of the security layer

We perform an extensive number of experiments to characterize and measure the costs introduced by the cryptographic communication protocol for data exchange between the AC and AS. The data transmission/receiving costs depend on the size of data to send/receive and on the network quality. For this reason, we measure the latency when sending and receiving objects of different size from an Android client to VMs running on two clouds with different capabilities and network quality. In particular, we present here only the costs related to data transmission; the results of data reception are similar and consistent with the results obtained with the transmission experiments. We use a Motorola Moto G phone with Quad-core 1.2GHz Cortex-A7 CPU, 1GB of memory, running Android 5.1 as the client device. We use two clouds for running the VMs: one private and one public, precisely the Amazon EC2 cloud[23]. The private cloud is deployed on a Dell machine with 2xIntel Xeon E5 2600 v3 2.40GHz processors (24 cores), 32GB (up to 1536GB) RAM, running Ubuntu Linux 14.04.3 LTS x86_64. The cloud platform is OpenStack[24].

The purpose of using two clouds is to perform the experiments using network connections with different conditions between the AC and the AS. As the *ping* measurements show, presented in Table IV, the connection towards the private cloud presents higher latency than the Amazon EC2 cloud. The Amazon Machine Image (AMI) we utilize for running the VM is a *m1.medium* instance, with 1 vCPU, 3.75GB RAM, 410GB of disk memory, not EBS optimized, and with moderate network performance[25]. We launch the instance in Amazon's *eu-west-1* region, more precisely in Ireland, the physical location of the Private cloud is Greece, and the physical location of the mobile device is Denmark. We configure the VM on the private cloud with 1 vCPU, 1GB RAM, 20GB of disk memory, and normal network priority. The VM in the private cloud and on Amazon EC2 runs the Android-x86 OS[26] 4.4 and 4.0, respectively. In particular, on Amazon EC2 we run our customized Android-x86 AMI, publicly available with ID *ami-ce5a4dba* [50].

We send/receive random data of size 4B, 100KB, and 1MB from the AC to the AS and vice-versa for 200, 100, and 20 times in clear and with SSL/TLS for each data size, respectively. We select these data sizes to represent applications needing to send small, average, and large data. Then, we plot the measurements as boxplots, showing the minimum, the 25th percentile, the median, the 75th percentile, and the maximum. We recreate the connection between the AC and the AS before sending the data, to avoid the caching optimization implemented in Java's *ObjectOutputStream*. The results of the experiments performed with the Private cloud and with the Amazon EC2 cloud are presented in Figure 6 and Figure 7, respectively. The results show that in all cases the encryption layer does not introduce any noticeable overhead compared to the clear communication.

We perform the same experiment also using the Linux version of the framework. The client device (AC) in this case is a Lenovo ThinkPad T460p running Ubuntu 16.04, Core i7-6820HQ (4C, 2.7 / 3.6GHz, 8MB) CPU, 4GB RAM, WiFi, and physically located in Denmark. The VM is

[23]https://aws.amazon.com/ec2/
[24]https://www.openstack.org/
[25]https://aws.amazon.com/ec2/previous-generation/
[26]http://www.android-x86.org/

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

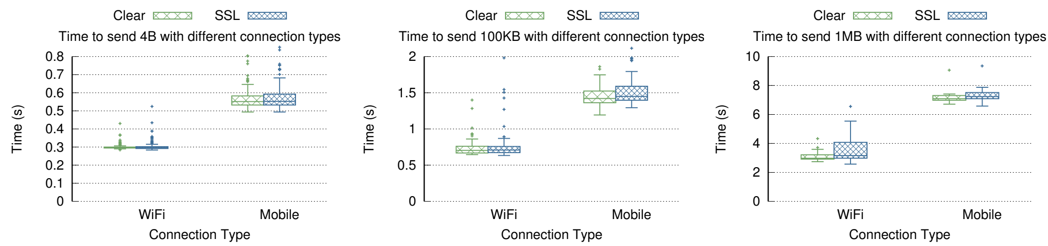

Figure 6. Latency of transmitting 4B, 100KB, and 1MB from the Motorola Moto G (AC) to the VM (AS) in clear and with SSL using WiFi and Mobile network. The VM runs on a private cloud with low data rate and high latency.

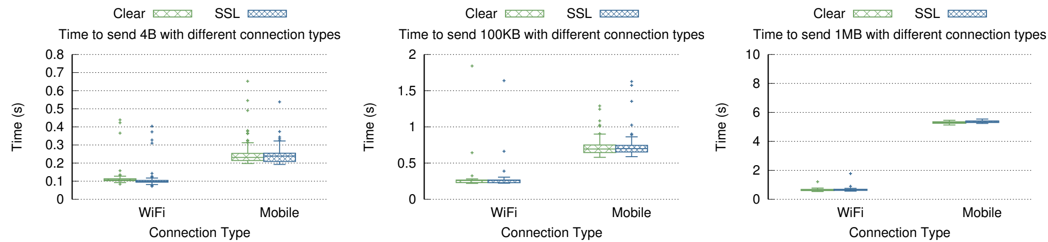

Figure 7. Latency of transmitting 4B, 100KB, and 1MB from the Motorola Moto G (AC) to the VM (AS) in clear and with SSL using WiFi and Mobile network. The VM runs on the Amazon EC2 cloud with moderate data rate and moderate latency.

Table IV. Network latency between the AC and AS when the AS runs on a Private cloud and on the Amazon EC2 cloud.

	Private Cloud		Amazon EC2	
	WiFi (ms)	Mobile (ms)	WiFi (ms)	Mobile (ms)
Motorola Moto G	87.526	162.539	57.651	132.380

configured with 1 vCPU, 1GB RAM, and normal network priority on the private cloud. On Amazon EC2, we use the *m3.medium* instance, which provides 1 vCPU, 3GB RAM, and moderate network priority. The VM runs Ubuntu 16.04 on both clouds. The network latency towards the private cloud and Amazon EC2, as measured by *ping*, is 78.846ms 37.718ms, respectively. The results of these experiments are consistent with the ones obtained with Android, showing that also in this case the encryption layer does not introduce any noticeable overhead compared to the clear communication. The plotted results are omitted for reasons of space.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the design and implementation of RAPID, the first, to the best of our knowledge, complete offloading suite for low-powered devices. RAPID supports CPU task offloading for Android and Java Linux applications and supports GPGPU CUDA-based offloading for Android and C/C++/Java Linux applications. Moreover, RAPID implements a lightweight security layer to protect the data transmitted during CPU task offloading from malicious users. We first presented a high-level perspective of the whole framework, describing its main components

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

with a focus on the GPGPU-related subsystem that deals with the GPGPU offloading process. We showed through preliminary experiments that GPGPU CUDA offloading on Android devices is not only possible but also convenient when the problem becomes large enough. Moreover, we showed the first impressive results of embedding the framework into a real world face recognition application, namely *BioSurveillance*. From these experiments we saw that transparent remote GPGPU offloading substantially reduced the execution time of workloads demanding highly compute-intensive operations. Finally, we showed that the security layer does not introduce any significant overhead in terms of data transmission delay.

As it is the case with computation offloading, RAPID is not a solution for all possible applications. Applications with heavy computation requirements, small-sized input/output problems, and non-strictly real-time could be the best candidates. However, there are applications that are characterised by a nature that makes the remoting approach not feasible, such as large input/output size, for example. The evaluation Android application we used for the results presented in this paper fosters a pretty common scenario in Android GPGPU offloading, where the network overhead is non-negligible. Considering just the GPGPU code offloading, Linux applications, such as the BioSurveillance and the WaComM model, best fit another scenario pictured out by the need of GPGPU computation and the availability of proximal accelerator servers (remote, but close and stable from the network point of view).

Currently, we are gearing the development of the proposed platform towards three future directions: *i)* As a first goal, we will improve the efficiency and architectural design of the GPGPU offload handler by relying on more developed and robust suites thanks to the availability of up-to-date hardware infrastructures, tools, and compiler technology. New communicators leveraging on parallel streams and InfiniBand are in development and early-testing stage. Regarding the local fallback in case of offloading failure, which is already supported for CPU computational tasks, we are working on implementing the same feature for GPGPU code as well. This delay is due to the current state of the available technology, which does not allow for fully automatic GPGPU code local fallback management. For this reason, we still rely on the developer's support for providing the source code for local execution along with that for remote acceleration. Moreover, we have started the integration of a Public Key Infrastructure for automatic key distribution into the project. *ii)* As medium-term goal, we will release the RAPID framework as open source software under Apache 2.0 license. This way, we will offer the availability of our cloud shared CPU and GPGPUs computing resources to a selected group of early adopters for testing their applications and for improving the overall quality of code; *iii)* As a long-term goal, we will leverage on a solid and stable CPU and GPGPU offloading infrastructure to implement real-world applications in the field of distributed computing and high-performance IoT [51],[52], paying special attention to the techniques developed to enhance the selection and transparent use of resources [53],[54].

At the time of writing, the GVirtuS' back-end is implemented as a Linux daemon, the GVirtuS' front-end is implemented as a Linux shared library, and the GVirtuS4j's front-end is implemented as a Java and Android library. Because of design and performance reasons, the implementation of the back-end as a Linux daemon is mandatory in our solution. However, there is nothing preventing the implementation of the front-end as a Windows Dynamic-link library (DLL), thus supporting also low-powered devices running Windows. In a similar way, GVirtuS4j could be implemented as an iOS framework, enabling remoted GPGPU computation also on Apple devices. As already

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

stated in the paper, we focused this ongoing research work on CUDA, but thanks to the GVirtuS architecture, our plans involve the development of a GVirtuS4j framework wrapping OpenCL. The GVirtuS' OpenCL plug-in is already available. The use of OpenCL could be a trailblazer to different acceleration architectures than just GPGPUs.

The RAPID acceleration framework and, in particular, the CPU[27], the GPU Linux[28] and Android[29] offloading middleware are available as open source under the business-friendly Apache 2.0 license.

ACKNOWLEDGEMENTS

This research has been funded by the European Commission under the Horizon 2020 program through Grant Agreement No 644312, corresponding to the "Heterogeneous Secure Multi-level Remote Acceleration Service for Low-Power Integrated Systems and Devices" (RAPID) project.

REFERENCES

1. 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*, pages 945–953, March 2012.
2. Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, Hollywood, CA, 2012.
3. Y. Li and W. Gao. Minimizing context migration in mobile code offload. *IEEE Transactions on Mobile Computing*, PP(99):1–1, 2016.
4. Yongin Kwon, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Yeongpil Cho, and Yunheung Paek. Precise execution offloading for applications with dynamic behavior in mobile cloud computing. *Pervasive and Mobile Computing*, 27:58 – 74, 2016.
5. A. Reiter and T. Zefferer. Paving the way for security in cloud-based mobile augmentation systems. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 89–98, March 2015.
6. Aaron Yi Ding, Bo Han, Yu Xiao, Pan Hui, Aravind Srinivasan, Markku Kojo, and Sasu Tarkoma. Enabling energy-aware collaborative mobile data offloading for smartphones. In *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*, pages 487–495. IEEE, 2013.
7. F. A. Silva, M. Rodrigues, P. Maciel, S. Kosta, and A. Mei. Planning mobile cloud infrastructures using stochastic petri nets and graphic processing units. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 471–474, Nov 2015.
8. Jose Duato, Antonio J Pena, Federico Silla, Juan C Fernandez, Rafael Mayo, and Enrique S Quintana-Orti. Enabling cuda acceleration within virtual machines using rcuda. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10. IEEE, 2011.
9. K. Choi, J. Lee, Y. Kim, S. Kang, and H. Han. Feasibility of the computation task offloading to gpgpu-enabled devices in mobile cloud. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 244–251, Sept 2015.
10. Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.
11. Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[27]https://github.com/RapidProjectH2020/offloading-framework-android
[28]https://github.com/RapidProjectH2020/GVirtuS
[29]https://github.com/RapidProjectH2020/gvirtus-android

Copyright © 0000 John Wiley & Sons, Ltd.

12. Richard Membarth, Oliver Reiche, Frank Hannig, and Jürgen Teich. Code generation for embedded heterogeneous architectures on android. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 86. European Design and Automation Association, 2014.
13. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.
14. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.
15. R. Montella, G. Coviello, G. Giunta, G. Laccetti, F. Isaila, and J.G. Blas. A general-purpose virtualization service for hpc on cloud computing: An application to gpus. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7203 LNCS(PART 1):740–749, 2012.
16. Seiichi Gohshi. Real-time super resolution algorithm for security cameras. In *e-Business and Telecommunications (ICETE), 2015 12th International Joint Conference on*, volume 5, pages 92–97. IEEE, 2015.
17. *International Technology Roadmap for Semiconductors (ITRS) 2.0*. Semiconductor Industry Association, 2015.
18. I Ascione, Giulio Giunta, P Mariani, Raffaele Montella, and Angelo Riccio. A grid computing based virtual laboratory for environmental simulations. *Euro-Par 2006 Parallel Processing*, pages 1085–1094, 2006.
19. R. Montella, D. Kelly, W. Xiong, A. Brizius, J. Elliott, R. Madduri, K. Maheshwari, C. Porter, P. Vilter, M. Wilde, et al. Face-it: A science gateway for food security research. *Concurrency and Computation: Practice and Experience*, 27(16):4423–4436, 2015.
20. Q. Pham, T. Malik, I. Foster, R. Di Lauro, and R. Montella. SOLE: Linking Research Papers with Science Objects. In *IPAW*, pages 203–208. Springer, 2012.
21. R. Montella, G. Giunta, and A. Riccio. Using grid computing based components in on demand environmental data delivery. In *Proceedings of the second workshop on Use of P2P, GRID and agents for the development of content networks*, pages 81–86. ACM, 2007.
22. R. Di Lauro, F. Giannone, L. Ambrosio, and R. Montella. Virtualizing general purpose gpus for high performance cloud computing: an application to a fluid simulator. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 863–864. IEEE, 2012.
23. Salvatore Cuomo, Ardelio Galletti, Giulio Giunta, and Livia Marcellino. Toward a multi-level parallel framework on gpu cluster with petsc-cuda for pde-based optical flow computation. *Procedia Computer Science*, 51:170–179, 2015.
24. Jieun Choi, Theodora Adufu, and Yoonhee Kim. Data-locality aware scientific workflow scheduling methods in hpc cloud environments. *International Journal of Parallel Programming*, pages 1–14, 2017.
25. V. Boccia, L. Carracciuolo, G. Laccetti, M. Lapegna, and V. Mele. Hadab: Enabling fault tolerance in parallel applications running in distributed environments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7203 LNCS(PART 1):700–709, 2012.
26. Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *International Conference on Cloud Computing*, pages 115–131. Springer, 2009.
27. Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.
28. Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. *Cuckoo: A Computation Offloading Framework for Smartphones*, pages 59–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
29. Ioana Giurgiu, Oriana Riva, and Gustavo Alonso. Dynamic software deployment from clouds to mobile devices. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 394–414, New York, NY, USA, 2012. Springer-Verlag New York, Inc.
30. F. Lordan and R. M. Badia. Compss-mobile: Parallel programming for mobile-cloud computing. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 497–500, May 2016.
31. A. Reiter and T. Zefferer. Power: A cloud-based mobile augmentation approach for web- and cross-platform applications. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 226–231, Oct 2015.
32. Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reano. Remote gpu virtualization: Is it useful? In *High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2016 2nd IEEE International Workshop on*, pages 41–48. IEEE, 2016.
33. Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par 2009 Parallel Processing*, pages 887–899. Springer, 2009.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

34. B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.
35. Carlos Reaño and Federico Silla. A performance comparison of cuda remote gpu virtualization frameworks. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 488–489. IEEE, 2015.
36. Carlos Reaño and Federico Silla. Reducing the performance gap of remote gpu virtualization with infiniband connect-ib. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pages 920–925. IEEE, 2016.
37. R. Montella, C. Ferraro, S. Kosta, V. Pelliccia, and G. Giunta. *Enabling Android-Based Devices to High-End GPGPUs*, pages 118–125. Springer International Publishing, Cham, 2016.
38. A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.
39. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.
40. K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Informational), November 2016.
41. K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, and M. Scott. PKCS #12: Personal Information Exchange Syntax v1.1. RFC 7292 (Informational), July 2014.
42. François Armand, Michel Gien, Gilles Maigné, and Gregory Mardinian. Shared device driver model for virtualized mobile handsets. In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 12–16. ACM, 2008.
43. George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130. ACM, 2008.
44. Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742. IEEE, 2011.
45. R. Montella, G. Giunta, G. Laccetti, M. Lapegna, C. Palmieri, C. Ferraro, and V. Pelliccia. Virtualizing cuda enabled gpgpus on arm clusters. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9574 LNCS:3–14, 2016.
46. Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, Valentina Pelliccia, Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. On the virtualization of cuda based gpu remoting on arm and x86 machines in the gvirtus framework. *International Journal of Parallel Programming*, pages 1–22, 2017.
47. Steve Crago, Kyle Dunn, Patrick Eads, Lorin Hochstein, Dong-In Kang, Mikyung Kang, Devendra Modium, Karandeep Singh, Jinwoo Suh, and John Paul Walters. Heterogeneous cloud computing. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 378–385. IEEE, 2011.
48. G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing*, volume 3821 of *Lecture Notes in Computer Science*, pages 379–391. Springer, 2010.
49. Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.
50. Sokol Kosta, C Perta, Julinda Stefa, Pan Hui, and Alessandro Mei. Clone2clone (c2c): Enable peer-to-peer networking of smartphones on the cloud. *T-Labs, Deutsche Telekom, Tech. Rep. TR-SK032012AM*, 2012.
51. Burak Kantarci and Hussein T Mouftah. Trustworthy sensing for public safety in cloud-centric internet of things. *Internet of Things Journal, IEEE*, 1(4):360–368, 2014.
52. Chiyoung Lee, Se-Won Kim, and Chuck Yoo. Vadi: Gpu virtualization for an automotive platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290, 2016.
53. M.R. Guarracino, G. Laccetti, and A. Murli. Application oriented brokering in medical imaging: Algorithms and software architecture. In *European Grid Conference, 2005*, volume 3470 LNCS, pages 972–981, 2005.
54. A. Murli, L. D'Amore, G. Laccetti, F. Gregoretti, and G. Oliva. A multi-grained distributed implementation of the parallel block conjugate gradient algorithm. *Concurrency and Computation: Practice and Experience*, 22(15):2053–2072, 2010.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe