UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de una herramienta de testeo automático para el middleware rCUDA

TRABAJO FIN DE MÁSTER

Máster Universitario en Ingeniería Informática

Autor: Antonio Díaz Román

Tutores: Federico Silla Jiménez
Carlos Reaño González

Curso 2017-2018

Resum

La verificació és una etapa crítica en qualsevol projecte de desenvolupament de programari. El principal objectiu de la verificació és que el programari que està sent desenvolupat es comporte d'acord amb la seua especificació de requisits funcionals i no funcionals. En este Treball Fi de Màster es dissenyarà, implementarà i posarà en producció una plataforma de verificació automàtica per a l'entorn de virtualización remota de GPUs rCUDA emprant el framework de verificació Boost.Test. Esta plataforma de verificació executarà unitats de test en què es llançaràn massivament aplicacions d'índoles molt variades. D'esta manera, la plataforma de verificació proporcionarà als desenvolupadors de rCUDA tota la informació que necessiten per a localitzar els possibles errors i corregir-los en el menor temps possible.

Paraules clau: rCUDA, testing, verificació, Boost, Boost.Test, GPU, virtualització

Resumen

La verificación es una etapa crítica en cualquier proyecto de desarrollo de software. El principal objetivo de la verificación es que el software que está siendo desarrollado se comporte conforme a su especificación de requisitos funcionales y no funcionales. En este Trabajo Fin de Máster se diseñará, implementará y pondrá en producción una plataforma de verificación automática para el entorno de virtualización remota de GPUs rCUDA empleando el framework de verificación Boost.Test. Esta plataforma de verificación ejecutará unidades de test en los que se lanzarán masivamente aplicaciones de índoles muy variadas. De esta forma, la plataforma de verificación proporcionará a los desarrolladores de rCUDA toda la información que necesitan para localizar los posibles errores y corregirlos en el menor tiempo posible.

Palabras clave: rCUDA, testing, verificación, Boost, Boost.Test, GPU, virtualización

Abstract

Verification is a critical stage in any software development project. The main aim of verification is that software being developed behaves accordingly to its functional and non-functional requirement specification. In this Master's Thesis, an automatic verification platform for the virtualization environment of remote GPUs rCUDA will be designed, implemented and put into production by using the Boost.Test testing framework. This verification platform will run test units where varied applications will be massively launched. In this way, the verification platform will provide rCUDA developers with all the information they need to find the possible errors and correct them in the shortest possible time.

Key words: rCUDA, testing, verification, Boost, Boost.Test, GPU, virtualization

Índice general

CAPÍTULO 1
Introducción

El plan de estudios del Grado en Ingeniería Informática de la Escuela Técnica Superior de Ingeniería Informática (ETSINF) de la Universitat Politècnica de València (UPV) está diseñado para proveer a sus egresados de una diversidad de conocimientos de manera que puedan desarrollar su posterior carrera profesional en diferentes temáticas relacionadas con la informática. Más todavía, si el egresado en cuestión combina los estudios de Grado en Ingeniería Informática con los del Máster Universitario en Ingeniería Informática (MUIINF), el espectro de conocimientos adquiridos se amplía hasta tal punto que resulta poco probable que el antiguo alumno acabe desarrollando su carrera profesional en un contexto en el que no haya recibido alguna formación. No obstante, a pesar de los seis años de duración que presenta este plan de estudios integrado, y como consecuencia del vasto campo que es la informática, la mayoría de sus egresados probablemente deba completar su formación con conocimientos concretos del área en el que desarrollen su profesión. De la misma manera, es probable que, a lo largo de la vida profesional del ingeniero, éste cambie de área con el tiempo. Y en caso de que no lo hiciera, también es probable que tuviera que actualizar constantemente sus conocimientos para incorporar los últimos avances y tecnologías del ámbito en cuestión.

Las áreas en las que un egresado del Máster Universitario en Ingeniería Informática podría desarrollar su vida profesional son muy variadas. Citando solo algunas, podría trabajar de analista programador, administrador de sistemas, arquitecto de software/hardware, programador de sistemas empotrados, consultor TIC[1], administrador y/o programador de sistemas ERP[2], etc. Y además de estas disciplinas, existe un área en la Ingeniería Informática en la que la Universitat Politècnica de València destaca especialmente por la reconocida calidad de su formación: la Ingeniería del Software.

La Ingeniería del Software es una disciplina que se cubre holgadamente durante el plan de estudios integrado que conforma el Grado en Ingeniería Informática combinado con el Máster Universitario en Ingeniería Informática. Son claros ejemplos las asignaturas de Ingeniería del Software, Gestión de Proyectos, Planificación y Dirección de Proyectos y, finalmente, Auditoría, Calidad y Gestión de los sistemas de Información, entre otras. Además, la ETSINF dispone de una rama específica para los estudiantes de Grado que está dedicada exclusivamente a la Ingeniería del Software. No en vano uno de los principales baluartes de la ETSINF es la Ingeniería del Software, puesto que ésta es un pilar fundamental en la Ingeniería Informática de nuestro tiempo.

Así pues, una posible salida profesional de un egresado del Máster Universitario en Ingeniería Informática es la participación en proyectos de Ingeniería del Software. Estos proyectos pueden abarcar multitud de disciplinas y ámbitos, como por ejemplo desarro-

[1] Tecnologías de la Información y Comunicación.
[2] Enterprise Resource Planning, Sistema de planificación de recursos empresariales.

llo de aplicaciones (tanto para escritorio como para dispositivos móviles), desarrollo de aplicaciones científicas, de sistemas operativos, de sistemas empotrados, etc. Siguiendo con estos ejemplos de Ingeniería del Software en los que un egresado del Máster Universitario en Ingeniería Informática podría ejercer su profesión se encuentra el software de computación de altas prestaciones (High Performance Computing, HPC [1]). En esta categoría podrían agruparse proyectos de software tales como librerías de comunicaciones (como es el caso de librerías MPI [2]), librerías de aceleración de aplicaciones (como podrían ser CUDA [3], OpenCL [4] u OpenACC [5]) o plataformas de virtualización de aceleradores en grandes centros de procesos de datos, como es el caso que ocupa a este trabajo.

Todos los proyectos software pasan por diversas fases. Esto es lo que se conoce como ciclo de vida del software. De hecho, uno de los primeros conceptos que los estudiantes de la Ingeniería del Software aprenden es el ciclo de vida del software. El ciclo de vida del software es una estructura bien definida que se aplica al desarrollo de un producto software, que consta de unas fases determinadas por las que debe pasar todo desarrollo de software. Esto implica que el proceso de Ingeniería del Software comienza mucho antes de escribir las primeras líneas de código, y continúa mucho después de que la primera versión del producto haya sido completada y puesta en producción.

Existen distintos modelos de desarrollo de software cuyos ciclos de vida difieren unos de otros. Estos modelos son una representación abstracta que presentan un enfoque común para el desarrollo del software. Por citar muy brevemente algunos:

- **Modelo de desarrollo clásico o en cascada**: modelo de desarrollo en el que cada una de las fases se comienza únicamente cuando ha finalizado completamente la fase anterior. Al finalizar la última fase se obtiene la versión final del producto.
- **Modelo clásico con prototipado**: modelo muy similar al clásico, con la diferencia de que, en lugar de obtener el producto final al finalizar el ciclo, se obtiene una versión sin desarrollar completamente o incorporando algunas características del sistema final, en lugar de la totalidad de las mismas.
- **Modelo de espiral**: modelo cuyas fases se disponen en forma de espiral de tal forma que cada bucle o iteración respresenta un conjunto de actividades que son seleccionadas en función del análisis de riesgo, realizado en todas las iteraciones.
- **Modelo iterativo e incremental**: modelo de desarrollo formado por tareas agrupadas en iteraciones donde cada iteración produce un incremento del producto. De este modo, el desarrollador puede sacar ventaja de lo aprendido en la iteración anterior y los cambios en los requisitos pueden ser corregidos más fácilmente que en los modelos anteriores.
- **Modelos de desarrollo ágil**: modelo iterativo que aboga por un punto de vista más ligero y más centrado en la funcionalidad que en la documentación y planificación. Los procesos ágiles se retroalimentan de las iteraciones anteriores por medio de pruebas periódicas y versiones frecuentes del producto.

Estos modelos de desarrollo de software son ciertamente distintos, pero tienen una característica común: todos ellos dedican al menos una de las fases a verificar el software que está en desarrollo. Incluso en algunos modelos, como son los casos de desarrollos incrementales y ágiles, la fase de verificación se encuentra presente en todas las fases restantes. Es más, el proceso de verificación es de tal importancia que incluso existe un tipo de desarrollo de software llamado desarrollo guiado por pruebas (Test-Driven Development, TDD) donde los requisitos son traducidos a pruebas en lugar de ser traducidos a

funcionalidades, para posteriormente implementar el software que hace que las pruebas se superen (es decir, se escriben las pruebas de verificación y se fuerza a que fallen, para después escribir el código que hace que las pruebas se satisfagan).

La fase de verificación debe comprobar que el software realiza la función que consta en su especificación de requisitos de forma correcta y que, además, su comportamiento es robusto. El proceso de verificación de un proyecto software determinado puede tener poco en común con otros proyectos de software, pero siempre hay una constante: el proceso de verificación es difícil de diseñar e implementar y ocupa un gran porcentaje del esfuerzo y del coste de todos los proyectos software (hasta el 40 % del coste total en algunos de ellos). A pesar de ello, la totalidad de los proyectos de software industriales pasan por un concienzudo proceso de verificación con objeto de que su funcionamiento esté lo más cerca posible de su especificación de requisitos.

Teniendo en cuenta todo lo comentado anteriormente, en este Trabajo Fin de Máster (TFM) se pretende desarrollar una plataforma de verificación específicamente diseñada para el middleware de virtualización remota de GPUs rCUDA (*Remote CUDA*) [6] [7]. Además, este TFM se enmarca dentro del contrato laboral que el alumno que escribe este trabajo tiene en el Departamento de Informática de Sistemas y Computadores (DISCA) [11], concretamente en el Grupo de Arquitecturas Paralelas (GAP) [12]. Desde sus inicios, el objetivo principal por el que este alumno fue contratado fue el desarrollo de esta plataforma de verificación, además de para otras tareas relacionadas con el proyecto rCUDA.

Esta plataforma de verificación tendrá como principal objetivo automatizar las tareas mediante las cuales el equipo de desarrollo de rCUDA comprueba y verifica que rCUDA se comporta conforme a su especificación. Sin una plataforma de verificación automática, los desarrolladores deben ejecutar una por una las aplicaciones a las que rCUDA da soporte, comprobando manualmente si la ejecución ha sido correcta y obteniendo tiempos, también manualmente, para comprobar que el rendimiento de rCUDA es el adecuado frente a los tiempos de referencia, esto es, los tiempos que proporciona CUDA con la misma aplicación.

Sin embargo, con una plataforma de verificación como la diseñada en este Trabajo Fin de Máster, los desarrolladores de rCUDA pueden, con un simple programa en línea de comandos que cuenta con una interfaz de usuario sencilla y homogénea para todas las aplicaciones a las que rCUDA da soporte, ejecutar de forma masiva y automática todas las aplicaciones que requieran en un momento dado. A todo esto, la plataforma de verificación proporciona al usuario, para cada aplicación ejecutada, el tiempo de ejecución y el resultado de la ejecución, tanto por la salida estándar de la aplicación como a través de unos ficheros de registro cuyos detalles se comentarán en capítulos posteriores.

La realización de este Trabajo Fin de Máster resulta enormemente beneficiosa para el alumno que escribe estas líneas por diversos motivos. En primer lugar, lo introduce de forma práctica en el uso de aceleradores de cómputo basados en GPUs. Estos aceleradores están presentes actualmente en multitud de centros de datos en producción. Por lo tanto, adquirir experiencia en el uso de estos aceleradores proveerá al alumno de una mayor empleabilidad. En segundo lugar, la realización de este trabajo introduce al alumno en la verificación de aplicaciones que se ejecutan en clústeres de computadores. Estos clústeres son el motor de cómputo en los centros de datos de infinidad de empresas tanto a nivel nacional como internacional. Por lo tanto, adquirir conocimientos relacionados con la programación y verificación de software en estos entornos industriales, al mismo tiempo que se aumenta la experiencia del alumno en el uso de estos clústeres de altas prestaciones, redundará en una mayor empleabilidad una vez acabe los estudios universitarios, dado que su formación será más atractiva para el mundo empresarial.

Por último, la realización del presente trabajo conlleva adquirir conocimientos a nivel de usuario y de administrador de centros de datos sobre diversas aplicaciones comúnmente utilizadas en dichos centros de datos, además de mejorar las habilidades del alumno en la programación de aplicaciones distribuidas y en la verificación del software.

Así pues, este Trabajo Fin de Máster tiene un carácter completamente orientado a la industria, pues su objetivo no es otro que proporcionar estabilidad y robustez al middleware de virtualización remota de GPUs rCUDA, plataforma que ha sido distribuida a cientos de centros de datos de multitud de compañías y organismos a lo largo de todo el mundo.

Finalmente, esta memoria se estructura como se presenta a continuación. En primer lugar, y tras la presentación de este trabajo mediante esta introducción, en el capítulo 2 se abordará el concepto de la técnica de virtualización remota de GPUs y se detallará el funcionamiento del middleware de virtualización remota de GPUs rCUDA. A continuación, en el capítulo 3 se introducirá brevemente el concepto de pruebas de software donde se comentarán algunas herramientas de verificación y donde, posteriormente, se detallará la instalación y forma de uso de la librería de verificación empleada en este Trabajo Fin de Máster. Seguidamente, en el capítulo 4 se detallará el proceso de desarrollo que ha seguido la plataforma de verificación desarrollada en este trabajo, desde la fase de análisis de requisitos, pasando por la fase de diseño y finalizando por la fase de implementación. Posteriormente, en el capítulo 5 se detallará el proceso de puesta en producción del programa diseñado, poniendo especial énfasis en la retroalimentación recibida por los usuarios de dicha plataforma. Para acabar, en el capítulo 6 se expondrán las conclusiones a las que se han llegado tras la realización de este trabajo.

CAPÍTULO 2
Qué es rCUDA

El objetivo de este Trabajo Fin de Máster es diseñar e implementar un sistema de verificación para el middleware de virtualización remota de GPUs rCUDA, el cual ha sido creado en la Universitat Politècnica de València. Por lo tanto, para entender el contexto en el que se desarrolla este trabajo, se hace imprescindible revisar el middleware rCUDA. Este capítulo proporciona una descripción detallada de esta tecnología de virtualización.

2.1 Introducción

Actualmente, las capacidades de las GPUs (Graphics Processing Units, Unidades de Procesamiento Gráfico) para ejecutar operaciones matemáticas de forma masivamente paralela están siendo utilizadas con el objetivo de acelerar partes específicas de diversas aplicaciones. A este respecto, los programadores explotan los recursos de las GPUs asignando las partes computacionalmente intensas a las mismas. Aunque los programadores deben especificar qué partes de la aplicación se ejecutan en CPU y qué partes en GPU, la existencia de librerías y modelos de programación tales como CUDA (Compute Unified Device Architecture) [3] facilitan enormemente la tarea. En este contexto, las GPUs reducen significativamente el tiempo de ejecución de aplicaciones pertenecientes a dominios tan distintos como Big Data [16], álgebra computacional [17], análisis de imagen [18], finanzas [19] y biología [20], entre otros.

CUDA es una plataforma de computación paralela creada por NVIDIA que proporciona un compilador y un conjunto de herramientas de desarrollo cuyo objetivo es dotar a los programadores de la capacidad de usar las GPUs de NVIDIA para acelerar las partes computacionalmente intensas de sus aplicaciones. En la práctica, se trata de una extensión del lenguaje de programación C y C++, aunque también existen *wrappers*[1] para poder usar CUDA con Python, Fortran y Java. CUDA explota las ventajas de las GPUs frente a las CPUs de propósito general usando el alto grado de paralelismo que presentan con sus múltiples núcleos, que además permiten el lanzamiento de un altísimo número de hilos de ejecución de forma concurrente.

Para usar CUDA, un programador necesitaría, en primer lugar, una GPU de NVIDIA compatible con CUDA, aunque la práctica totalidad de las GPUs actuales de NVIDIA lo son. Además, necesitaría tener instalado un driver apropiado para dicha GPU, siendo los drivers más recientes apropiados para ello. Por último, necesitaría descargar e instalar el *CUDA Toolkit* de la web de NVIDIA, siendo posible a través de este enlace. Una vez instalado, dicho *toolkit* proporciona, principalmente, un compilador llamado `nvcc` basado

[1] Nombre que se le da a algunas funciones o programas cuyo propósito principal es llamar a una segunda función o programa.

en gcc[2] modificado para ser compatible con la API de CUDA, y un conjunto de librerías que contienen toda la funcionalidad que proporciona dicha API.

La figura de código fuente 2.1 proporciona un ejemplo de programa escrito en C que utiliza la API de CUDA para calcular la suma de dos matrices.

```
1  #include <stdio.h>
2  #define N 4
3  #define M 5
4
5  //definicion del kernel
6  __global__ void add(int *a, int *b, int *c) {
7      int tidx=threadIdx.x + blockIdx.x * blockDim.x;
8      int tidy=threadIdx.y + blockIdx.y * blockDim.y;
9
10     if (tidx < N && tidy < M) {
11         c[tidx + tidy*N] = a[tidx + tidy*N] + b[tidx + tidy*N];
12     }
13 }
14
15 int main() {
16     int a[N*M], b[N*M], c[N*M];
17     int *dev_a, *dev_b, *dev_c,i;
18
19     //reservar memoria en GPU
20     cudaMalloc((void **) &dev_a, N*M*sizeof(int) );
21     cudaMalloc((void **) &dev_b, N*M*sizeof(int) );
22     cudaMalloc((void **) &dev_c, N*M*sizeof(int) );
23
24     //rellenar matrices en CPU
25     for (i=0;i<N*M;i++) {
26         a[i]=i;
27         b[i]=i;
28     }
29
30     //enviar matrices a GPU
31     cudaMemcpy( dev_a, a, N*M*sizeof(int) , cudaMemcpyHostToDevice );
32     cudaMemcpy( dev_b, b, N*M*sizeof(int) , cudaMemcpyHostToDevice );
33     cudaMemcpy( dev_c, c, N*M*sizeof(int) , cudaMemcpyHostToDevice );
34
35     dim3 thr_p_block(N, M);
36
37     //llamar al Kernel
38     add<<<1, thr_p_block>>>(dev_a,dev_b,dev_c);
39
40     //obtener el resultado de vuelta en la CPU
41     cudaMemcpy( c, dev_c, N*M*sizeof(int), cudaMemcpyDeviceToHost );
42
43     //liberar memoria
44     cudaFree(dev_a) ;
45     cudaFree(dev_b) ;
46     cudaFree(dev_c) ;
47 }
```

Código fuente 2.1: Ejemplo de programa CUDA que realiza la suma de dos matrices usando GPUs

Tal y como puede verse en la figura de código 2.1, la función encargada de realizar el cómputo en la GPU se denomina *kernel*, y es la ubicada en la línea 6. El programa principal utiliza llamadas a funciones CUDA para, en primer lugar, reservar memoria

[2]Compilador creado por el proyecto GNU que permite compilar y generar binarios a través del código fuente creado con los lenguajes C, C++, Fortran y Ada, entre otros.

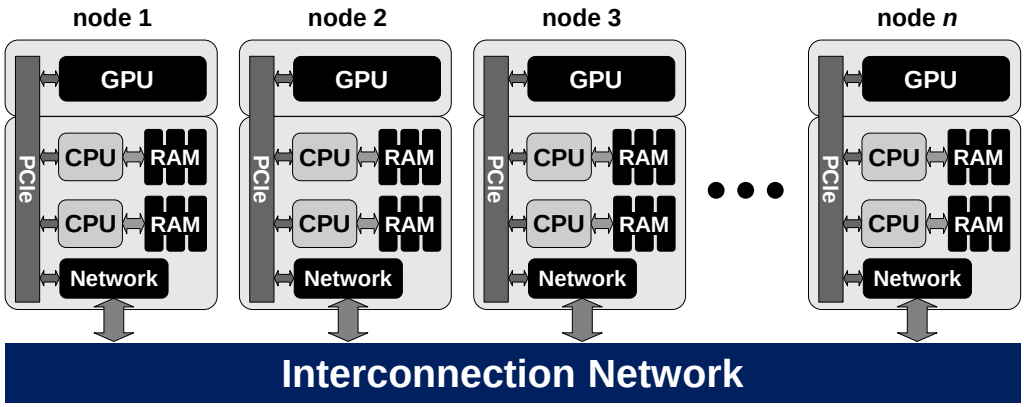


Figura 2.1: Ejemplo de un clúster acelerado con GPUs (cortesía de [8])

en la GPU usando llamadas `cudaMalloc`. En segundo lugar envía las matrices ubicadas en memoria principal a la memoria de la GPU empleando las llamadas `cudaMemcpy`. A continuación, realiza la llamada al kernel de nombre `add` en la línea 38, el cual realiza el cálculo de la suma en la GPU instalada en la máquina. Y por último, el programa retorna los resultados desde la memoria de la GPU a la memoria principal del sistema empleando de nuevo la función `cudaMemcpy`, para después liberar la memoria de GPU usando la función `cudaFree`.

Los centros de datos actuales incluyen típicamente una o más GPUs en los nodos de su clúster. Dependiendo de la configuración exacta del clúster, las GPUs podrían estar presentes únicamente en algunos de los nodos o, por el contrario, estar instaladas en todos ellos. La figura 2.1 muestra un ejemplo de clúster típico, compuesto de n nodos, donde cada nodo incluye una GPU.

A todo esto, la red de interconexión entre los nodos podría ser, por ejemplo, Ethernet o Infiniband [15], entre otras. Cabe destacar, no obstante, que usar GPUs con esta configuración no está exento de efectos colaterales. Por ejemplo, consideremos la ejecución de una aplicación paralela distribuida a través de MPI que no requiere el uso de GPUs. Típicamente, esta aplicación se propagará por diversos nodos del clúster, ocupando de esta manera todos los núcleos disponibles en ellos. En este escenario, las GPUs de los nodos involucrados en la ejecución de dicha aplicación MPI serán inaccesibles por otras aplicaciones dado que todos los cores están ocupados en la ejecución de la aplicación MPI, y por lo tanto no es posible lanzar nuevas aplicaciones en esos nodos. Esto causará que las GPUs permanezcan ociosas durante algunos períodos de tiempo, reduciendo de este modo la utilización global de las GPUs y haciendo que la inversión de capital inicial realizada durante la adquisición del clúster necesite más tiempo para ser amortizada.

Otro ejemplo de limitación asociada a cómo se usan actualmente las GPUs en clústeres está relacionado con la forma en que los planificadores de trabajos como Slurm [21] realizan la gestión de los recursos en un clúster. Estos planificadores son capaces de planificar con granularidad fina recursos tales como CPUs o memoria, pero no recursos como las GPUs. Por ejemplo, los planificadores pueden asignar recursos de CPU por núcleo, siendo así capaces de compartir los sockets de CPU presentes en un servidor entre muchas aplicaciones. En el caso de la memoria, los planificadores también pueden asignar, en un enfoque de memoria compartida, la memoria presente en un nodo dado a distintas aplicaciones que se ejecutarán concurrentemente en ese nodo. Sin embargo, en el caso de las GPUs, los planificadores usan una granularidad de GPU. A este respecto, las GPUs son asignadas a las aplicaciones de forma totalmente exclusiva. Por lo tanto, una GPU no puede ser compartida entre distintas aplicaciones incluso cuando tiene recursos

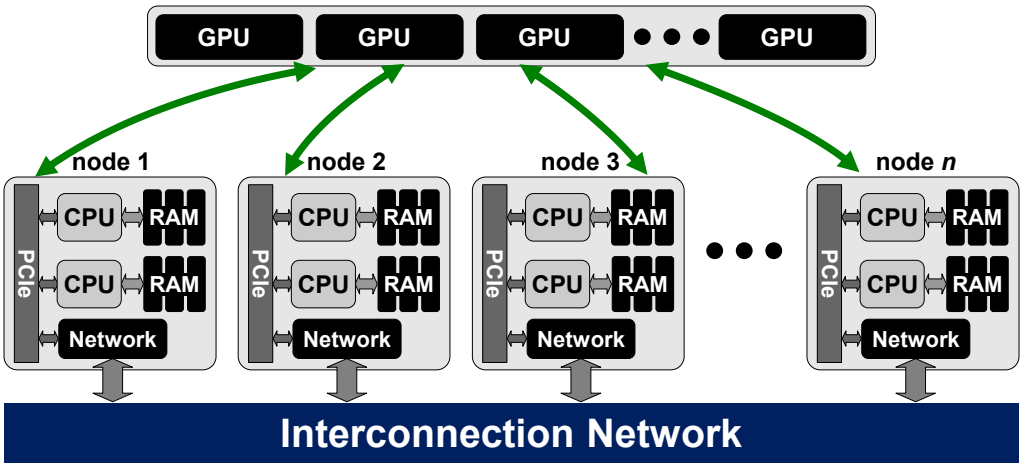


Figura 2.2: Configuración lógica de un clúster cuando se utiliza la técnica de virtualización de GPUs remotas (cortesía de [8])

suficientes para permitir la ejecución concurrente de esas aplicaciones, causando que la utilización global de las GPUs sea, en general, baja. Este hecho no solo reduce la potencia de cómputo efectiva sino también causa un desperdicio de energía nada despreciable.

Con objeto de reducir algunos de los efectos colaterales relacionados con el uso de GPUs en los clústeres de altas prestaciones, es posible utilizar el mecanismo de virtualización remota de GPUs. Este mecanismo permite a una aplicación ser ejecutada en un nodo que no posee ninguna GPU mientras que usa aceleradores instalados en otros nodos del clúster. En otras palabras, la técnica de virtualización remota de GPUs permite que las GPUs sean desacopladas de los nodos desde un punto de vista lógico, permitiendo de este modo que las GPUs desacopladas puedan ser compartidas simultáneamente por todos los nodos de la instalación de forma transparente a las aplicaciones. La figura 2.2 muestra la configuración lógica de un clúster tras aplicar el mecanismo de virtualización remota de GPUs.

En la nueva configuración, las GPUs son desacopladas de sus nodos anfitriones desde el punto de vista lógico creando, de esta manera, un pool de GPUs donde cada GPU puede ser accedida desde cualquier nodo del clúster. Además, una GPU dada podría servir concurrentemente a más de una aplicación. Esta forma de compartir GPUs no solo incrementa la utilización global de las GPUs, sino que también permite crear configuraciones donde no todos los nodos del clúster poseen GPUs localmente instaladas, a pesar de que todos los nodos podrán ejecutar aplicaciones aceleradas por GPUs. Esta configuración reduciría los costes asociados a la adquisición y posterior uso de GPUs. A este respecto, la energía total requerida para operar un centro de datos acelerado por GPUs se reduciría [9], lo que relaja en cierto modo las grandes preocupaciones energéticas de las futuras instalaciones de computación exascale[3].

Así pues, rCUDA (Remote CUDA, CUDA remoto) es un middleware de virtualización de GPUs que emplea una arquitectura cliente/servidor y que ha sido desarrollado por un grupo de investigadores de la Universitat Politècnica de València. rCUDA es totalmente compatible con la interfaz de programación de aplicaciones de CUDA y, entre otras cosas, permite la asignación de múltiples instancias virtuales de GPUs físicas a una misma aplicación sin necesidad de modificar el código fuente de las aplicaciones y sin necesidad de emplear técnicas de programación distribuida como MPI. Además, tal y co-

[3] La computación exascale hace referencia a los computadores capaces de realizar un mínimo de 1 exaflop, esto es, 10^{18} operaciones de coma flotante por segundo.

mo se comentará en las secciones posteriores, la utilización de rCUDA proporciona una serie de beneficios adicionales en otros campos como puede ser el de utilizar GPUs en máquinas virtuales de forma más eficiente a como se hace actualmente o el de tener la capacidad de migrar trabajos de una GPU hacia otras GPUs más ociosas con el objetivo de ahorrar energía o equilibrar la carga.

En esencia, rCUDA es una tecnología que proporciona mucha flexibilidad en la utilización de GPUs para la computación de propósito general (GPGPU, General-Purpose Computing on Graphics Processing Units) [22]. En el momento de escribir estas líneas, la última versión estable de rCUDA es la 16.11, versión preparada para ejecutar la mayoría de aplicaciones del género HPC [1]. No obstante, se espera que en un breve espacio de tiempo se presente la nueva versión estable que incluirá soporte para los frameworks de Deep Learning más populares del mercado, como por ejemplo Caffe [23] y TensorFlow [24]. En estos momentos está liberada la versión 18.03 beta que incluye soporte sin optimizar para dichos frameworks además de soporte para programas de renderizado como Blender y librerías tales como cuSolver [25], cuBLAS-XT [26] y nvGRAPH [53], entre otras. Puede solicitarse una copia de los ficheros binarios de rCUDA cumplimentando un breve formulario a través de la web del proyecto rCUDA www.rcuda.net.

En las siguientes secciones se introduce un análisis sobre la técnica de virtualización remota de GPUs a nivel general y, posteriormente, un análisis en profundidad de lo que ofrece rCUDA en el ámbito de esta técnica.

2.2 Análisis de la técnica de virtualización remota de GPUs

Las librerías de software tales como CUDA permiten a los programadores usar GPUs para computación de propósito general. Existen multitud de soluciones de virtualización de GPUs que usan CUDA, tales como GridCuda [27], DS-CUDA [28], gVirtuS [29], vCUDA [30], GViM [31] y rCUDA [6]. En esencia, estas soluciones comparten las GPUs virtualizándolas. En este sentido, estos frameworks proveen a las aplicaciones instancias virtuales del dispositivo real, que por lo tanto puede ser compartido concurrentemente. Típicamente, estas soluciones para compartir GPUs sitúan el límite de la virtualización en el nivel de la API (CUDA en el caso de GPUs Nvidia). En general, las soluciones de virtualización basadas en CUDA ofrecen la misma API que la Runtime API[4] [32] de CUDA.

La figura 2.3 muestra la arquitectura que poseen la mayoría de estas soluciones de virtualización, que siguen un enfoque distribuido cliente/servidor. La parte cliente del middleware se instala en el nodo del clúster que ejecuta la aplicación que necesita ser acelerada con GPUs, mientras que la parte del servidor se ejecuta en el nodo que posee actualmente la o las GPUs. De esta forma, el cliente recibe una petición CUDA desde la aplicación que está siendo acelerada, la interpreta y la reenvía convenientemente al servidor remoto. En el servidor, el middleware recibe la petición, la interpreta y la envía a la GPU, que completa la ejecución de la petición y provee los resultados de la ejecución al servidor. Como respuesta, el servidor envía los resultados al cliente, que los reenvía a su vez a la aplicación original, la cual no es consciente de que su petición CUDA ha sido servida por una GPU remota en lugar de por una local.

Las soluciones de virtualización remota de GPUs basadas en CUDA puede ser clasificadas en dos tipos: (1) aquellas destinadas a usarse en el contexto de máquinas virtuales y (2) aquellas ideadas como soluciones de virtualización de propósito general, para ser

[4]CUDA proporciona dos APIs distintas. Mientras que la Driver API ofrece funciones de bajo nivel de abstracción pero de alta complejidad, la Runtime API ofrece funciones de alto nivel con una complejidad sensiblemente menor.

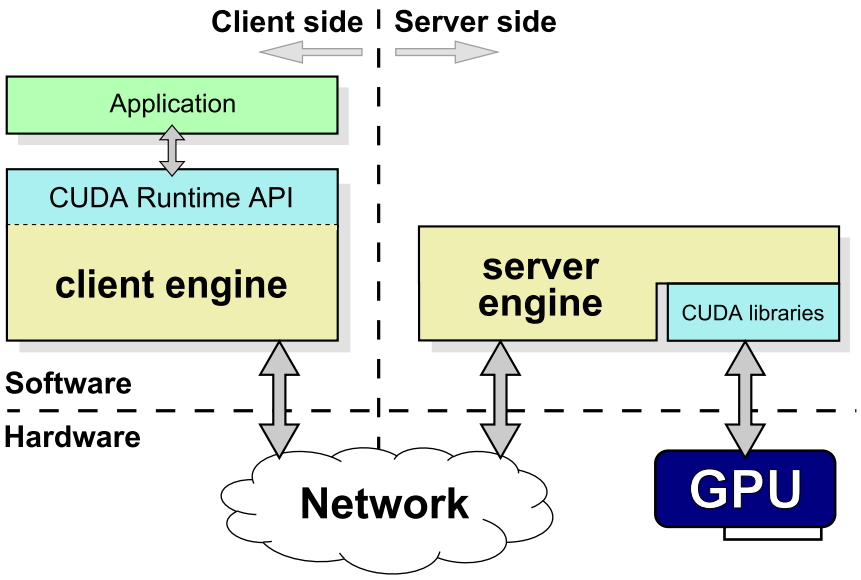


Figura 2.3: Arquitectura típica de los frameworks de virtualización remota de GPUs (cortesía de [8])

usadas en dominios nativos (considérese que este último enfoque también puede ser aplicado en el contexto de máquinas virtuales). Los frameworks pertenecientes a la primera categoría hacen uso de mecanismos de memoria compartida para transferir datos de memoria principal de la máquina virtual a la GPU en el host anfitrión, mientras que las soluciones del segundo tipo hacen uso de la estructura de red del clúster para transferir datos de la memoria principal del cliente a la GPU remota localizada en el servidor. Esta es la razón por la que las soluciones de la segunda aproximación son conocidas comúnmente como soluciones de virtualización remota de GPUs.

Al respecto del primer tipo de virtualización remota de GPUs descrito en el párrafo anterior, existen múltiples soluciones que han sido desarrolladas para ser usadas específicamente en máquinas virtuales, tales como vCUDA, GViM, gVirtuS y Shadowfax [33]. La tecnología vCUDA, destinada para máquinas virtuales Xen, solamente soporta una versión de CUDA antigua (3.2) e implementa un subconjunto no especificado de la Runtime API de CUDA. Además, su protocolo de comunicaciones presenta una considerable sobrecarga por el coste de codificar y descodificar las llamadas. Esta sobrecarga causa una notoria caída en el rendimiento global. Por otra parte, GViM, también destinado a entornos Xen, se basa en la versión obsoleta de CUDA 1.1 y, en principio, no implementa el conjunto completo de la Runtime API de CUDA. gVirtuS se basa en la versión de CUDA 6.5 y también implementa un pequeño subconjunto de las funciones de la Runtime API. A pesar de estar diseñado para máquinas virtuales, provee también comunicaciones TCP/IP para virtualización de GPUs remotas, permitiendo a las aplicaciones que se ejecutan en un entorno no acelerado por GPUs acceder a las GPUs ubicadas en otros nodos. Al respecto de Shadowfax, esta solución permite a las máquinas virtuales Xen acceder a las GPUs ubicadas en el mismo nodo, aunque también sería posible acceder a las GPUs localizadas en otros nodos del clúster. Soporta la versión 1.1 de CUDA y, además, ni el código fuente ni los binarios están disponibles con objeto de evaluar su rendimiento.

Con respecto al segundo tipo de virtualización mencionado, que provee virtualización remota de GPUs para computación de propósito general tanto para máquinas virtuales como para entornos reales, se pueden encontrar las soluciones rCUDA, GridCUDA y DS-CUDA. rCUDA, que será descrito convenientemente en la sección 2.3, es compatible con CUDA 9.1 y provee soporte específico para comunicaciones en redes compatibles

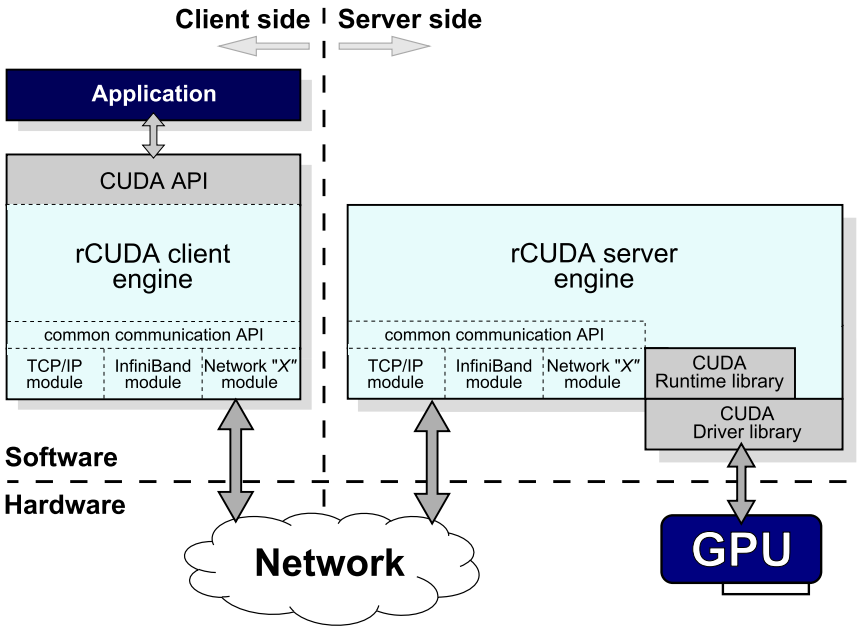


Figura 2.4: Arquitectura detallada de rCUDA (cortesía de [8])

con TCP/IP, Infiniband y RoCE (rCUDA usa la API Infiniband Verbs con objeto de aprovechar las características RDMA[5] de esta red). GridCUDA también ofrece acceso a GPUs remotas en un clúster, pero soporta la antigua versión de CUDA 2.3. Además, no existe actualmente una versión pública de GridCUDA de tal forma que pueda ser probada. Con respecto a DS-CUDA, integra una versión más reciente de CUDA (4.1) e incluye soporte específico para comunicaciones Infiniband usando la API Infiniband Verbs. Sin embargo, DS-CUDA presenta múltiples limitaciones, como por ejemplo no permitir transferencias con memoria *pinned*[6].

2.3 Análisis del middleware de virtualización de GPUs rCUDA

En esta sección se describe la tecnología rCUDA con un mayor nivel de detalle. La figura 2.4 representa una vista detallada de la arquitectura del framework rCUDA. rCUDA soporta actualmente las versiones de CUDA 7, 7.5, 8.0, 9.0 y 9.1, siendo compatible binario con estas versiones, lo que implica que los programas CUDA no necesitan ser modificados con objeto de usar rCUDA en su lugar. Además, implementa la Runtime API y Driver API (excepto las funciones para gráficos) y también proporciona soporte para las librerías incluídas en CUDA, tales como cuFFT [34], cuBLAS [35], cuSPARSE [36], etc. A todo esto, el middleware rCUDA permite que un solo servidor preste servicio concurrentemente a múltiples clientes remotos que le solicitan acceso a sus GPUs. Esto es posible creando contextos de GPU independientes, todos ellos siendo asignados a clientes diferentes. Estos contextos de GPU independientes proporcionan aislamiento, y por lo tanto robustez, frente a los posibles fallos de un cliente.

rCUDA porporciona soporte específico para diferentes redes de interconexión. Este soporte es llevado a cabo haciendo uso de un conjunto de módulos de comunicación

[5]Remote Direct Memory Access (acceso remoto directo a memoria), técnica para realizar transferencias desde memoria principal de un nodo a la de otro sin la intervención del microprocesador.

[6]Tipo de memoria que el sistema operativo no pagina nunca a disco, por lo que su rendimiento es muy superior al de la memoria paginable.

específicos que se cargan en tiempo de ejecución, y que han sido implementados específicamente con objeto de conseguir tanto rendimiento como sea posible de la red de interconexión subyacente. Actualmente, hay disponibles tres módulos de comunicaciones: uno destinado a redes compatibles con TCP/IP, otro específicamente diseñado para Infiniband y un tercero para dar soporte a redes RoCE.

Además, independientemente de la red usada, el intercambio de datos entre los clientes de rCUDA y las GPUs gestionadas por los servidores rCUDA está segmentado de tal forma que se consiga el máximo ancho de banda. Los buffers internos utilizados en esta segmentación de rCUDA utilizan memoria pinned preasignada dado el ancho de banda más elevado de este tipo de memoria.

Usar rCUDA es muy sencillo. Simplemente se requiere prestar atención a un conjunto de variables de entorno del sistema operativo antes de lanzar la ejecución propiamente dicha de la aplicación: `RCUDA_DEVICE_COUNT`, `RCUDA_DEVICE_j` y `RCUDA_NETWORK`. La primera variable indica la cantidad de GPUs accesibles por la aplicación. Por ejemplo, si se asigna dos GPUs a la aplicación que debe ser acelerada, entonces el usuario ejecutaría el comando "`export RCUDA_DEVICE_COUNT=2`". La segunda variable de entorno, `RCUDA_DEVICE_j`, indica, para cada GPU asignada a la aplicación, en qué nodo del clúster se localiza la GPU cuyo identificador es `j`. En el ejemplo anterior, el usuario ejecutaría los comandos "`export RCUDA_DEVICE_0=192.168.0.1`" y "`export RCUDA_DEVICE_1=192.168.0.2`", asumiendo que los nodos con estas IPs se utilizan como servidores. Por último, la variable de entorno `RCUDA_NETWORK` determina el módulo de comunicaciones que el usuario desea utilizar en la ejecución de la aplicación. Por ejemplo, el usuario ejecutaría el comando "`export RCUDA_NETWORK=IB`" para aprovechar la API de Infiniband Verbs a través del módulo de comunicaciones implementado a tal efecto. En el caso de querer usar el módulo de TCP/IP, el usuario ejecutaría el comando "`export RCUDA_NETWORK=TCP`" o simplemente dejaría la variable sin establecer, ya que es el módulo de comunicaciones usado por defecto.

2.4 Beneficios del uso de rCUDA

En esta sección se introducen seis de los beneficios que el mecanismo de virtualización remota de GPUs en general, y rCUDA en particular, presenta para la ejecución de aplicaciones en clústeres de altas prestaciones. Estos beneficios, que serán detallados a continuación, son los siguientes:

1. Más GPUs disponibles para una sola aplicación.
2. Los sockets de CPU ocupados en un servidor no bloquean el uso de las GPUs ubicadas en ese servidor.
3. La productividad del clúster se incrementa al mismo tiempo que se reduce el consumo de energía. La utilización global de las GPUs también se incrementa.
4. Las actualizaciones físicas del clúster son más sencillas y baratas simplemente añadiendo servidores con GPUs a un clúster que no está acelerado con GPUs.
5. Múltiples máquinas virtuales pueden acceder simultáneamente a la misma GPU de forma compartida.
6. Los trabajos de GPU pueden migrarse fácilmente a través del clúster para que sean ejecutados en un número menor de nodos.

Las siguientes subsecciones describen con más detalle estos beneficios.

Beneficio 1: Más GPUs disponibles para una sola aplicación

Cuando se utiliza CUDA, una aplicación MPI puede ser distribuida a través de múltiples nodos del clúster con objeto de usar las GPUs instaladas en esos nodos. No obstante, una aplicación paralela de memoria compartida basada en el uso de hilos de ejecución solo puede ser ejecutada en un único nodo y, por lo tanto, solo puede beneficiarse de las GPUs que están localmente instaladas en ese nodo. Por el contrario, cuando se utiliza rCUDA, una aplicación que se ejecuta en un solo nodo puede hacer uso de todas las GPUs del clúster, aumentando así su rendimiento. En este caso, la única limitación para incrementar el rendimiento de la aplicación sería la habilidad del programador de organizar la aplicación de la forma adecuada para aprovechar tantas GPUs como haya disponibles.

La figura 2.5 muestra parte de la salida proporcionada por la ejecución del programa `deviceQuery` del Software Development Kit de CUDA. En este caso, las 64 GPUs instaladas en uno de los clústeres del Barcelona Supercomputing Center (BSC) fueron asignadas a la aplicación, mostrando de esta forma las posibilidades que ofrece la virtualización remota de GPUs.

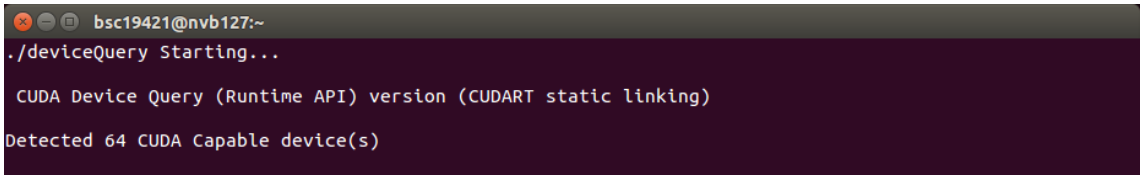


Figura 2.5: Programa deviceQuery del SDK de CUDA empleando 64 GPUs remotas (cortesía de [8])

Beneficio 2: Los sockets de CPU ocupados en un servidor no bloquean el uso de las GPUs ubicadas en ese servidor

Los usuarios de un clúster tienden a requerir tantos recursos como sea posible para ejecutar sus aplicaciones con objeto de reducir sus tiempos de ejecución. El hecho de requerir tantos recursos como sea posible puede lograrse de múltiples formas. Por ejemplo, es muy común que usuarios que están enviando aplicaciones paralelas de memoria compartida a las colas del planificador requieran para sus aplicaciones tantos núcleos CPU como haya disponibles en el nodo en cuestión. En la práctica, este requisito se traduce en que esa aplicación usará todos los núcleos del nodo en el que la aplicación se ejecute. De esta forma, durante la ejecución de la aplicación, no podrá ejecutarse ninguna otra debido a la ausencia de núcleos CPU disponibles.

De una forma similar, los usuarios también podrían enviar a las colas del planificador peticiones para ejecutar aplicaciones MPI de memoria compartida no aceleradas por GPUs. Estas aplicaciones abarcan múltiples nodos del clúster, tradicionalmente ocupando todos los núcleos de CPU presentes en dichos nodos. Tal y como se ha comentado en el ejemplo anterior, durante todo el tiempo en que la aplicación está en ejecución, ninguna otra aplicación tiene sitio en los nodos en los que se ejecuta esa aplicación ya que no existen núcleos de CPU disponibles.

Aunque la ejecución de tales aplicaciones podría llevar a un tiempo de ejecución reducido y por lo tanto, a una utilización global de recursos de CPU alta, cuando los nodos involucrados incluyen una o más GPUs, estos aceleradores permanecerán ociosos durante el tiempo que la aplicación esté en ejecución. Los aceleradores en esos nodos no están, por lo tanto, disponibles para otras aplicaciones porque, con objeto de usarlos, se requiere

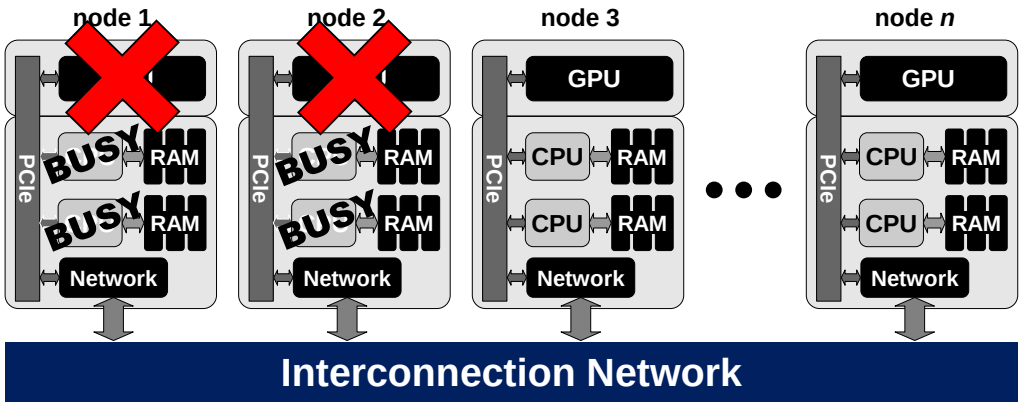


Figura 2.6: Las GPUs de los nodos 1 y 2 no están disponibles debido a que todos los núcleos de CPU de los nodos están ocupados con la ejecución de aplicaciones no aceleradas con GPU (cortesía de [8])

lanzar esas aplicaciones en dichos nodos. Sin embargo, no es posible porque esas aplicaciones requerirían al menos un núcleo de CPU disponible, pero todos los núcleos CPU han sido asignados a la aplicación no acelerada y, por lo tanto, el planificador de trabajos no lanzará ninguna aplicación en ese nodo en cuestión. Esta condición está representada en la figura 2.6, donde las GPUs en los nodos 1 y 2 del clúster no están disponibles porque todos los núcleos de CPU de esos nodos están ocupados con la ejecución de aplicaciones no aceleradas. Nótese que esas GPUs bloqueadas no solo provocan una reducción temporal de la potencia de cómputo global del clúster, sino que siguen consumiendo energía. Por ejemplo, la GPU Nvidia Tesla K20m consume 25W durante el tiempo de inactividad. De una forma similar, la GPU Nvidia Tesla K40m consume 20W.

El mecanismo de virtualización remota de GPUs podría ser útil en esos escenarios, tal y como se muestra en la figura 2.7. Cuando alguna aplicación no acelerada bloquea el uso de las GPUs en uno o varios nodos del clúster, los frameworks como rCUDA podrían hacer posible el uso de estas GPUs asignándolas a aplicaciones que se están ejecutando en otros nodos del clúster. El resultado final es que, además de incrementar la utilización global de los procesadores, también se incrementa la utilización global de las GPUs. Por otra parte, cabe recordar que rCUDA hace uso del servidor rCUDA con objeto de proveer acceso a las GPUs remotas. Dicho servidor, que se ejecuta como demonio, debe ser ejecutado en uno de los núcleos de CPU en el nodo que posee las GPUs localmente instaladas. No obstante, este demonio introduce una sobrecarga media-baja.

Beneficio 3: La productividad del clúster se incrementa al mismo tiempo que se reduce el consumo de energía. La utilización global de las GPUs también se incrementa

Cuando se utiliza la técnica de virtualización remota de GPUs en un clúster, las GPUs pueden ser compartidas concurrentemente entre múltiples aplicaciones tanto como la memoria disponible en dichas GPUs permita. Además, dado que una GPU puede ser usada por aplicaciones que se están ejecutando en nodos distintos a los que tienen las GPUs localmente instaladas, cuando todos los núcleos de CPU en el nodo que posee una GPU están ocupados con una aplicación no acelerada, la GPU puede ser usada por otro nodo del clúster, tal y como se describió en el beneficio 2. Estas características contribuyen a una tasa de utilización de GPU más alta, que se traduce en un incremento de la productividad del clúster (medida en trabajos realizados por unidad de tiempo) y en una reducción de la energía consumida.

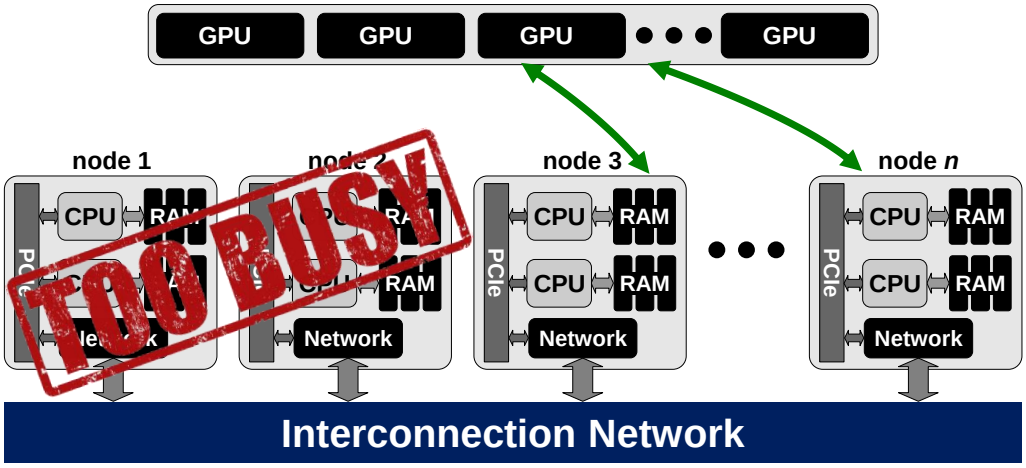


Figura 2.7: La técnica de virtualización remota de GPUs permite que las GPUs en nodos cuyos núcleos de CPU están ocupados puedan ser usadas por aplicaciones que están siendo ejecutadas en otros nodos del clúster (cortesía de [8])

Beneficio 4: Las actualizaciones físicas del clúster son más sencillas y baratas simplemente añadiendo servidores con GPUs a un clúster que no está acelerado con GPUs

El uso de GPUs en un clúster impone múltiples consideraciones al respecto de la configuración física de los nodos del clúster. Por ejemplo, los nodos que tienen GPUs necesitan incluir fuentes de alimentación más potentes capaces de suministrar la energía requerida por las mismas. Además, las GPUs no son precisamente dispositivos pequeños, por lo que requieren una nada despreciable cantidad de espacio en los nodos donde están instaladas. Estos requisitos hacen que instalar GPUs en un clúster que inicialmente no las incluía pueda llegar a resultar bastante caro (las fuentes de alimentación deben ser actualizadas con objeto de incrementar su potencia) o simplemente imposible (los nodos no tienen espacio físico suficiente para las GPUs). Sin embargo, y a pesar de todo esto, las cargas de trabajo de algunos centros de datos podrían evolucionar hacia el uso de GPUs. En ese punto, la preocupación principal es cómo llevar a cabo la introducción de GPUs en una instalación computacional que no incluía GPUs en el momento de la adquisición.

Una posible solución a este problema es adquirir una cantidad determinada de servidores que incluyan GPUs y desviar la ejecución de las aplicaciones aceleradas con GPUs hacia esos nodos. El planificador Slurm se encargaría automáticamente de enviar los trabajos de aplicaciones aceleradas a esos nuevos servidores. Sin embargo, aunque esta aproximación es factible, presenta la limitación de que los trabajos que incluyen GPUs probablemente tendrán que esperar durante un tiempo hasta que alguno de los servidores con GPUs esté disponible incluso aunque la tasa de utilización de la GPU sea baja. Otra limitación es que las aplicaciones aceleradas a través de MPI solamente serán capaces de abarcar tantos nodos como servidores con GPUs fueron adquiridos inicialmente. Dadas estas limitaciones, una mejor aproximación podría ser adquirir unos cuantos servidores con GPUs y utilizar rCUDA para ejecutar las aplicaciones aceleradas con GPUs en cualquiera de los nodos mientras que se usan las GPUs de los nuevos servidores, tal y como muestra la figura 2.8. Esta solución no solamente incrementaría la utilización global de las GPUs con respecto al uso de CUDA en el escenario anterior, sino que permitiría también que las aplicaciones MPI pudieran ser ejecutadas en tantos nodos como requiriesen ya que los procesos MPI podrían acceder remotamente a las GPUs a través de rCUDA. En resumen, la técnica de virtualización remota de GPUs permite a los clústeres que no incluyeron inicialmente GPUs actualizarse de forma fácil y barata incluyendo

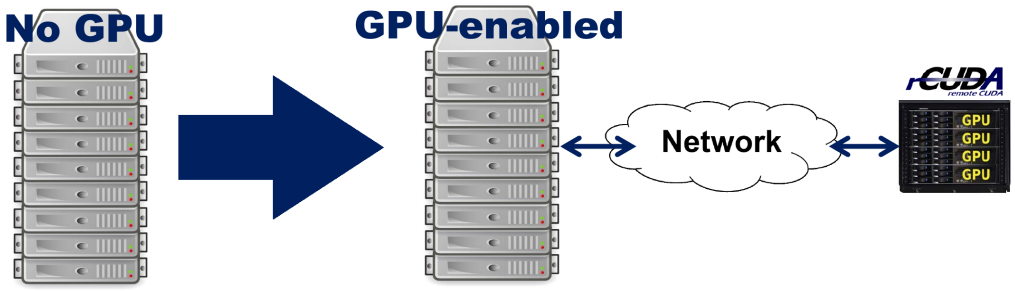


Figura 2.8: Actualización física de un clúster que no dispone inicialmente de GPUs añadiendo unos pocos nodos con GPUs. Así, las GPUs pueden ser usadas tanto por los nodos que las poseen como por los nodos que no las poseen, usando en este último caso la red de interconexión para acceder a las GPUs (cortesía de [8])

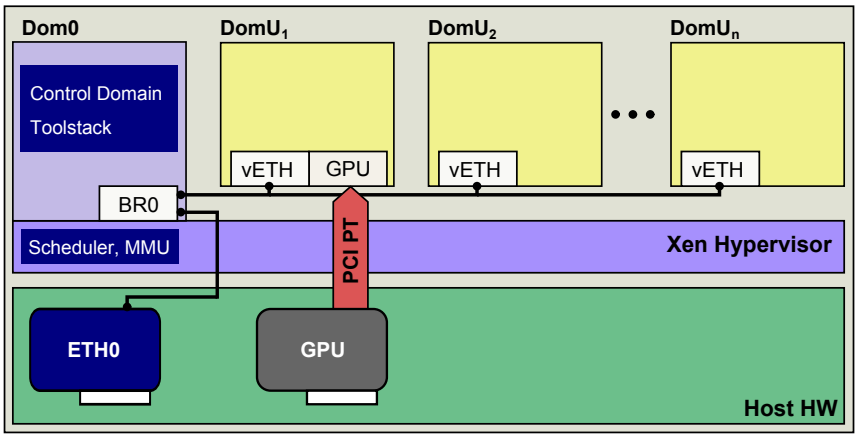


Figura 2.9: Configuración típica de un sistema de virtualización Xen mostrando la forma en que se le proporcionan a la máquina virtual el adaptador de red y la GPU disponibles en el host (cortesía de [8])

servidores que contienen GPUs. De esta forma, los nodos originales podrán acceder a las GPUs instaladas en los nuevos, que pasarán a ser "servidores de GPUs". Se utilizaría, en este caso, una versión modificada de Slurm para planificar el uso de las GPUs en los nuevos servidores.

Beneficio 5: Múltiples máquinas virtuales pueden acceder simultáneamente a la misma GPU de forma compartida

Es habitual proporcionar aceleración CUDA a máquinas virtuales a través de la técnica PCI Passthrough[7]. Esta técnica está basada en el uso de las extensiones de virtualización ampliamente disponibles en los servidores actuales de computación de altas prestaciones. La técnica PCI Passthrough permite asignar una GPU, de forma exclusiva, a una de las máquinas virtuales que se está ejecutando en el sistema anfitrión. Además, cuando se usa este mecanismo, el rendimiento que se obtiene con los aceleradores está muy cerca del conseguido cuando se emplean GPUs en dominios nativos. La figura 2.9 muestra un despliegue típico de máquina virtual en el hipervisor Xen, mostrando un host alojando múltiples máquinas virtuales.

[7]Técnica para asignar de forma exclusiva un dispositivo PCI del sistema anfitrión a una máquina virtual que se ejecuta en dicho anfitrión.

Se puede ver en la figura 2.9 que el hardware del sistema anfitrión está compuesto por, entre otros dispositivos, un adaptador de red Ethernet y una GPU. Por encima de la capa de hardware se instala una delgada capa de software consistente en el hipervisor Xen. Sobre el hipervisor residen las máquinas virtuales (Dom0 y DomU_i). Nótese que la máquina virtual Dom0 es una máquina virtual predefinida usando el kernel Linux Xen y se comporta como la interfaz de configuración y administración del hipervisor. El resto de máquinas virtuales (desde DomU_1 hasta DomU_n) son máquinas virtuales sin privilegios de administración que pueden ser proporcionadas a los usuarios. La figura 2.9 muestra la forma en que se proporcionan el adaptador Ethernet y la GPU a la máquina virtual. Por otra parte, el adaptador Ethernet está asignado a la máquina virtual Dom0, que proporciona conectividad al resto de máquinas usando un switch Ethernet software, creando así una red virtual entre las máquinas virtuales. La GPU, sin embargo, está asignada a una de las máquinas virtuales usando la técnica de PCI Passthrough. En el caso de otros hipervisores tales como KVM [10], el despliegue global es similar a excepción de algunos detalles de configuración.

Sin embargo, el enfoque de usar PCI Passthrough asigna las GPUs de forma exclusiva a las máquinas virtuales y, por lo tanto, no permite compartir las GPUs de forma concurrente entre máquinas virtuales que se están ejecutando simultáneamente en el mismo host. En el caso de la figura 2.9, la máquina virtual DomU_1 es la única que tiene acceso a la GPU. El resto de máquinas virtuales que están en ejecución en ese host no podrán hacer uso de ese acelerador hasta que deje de estar asignado a la máquina DomU_1. Además, es importante destacar que solamente una de las restantes máquinas podrá utilizar la GPU una vez sea liberada de DomU_1. Es decir, la GPU no podrá ser compartida por más de una máquina virtual simultáneamente. Dicho esto, es notoria la poca flexibilidad que esta configuración ofrece al respecto de la gestión de GPUs en el contexto de máquinas virtuales, dado que solamente una de ellas puede acceder a la GPU en un momento dado.

Con la técnica de virtualización remota de GPUs es posible asignar de forma concurrente una GPU dada a múltiples máquinas virtuales, de tal forma que las aplicaciones que se están ejecutando en ellas pueden compartir los recursos de GPU. Pueden considerarse dos escenarios distintos: uno donde las máquinas virtuales acceden a la GPU localizada en el mismo host que ejecuta las máquinas virtuales, y otro donde se tiene una instalación de red Infiniband en el clúster y por lo tanto las máquinas virtuales pueden acceder a las GPUs instaladas en otros nodos del clúster. La figura 2.10 muestra el primer escenario, mientras que la figura 2.11 muestra el segundo.

En el primer escenario, una de las máquinas virtuales tendrá acceso exclusivo a la GPU haciendo uso de la técnica de PCI Passthrough. Esta máquina virtual otorgará acceso a la GPU al resto de máquinas virtuales usando el middleware rCUDA: el servidor rCUDA se ejecutará en la máquina virtual que tiene en exclusiva la GPU mientras que las otras máquinas usarán el cliente rCUDA para acceder a la GPU a través de la red virtual de Xen. En este escenario se utilizarán las comunicaciones basadas en TCP/IP para comunicar los clientes rCUDA con el servidor rCUDA. En consecuencia, las máquinas virtuales que ejecuten el cliente rCUDA tendrán una o más instancias virtuales (vGPU) de la GPU real, que estará físicamente conectada a la máquina virtual DomU_1. Además, está máquina podrá usar tanto la GPU real como sus instancias virtuales.

Con respecto al segundo escenario, mostrado en la figura 2.11, que usa la red Infiniband ya presente en el clúster para acceder a las GPUs remotas, se debe cambiar el firmware del adaptador de red Infiniband tal y como muestran las directrices de la Mellanox User's Guide [37], con objeto de proporcionar instancias virtuales (virtual functions, VF's) del adaptador de red Infiniband, además de la instancia real (physical funtion, PF). Cada una de estas funciones virtuales serán proporcionadas de forma exclusiva a una

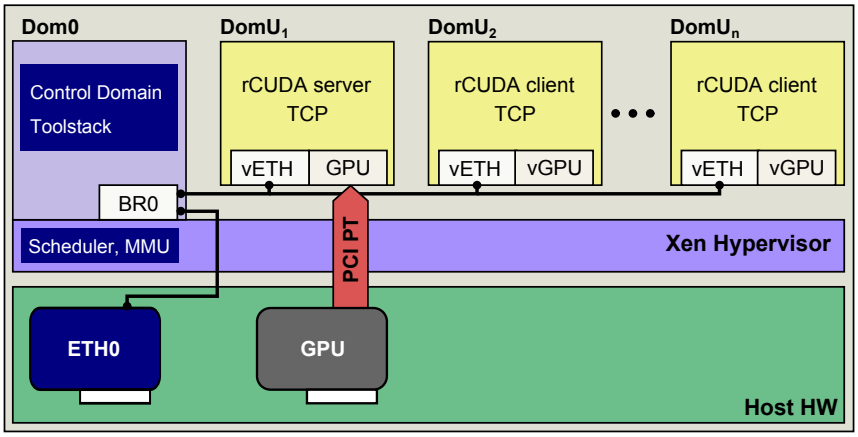


Figura 2.10: Escenario donde las máquinas virtuales acceden a la GPU del host (cortesía de [8])

máquina virtual Xen usando la técnica de PCI Passthrough. Además, dado que existe una red de Infiniband disponible en el clúster, la comunicación entre los clientes rCUDA en las máquinas virtuales y el servidor rCUDA se realizará a través de la API de altas prestaciones de Infiniband Verbs.

Beneficio 6: Los trabajos de GPU pueden migrarse fácilmente a través del clúster

Maximizar la tasa de utilización de los recursos es uno de los objetivos perseguidos cuando se instala un centro de datos. Maximizando la utilización de los distintos recursos en la instalación se puede obtener un mayor beneficio, causando de esta forma una amortización más rápida de la adquisición. Otra posibilidad es utilizar técnicas de balanceo de carga, de manera que los usuarios del centro de datos obtienen unos tiempos de respuesta óptimos. El balanceo de carga puede hacerse atendiendo a diversos criterios, entre los cuales se puede incluir el uso de prioridades para diferentes usuarios del centro de datos (usuarios con mayor prioridad usan los recursos de manera exclusiva obteniendo por tanto mejores tiempos de respuesta).

Sea cual sea la política usada en el centro de datos (maximizar la utilización o balancear la carga), la utilización de los recursos evoluciona con el tiempo, dependiendo de las cargas de trabajo existentes en cada momento. Por lo tanto, y en algún momento, la utilización de las GPUs en el clúster podría ser similar a la que se muestra en la figura 2.12a.

Esta figura muestra un pequeño clúster de 14 nodos, cada uno de ellos con una GPU. Se muestra a la derecha de cada nodo el porcentaje de utilización de la GPU. Puede verse que algunos nodos presentan un alto porcentaje de utilización. Por ejemplo, los nodos 9 y 12 están usando su GPU en un 90 % aproximadamente. Por contra, otros nodos presentan una tasa de utilización mucho más baja, como puede ser el caso de los nodos 6 y 11, cuyas GPUs prácticamente están ociosas.

En este escenario donde algunos nodos del clúster presentan una baja tasa de utilización, sería útil migrar los trabajos que se están ejecutando en esos nodos hacia otros nodos. Esto es, sería interesante consolidar los trabajos en un número menor de servidores, de tal forma que los nodos que quedaran libres pudieran ser apagados o puestos en estados de bajo consumo, reduciendo así la energía consumida por el centro de datos. La figura 2.12b muestra esta consolidación, donde los trabajos que presentan una baja tasa de utilización de la GPU, como es el caso de los nodos 2, 4, 5, 6, 8, 10 y 11 han sido

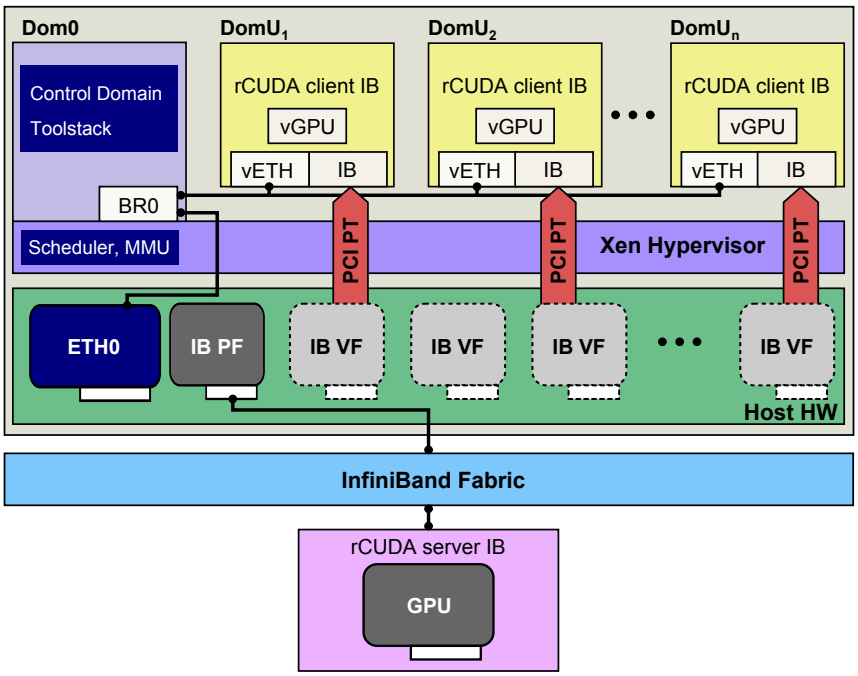
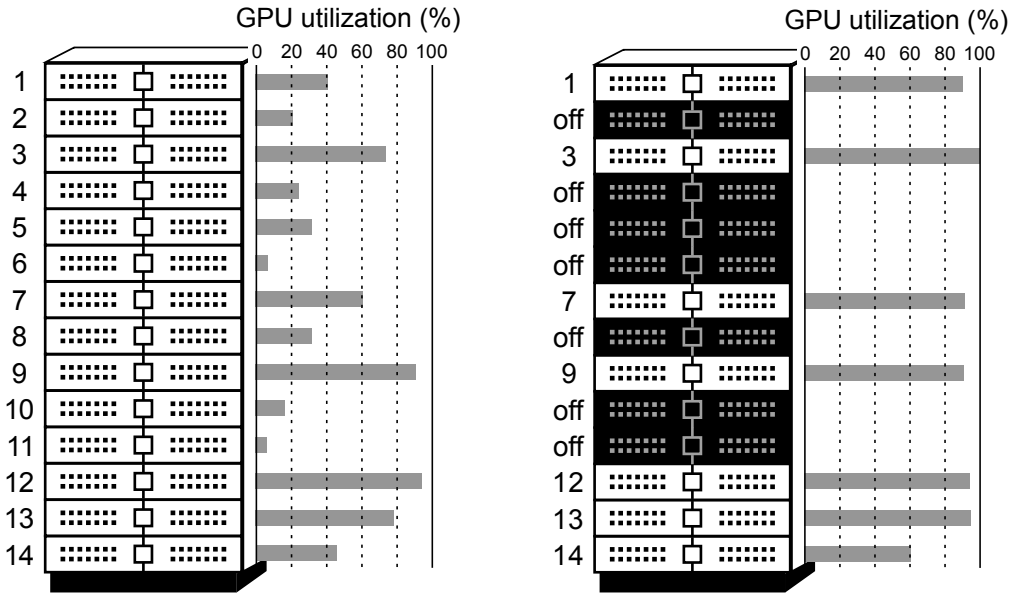


Figura 2.11: Escenario donde las máquinas virtuales acceden a las GPUs instaladas en otros nodos del clúster (cortesía de [8])



(a) La tasa de utilización de las GPUs antes de la puesta en marcha de los servidores rCUDA. Todos los nodos del clúster están en marcha.

(b) Estado del clúster tras poner en marcha los servidores rCUDA y migrar trabajos de GPU. Únicamente siete nodos están en marcha.

Figura 2.12: Uso de la migración de rCUDA en un clúster con objeto de consolidar los trabajos de GPU y reducir energía (cortesía de [8])

migrados a otros nodos. Tras la migración de esos trabajos, los nodos liberados han sido apagados, reduciendo por lo tanto el consumo de energía del clúster.

Llevar a cabo la migración de trabajos que usan GPU requiere migrar tanto los procesos que se están ejecutando en la GPU como en la CPU que está usando la aplicación. La migración de trabajos de CPU se ha llevado a cabo a través de distintos frameworks y es un mecanismo ampliamente conocido. Sin embargo, migrar la parte GPU de una aplicación CUDA es mucho más complejo por dos razones: (1) los kernels que se están ejecutando en la GPU se ejecutan de forma asíncrona con la CPU y, por lo tanto, cuando se pone en marcha la migración, el kernel en la GPU podría estar todavía en ejecución, y (2) la memoria de GPU reservada para la aplicación, que no está siendo controlada por el sistema operativo, debe copiarse a la GPU de destino.

Llevar a cabo el primer punto no debería ser difícil. Una vez se ha puesto en marcha la migración, el framework de migración podría ejecutar una llamada de sincronización (por ejemplo la función `cudaDeviceSynchronize`) con objeto de esperar a que los kernels que se están ejecutando en la GPU se completen. Sin embargo, abordar el segundo punto no es tan sencillo dado que el mapa de memoria usado por la aplicación en la GPU no está incluído en las tablas del sistema operativo de ese proceso. Por lo tanto, a no ser que se implemente algún tipo de soporte adicional, no es posible recuperar las regiones de memoria usadas por la aplicación en la GPU.

Con objeto de implementar este soporte adicional, se podrían interceptar las llamadas de gestión de memoria de GPU ejecutadas por la aplicación (tales como las funciones `cudaMalloc` y `cudaFree`). Al hacerlo, es posible obtener la información requerida para recuperar las regiones de memoria usadas en la GPU.

No obstante, aunque obtener la información requerida acerca de las regiones de memoria usadas en la GPU hace posible migrar aplicaciones CUDA de un nodo a otro, es posible hacer una migración todavía más efectiva cuando se utiliza la técnica de virtualización remota de GPUs. A este respecto, cuando no se utiliza esta técnica, migrar una aplicación CUDA a otro nodo del clúster implica que el nodo de destino tiene, en primer lugar, un número suficiente de núcleos disponibles para la aplicación que se está migrando. En segundo lugar, memoria principal suficiente para alojar los datos de la aplicación, y por último, suficiente memoria de GPU para almacenar los datos de la GPU de origen. Encontrar un nodo de destino en el clúster que satisfaga estos requisitos podría no ser complejo. Sin embargo, cuando se usa la técnica de virtualización remota de GPUs, podría seguirse otro enfoque con objeto de hacer el proceso de migración más eficiente. Este nuevo enfoque está basado en el hecho de que la técnica de virtualización remota de GPUs desacopla las GPUs de los nodos desde un punto de vista lógico y, por lo tanto, las partes de CPU y GPU de la aplicación pueden ser migradas de forma independiente a nodos distintos. De esta forma, no sería necesario que los tres requisitos descritos se satisficieran en un nodo concreto, si no que estos requisitos podrían dividirse en dos conjuntos: encontrar un nodo que satisfaga los primeros dos requisitos, y encontrar otro nodo que satisfaga el tercer requisito. El primer conjunto de requisitos hace referencia a la migración de la parte CPU de la aplicación, mientras que el segundo hace referencia a la parte GPU.

Este nuevo enfoque de migración haría más sencillo encontrar mejores nodos candidatos que cuando es necesario que los tres requisitos se satisfagan en el mismo nodo. Además, dado que la parte GPU se estaría ejecutando probablemente en un nodo distinto al de la parte CPU, sería incluso posible que solamente una de las dos partes necesitara ser migrada. En este contexto, el hecho de elegir si migrar la parte CPU o la parte GPU de una aplicación estaría basado en el estado actual del clúster y en las políticas de optimización. Nótese que si en lugar de buscar la consolidación de servidores se busca el

balanceo de carga, todo lo que se ha comentado sigue siendo útil con la única diferencia de que las políticas para decidir los nodos destino de las migraciones cambiaría.

El middleware rCUDA soporta la migración de la parte GPU de las aplicaciones CUDA. A este respecto, con rCUDA es posible seleccionar uno de los múltiples trabajos de GPU que se están ejecutando de forma concurrente en un acelerador dado y moverlo a otra GPU en el mismo nodo o en otro nodo del clúster. Este proceso es transparente a la aplicación que usa la GPU, que no tiene constancia alguna de la migración llevada a cabo.

2.5 Conclusiones

Como se ha podido ver en este capítulo, rCUDA es un middleware de virtualización remota de GPUs que proporciona múltiples beneficios en la utilización de aceleradores en centros de datos e incrementa enormemente la flexibilidad en el empleo de dichos aceleradores con respecto al enfoque ofrecido por CUDA. Sin embargo, estos beneficios proporcionados por rCUDA son el resultado de un proyecto de software de gran tamaño y de elevada complejidad que requiere del proceso de aseguramiento de calidad más estricto posible. Es por ello que se plantea la plataforma de verificación diseñada e implementada en este Trabajo Fin de Máster, cuyo cometido no es otro que verificar que todas las funcionalidades que provee rCUDA se comportan conforme dictan sus especificaciones de requisitos. Así pues, en el capítulo siguiente se proporcionan algunos detalles acerca de las pruebas de software en el mundo de la Ingeniería del Software para, posteriormente, detallar en el capítulo 4 cómo fue el proceso de desarrollo de la plataforma de verificación diseñada en este trabajo.

CAPÍTULO 3
Pruebas de software

Tras presentar en el capítulo anterior la tecnología rCUDA, objeto del programa de verificación desarrollado en este TFM, en este capítulo se introducen diversos conceptos relacionados con la verificación del software. También se presenta la librería Boost.Test, que será utilizada de forma intensiva durante el desarrollo del programa de verificación.

3.1 Introducción

Como se ha comentado en el capítulo 1 de esta memoria, los procesos de verificación deberían estar presentes en la totalidad de los desarrollos de software. Es más, cuando se habla de productos software de calidad, no existe software no verificado que llegue a manos de un potencial cliente. Así pues, es imprescindible tener en cuenta que la verificación del software no es un proceso opcional, sino un proceso totalmente necesario que permite a los desarrolladores garantizar que las aplicaciones que desarrollan cumplen las funcionalidades que se espera de ellas, ayudando de este modo a encontrar errores o defectos que aún no se han descubierto. Además, detectar los errores en fases tempranas del ciclo de desarrollo reduce el coste del mismo, ya que cuanto más tarde se detecte un error en el ciclo de desarrollo de un producto software, más cara resultará su resolución.

Así pues, las pruebas de software podrían definirse como la actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y se registran, y se realiza una evaluación de algún aspecto, ya sea corrección, robustez, eficiencia, etc. Teniendo en cuenta esta definición, podrían considerarse dos opciones acerca de la corrección del resultado del proceso de desarrollo en sí mismo:

- ¿Se está construyendo el producto correcto?
- ¿Se está construyendo correctamente el producto?

Ambas opciones, aunque parezcan similares, no son equivalentes. La primera hace referencia a la **validación** del producto, esto es, si el software hace lo que el usuario espera que haga. Esta opción, por lo tanto, es un proceso genérico que debe asegurar que el sistema satisface las espectativas del usuario, por lo que el usuario final se convierte en el foco principal del proceso. Es importante llevar a cabo la validación de los requisitos del sistema en las fases iniciales del ciclo de desarrollo. Es fácil cometer errores y omisiones durante la fase de análisis de requisitos del sistema y, en tales casos, el software final no cumpliría las espectativas de los clientes. No obstante, en la práctica, la validación de los requisitos no puede descubrir todos los problemas que presenta el producto final. Algunos defectos en los requisitos solo pueden ser descubiertos cuando la implementación del

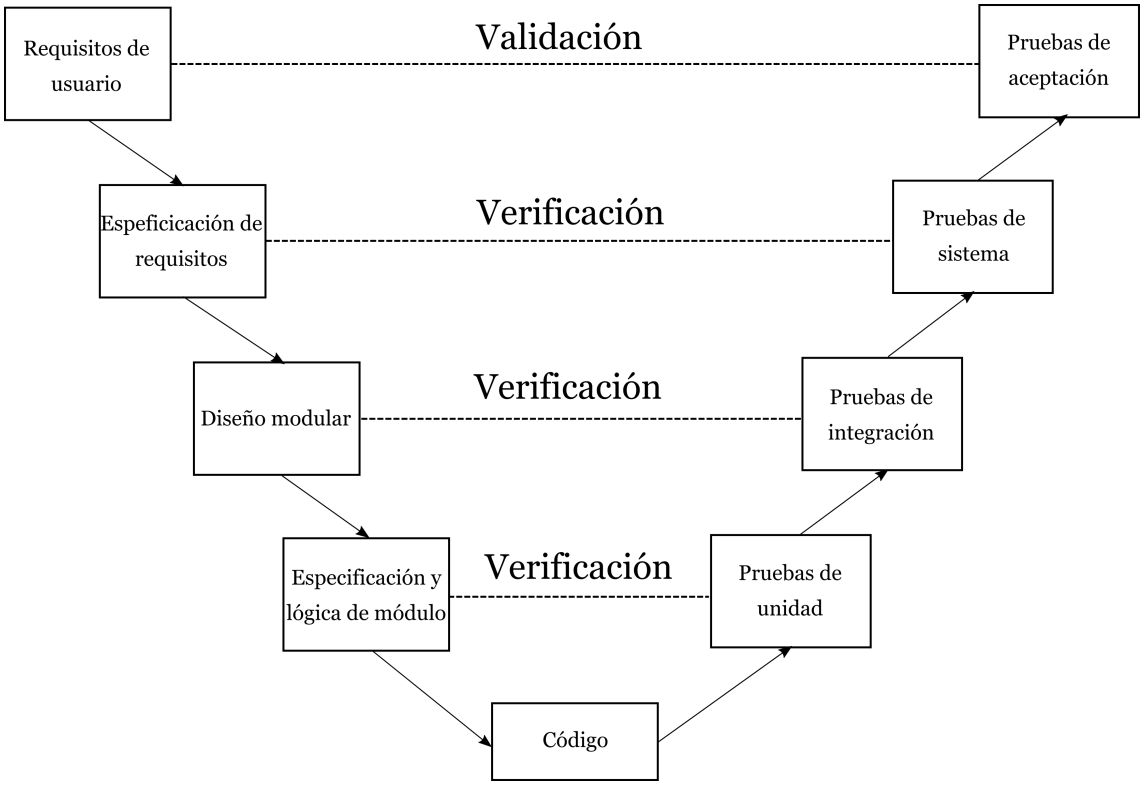


Figura 3.1: El proceso de prueba

sistema es completa. Es por ello que tiene lugar **la segunda opción**. Ésta hace referencia a la **verificación** del resultado del proyecto software. Esto es, si el software construido cumple con las especificaciones de requisitos funcionales y no funcionales que se han especificado en la fase de análisis. Esta opción es, por lo tanto, la que se va a tratar en este Trabajo Fin de Máster.

El proceso de prueba de un proyecto software consiste en una serie de fases bien definidas cuyo objetivo es realizar un tipo determinado de pruebas en un momento determinado del ciclo de desarrollo. Así pues, y según puede verse en la figura 3.1, el proceso de prueba dispone a su vez de distintos tipos de pruebas. Estas pruebas representadas en la figura 3.1 son las siguientes.

- **Pruebas de unidad o unitarias**: pruebas realizadas a cada módulo de forma individual con objeto de comprobar su funcionamiento autónomo (véase el capítulo 3.2 para más información)
- **Pruebas de integración**: el software totalmente ensamblado se prueba como un todo, con objeto de comprobar si cumple tanto los requisitos funcionales como los no funcionales.
- **Pruebas de sistema**: el software ya verificado se integra con el resto del sistema para probar su funcionamiento conjunto.
- **Pruebas de aceptación**: el producto final se comprueba por el usuario final en su propio entorno de explotación para determinar si lo acepta o no.

Además, pueden considerarse dos enfoques distintos con respecto al tipo de pruebas que pueden efectuarse en un momento dado:

- **Pruebas de caja blanca**: también llamadas pruebas estructurales, las cuales se basan en el estudio minucioso de toda la operatividad de una parte del sistema, considerando los detalles procedimentales.
- **Pruebas de caja negra**: también llamadas pruebas funcionales, las cuales analizan principalmente la compatibilidad en cuanto a las interfaces de cada uno de los componentes software entre sí. Estas pruebas se centran en el punto de vista del usuario final, considerando al producto software como una caja negra de la que se desconoce por completo su funcionamiento. Esto es, no es necesario conocer los detalles procedimentales, limitándose a interactuar con la aplicación mediante sus interfaces de entrada.

La aplicación desarrollada en este Trabajo Fin de Máster tiene como objetivo elaborar una plataforma de verificación que integre un conjunto de pruebas de unidad de caja negra en el que el framework rCUDA es probado durante todo el ciclo de su desarrollo. La integración de estas pruebas de unidad servirá para llevar a cabo el nivel de pruebas de integración descrito en la figura 3.1.

3.2 Pruebas de unidad

En programación, una prueba de unidad o unitaria es una comprobación del funcionamiento correcto de una unidad de código, siendo unidades de código, por ejemplo, las funciones o métodos en programación estructurada o funcional, y las clases en programación orientada a objetos. El objetivo de las pruebas de unidad es cerciorarse de que cada unidad funciona de forma correcta y eficiente sin tener en cuenta la vinculación con el resto de unidades del programa. Así, una vez se ha comprobado que cada unidad del código funciona individualmente conforme a su especificación, puede pasarse a implementar las pruebas de integración para asegurar el correcto funcionamiento del sistema en su conjunto.

Con objeto de diseñar pruebas de unidad de forma correcta, deben considerarse los siguientes aspectos:

- Las pruebas de unidad deben ser **automatizables**, de modo que puedan ejecutarse sin necesidad de intervención manual.
- Además, deben ser **repetibles**, dado que se busca ejecutar las pruebas tantas veces como se requiera. Por ello, la rapidez de las mismas es un factor clave y, por lo tanto, deben estar lo más optimizadas posible.
- Las pruebas de unidad deben **cubrir la mayor cantidad de código** posible, de esta forma será probado un mayor área del código de la aplicación que está siendo probada, que es finalmente lo que se pretende.
- Además, las pruebas de unidad deben poder ser ejecutadas de **forma independiente** del entorno, lo que implica que deben poder ejecutarse en cualquier máquina del equipo de desarrollo.
- También debe considerarse que la ejecución de una prueba de unidad no debe **afectar a la ejecución de otra** u otras pruebas de unidad, por lo que una prueba debería dejar el entorno en el mismo estado en el que se lo encontró.

Realizar pruebas de unidad en cualquier proyecto software proporciona una serie de innegables ventajas. Empezando por el equipo de desarrollo, la calidad del software que

éste produce se incrementará en la misma proporción que se reducirá el tiempo de depuración. Además, las pruebas de unidad fomentan el cambio de componentes de código que tienen un diseño deficiente, ya que las pruebas permiten al desarrollador cerciorarse de que su código modificado sigue cumpliendo con la especificación de requisitos. A todo esto, la ejecución de pruebas de unidad simplifica enormemente las pruebas de integración, puesto que al haber asegurado la calidad de cada módulo del programa de forma individual, cualquier fallo en la integración de los componentes debería deberse al proceso de integración en sí mismo o, en su defecto, a cualquier otro aspecto ajeno a la implementación de los componentes individuales del programa. Y por último, escribir pruebas de unidad, además de servir como documentación acerca del comportamiento esperado de cada módulo, le sirve al programador para probar o depurar un módulo sin necesidad de disponer del sistema al completo. Esto resulta especialmente útil en situaciones en las que montar todo el entorno en el que se ejecuta el programa en desarrollo es más costoso que corregir el problema en sí mismo.

3.3 Pruebas de integración

Tras haber llevado a cabo las pruebas de unidad, el siguiente paso natural es realizar las pruebas de integración, tal y como muestra la figura 3.1. Las pruebas de integración tienen como objetivo probar el software como un todo, de forma que sea posible comprobar que dicho software cumple con los requisitos funcionales y no funcionales establecidos en la fase de análisis de requisitos. Las pruebas de integración se centran, principalmente, en probar la comunicación entre los componentes a través de sus interfaces de entrada y salida. Con esta comprobación, es posible ver que el software desarrollado funciona de manera adecuada en conjunto, y que no se presentan problemas en la combinación de los distintos elementos unitarios.

Así pues, la herramienta de verificación desarrollada en este TFM actúa en los niveles de pruebas de unidad y de integración. Las pruebas de unidad están representadas por aquellos programas que el equipo de desarrollo de rCUDA implementa ad-hoc para probar una funcionalidad concreta y específica, como por ejemplo podrían ser las distintas funciones de la Runtime API del SDK[1] de CUDA. Las pruebas de integración, sin embargo, estarían representadas por aquellas aplicaciones que hacen un uso intensivo de todo el conjunto de funciones de la Runtime API y Driver API del SDK de CUDA.

3.4 Herramientas de verificación

Existen multitud de frameworks que facilitan la elaboración de pruebas de software, como por ejemplo podrían ser los frameworks CppUnit [38], NanoCppUnit [39], Unit++ [40], CxxTest [41] y Google Test [42].

- **CppUnit**: framework de pruebas de unidad para C y C++ que se encuentra entre los más empleados. Nació como un fork de JUnit [43] para Windows y fue portado posteriormente a Unix. Está licenciado bajo la licencia GNU GPL. Existe un fork de CppUnit usado en freedesktop.org que se utiliza para realizar las pruebas de verificación de Libreoffice y sistemas basados en Linux tales como Debian, Ubuntu, Gentoo y Arch Linux.

[1] *Software Development Kit* (Kit de desarrollo de software), es un conjunto de herramientas de software que permiten al programador crear un programa informático para un sistema determinado.

- **NanoCppUnit**: framework de pruebas unitarias también para C++ que solo funciona bajo entornos Windows. Su documentación es escasa y la implementación de los árboles de casos de test no es tan amigable con en otros casos, además de no seguir los estándares habituales.
- **Unit++**: framework también para C++ cuya sintaxis es extremadamente similar a la nativa del propio lenguaje al que está destinado. Por otra parte, este framework no posee macros que faciliten al programador la inclusión de suites de test[2] y casos de test, por lo que el procedimiento se limita a crear clases derivadas de otras clases que representan suites de test.
- **CxxTest**: framework para C++ con una gran bibliografía que utiliza Perl para implementar las macros que posteriormente se convertirán en las suites y casos de test. No requiere de demasiado código para conseguir funcionalidades y es portable a otras plataformas.
- **Google Test**: framework para C++ de reciente aparición, es portable y cuenta con versiones para Linux, Windows, Mac OS X y algunos sistemas operativos móviles. Tiene características avanzadas como casos de test de distintos tamaños específicamente diseñados para pruebas de unidad (small), integración (medium) y aceptación (large). Además de los proyectos diseñados por Google, otros proyectos como Chromium, Protobuf, OpenCV, Caffe y Gromacs también emplean este framework.

Además de los anteriores, existe otro framework llamado **Boost.Test** [13] que forma parte de la librería de C++ Boost [14]. Boost es un gran conjunto de librerías que se ha diseñado para dotar a C++ de una serie de funcionalidades de las que por sí mismo carece o están implementadas de formas distintas (a menudo, en un nivel de abstracción más bajo). Citando solamente algunas de estas librerías incluidas en Boost:

- Procesamiento de cadenas y texto
- Contenedores
- Iteradores
- Programación paralela
- Parámetros de programas
- Acceso a sistemas de ficheros
- Fecha/hora y control de tiempos
- Soporte a otros lenguajes de programación
- Entrada/salida
- Comunicación entre procesos
- Correción y prueba

Esta última librería, corrección y prueba, posee el nombre de Boost.Test y es el framework de verificación que se ha empleado en este Trabajo Fin de Máster. Las razones por las que se ha elegido este framework de entre todos los disponibles son múltiples:

[2]Como se verá más adelante, una suite de test es una agrupación lógica de un conjunto de casos de test cuya función es permitir al programador ejecutar ese conjunto de casos de test de una forma más simplificada.

- Posee una magnífica documentación a prueba de los menos experimentados en frameworks de verificación.
- Es ampliamente usado en multitud de proyectos de Ingeniería del Software a lo largo de todo el mundo.
- Su metodología de uso es muy fácil de comprender e implementar, ya que principalmente se basa en macros que facilitan enormemente al programador la tarea de desarrollar suites y casos de test. Además, el número de macros disponibles es muy elevado, proporcionando así al framework una elevada funcionalidad.
- Posee dos modalidades principales con respecto a la implementación de árboles de test: registro manual y automático. Este último simplifica enormemente la creación de árboles de casos de test con respecto al registro manual, encargándose de forma automática de los detalles de implementación que no están relacionados con la tarea de verificación en sí misma, y por lo tanto ofreciendo al programador la posibilidad de centrarse únicamente en la tarea de verificación.

En las siguientes secciones se proporciona información mucho más detallada acerca del funcionamiento de Boost.Test así como algunos ejemplos de uso.

3.5 Boost.Test

Boost.Test es la librería perteneciente al conjunto de librerías Boost que se encarga de llevar a cabo la verificación de programas escritos en C++. Esta librería proporciona una forma cómoda y flexible de escribir programas de prueba, organizar tests en casos de test y suites de test, y controlar la ejecución de los mismos.

Con objeto de usar Boost.Test, lo primero que haría un programador es proceder a la instalación de dicha librería. Para ello, podría seguir una de las siguientes aproximaciones:

- Descargar los paquetes asociados a través de los repositorios de su sistema operativo
- Descargar el código fuente y compilarlo para generar los ficheros binarios (principalmente ficheros de cabeceras y librerías, tanto estáticas como dinámicas)

En el caso de usar los paquetes a través de los repositorios de su sistema operativo, el programador podría proceder tal como sigue (para el caso de utilizar un sistema basado en Debian):

```
# apt-get install libboost-test-dev
```

Así pues, tras la ejecución del comando anterior en un sistema operativo Debian Stretch, se instalarían los ficheros de la librería de Boost.Test a través de los repositorios oficiales de dicho sistema[3].

En la segunda aproximación, el programador descargaría el directorio comprimido de Boost a través del enlace que proporciona su web, tras lo cual realizaría los siguientes

[3]En cualquier versión estable del sistema operativo Debian GNU/Linux, cualquier paquete alojado en sus repositorios permanecerá siempre en la misma versión, por lo que en Debian Stretch, que es el ejemplo usado en este caso, la versión de Boost alojada en sus repositorios será siempre la versión 1.62.

pasos para prodecer a su compilación e instalación (nótese que no existe opción de descargar únicamente la librería Boost.Test, si no que se requiere descargar todo el conjunto de librerías Boost):

```
$ wget -c https://dl.bintray.com/boostorg/release/1.67.0/source/
  boost_1_67_0.tar.bz2
$ tar xjf boost_1_67_0.tar.bz2
$ cd boost_1_67_0/
$ ./bootstrap.sh --prefix=/opt/boost/1.67/
$ ./b2 install -j $(nproc)
```

Tras la ejecución de la secuencia de comandos anteriores, el programador obtendría, al listar el directorio empleado como objetivo durante la instalación, los directorios `include` y `lib` que contendrían, respectivamente, los ficheros de cabeceras con extensión `.hpp` y los ficheros de librerías con extensiones `.a` (librerías estáticas) y `.so` (librerías dinámicas).

La elección de qué aproximación debe escogerse para instalar Boost.Test depende exclusivamente del programador, que deberá considerar las ventajas e inconvenientes de cada una de ellas. La primera aproximación permitirá la instalación de una forma cómoda de la librería individual que necesita y, además, esa instalación se realizará en los directorios de librerías del sistema operativo (y por lo tanto, el sistema operativo la encontrará de forma automática siempre que el programador la utilice). Sin embargo, la versión de la librería alojada en los repositorios del sistema operativo podría no satisfacer los requisitos del programador, ya que en el ejemplo anterior la versión de Boost disponible en los repositorios es la 1.62, mientras que la versión estable actual es la 1.67 (nótese que las versiones de las aplicaciones en sus páginas web oficiales suelen estar adelantadas con respecto a las versiones alojadas en los repositorios de los sistemas operativos).

Por otra parte, la segunda aproximación permite elegir qué versión de Boost utilizar, pero por contra debe descargarse, compilarse e instalarse de forma manual. Además, dado que se trata de una instalación manual, los ficheros de librerías y cabeceras resultantes no se alojarán en los directorios del sistema operativo destinados a tal efecto, lo que provocará que el sistema operativo no los encuentre de forma automática, recayendo esta tarea sobre el programador en todo momento en que quiera utilizar la librería[4].

Una vez instalado Boost.Test, el siguiente paso para el programador sería elegir, de entre los tres siguientes, el modo en que incluirá el framework en sus programas:

- Librería de encabezado único
- Librería de enlazado estático
- Librería de enlazado dinámico

El primer caso es válido cuando no se tiene acceso a las librerías compiladas de Boost.Test, esto es, cuando el usuario no ha instalado estas librerías mediante una de las dos formas mencionadas anteriormente, de tal forma que únicamente incluyendo el fichero de cabecera `unit_test.hpp` sería posible usar el framework, aunque únicamente en una sola "Translation Unit", esto es, un fichero de código fuente más los ficheros de cabeceras incluidos directa o indirectamente. Si el programador ha compilado y/o instalado la librería de Boost.Test de una de las dos formas mencionadas, como es el caso (o

[4]Otra posibilidad es que el programador tenga permisos de administración en la máquina donde ejecutará Boost y, por lo tanto, pueda copiar los ficheros de librerías a la ubicación por defecto, típicamente `/usr/lib`. De esta forma el sistema ya estará en disposición de encontrar la librería de forma automática.

tiene acceso a la misma a través de una fuente externa, como podría ser un sistema de ficheros compartido), entonces podría hacer uso de dicha librería y evadir esta limitación, por lo que podría escoger uno de los métodos restantes. Estos dos métodos restantes son similares, con la salvedad de que el primero emplea la versión estática de la librería (esto es, los ficheros con extensión `.a`) y el segundo emplea la versión dinámica (ficheros `.so`). La diferencia principal entre una versión y otra es que usando la versión estática, el programa incluirá la implementación de las funciones de las librerías que necesite en su propio código objeto, de tal forma que no necesitará buscar dicha implementación de forma externa en tiempo de ejecución. Por contra, la versión dinámica de la librería no incluirá ningún tipo de implementación, y buscará en tiempo de ejecución los ficheros de la librería dinámica con tal de acceder a la implementación de sus funciones. A pesar de ello, para proyectos software cuyo tamaño excede un cierto umbral, puede ser más interesante emplear la versión dinámica de las librerías que emplea, puesto que de esta manera no se verá incrementado el tamaño final del programa con la inclusión de dichas librerías.

Una vez escogido el modo en que incluirá Boost.Test en su programa, el programador debería plantearse la cuestión de qué tipo de registro de casos de test desea llevar a cabo en su programa: registro manual o automático. El primer caso permite al programador tener un control más exhaustivo de los casos de test que se ejecutan y en qué orden lo hacen, pero por contra obliga al programador a llevar a cabo todas las tareas de registro de dichos casos de test. La segunda opción, sin embargo, permite al programador abstraerse de los detalles de registro y centrarse en la tarea de verificación exclusivamente, pero por contra pierde cierto control sobre su ejecución.

Considérese el ejemplo de la figura de código fuente 3.1. En él se aprecia un fragmento de código en el que se registra de forma manual un caso de test que siempre se evalúa a cierto a través de la función `free_test_function()`:

```
1  #include <boost/test/included/unit_test.hpp>
2
3  void free_test_function()
4  {
5    BOOST_TEST(true);
6  }
7
8  test_suite* init_unit_test_suite(int argc, char* argv) {
9    boost::unit_test::framework::master_test_suite().add(
       ↪  BOOST_TEST_CASE(&free_test_function));
10
11   return 0;
12 }
```

Código fuente 3.1: Programa con caso de test registrado de forma manual

Tal y como se aprecia en la figura de código fuente 3.1, la línea 1 es una directiva del preprocesador que incluye el fichero de cabeceras `unit_tests.hpp` que a su vez incluye el resto de ficheros de cabeceras que necesita para poder acceder a toda la funcionalidad que proporciona el framework. Para hacer efectivo el registro manual mencionado se necesita, en primer lugar, la función de inicialización de módulos de test, esto es, la función `init_unit_test_suite()`, que hace las veces de función `main` del programa. Esta función se encargará de crear y registrar todos los casos de test que se encuentren declarados en su cuerpo. Para ello, los casos de test se registrarán a través de la función `add()` mostrada en la línea 9 de la figura de código fuente 3.1. En este caso, la función `add()` recibe un puntero a una instancia de la clase `boost::unit_test::test_case` que le es devuelto a través de la macro `BOOST_TEST_CASE`.

Considérese ahora el código de la figura de código 3.2. En dicha figura se aprecia un código que realiza la misma función que el código de la figura 3.1, pero empleando registro automático.

```
1 #define BOOST_TEST_MODULE example
2 #include <boost/test/included/unit_test.hpp>
3
4 BOOST_AUTO_TEST_CASE(free_test_function) {
5   BOOST_TEST(true);
6 }
```

Código fuente 3.2: Programa con caso de test registrado de forma automática

Como puede apreciarse, la simplicidad de esta segunda aproximación es muy superior a la de la primera, ya que la macro `BOOST_AUTO_TEST_CASE` registra de forma automática el caso de test sin necesidad de usar la función `init_unit_test_suite()`. Así pues, la elección de qué aproximación escoger depende exclusivamente de los intereses del programador, teniendo en cuenta que el registro automático probablemente sea válido en la mayoría de las situaciones.

Tras haber escogido el programador un tipo de registro (en el caso que ocupa a los siguientes ejemplos, se ha escogido el automático), el programador ya estaría en disposición de utilizar las funciones que proporciona el framework para escribir sus primeros árboles de test. Para ello, como primer programa y con objeto de familiarizarse con el framework, podría considerarse el representado en la figura de código fuente 3.3:

```
1 #define BOOST_TEST_MODULE My Test
2 #include <boost/test/included/unit_test.hpp>
3
4 BOOST_AUTO_TEST_CASE(first_test) {
5   int i = 1;
6   BOOST_TEST(i);
7   BOOST_TEST(i == 2);
8 }
```

Código fuente 3.3: Primer programa con Boost.Test

Como puede apreciarse en el código 3.3, la línea 4 contiene la macro `BOOST_AUTO_TEST_CASE` que declara un caso de test con registro automático llamado `first_test` que ejecutará todo lo contenido en su entorno de pruebas delimitado por sus llaves. En este caso en concreto, este caso de test ejecutará dos tests. El primero comprobará que la variable entera `i` no sea cero, y el segundo comprobará que la misma variable tenga el valor 2. Dado que el valor inicial de la variable es 1, el primer test se satisfará aunque no así el segundo. Si el programador compila y ejecuta el programa, obtendrá la siguiente salida:

```
Running 1 test case...
test_file.cpp(8): error: in "first_test": check i == 2 has failed [1 !=
  2]

*** 1 failure is detected in the test module "My Test"
```

En la salida, el framework notifica al usuario que se ha ejecutado un caso de test y que se ha producido un error en el módulo de test “My Test”. El motivo por el que el caso de test ha fallado es la comprobación de que la variable `i`, con valor 1, tiene un valor distinto al que se espera que tenga, esto es, 2.

El ejemplo de la figura de código 3.3 emplea un árbol de casos de test de una sola hoja, donde dicha hoja está representada por el caso de test `first_test`. Sin embargo, una de las características más interesantes de Boost.Test es la posibilidad de declarar árboles de casos de test más complejos, donde se le concede al programador la posibilidad de agrupar y anidar tantos casos de test como necesite. En el ejemplo de código 3.4 se muestra un nuevo programa donde se genera un árbol de casos de test más complejo en el que se emplea la macro `BOOST_AUTO_TEST_SUITE` para agrupar casos de test más sencillos.

```
1  #define BOOST_TEST_MODULE example
2  #include <boost/test/included/unit_test.hpp>
3
4  BOOST_AUTO_TEST_SUITE(test_suite1)
5
6  BOOST_AUTO_TEST_CASE(test_case1) {
7    BOOST_TEST_WARN(sizeof(int) < 4U);
8  }
9
10 BOOST_AUTO_TEST_CASE(test_case2) {
11   BOOST_TEST_REQUIRE(1 == 2);
12   BOOST_FAIL("Should never reach this line");
13 }
14
15 BOOST_AUTO_TEST_SUITE_END()
16
17 BOOST_AUTO_TEST_SUITE(test_suite2)
18
19 BOOST_AUTO_TEST_CASE(test_case3) {
20   BOOST_TEST(true);
21 }
22
23 BOOST_AUTO_TEST_CASE(test_case4) {
24   BOOST_TEST(false);
25 }
26
27 BOOST_AUTO_TEST_SUITE_END()
```

Código fuente 3.4: Ejemplo sencillo de agrupación de casos de test

En la figura de código 3.4 se generan dos suites de test (líneas 4 y 17) que contienen cada una dos casos de test. El caso de test `test_case1` emplea la macro `BOOST_TEST_WARN` que presenta un nivel de severidad leve, por lo que en caso de que no se satisfaga el caso de test, se mostrará una alerta pero no se incrementará el número de errores y el programa continuará. El caso de test `test_case2`, sin embargo, emplea la macro `BOOST_TEST_REQUIRE`, que presenta un nivel de severidad alto y, por lo tanto, en caso de que el caso de test falle, el número de casos de test fallidos se incrementará y el programa abortará su ejecución.

Nótese que todo árbol de casos de test siempre tiene como nodo raíz a una instancia de la clase `master_test_suite`, por lo que el framework siempre mantiene activa una instancia de esta clase. Todos los casos de test definidos en un programa están registrados, de forma directa o indirecta, como clases hijas de la clase `master_test_suite`.

Otra característica muy interesante y que le proporciona al programador mucha flexibilidad a la hora de ejecutar los casos de test son los denominados ***decorators***. Los *decorators* proporcionan un mecanismo para modificar ciertos atributos en tiempo de ejecución de las unidades de test registradas automáticamente. Entre otros, estos atributos permiten controlar la descripción de las unidades de test así como las etiquetas, timeout máximo, número de fallos esperados (por ejemplo, para unidades funcionales que no están completadas todavía), dependencias entre unidades de test, etc. Estos *decorators* se definen como segundo parámetro de las macros `BOOST_AUTO_TEST_CASE` y

BOOST_AUTO_TEST_SUITE, aunque es posible definirlos de forma explícita mediante una macro destinada a tal efecto. La figura de código 3.5 muestra un programa haciendo uso de los *decorators* label, description, timeout y depends_on.

```
#define BOOST_TEST_MODULE decorator_01
#include <boost/test/included/unit_test.hpp>
namespace utf = boost::unit_test;

BOOST_AUTO_TEST_CASE(test_case1, * utf::label("trivial") * utf::timeout(60)) {
  BOOST_TEST(true);
}

BOOST_AUTO_TEST_CASE(test_case2, * utf::label("trivial") * utf::label("cmp") *
   ↪  utf::description("testing equality of ones") * utf::depends_on(test_case1)) {
  BOOST_TEST(1 == 1);
}
```

Código fuente 3.5: Ejemplo de uso de *decorators*

El *decorator* label le proporciona al usuario mucha versatilidad al permiterle agrupar las unidades de test que comparten la misma etiqueta, lo que será muy útil en el momento de ejecutar dichas unidades. El *decorator* description asigna a una unidad de test una descripción que se mostrará cuando el usuario del framework liste las unidades de test disponibles en el programa. El *decorator* timeout establece un tiempo máximo que la unidad de test puede estar en ejecución. Si se excede dicho tiempo, la unidad de test fallará incrementándose el contador de errores y el programa continuará con la siguiente unidad de test. Y por último, el *decorator* depends_on es útil para establecer una relación de dependencia entre dos unidades de test. En el caso de la figura de código 3.5, se establece que el caso de test test_case2 depende del resultado satisfactorio del caso de test test_case1, por lo que si test_case1 falla, test_case2 no se ejecutará.

A todo esto, existe el *decorator* disabled que permite deshabilitar en tiempo de compilación una unidad de test, ya sea un caso de test o una suite de test (en cuyo caso, el estado deshabilitado será heredado por todos los casos de test que la contengan). Nótese que el estado por defecto de todas las unidades de test declaradas es enabled. Además, también se ofrece el *decorator* disabled_if, que permite deshabilitar un test en función de una condición.

Como ya se ha introducido en el figura de código 3.4, Boost.Test proporciona tres niveles distintos de severidad que determinará el número total de errores de un programa y el flujo de ejecución del mismo. La tabla 3.1 muestra un resumen de los tres niveles mencionados:

Nivel de severidad	Contador de errores del programa	Ejecución del programa
WARN	Sin efecto	Continúa
CHECK	Se incrementa	Continúa
REQUIRE	Se incrementa	Aborta

Tabla 3.1: Niveles de severidad disponibles

La tabla 3.1 muestra que el programador puede adaptar el nivel de severidad según sus necesidades, por lo que según la misma, las macros BOOST_WARN_EQUAL, BOOST_CHECK_EQUAL y BOOST_REQUIRE_EQUAL tendrán comportamientos distintos.

Toda esta flexibilidad que Boost.Test proporciona al programador para diseñar y codificar los casos de test también la proporciona cuando el usuario se dispone a ejecutar el

programa con objeto de que se ejecuten dichos casos de test. El framework de Boost.Test soporta múltiples parámetros que afectan a la ejecución del programa. Estos parámetros se pueden pasar por la línea de comandos o, en su defecto, mediante variables de entorno del sistema operativo, teniendo en cuenta que un parámetro pasado a través de la línea de comandos sobreescribirá a la variable de entorno correspondiente.

Uno de los parámetros más importantes que proporciona Boost.Test es el parámetro `run_test`. Este parámetro concede al usuario la posibilidad de ejecutar cualquier combinación y con cualquier granularidad de las unidades de test disponibles en el código del programa. Para ilustrarlo, considérese la figura de código 3.6, en el que se dispone de un árbol de unidades de test formado por diversos casos de test y suites de test.

```
#define BOOST_TEST_MODULE example
#include <boost/test/included/unit_test.hpp>

using boost::unit_test::label;

BOOST_AUTO_TEST_CASE(test_1, *label("L1")) {}
BOOST_AUTO_TEST_CASE(test_2, *label("L1")) {}

BOOST_AUTO_TEST_SUITE(suite_1)

  BOOST_AUTO_TEST_SUITE(suite_2)
    BOOST_AUTO_TEST_CASE(test_3) {}
    BOOST_AUTO_TEST_CASE(test_4) {}
  BOOST_AUTO_TEST_SUITE_END()

  BOOST_AUTO_TEST_SUITE(suite_3)
    BOOST_AUTO_TEST_CASE(test_5, *label("L2")) {}
    BOOST_AUTO_TEST_CASE(test_6, *label("L2")) {}
  BOOST_AUTO_TEST_SUITE_END()

  BOOST_AUTO_TEST_CASE(test_7, *label("L1")) {}
  BOOST_AUTO_TEST_CASE(test_8) {}
  BOOST_AUTO_TEST_CASE(test_9) {}

BOOST_AUTO_TEST_SUITE_END()
```

Código fuente 3.6: Ejemplo de uso del parámetro `run_test`

Si el usuario que ejecuta el programa omitiera el parámetro `run_test`, el programa se ejecutaría en su totalidad. Sin embargo, el usuario podría hacer uso de dicho parámetro para filtrar los casos de test que serán ejecutados finalmente, evitando así tener que ejecutar todo el árbol completo cuando podría no haber necesidad de hacerlo. Los siguientes ejemplos ilustran el uso del parámetro `run_test`:

- `run_test=test_1`: en este caso, se ejecutaría el caso de test de nombre `test_1` que se encuentra en el nivel más externo de la jerarquía del árbol de casos de test.
- `run_test=suite_1/suite_2/test_3`: se ejecutaría el caso de test de nombre `test_3` que se encuentra en la ruta del árbol de casos de test `suite_1/suite_2`.
- `run_test=suite_1/suite_3`: se ejecutaría la suite de casos de test con nombre `suite_3` que contendría, a su vez, los casos de test `test_5` y `test_6`.
- `run_test=suite_1/test_7,suite_2`: en este caso, se ejecutaría el caso de test `test_7` perteneciente a la suite `suite_1` y además se ejecutaría la suite de nombre `suite_2` al completo. En este caso, el carácter coma (“,”) sirve para ejecutar unidades de test hermanas, esto es, que comparten la misma raíz del árbol de casos de test.

- `run_test=suite_1/test_8:test_1`: si el propósito es ejecutar dos unidades de test que no comparten la raiz del árbol de casos de test, el usuario podría emplear el carácter dos puntos (":") para ejecutar, como en este caso, el caso de test `test_8` de la suite `suite_1` junto con el caso de test `test_1` que se encuentra en la raíz del árbol de casos de test.

- `run_test=@L1`: en este caso, el carácter arroba ("@") permite al usuario la ejecución de todos los casos y suites de test que tengan definida la etiqueta `L1`, que serían los casos de test `test_1`, `test_2` y `suite_1/test_7`.

- `run_test=suite_1/suite_2:!suite_1/suite_2/test_4`: el carácter cierre de exclamación ("!") concede la usuario la posibilidad de deshabilitar una unidad de test en tiempo de ejecución. En este caso, se ejecutaría la suite `suite_1/suite_2` al completo, a excepción del caso de test `suite_1/suite_2/test_4`, por lo que en la práctica se ejecutaría únicamente el caso de test `suite_1/suite_2/test_3`.

- `run_test=suite_1/suite_2:+suite_1/suite_2/test_4`: en este caso, y suponiendo que el caso de test `suite_1/suite_2/test_4` estuviera deshabilitado con el *decorator* `disabled`, el carácter más ("+") permitiría habilitarlo en tiempo de ejecución.

Otros parámetros útiles que proporciona el framework Boost.Test y que son ampliamente usados en la práctica son los siguientes (es posible consultar la información detallada de todos los parámetros en el apéndice A):

- `--help[=<nombre_parametro>]`: muestra la ayuda asociada al parámetro que se le pasa opcionalmente o, en su defecto, proporciona la ayuda general.

- `--list_content`: muestra por la salida estándar una lista de todas las suites y casos de test que están definidos en el programa, proporcionando para cada suite y caso de test su posición en el árbol de unidades de test. Puede verse un ejemplo simplificado de esta ejecución con el programa desarrollado en este TFM al final del apéndice A (la ejecución completa en este programa contiene más de 1.000 líneas).

- `--list_labels`: es el equivalente al parámetro anterior pero en este caso muestra una lista sencilla de las etiquitas definidas en el programa mediante el *decorator* `label`. Puede verse un ejemplo de esta ejecución al final del apéndice A.

- `--random`: ejecuta los casos de test en orden aleatorio, en lugar de como están definidos en el código fuente del programa.

- `--log_format[=<opción>]`: permite cambiar la forma en que la salida del programa (no sus resultados finales) es mostrada por la salida estándar. Las opciones que admite este parámetro son `HRF` (Human Readable Format, opción por defecto), `XML` y `JUnit`. También está disponible el parámetro `--report_format`, que es equivalente pero en lugar de para la salida del programa, para el informe final.

- `--log_level[=<opción>]`: este parámetro determina la cantidad de información que el framework proporcionará por la salida estándar. De este modo, se puede establecer que muestre desde ninguna información hasta toda la información posible. La opción por defecto es mostrar únicamente la información relativa a los errores. Su opción equivalente para el informe final sería `--report_level`.

- `--log_sink[=<opción>]`: permite al usuario establecer el lugar donde el framework imprimirá la salida del programa. Por defecto sería la salida estándar, pero también son válidas las opciones de salida de error (`stderr`) y un fichero. Al igual que las

dos anteriores, también tiene una versión para el informe final que se determina con el parámetro `--report_sink`.

CAPÍTULO 4
Desarrollo

Desde sus comienzos, el equipo de desarrollo del middleware de virtualización remota de GPUs rCUDA contemplaba en su hoja de ruta el desarrollo de una plataforma de verificación automática. Las razones para ello eran, entre otras, la posibilidad de realizar ejecuciones masivas de las aplicaciones a las que rCUDA da soporte y detectar los posibles errores. Otra razón era la heterogeneidad de las interfaces de usuario de dichas aplicaciones, ya que éstas tienen formas muy variadas de introducir la información que necesitan para funcionar y eso repercute en una mayor curva de aprendizaje y, por consiguiente, en un mayor tiempo de puesta en marcha. Además, también es frecuente modificar el código fuente de rCUDA para solucionar algún fallo descubierto y, sin tener conocimiento explícito de ello, producir otro fallo cuya detección no se produzca en un plazo de tiempo razonable. Por lo tanto, se necesitaba un mecanismo que permitiera verificar de forma automática el código fuente de rCUDA en un instante de tiempo determinado, de tal forma que pudiera comprobarse que los cambios introducidos no habían provocado la pérdida de alguna otra funcionalidad.

Es por todo esto que el equipo de desarrollo de rCUDA puso en marcha la implementación de una plataforma de verificación que pudiera satisfacer todos estos requisitos con objeto de facilitar y mejorar el desarrollo de rCUDA. Así pues, en este capítulo se detalla el proceso de desarrollo en cascada de la **primera versión** de dicha plataforma de verificación, desde la fase inicial de análisis de requisitos, hasta la fase de puesta en marcha. En capítulos posteriores se presentará cómo ha evolucionado esta herramienta de verificación gracias a la retroalimentación proporcionada por sus usuarios tras la puesta en marcha.

4.1 Análisis de requisitos

Mediante diversas reuniones, el equipo de desarrollo de rCUDA definió los requisitos funcionales y no funcionales que deberían ser considerados durante el desarrollo de la plataforma de verificación para rCUDA. Este análisis de requisitos sirvió para determinar las necesidades y las condiciones a satisfacer por el producto final tomando en cuenta a todas las partes interesadas.

Los **requisitos funcionales** de un sistema software determinan el comportamiento de dicho sistema, esto es, todas aquellas funcionalidades de las que se le dotará en la fase de implementación. En este caso, los requisitos funcionales fueron los que se detallan a continuación.

En primer lugar, el programa resultante debía ser capaz de ejecutar en masa y de forma automática todos aquellos casos de test que el usuario del programa necesitara ejecu-

tar en un momento determinado. Esto es, debía ser posible seleccionar bajo demanda un subconjunto de casos de test del conjunto total de los mismos.

En segundo lugar, el programa debía tener una interfaz de usuario sencilla y homogénea con la que ejecutar y controlar los parámetros más comunes de las aplicaciones a las que rCUDA da soporte. La razón de este requisito es simple: aunque la inmensa mayoría de aplicaciones implementan formas de que el usuario introduzca datos que la aplicación necesita para funcionar (por ejemplo, parámetros de entrada y/o ficheros de configuración), la forma de introducirlos rara vez coincide, y esto añade mucha complejidad en el momento en que el usuario desea ejecutar dichas aplicaciones, puesto que el usuario debe considerar tantas formas de introducir información como aplicaciones desea ejecutar. Así pues, la plataforma de verificación debía lidiar con este problema implementando una interfaz homogénea de tal forma que el usuario se abstrajera de los detalles de cada aplicación.

El siguiente requisito que se determinó fue el de proporcionar a los desarrolladores de rCUDA la capacidad de garantizar con un margen de error lo más pequeño posible que los cambios introducidos desde su último *commit*[1] no habían introducido algún error del que no tuvieran conocimiento. Esto es, el programa debía de proporcionar al usuario la capacidad de ejecutar un *sanity check* para comprobar que lo que antes funcionaba, seguía funcionando tras los cambios introducidos. Por lo tanto, este requisito también llevaba implícito otro en el que la aplicación debía ser capaz de probar distintas versiones de rCUDA (al menos, una por usuario). Cabe mencionar que el *sanity check* debía estar formado por un subconjunto representativo de casos de prueba con una duración total tolerable para un desarrollo de software dinámico.

También se definió el requisito de otorgar a los usuarios del programa la posibilidad de ejecutar los casos de test con tres duraciones distintas, de tal forma que en función de lo que se necesitara probar en un momento determinado y del tiempo disponible para ello, el usuario pudiera ejecutar el caso de test con una duración adecuada a ese contexto. Así pues, se definieron para cada caso de test tres duraciones distintas, *small*, *medium* y *large*. Nótese que, en la práctica, estas tres duraciones distintas a menudo significan tres conjuntos de datos distintos, con lo que una aplicación determinada se acaba probando de forma mucho más intensa.

Un requisito imprescindible que también quedó definido durante esta fase es que los casos de test debían estar agrupados en función de su naturaleza, con objeto de facilitar la ejecución de los casos de test que compartieran naturaleza y/o tiempo de ejecución. La naturaleza de los casos de test implicaba que debían agruparse en función del tipo de aplicación que se requería probar. Por ejemplo, las aplicaciones pertenecientes al género HPC debían estar agrupadas bajo el mismo nombre, y lo mismo ocurría con las aplicaciones del género Deep Learning, las aplicaciones del SDK de CUDA, etc.

Como último requisito funcional, también se determinó que la aplicación debía almacenar absolutamente toda la información que cada aplicación ejecutada proporcionara. Además, la plataforma debía mostrar al usuario de una forma inequívoca y directa qué casos de test habían resultado fallidos.

Por otra parte, los **requisitos no funcionales** de un sistema software hacen referencia a las características del funcionamiento del software tales como la usabilidad, seguridad, sobrecarga, escalabilidad, rendimiento, estabilidad, etc. Así pues, para el caso que ocupa a este trabajo, se definieron los requisitos no funcionales que se presentan a continuación

[1] El equipo de desarrollo de rCUDA emplea el sistema de control de versiones Git, donde un *commit* implica confirmar los cambios realizados y almacenar una instantánea del estado actual del proyecto.

En primer lugar el programa resultante debía ofrecer un rendimiento similar al ofrecido por las aplicaciones nativas, esto es, la sobrecarga que llevara implícita el programa debía de ser despreciable.

Por otra parte, también se determinó que la plataforma de pruebas debía ofrecer al usuario una interfaz cómoda que asumiera determinados valores por defecto, de tal forma que el usuario pudiera ejecutar la aplicación considerando el menor número de parámetros posibles. Sin embargo, también se determinó que debía existir la posibilidad de que el usuario cambiara en tiempo de ejecución ese comportamiento por defecto.

Además, se estableció que el programa debía permanecer en ejecución al menos tanto tiempo como las apliciones que ejecutaba. Es decir, si una aplicación determinada estaba en ejecución durante seis horas, el programa que lanzaba dicha aplicación debía permanecer operativo durante todo ese tiempo. El motivo no era otro que si el programa de verificación acababa (de forma ordenada o no) antes que la aplicación que lanzaba, no sería posible verificar el estado de ejecución de dicha aplicación, perdiéndose de este modo todo el sentido en sí mismo de la plataforma de verificación.

4.2 Diseño de la plataforma de verificación

En esta fase se definió, principalmente, la arquitectura de la aplicación. Se determinó que la aplicación debía contar con dos capas de software con objeto de separar el motor de verificación de la interfaz de usuario. Así pues, la capa externa, a la que se le llamó `autotests`, sería el programa que incluiría toda la gestión de entrada de parámetros así como la salida y almacenamiento de la información, y la capa interna, llamada `unit_tests`, sería el programa que incluiría el árbol de unidades de test al completo.

Por otra parte, y con objeto de satisfacer el requisito de agrupar los casos de test en función de su naturaleza y duración, se diseñó la estructura representada en la figura 4.1 para cada una de las diferentes tipologías de las aplicaciones (inicialmente, se definieron tres tipologías que serían HPC, Deep Learning y SDK de CUDA).

En esta arquitectura, el nodo raíz del árbol consiste en una suite de casos de test de Boost.Test que agrupa a todas las unidades de test (ya sean suites o casos de test) de un tipo específico de aplicación. Así, se definieron en un primer momento los nodos raíz `HPC` (*High Performance Computing)*, `DL` (*Deep Learning*) y `cuda_samples`[2]. Dentro de cada nodo raíz se definieron, tal y como muestra la figura 4.1, tres suites para los tamaños *small*, *medium* y *large*. A su vez, dentro de cada una de estas suites se generaron tantas suites como aplicaciones se integrarían dentro del programa. Por ejemplo, la suite `HPC/HPC_small` contendría las suites `Gromacs`, `Barracuda`, `Namd`, etc. Y por último, las hojas del árbol de unidades de test las formarían los casos de test pertenecientes a cada aplicación, esto es, lo que en última instancia se ejecutaría cuando el usuario utilizara el programa.

Esta arquitectura permite ejecutar casos de test con cualquier granularidad que se necesite, ya que si el usuario desea ejecutar un conjunto de casos de test agrupados por clase o por tamaño, podría disponer de los nodos de la parte alta de cada rama del árbol jerárquico. Si por el contrario, necesitara ser más preciso con respecto a los casos de test a ejecutar, podría usar las hojas del árbol. Y si aún así estas facilidades no le resultaran suficientes y necesitara, por ejemplo, combinar la ejecución de suites de casos de test con casos de test individuales, podría emplear las características que el framework Boost.Test

[2]Programas de pequeño tamaño que provee el SDK de CUDA junto con su código fuente y que, entre otras cosas, sirven para iniciarse en la programación de GPUs con CUDA puesto que hacen uso de múltiples funciones de la API de CUDA.

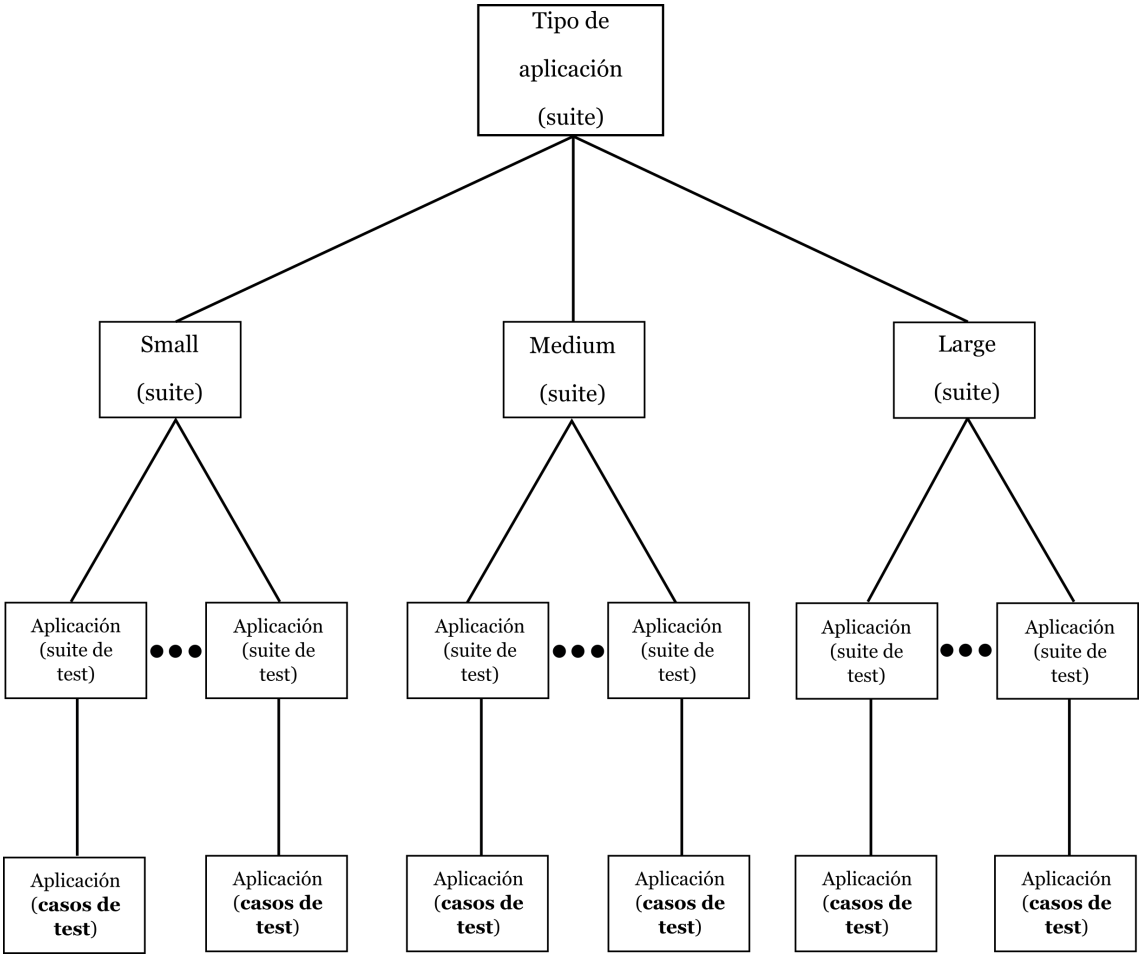


Figura 4.1: Arquitectura de las unidades de test

proporciona para ello, y que han sido descritas cuando se ha introducido el parámetro `run_test`.

Sin embargo, a pesar de que esta arquitectura proporciona mucha flexibilidad para ejecutar cualquier combinación de casos de test, presenta un pequeño hándicap: si un usuario necesitara ejecutar todos los casos de test pertenecientes a una aplicación, necesitaría usar el parámetro `run_test` con una secuencia relativamente larga de rutas en el árbol jerárquico, y esto es debido a que las aplicaciones están segmentadas en función del tamaño de los casos de test. Para abordar este problema, se decidió que todos los casos de test relativos a una aplicación también se agruparían desde una perspectiva lógica. Generar nuevas rutas y nodos en el árbol de unidades de test quedó descartado inmediatamente, pues implicaría repetir casos de test en el código fuente. Así pues, se determinó que la forma más apropiada de resolver el problema era empleando el *decorator* **label**, de tal forma que sería posible agrupar todos las casos de test de una aplicación desde un punto de vista lógico, lo que facilitaría enormemente al usuario el poder ejecutar todos los casos de test asociados a esa aplicación.

Con respecto al almacenamiento de la información proporcionada por cada caso de test ejecutado, se determinó que cada ejecución debía almacenar en una ubicación específica del sistema de ficheros compartido de que dispone el clúster de trabajo toda información que cada una de las aplicaciones ejecutadas a través de dicho programa proporcionara. Y además de esta información, se determinó que era necesario que junto con dicha información, el programa almacenara cierta información ajena a la aplica-

ción ejecutada pero igualmente relevante para los desarrolladores de rCUDA, como por ejemplo podrían ser las variables de entorno utilizadas (`LD_LIBRARY_PATH`, `PYTHONPATH`, `RCUDA_NETWORK`, `RCUDA_DEVICE_COUNT`, etc.), el tiempo consumido por el caso de test en cuestión, y el nodo del clúster desde el que se realizaba la ejecución.

Además, se requería que el usuario pudiera acceder a toda esta información almacenada en un momento distinto al empleado para ejecutar el programa de verificación. Esto, entre otras cosas, podría deberse a que la ejecución podría ser prolongada en el tiempo y el usuario necesitaría acceder a los ficheros de registro con un cierto tiempo de demora. Con objeto de abordar este tema, se decidió que la capa externa del programa generaría un *timestamp* compuesto por la fecha y la hora de ejecución del test y, a partir de este *timestamp*, se generaría una cadena que le daría nombre a los directorios que almacenarían la información proporcionada por la plataforma de verificación. Además, los directorios de ejecuciones previas debían ser movidos a otro directorio para dejar el directorio de la ejecución actual limpio, de tal forma que acceder a él fuera instantáneo.

Por último, y con objeto de facilitar la programación de la aplicación y mantener la coherencia en el sistema de ficheros compartido, se determinó que los ficheros asociados a cada aplicación integrada en la plataforma de verificación (ya fueran binarios, librerías o ficheros asociados a la ejecución de *samples*[3]) deberían de estar convenientemente agrupados en una ubicación determinada dentro de dicho sistema de ficheros.

4.3 Implementación

Una vez culminó la fase de diseño del programa que ocupa a este trabajo, se procedió con la fase de implementación. En primer lugar, se consideró como primera tarea a abordar la creación de la estructura de directorios que almacenaría los ficheros asociados a todas las aplicaciones que posteriormente se integrarían en el programa. A partir de esta estructura, el código fuente del programa referenciaría a estas ubicaciones para ejecutar cada aplicación, esto es, realizaría llamadas a aplicaciones compiladas e instaladas en el sistema de ficheros compartido. Para el caso de binarios y librerías de las aplicaciones, se decidió que la forma de proceder más interesante desde el punto de vista práctico era la creación de enlaces simbólicos a la ubicación original dentro del sistema de ficheros compartido, mientras que los ficheros asociados a los *samples* serían simplemente almacenados en bruto en esta nueva estructura de directorios. Así pues, inicialmente se comenzó generando los enlaces simbólicos y los ficheros de los *samples* asociados a los programas que vienen integrados en el Software Development Kit de CUDA.

Sin embargo, la tarea de definir la estructura de directorios requería que las aplicaciones a las que se pretendía dar soporte estuvieran adecuadamente instaladas y compiladas para la versión actual de CUDA, en ese momento la versión 8.0. Así pues, fue necesario emplear una gran parte del tiempo en estudiar, compilar e instalar cada una de las aplicaciones que finalmente se integraron en el motor de verificación. La figura 4.2 muestra como ejemplo la estructura de directorios de la aplicación del `HPC CUDASW++` tras haber realizado su compilación, instalación y organización para que el programa de verificación tenga acceso a todos sus ficheros.

[3]Nombre que se le da a ciertos programas de ejemplo de algunas aplicaciones con objeto de probar su funcionalidad.

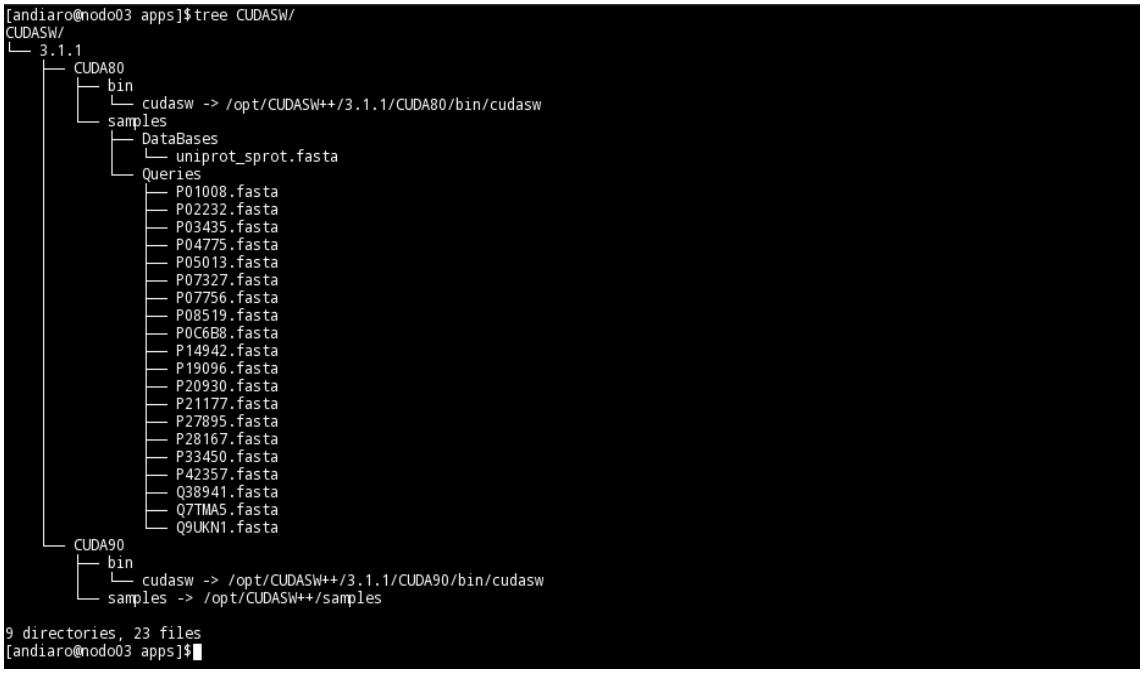


Figura 4.2: Organización de los ficheros y directorios de la aplicación de HPC CUDASW++

4.3.1. Implementación de `unit_tests`

Una vez generados los enlaces simbólicos y copiados el resto de ficheros necesarios para hacerlos funcionar, se comenzó con los cimientos del programa de verificación automática `unit_tests` (es decir, la capa interna). Para ello, se generaron ciertos ficheros de cabeceras con extensión `.hpp` conteniendo la declaración de todas aquellas variables necesarias para referenciar a las aplicaciones que se ejecutarían en la plataforma. Posteriormente, se crearon otros ficheros de código con extensión `.cpp` que contendrían, a su vez, la definición de todas las variables declaradas en sus homólogos de extensión `.hpp`. En particular, se definieron los siguientes ficheros de cabeceras.

- `unit_tests_functions.hpp`: fichero de cabeceras que incluye los prototipos de multitud de funciones personalizadas que le otorgan al programa toda su funcionalidad
- `unit_tests_constants.hpp`: fichero de cabeceras que incluye la declaración de todas las variables necesarias para que el programa pueda acceder al sistema de ficheros compartido y localizar correctamente las aplicaciones allí alojadas
- `unit_tests_commands.hpp`: fichero de cabeceras que incluye la declaración de todas las variables necesarias para que las aplicaciones puedan ser ejecutadas de forma automática desde una shell

Posteriomente, se generaron los ficheros de código fuente que contendrían las suites y casos de test que definirían la estructura jerárquica de unidades de test del motor de verificación. La figura de código 4.1 representa la suite de test `GROMACS_small` para la aplicación `GROMACS` ubicada en la ruta `HPC/HPC_small` tal y como fue implementada en la primera versión del programa.

```
1  #define BOOST_TEST_DYN_LINK
2  #include <boost/test/unit_test.hpp>
3  #include <iostream>
4  #include <string>
5  #include <stdio.h>
6  #include "unit_tests_functions.hpp"
7  #include "unit_tests_constants.hpp"
8  #include "unit_tests_commands.hpp"
9
10 using namespace std;
11 using namespace boost::unit_test;
12 using namespace unit_tests_constants;
13 using namespace unit_tests_commands;
14
15 BOOST_AUTO_TEST_SUITE(HPC);
16
17 BOOST_AUTO_TEST_SUITE(HPC_small);
18
19 BOOST_AUTO_TEST_SUITE(GROMACS_small, * label("GROMACS"));
20
21 BOOST_AUTO_TEST_CASE(GROMACS_gpu_sample_small) {
22     std::string final_complete_cmd_gromacs_5_1_4_cuda80_gpu_small =
        ↪  appendArgsToGromacsCommand( complete_cmd_gromacs_5_1_4_cuda80_gpu_small);
23     final_complete_cmd_gromacs_5_1_4_cuda80_gpu_small = appendForwardingToCommand
        ↪  (final_complete_cmd_gromacs_5_1_4_cuda80_gpu_small,
        ↪  boost::unit_test::framework::current_test_case().p_name);
24     tic();
25     BOOST_CHECK_EQUAL(system( final_complete_cmd_gromacs_5_1_4_cuda80_gpu_small.c_str()),
        ↪  success_value);
26     toc(boost::unit_test::framework::current_test_case().p_name);
27     appendExecutionInfo( boost::unit_test::framework::current_test_case().p_name);
28 }
29
30 BOOST_AUTO_TEST_CASE(GROMACS_gpu_nvml_sample_small) {
31     std::string final_complete_cmd_gromacs_5_1_4_cuda80_gpu_nvml_small =
        ↪  appendArgsToGromacsCommand( complete_cmd_gromacs_5_1_4_cuda80_gpu_nvml_small);
32     final_complete_cmd_gromacs_5_1_4_cuda80_gpu_nvml_small = appendForwardingToCommand(
        ↪  final_complete_cmd_gromacs_5_1_4_cuda80_gpu_nvml_small,
        ↪  boost::unit_test::framework::current_test_case().p_name);
33     tic();
34     BOOST_CHECK_EQUAL(system(
        ↪  final_complete_cmd_gromacs_5_1_4_cuda80_gpu_nvml_small.c_str()),
        ↪  success_value);
35     toc(boost::unit_test::framework::current_test_case().p_name);
36     appendExecutionInfo( boost::unit_test::framework::current_test_case().p_name);
37 }
38
39 BOOST_AUTO_TEST_SUITE_END();
40
41 BOOST_AUTO_TEST_SUITE_END();
42
43 BOOST_AUTO_TEST_SUITE_END();
```

Código fuente 4.1: suite `GROMACS_small` en la primera versión del programa

Lo que se muestra en la figura 4.1 consiste en el contenido del fichero de código fuente del programa `unit_tests` llamado `hpc_gromacs_small.cpp`, que contiene dos casos de test para la aplicación de HPC GROMACS. En el código mostrado se visualiza que, en primer lugar, se incluyen las directivas del preprocesador necesarias para que el programa funcione como se espera. Por ejemplo, la línea 1 tiene como objetivo forzar al framework Boost.Test a que sus librerías se carguen de forma dinámica, esto es, en tiempo de ejecu-

ción. Las líneas 6, 7 y 8 incluyen los ficheros de cabeceras definidos específicamente, tal y como se ha comentado previamente.

Como puede verse en las líneas 15, 17 y 19, se definen tres suites de test anidadas donde la suite `HPC` es la más externa y contiene a todas las demás. Dentro de la suite más interna, `GROMACS_small`, se definen los casos de test propiamente dichos. En este caso, se definen dos casos de test con los nombres `GROMACS_gpu_sample_small` y `GROMACS_gpu_nvml_sample_small`.

Desde las fases más tempranas del desarrollo se consideró que sería conveniente organizar los ficheros del código fuente de tal forma que éste fuera mantenible a lo largo del tiempo, por ejemplo en los casos en los que se tuviera acceso a nuevos casos de test y hubiera que añadirlos a los ya existentes en la plataforma de verificación. Por ello, se decidió generar un fichero de código fuente con extensión `.cpp` por cada tipo, aplicación y tamaño de caso de test. En el caso del ejemplo de la figura de código 4.1, se muestra el fichero `hpc_gromacs_small.cpp`, teniendo en cuenta que para la misma aplicación también se generaron los ficheros `hpc_gromacs_medium.cpp` y `hpc_gromacs_large.cpp`. Además, debe considerarse también que el framework de verificación Boost.Test permite sin ningún tipo de inconveniente que una misma suite de casos de test esté definida en más de un fichero de código fuente, por lo que esta característica resultó muy útil al proporcionar la posibilidad de agrupar diversos casos de test en distintos ficheros de código fuente bajo una misma suite de casos de test.

En esta primera versión del programa, los casos de test comparten, salvo algunas excepciones, la misma estructura. En primer lugar, se declara una nueva variable de tipo `std::string` que recibe un string del estándar de C++ devuelto por la función creada ad-hoc `appendArgsToGromacsCommand()`, y que se encarga de procesar los parámetros que el usuario introduce a través de la capa externa del programa, esto es, el programa `autotests`. Posteriormente, ese string se le pasa por valor a la función, también creada ad-hoc, `appendForwardingToCommand()` junto con el nombre del caso de test, que se obtiene de forma dinámica a través del atributo `boost::unit_test::framework::current_test_case().p_name`. Esta función se encargará de añadir al comando que posteriormente ejecutará la aplicación los datos relativos a la redirección de la salida estándar y de error, con objeto de almacenar toda la información que dicha aplicación procure en los directorios apropiados del sistema de ficheros compartido.

Seguidamente, y tras realizar una llamada a la función personalizada `tic()`, que se encarga de inicializar un cronómetro, se ejecutará el test propiamente dicho de este caso de test. Este test consiste en ejecutar la aplicación a través de la función POSIX `system`, pasándole como parámetro la variable que contiene el comando asociado a la aplicación que se desea ejecutar. La función `system` utiliza internamente las llamadas al sistema `fork` y `exec` para crear un proceso hijo que ejecuta el comando especificado a través de su único parámetro. En esencia, la llamada `system` facilita al programador la ejecución de comandos en la shell siempre y cuando el programador esté dispuesto a renunciar a cierto control sobre los procesos que se crean mediante esa llamada (en la sección 5.1.7 de este trabajo se verá que esta pérdida de control podría no ser aceptable en muchos casos).

Tal y como proclama el estándar POSIX con respecto a los valores de retorno de los comandos ejecutados en una shell, un programa ejecutado devolverá el valor 0 si la ejecución ha sido correcta. En cualquier otro caso, devolverá un valor distinto de cero. Así pues, el programa `unit_tests` aprovecha esta característica para identificar si una aplicación lanzada a través de la plataforma de verificación se ha ejecutado correctamente. Es por ello que la llamada a la función `system` se enmarca dentro de la macro de Boost.Test `BOOST_CHECK_EQUAL(param1, param2)`, que compara el valor del primer parámetro con el valor del segundo, y devuelve falso si ambos valores no coinciden. En este caso, se

compara el valor que devuelve la llamada a `system` con el valor almacenado en la variable entera `success_value`, que para el caso, contiene un cero.

Cuando la aplicación ha finalizado y devuelve el control de nuevo al programa, se realiza una llamada a la función `toc()` que detiene el cronómetro iniciado en la función `tic()` e imprime el tiempo consumido por la salida estándar. Por último, se realiza una llamada a la función creada ad-hoc `appendExecutionInfo()` que añade toda la información relativa a la ejecución a los ficheros de registro resultantes de dicha ejecución, como por ejemplo las variables de entorno, el tiempo consumido, el nombre del nodo cliente rCUDA, etc. Para finalizar, se ejecutan las macros que permiten dar por finalizada la definición de una suite de test, esto es, la macro `BOOST_AUTO_TEST_SUITE_END()`. Puede apreciarse también en la línea 19 que, junto con la definición de la suite `GROMACS_small`, aparece el *decorator* `label` definidiendo una etiqueta llamada `GROMACS`. Como se ha comentado anteriormente, esto resuelve el problema de la necesidad de duplicar código en los casos en que el usuario pretende ejecutar todos los casos de test del programa pertenecientes a una sola aplicación.

Así pues, el desarrollo de la primera versión de `unit_tests` finalizó cuando se implementaron las unidades de test para un conjunto de aplicaciones y librerías determinado, que son las que se listan a continuación: `Cuda Samples`, `BarraCUDA` [44], `Cuda-Meme` [45], `Cuda-SW++` [46], `GPU-Blast` [47], `Gromacs` [48], `Lammps` [49], `Magma` [50] y `Namd` [51]. La figura 4.3 muestra la arquitectura real que se implementó en el motor de verificación `unit_test` para las aplicaciones de `HPC Gromacs`, `Lammps` y `Namd`. La inclusión de aplicaciones de Deep Learning no se efectuó en un primer momento puesto que rCUDA se encontraba en pleno desarrollo de las funcionalidades que le darían soporte a este tipo de aplicaciones, mientras que el sorpote para los *samples* de CUDA y las aplicaciones `HPC` estaba completamente terminado.

En definitiva, el proceso de implementación de `unit_tests` resultó ciertamente laborioso, en primer lugar por la curva de aprendizaje que implicó el framework de Boost.Test, teniendo en cuenta que no se contaba con experiencia alguna en ningún otro framework de verificación (si bien, tal y como se ha comentado previamente, la magnífica documentación y la ingente cantidad de macros facilitó enormemente la tarea). En segundo lugar, la definición de los casos de test resultó en algunas ocasiones una tarea repetitiva, puesto que algunas aplicaciones cuentan con cientos de casos de test, como es el caso de `Magma` y `GPU-Blast`. Para estos casos, se contó con la ayuda de editores de código compatibles con expresiones regulares (que ayudaron enormemente a procesar texto repetido) y scripts creados ad-hoc cuyo cometido fue la generación de código con estructura similar.

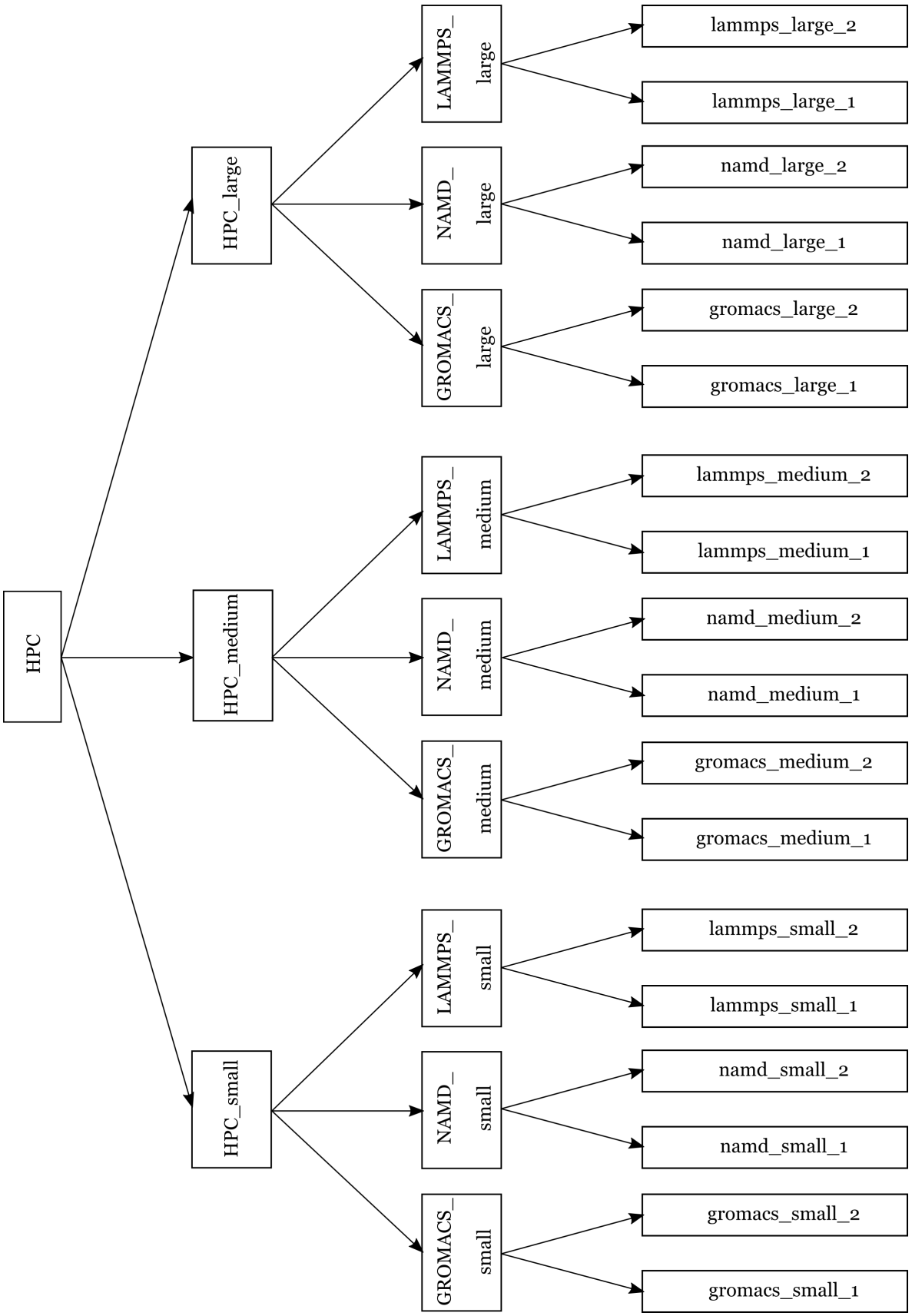


Figura 4.3: Arquitectura de las unidades de test para las aplicaciones `HPC Gromacs`, `Lammps` y `Namd`.

4.3.2. Implementación de `autotests`

Con respecto al programa `autotests`, la capa más externa de la plataforma de verificación, era necesaria, para empezar, una correcta gestión de los parámetros de entrada, pues entre otros, el objetivo de este programa es ser la interfaz entre el usuario y el programa `unit_tests`. En una versión inicial del programa, esta tarea se llevó a cabo empleando la gestión nativa de parámetros del lenguaje C. No obstante, rápidamente se pasó a emplear la librería de Boost Program.Options [54] dada su versatilidad y alto nivel de abstracción (nótese que esta librería no tiene relación alguna con Boost.Test más allá de que forman parte del conjunto de librerías para C++ Boost). La figura de código 4.2 muestra la gestión de parámetros una vez finalizada la implementación de la primera versión del programa.

```
1  boost::program_options::options_description description("Allowed options");
2  description.add_options()
3    ("help", "Show help message.")
4    ("list_content", "Lists the tests units, their organization in the test tree, their
       ↪  enabled/disabled state, etc.")
5    ("list_labels", "Lists the labels defined in the application.")
6    ("build_info", "Instructs the framework to display library build information prior to
       ↪  show execution information.")
7    ("rcuda_lib", boost::program_options::value<std::string>(), "Path to rCUDA lib path.
       ↪  Mandatory argument.")
8    ("run_test", boost::program_options::value<std::string>(), "Tests to run with
       ↪  UNIT_TESTS_rCUDA.")
9    ("log_level", boost::program_options::value<std::string>(), "Log level. Default is
       ↪  error. Others are all, test_suite, message, warning, cpp_exception,
       ↪  system_error, fatal_error and nothing.")
10   ("random", "Run the tests cases in random order.")
11   ("log_sink", boost::program_options::value<std::string>(), "Name of the file which will
       ↪  be used to store the log.")
12   ("report_sink", boost::program_options::value<std::string>(), "Name of the file which
       ↪  will be used to store the report.")
13   ("gpu_num", boost::program_options::value<int>(), "Number of the GPU to use in
       ↪  CUDA-MEME.")
14   ("num_threads_omp", boost::program_options::value<int>(), "Number of OpenMP threads to
       ↪  use in CUDA-MEME, GPU-BLAST and GROMACS.")
15   ("num_gpus", boost::program_options::value<int>(), "Number of GPUs to use in CUDASW and
       ↪  MAGMA.")
16   ("ntmpi", boost::program_options::value<int>(), "Number of threads MPI for GROMACS.")
17   ("nsteps", boost::program_options::value<int>(), "Number of steps to run for GROMACS.")
18   ("num_procs", boost::program_options::value<int>(), "Number of MPI processes for LAMMPS
       ↪  and AMBER.")
19   ;
20
21 boost::program_options::variables_map vm;
22 boost::program_options::store(boost::program_options::parse_command_line( argc, argv,
     ↪  description), vm);
23 boost::program_options::notify(vm);
```

Código fuente 4.2: Gestión de parámetros mediante Boost Program.Options

Como puede apreciarse en la figura de código 4.2, algunos parámetros son comunes para diversas aplicaciones, como es el caso del parámetro `num_threads_omp`, que es aplicable a las aplicaciones `Cuda-Meme`, `GPU-Blast` y `Gromacs`. No obstante, algunas otras aplicaciones emplean parámetros cuyo comportamiento es ligeramente distinto a pesar de que pretenden controlar la misma característica de la aplicación, por lo que en estos casos se prefirió definir una gestión aparte a través de un parámetro de distinto nombre (generalmente el nombre original). Es el caso, por ejemplo, de la aplicación Gromacs. Esta aplicación ofrece dos parámetros para gestionar el número de hilos de CPU en lugar de

uno, como así hacen las aplicaciones `Cuda-Meme` y `GPU-Blast`. En Gromacs, el parámetro `ntmpi` define el número de hilos de CPU usando la tecnología `thread-MPI`, empleada en Gromacs para ejecución en memoria compartida, y el parámetro `ntomp` define el número de hilos de CPU empleando la tecnología OpenMP. Así pues, en este caso se decidió aunar el parámetro `ntomp` de Gromacs sobre el parámetro de `autotests` `num_threads_omp`, y proporcionar de forma independiente el parámetro `ntmpi`.

Para continuar, el programa `autotests` obtiene un *timestamp* a través de una función definida para tal cometido y lo emplea para sincronizarse con el programa que pretende ejecutar, `unit_tests`. Esto es, el programa `autotests` genera un `timestamp` y lo comparte con `unit_test`, de tal forma que esa cadena es la que, en combinación con otras tales como el nombre de usuario y el nodo utilizado como cliente de rCUDA, le sirve a ambos programas para generar y obtener los ficheros de registro necesarios para depurar rCUDA. Además de esta, también era necesario compartir más información entre ambos procesos, por lo que se estudiaron los distintos métodos que podrían emplearse para ello. Finalmente, aprovechando el mecanismo de herencia que proveen los sistemas Unix, se decidió implementar esta comunicación en sentido descendente a través de variables de entorno del sistema operativo, puesto que una variable de entorno definida en un proceso es heredada por los procesos hijos de este proceso. La figura de código 4.3 muestra la implementación de la función encargada de calcular el *timestamp* a través de la librería Date_Time [55] de Boost. Esta función devuelve un string del estándar de C++ llamado `timestamp` que, posteriormente, es compartido entre ambos procesos a través de las funciones de C `setenv()` y `getenv()`, que sirven , respectivamente, para establecer y obtener una variable de entorno.

```
1  std::string getTimestamp(void) {
2      boost::posix_time::ptime now = boost::posix_time::second_clock::local_time();
3      std::stringstream ss_month, ss_day, ss_hours, ss_minutes, ss_seconds;
4      std::string year, month, day, hours, minutes, seconds, timestamp;
5
6      year = std::to_string(now.date().year());
7      ss_month << setw(2) << setfill('0') << now.date().month().as_number();
8      month = ss_month.str();
9      ss_day << setw(2) << setfill('0') << now.date().day().as_number();
10     day = ss_day.str();
11     ss_hours << setw(2) << setfill('0') << now.time_of_day().hours();
12     hours = ss_hours.str();
13     ss_minutes << setw(2) << setfill('0') << now.time_of_day().minutes();
14     minutes = ss_minutes.str();
15     ss_seconds << setw(2) << setfill('0') << now.time_of_day().seconds();
16     seconds = ss_seconds.str();
17     timestamp = "D" + year + month + day + "_T" + hours + minutes + seconds;
18
19     return timestamp;
20 }
```

Código fuente 4.3: Función creada para calcular el *timestamp*

Como se ha mencionado al principio de este capítulo, otra tarea de la capa externa de la plataforma de verificación es la correcta gestión de los directorios y ficheros que se utilizan para almacenar toda la información proporcionada por las aplicaciones ejecutadas más la añadida posteriormente. Así pues, tras la gestión de los parámetros era necesario implementar el código encargado de realizar esta tarea. Para ello, se empleó una nueva librería de Boost llamada Boost.Filesystem [57], cuyo cometido es precisamente gestionar y administrar sistemas de ficheros. De esta forma, se generaron nuevas funciones personalizadas que mediante dicha librería, llevaban a cabo las tareas de gestión del sistema de ficheros.

Uno de los requisitos establecidos en la fase de análisis de requisitos era el de que los usuarios del programa de verificación debían ser capaces de poder consultar ejecuciones anteriores. Para ello, los ficheros relativos a ejecuciones anteriores debían moverse a otro directorio con objeto de mantener el directorio de la ejecución actual precisamente con la información de la ejecución actual. De esta forma, acceder a este directorio sería instantáneo, además de conceder la posibilidad de consultar cualquier otra ejecución previa. La figura de código 4.4 muestra como ejemplo la función `moveDirectoriesInsideDirectory()` creada para tal cometido empleando la librería Boost.Filesystem.

```
1  void moveDirectoriesInsideDirectory(boost::filesystem::path const & source,
   ↪  boost::filesystem::path const & destination) {
2      boost::filesystem::directory_iterator end_iterator;
3      unsigned int dir_count = 0;
4
5      if (boost::filesystem::is_directory(source) &&
       ↪  boost::filesystem::is_directory(destination)) {
6          for (boost::filesystem::directory_iterator iterator(source); iterator !=
           ↪  end_iterator; ++iterator) {
7              try {
8                  if (boost::filesystem::is_directory(iterator->status())) {
9                      std::string destination_string = destination.string() + "/" +
                       ↪  iterator->path().filename().string();
10                     boost::filesystem::path final_destination(destination_string);
11                     ++dir_count;
12                     boost::filesystem::rename(iterator->path(), final_destination);
13                 }
14             }
15             catch (const std::exception & ex) {
16                 std::cerr << iterator->path().filename() << " " << ex.what() <<
                   ↪  std::endl;
17             }
18         }
19     }
20 }
```

Código fuente 4.4: Función creada para mover el contenido de un directorio a otro

Una vez definida la función, el código principal hace uso de dicha función tal y como muestra la figura de código 4.5.

```
1 boost::filesystem::path source(path_tests_logs_user_hostname_current);
2 boost::filesystem::path destination(path_tests_logs_user_hostname_past);
3 moveDirectoriesInsideDirectory(source, destination);
```

Código fuente 4.5: Llamada a la función `moveDirectoriesInsideDirectory`

Por otra parte, el programa `autotests` recibe mediante el parámetro `--rcuda_lib` un argumento fundamental para el funcionamiento del mismo: la ruta a la ubicación donde se encuentran los ficheros de la librería del cliente de rCUDA de cada desarrollador. De esta forma, el programa `autotests` adquiere la capacidad de utilizar estas librerías para reemplazar las librerías de CUDA en la ejecución de todos los programas que hagan uso de dicha librería.

Por último, una vez gestionados los parámetros de entrada, las variables de entorno para compartir datos entre los procesos `autotests` y `unit_tests` y los ficheros y directorios de registro, se procede a la ejecución del motor de verificación, esto es, el programa `unit_tests`. Para ello, en esta primera versión se empleó de nuevo la función `system`, tal y como muestra la figura de código 4.6.

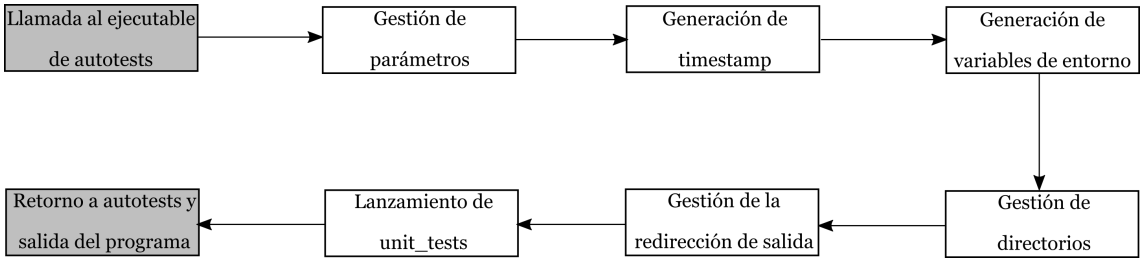


Figura 4.4: Flujo de ejecución de la primera versión de la plataforma de verificación diseñada e implementada en este trabajo.

```
1 std::string command = path_unit_tests_rcuda + "/.unit_tests";
2
3 // La variable "command" se modifica en este punto del programa en función de los
    ↪  parámetros introducidos por el usuario
4
5 system(command.c_str());
```

Código fuente 4.6: Lanzamiento del motor de verificación `unit_tests`

La figura 4.4 muestra de una forma simplificada el flujo de ejecución de la plataforma de verificación en su primera versión.

Una vez finalizada la implementación de ambas capas de software, el resultado final fue que el programa `autotests` contendría cerca de 400 líneas de código, y el programa `unit_tests` contendría cerca de las 11.000 líneas. Éste último programa, además, contendría una suma aproximada de 800 casos de test distribuidos en 47 suites únicas de test tal y como muestra la tabla 4.1. Nótese que esta versión de la plataforma de verificación consiste en la primera versión de la misma, pues tal y como se presentará en el capítulo de puesta en producción, esta versión del programa fue modificada en una gran parte hasta alcanzar la versión que se encuentra actualmente en producción.

Tipo de casos de test	Tamaño de casos de test	Aplicación	Casos de test
HPC	Small Medium Large	Gromacs	gromacs_(tamaño)_1
			gromacs_(tamaño)_2
			...
		Namd	namd_(tamaño)_1
			namd_(tamaño)_2
			...
		Lammps	lammps_(tamaño)_1
			lammps_(tamaño)_2
			...
		BarraCUDA	barracuda_(tamaño)_1
			barracuda_(tamaño)_2
			...
		Cuda-Meme	cuda_meme_(tamaño)_1
			cuda_meme_(tamaño)_2
			...
		Cuda-SW++	cudasw_(tamaño)_1
			cudasw_(tamaño)_2
			...
		GPU-Blast	gpu_blast_(tamaño)_1
			gpu_blast_(tamaño)_2
			...
		Magma	magma_(tamaño)_1
			magma_(tamaño)_2
			...
Cuda Samples	Simple	asyncAPI	
		cdpSimplePrint	
		cdpSimpleQuicksort	
		...	
	Utilities	deviceQuery	
		bandwidthTest	
		p2pBandwidthLatencyTest	
		...	
	Advanced	alignedTypes	
		cpp11_cuda	
		cdpAdvancedQuicksort	
		...	
	Finance	binomialOptions	
		BlackScholes	
		MonteCarloMultiGPU	
		...	
	...	...	

Tabla 4.1: Distribución de las suites y casos de test de la plataforma de verificación una vez completada la primera versión del programa.

CAPÍTULO 5
Puesta en producción

La liberación y puesta en marcha de la aplicación desarrollada para la verificación del middleware rCUDA se realizó sin mayores inconvenientes. Se generaron los binarios definitivos y se ubicaron en una posición determinada del sistema de ficheros compartido de tal forma que todos los desarrolladores de rCUDA tuvieran acceso a la nueva aplicación de verificación. Además, se generó un manual de usuario que se compartió con el equipo de desarrollo y que se adjunta en este trabajo como apéndice A. Así pues, a partir del momento de la puesta en producción del programa, los desarrolladores eran por fin capaces de ejecutar de forma masiva casos de test que ejecutaban a su vez aplicaciones que probaban su copia local de los ficheros binarios de rCUDA compilados a partir de su código fuente. Además, también eran capaces de comprobar dichas ejecuciones en tiempo real, aplicaciones que de otra forma deberían ejecutar una a una a través de las interfaces individuales de cada una de ellas. Esto supuso un cambio muy importante con respecto al estado anterior, ya que debido a la dificultad de ejecutar un gran número de aplicaciones, la frecuencia con la que antes se ejecutaban dichas aplicaciones para comprobar el estado de rCUDA era muy inferior a la actual.

En este capítulo se detalla cómo se llevaron a cabo las modificaciones efectuadas a la plataforma de verificación desarrollada en este TFM gracias a la retroalimentación ofrecida por sus usuarios. También se detalla la nueva interfaz de usuario generada a partir de dicha retroalimentación.

5.1 Retroalimentación de los desarrolladores

Los usuarios de la aplicación desarrollada en este trabajo no tardaron en proporcionar al desarrollador de la misma retroalimentación con objeto de pulir y mejorar ciertos aspectos. Y gracias a este feedback, la aplicación que está siendo utilizada actualmente para verificar el desarrollo de rCUDA presenta una notable evolución frente a la liberada como primera versión. Así pues, en este capítulo se detallan todas las mejoras que han sido añadidas a la primera versión del programa como resultado de la retroalimentación del equipo de desarrollo de rCUDA.

5.1.1. Fichero de registro global

La primera versión del programa ofrecía, en tiempo real y por la salida estándar, la información asociada a cada caso de test, como el estado de dicha ejecución (esto es, si se había producido fallo o no) y el tiempo de ejecución de cada caso de test. Además, se almacenaban en los directorios destinados a tal efecto los registros de cada aplicación y la información extra añadida para cada una de ellas.

No obstante, considérese el caso de que un usuario de la plataforma de verificación, y tras haber pasado un tiempo de haber lanzado y terminado una ejecución en particular, necesitara acceder de nuevo al registro que proporciona dicha plataforma por la salida estándar. La forma de abordar esto en la primera versión del programa habría sido redirigir de forma manual la salida estándar a un fichero y, de esta manera, tener la posibilidad de consultar posteriormente dicha salida accediendo a ese fichero.

Así pues, una de las primeras características solicitadas por los usuarios de la plataforma fue precisamente la de resolver este inconveniente de forma automática. Para ello, se decidió generar un nuevo fichero y ubicarlo en el directorio asociado a la ejecución actual. Este fichero contendría una copia de la salida estándar proporcionada por el programa. La implementación de esta característica consistió simplemente en modificar la variable `command`[1] de tal forma que, mediante el programa `tee`, se redirigiera la salida estándar al fichero mencionado. Esta variable consistía en un string del estándar de C++ que contenía la línea de ejecución asociada a la aplicación que se pretendía ejecutar en el caso de test en cuestión y que, posteriormente, debía ejecutarse a través de una shell. La figura de código 5.1 ilustra esta implementación.

```
1 command += " 2>&1 | tee -a " + current_dir_logs + "/_globalResults";
```

Código fuente 5.1: Generación del fichero de resultados globales

5.1.2. Desactivar casos de test en tiempo de ejecución

Algunos casos de test implementados en el programa resultaban en fallo simplemente porque no se cumplían determinados requisitos de diversa índole, aunque dicho fallo no tuviera relación alguna con rCUDA, como por ejemplo los casos en que una determinada aplicación necesitaba un *Compute Capability*[2] determinado, y éste no era satisfecho por las GPUs presentes en el nodo donde se ejecutaba el servidor de rCUDA. Por lo tanto, los usuarios de la plataforma de verificación solicitaron que, en circunstancias en las que no fuera posible satisfacer estos requisitos, el programa desactivara en tiempo de ejecución estos casos de test.

Para llevar a cabo la implementación, inicialmente se usaron los *decorators* `disabled` y `disabled_if`, que como se ha comentado anteriormente proveen un mecanismo para deshabilitar casos de test incondicionalmente (en el caso de `disabled`) y en función de una o varias condiciones (en el caso de `disabled_if`). No obstante, posteriormente se descartó el uso de estos *decorators* dado que se requería que, además de deshabilitar el caso de test en cuestión, se le proporcionara al usuario un mensaje que mostrase el motivo que llevaba a la desactivación de dicho caso de test. Así pues, tras realizar las pruebas pertinentes, la solución que se llevó a cabo fue la de realizar ciertas comprobaciones dentro del código de cada caso de test que necesitara ser deshabilitado en tiempo de ejecución si no se satisfacían las restricciones antes mencionadas y, en caso de ser deshabilitado, mostrar al usuario un mensaje proporcionándole la razón que llevaba a su desactivación.

Tras la implementación de la solución, aunque el usuario percibía que el caso de test no se había ejecutado al no haber visto nada al respecto por la salida estándar ni tampoco en los ficheros de registro, en la práctica el caso de test sí se había ejecutado (macro `BOOST_AUTO_TEST_CASE()`), aunque no así los tests propiamente dichos (macros

[1] Recuérdese que la variable `command` es la variable que contiene el comando que se ejecutará en el proceso `unit_test`, esto es, la aplicación que finalmente será ejecutada mediante la llamada al sistema `system` en la versión inicial de la plataforma.

[2] Nombre dado por NVIDIA a la versión del hardware de sus GPUs.

BOOST_CHECK_EQUAL()). La figura de código 5.2 ilustra un caso de test en el que el programa ejecutado necesita un *Compute Capability* mínimo de 5.3 para funcionar correctamente, y en caso de que no se satisfaga, el caso de test fallará.

```
1  BOOST_AUTO_TEST_CASE(fp16ScalarProduct) {
2      if (std::stod(unit_tests_constants::cc) >= 5.3) {
3          std::string final_complete_cmd_fp16ScalarProduct =
               ↪ appendForwardingToCommand(complete_cmd_fp16ScalarProduct,
               ↪ boost::unit_test::framework::current_test_case().p_name);
4          tic();
5          BOOST_CHECK_EQUAL(system(final_complete_cmd_fp16ScalarProduct.c_str()),
               ↪ success_value);
6          toc(boost::unit_test::framework::current_test_case().p_name);
7          appendExecutionInfo(boost::unit_test::framework::current_test_case().p_name);
8      }
9      else {
10         std::cout << Color::yellow << "Test " <<
               ↪ boost::unit_test::framework::current_test_case().p_name << " disabled in
               ↪ runtime due to minimum compute capability not fulfilled" << Color::def <<
               ↪ std::endl;
11     }
12 }
```

Código fuente 5.2: Test deshabilitado en tiempo de ejecución

En la figura 5.2, para realizar la comprobación inicial se accede a la variable `unit_tests_constants::cc` que contiene la *Compute Capability* de la GPU de índice 0 del nodo del clúster que está ejecutando el servidor rCUDA. Este valor es obtenido mediante una función implementada ad-hoc que hace uso de la API de CUDA cuando comienza el programa principal.

Así pues, se ha seguido esta misma aproximación para deshabilitar aquellos casos de test que requieren de circunstancias especiales para funcionar. Algunos de estos ejemplos los podemos encontrar en el SDK de CUDA, como pueden ser los programas `simpleCUFFT_2d_MGPU`, que necesita un mínimo de dos GPUs para poder ejecutarse, y `simpleP2P`, que además de tener el mismo requisito, ambas GPUs también deben soportar transferencias de memoria P2P entre ellas y estar ambas ubicadas en el mismo árbol PCI.

Otro ejemplo, ya fuera del SDK de CUDA, es la aplicación de HPC `Barracuda`, que para determinados casos de test exige un mínimo de 6 GB de memoria de GPU disponible, por lo que estos casos de test fallaban cuando eran ejecutados en nodos con GPUs Nvidia Tesla K20m, cuya memoria máxima es de 5 GB. Al igual que en los casos anteriores, se ha abordado este problema con llamadas a la API de CUDA para obtener la información necesaria para deshabilitar el test en tiempo de ejecución en caso de que no se satisfagan las condiciones necesarias.

5.1.3. Versión *verbose*

La salida estándar de la primera versión de la aplicación de verificación en su ejecución por defecto mostraba el nombre de todos los casos de test ejecutados y el tiempo empleado en cada uno de ellos. No obstante, podría ser interesante que la salida fuera algo más rica en detalles, y eso fue precisamente lo que se solicitó por parte de sus usuarios.

Así pues, se optó por elaborar una versión *verbose* de la aplicación que se activara con el parámetro `--verbose` o `-v`. Esta versión mostraría, además de lo ya mostrado en la versión estándar, esto es, el nombre del caso de test y el tiempo de ejecución del mismo, la fecha y la hora de inicio y de fin de cada caso de test, y el *timestamp* de la ejecución

actual, con objeto de facilitar la búsqueda de dicha ejecución en los ficheros de registro si dicha búsqueda se realizara pasado un tiempo.

5.1.4. Comando a ejecutar por los casos de test

Otra solicitud por parte de los usuarios de la aplicación fue la de incluir en la salida estándar el comando "real" de la aplicación que era ejecutada bajo la interfaz del programa de verificación. De esta forma, se podría comprobar, por ejemplo, cuántos hilos de CPU o cuántas GPUs se habían activado en la ejecución, entre otras comprobaciones. No obstante, existía el inconveniente de que mostrar por pantalla esta información sobrecargaría ligeramente la salida original, por lo que se optó por incluirla únicamente en la versión *verbose*, además de en la información extra almacenada junto a cada caso de test ejecutado.

Así pues, la tarea consistió en modificar el prototipo de la función `appendExecutionInfo()` para que recibiera un segundo parámetro. Este nuevo parámetro sería el nombre de la variable que contuviera el comando a ejecutar, almacenado debidamente en el fichero de comandos `unit_tests_constants.cpp`. La función, por lo tanto, únicamente debía comprobar si el usuario estaba ejecutando la versión verbose del programa y, si así era, imprimir por pantalla el comando en cuestión.

5.1.5. Tiempos de ejecución adaptativos

La primera versión del programa mostraba los tiempos de ejecución de cada caso de test en nanosegundos, independientemente del tiempo de ejecución total. Esta era una unidad de magnitud apropiada para muchos casos de test, pero otros tenían un tiempo de ejecución del orden de horas, por lo que mostrar los resultados en nanosegundos era poco apropiado. Así pues, los usuarios solicitaron que las magnitudes de los tiempos de ejecución se adaptaran al tiempo de ejecución.

Se decidió, finalmente, que la forma más apropiada de mostrar el tiempo de ejecución era la empleada por el programa Unix `time`, por lo que se decidió implementar una gestión de tiempo similar. La figura de código 5.3 muestra la implementación de la función `tic()` encargada de llevar a cabo la tarea de inicializar el cronómetro, para la que se utilizó la librería Boost.Chrono [56]. Nótese que la función comprueba a través de la función `isVerbose()` si el usuario ha ejecutado la versión `verbose` del programa. En función de eso, la cantidad de información a proporcionar será mayor o menor.

```
1  void tic(std::string test, std::string command) {
2      if (isVerbose()) {
3          boost::posix_time::ptime now = boost::posix_time::second_clock::local_time();
4          std::stringstream ss_month, ss_day, ss_hours, ss_minutes, ss_seconds;
5          std::string year, month, day, hours, minutes, seconds;
6
7          year = std::to_string(now.date().year());
8          ss_month << setw(2) << setfill('0') << now.date().month().as_number();
9          month = ss_month.str();
10         ss_day << setw(2) << setfill('0') << now.date().day().as_number();
11         day = ss_day.str();
12         ss_hours << setw(2) << setfill('0') << now.time_of_day().hours();
13         hours = ss_hours.str();
14         ss_minutes << setw(2) << setfill('0') << now.time_of_day().minutes();
15         minutes = ss_minutes.str();
16         ss_seconds << setw(2) << setfill('0') << now.time_of_day().seconds();
17         seconds = ss_seconds.str();
18
```

```
19          std::string init_time = hours + ":" + minutes + ":" + seconds;
20          std::string init_date = day + "-" + month + "-" + year;
21
22
23          std::cout << std::endl << "Test " << test << " has begun at " << Color::blue <<
                ↪  init_time << Color::def << " on " << Color::blue << init_date <<
                ↪  Color::def << std::endl;
24
25          start = boost::chrono::high_resolution_clock::now();
26      }
27      else {
28          start = boost::chrono::high_resolution_clock::now();
29      }
30  }
```

Código fuente 5.3: Implementación de las función `tic()` encargada de inicializar el cronómetro

Por otra parte, la figura de código 5.4 muestra la implementación de la función `toc()`, que se encarga de detener el cronómetro e imprimir el tiempo de ejecución por pantalla (usándose de nuevo la librería Boost.Chrono).

```
1   void toc(std::string test, std::string command) {
2       boost::chrono::milliseconds duration_milliseconds =
            ↪  boost::chrono::duration_cast<boost::chrono::milliseconds>
            ↪  (boost::chrono::high_resolution_clock::now() - start);
3       boost::chrono::seconds duration_seconds = boost::chrono::duration_cast
            ↪  <boost::chrono::seconds>(duration_milliseconds);
4       boost::chrono::minutes duration_minutes = boost::chrono::duration_cast
            ↪  <boost::chrono::minutes>(duration_milliseconds);
5       boost::chrono::hours duration_hours = boost::chrono::duration_cast
            ↪  <boost::chrono::hours>(duration_milliseconds);
6
7       final_time = std::to_string(duration_hours.count()) + "h " +
            ↪  std::to_string((duration_minutes % boost::chrono::hours(1)).count()) + "m " +
            ↪  std::to_string((duration_seconds % boost::chrono::minutes(1)).count()) + "s "
            ↪  + std::to_string((duration_milliseconds % boost::chrono::seconds(1)).count())
            ↪  + "ms";
8
9       if (isVerbose()) {
10          boost::posix_time::ptime now = boost::posix_time::second_clock::local_time();
11          std::stringstream ss_month, ss_day, ss_hours, ss_minutes, ss_seconds;
12          std::string year, month, day, hours, minutes, seconds;
13
14          year = std::to_string(now.date().year());
15          ss_month << setw(2) << setfill('0') << now.date().month().as_number();
16          month = ss_month.str();
17          ss_day << setw(2) << setfill('0') << now.date().day().as_number();
18          day = ss_day.str();
19          ss_hours << setw(2) << setfill('0') << now.time_of_day().hours();
20          hours = ss_hours.str();
21          ss_minutes << setw(2) << setfill('0') << now.time_of_day().minutes();
22          minutes = ss_minutes.str();
23          ss_seconds << setw(2) << setfill('0') << now.time_of_day().seconds();
24          seconds = ss_seconds.str();
25
26          std::string end_time = hours + ":" + minutes + ":" + seconds;
27          std::string end_date = day + "-" + month + "-" + year;
28
29          std::cout << "Test " << test << " has finished at " << Color::blue << end_time <<
                ↪  Color::def << " on " << Color::blue << end_date << Color::def <<
                ↪  std::endl;
30
```

```
31          std::cout << "Executed command: " << Color::magenta << command << Color::def <<
               ↪  std::endl;
32
33
34          std::cout << "Time elapsed for test " << Color::cyan << test << Color::def << ":
               ↪  " << final_time << std::endl << std::endl;
35      }
36      else {
37          std::cout << "Time elapsed for test " << Color::cyan << test << Color::def << ":
               ↪  " << final_time << std::endl;
38      }
39  }
```

Código fuente 5.4: Implementación de la función `toc()` que detiene el cronómetro y proporciona al usuario el tiempo de ejecución del caso de test en cuestión

5.1.6. Tiempo total de ejecución

En esta solicitud, los usuarios preguntaron por la posibilidad de incluir el tiempo total de ejecución en la salida estándar del programa y, por consiguiente, en el fichero de registro global, siendo el tiempo total el tiempo transcurrido desde que el usuario lanza la aplicación de verificación hasta que ésta finaliza totalmente. En la primera versión del programa, esto se efectuaba de forma manual mediante el programa Unix `time`, pero al igual que ocurría con el fichero de registro global, el hecho de que el programa hiciera esto por el usuario de forma automática les facilitaría su trabajo.

Así pues, esta ampliación se abordó de una forma similar a como se implementó la función encargada de calcular y mostrar el tiempo de ejecución de cada caso de test, aunque en este caso, el encargado de realizar los cálculos ya no sería el motor de testing (`unit_tests`), si no el programa de interfaz de usuario (`autotests`). Para ello, en el programa principal, y en el instante inmediatamente anterior al que el proceso `autotests` lanzara el programa `system` para ejecutar `unit_tests`, se inicializaba un nuevo cronómetro (función `globalTic()`) y, cuando finalizara el proceso `unit_tests` y retornara el control al proceso padre, éste detendría el cronómetro e imprimiría por pantalla el tiempo total (función `globalToc()`).

5.1.7. Timeout para casos de test

Esta ampliación del programa fue una de las más interesantes desde el punto de vista práctico. Dado que rCUDA es un software que está en continuo desarrollo, e incluso en ocasiones existe más de una rama de desarrollo en paralelo, se están probando y creando continuamente nuevas funcionalidades. Estas funcionalidades pasan por todos los estados de cualquier desarrollo software: desde la fase alpha más temprana hasta la fase final estable. Por lo tanto, probando las nuevas funcionalidades en desarrollo no es extraño encontrarse con que un caso de test se queda bloqueado u ocupa un tiempo demasiado elevado como para ser aceptable. En cualquiera de estos dos casos, la ejecución de la plataforma de verificación permanecía bloqueada en un caso de test en concreto y no avanzaba con el resto de casos de test. Por lo tanto, una solicitud unánime por parte de los desarrolladores de rCUDA fue que el programa de verificación debía ser capaz de cancelar de forma automática un caso de test en ejecución que ocupara un tiempo de ejecución más allá de lo permitido.

Así pues, para abordar esta ampliación se utilizó inicialmente, en la propia definición de los casos de test, el *decorator* `timeout` que proporciona el propio framework de

Boost.Test. No obstante, y tras las primeras pruebas, se consideró que esta solución no era la apropiada. El motivo fue que, a pesar de que el *decorator* `timeout` llevaba a cabo correctamente su función, es decir, cancelaba un caso de test de forma automática tras cumplirse el tiempo permitido y continuaba con el siguiente caso de test, sin embargo dejaba en marcha y en background la ejecución de la aplicación que lanzaba (recuérdese que la aplicación de verificación usaba en su primera versión la llamada al sistema `system` para poner en marcha una aplicación que usaba las librerías de rCUDA). Esto era un problema ya que en algunos casos, como podría ser el framework de Deep Learning de Google TensorFlow, se requería que solamente hubiera un caso de test en ejecución y, si había más de uno, el caso de test fallaba.

Por lo tanto, la solución que se decidió implementar implicó modificar una gran parte del núcleo de la plataforma de verificación. En primer lugar, resultó evidente en seguida que el uso de la función `system` de Unix dejó de ser apropiada. El motivo fue que esta llamada proporcionaba comodidad al programador al abstraerlo de los detalles de emplear manualmente las funciones `fork` y `exec`, pero pagando el precio de perder el control sobre los procesos ejecutados. Sin embargo, en este nuevo contexto se hacía necesario recuperar este control, ya que tras vencer el *timeout* establecido, el programa `autotests` debía *matar* sus procesos hijos empleando para ello sus identificadores de proceso (`PID`) o, en su defecto, sus identificadores de grupo (GPID). Estos identificadores únicamente podían ser obtenidos a través de las llamadas al sistema `fork` y `exec`, por lo que en última instancia se decidió sustituir la función `system` por una función creada ad-hoc llamada `testLaunch()` que hacía uso internamente de las llamadas al sistema `fork` y `exec`, y que se proporciona en la figura de código 5.5.

```
1  int testLaunch(std::string test, std::string command, Test_size test_size) {
2      pid_t cpid, wpid, cpgid;
3      int status;
4      std::time_t t1, t2;
5      bool killed = false;
6      unsigned int timeout_seconds, timeout_minutes, sleep_time = 30;
7
8      //User has set timeout parameter
9      if (!isSetEnvVarToNull(unit_tests_constants::ev_timeout)) {
10         timeout_minutes = std::stoi(getEnvVar(unit_tests_constants::ev_timeout));
11         timeout_seconds = timeout_minutes * 60;
12     }
13     else {
14         switch (test_size) {
15             case Test_size::small:
16                 timeout_seconds = unit_tests_constants::testLaunch_timeout_small;
17                 break;
18             case Test_size::medium:
19                 timeout_seconds = unit_tests_constants::testLaunch_timeout_medium;
20                 break;
21             case Test_size::large:
22                 timeout_seconds = unit_tests_constants::testLaunch_timeout_large;
23                 break;
24         }
25     }
26
27     cpid = fork();
28
29     if (cpid == -1) {
30         perror("fork cannot be executed");
31         exit(EXIT_FAILURE);
32     }
33
```

```
34      //Parent code
35      if (cpid) {
36          std::time(&t1);
37          setpgid(cpid, 0);
38          cpgid = getpgid(cpid);
39
40          do {
41              std::time(&t2);
42              wpid = waitpid(cpid, &status, WNOHANG);
43
44              if (wpid == -1) {
45                  perror("waitpid cannot be executed");
46                  exit(EXIT_FAILURE);
47              }
48
49              //Timeout reached
50              if ((t2 - t1) >= timeout_seconds && !killed) {
51                  if (!kill(-cpgid, SIGKILL)) {
52                      std::cout << std::endl << Color::red << "Timeout reached, test " <<
                            ↪  Color::yellow << test << Color::red << " killed, waiting " <<
                            ↪  sleep_time << " seconds for recovering purposes" << Color::def
                            ↪  << std::endl << std::endl;
53                      sleep(sleep_time);
54                      killed = true;
55                  }
56              }
57              sleep(1);
58          }
59          while (!wpid);
60
61          if (WIFEXITED(status)) {
62              return WEXITSTATUS(status);
63          }
64          else if (WIFSIGNALED(status)) {
65              std::cout << "Test exited by signal " << WTERMSIG(status) << std::endl;
66              return -101;
67          }
68          else if (WIFSTOPPED(status)) {
69              std::cout << "Test stopped by signal " << WSTOPSIG(status) << std::endl;
70              return -102;
71          }
72          else {
73              std::cout << "Test exited by unknown reason" << std::endl;
74              return -103;
75          }
76      }
77      //Child code
78      else {
79          if (execl("/bin/sh", "/bin/sh", "-c", command.c_str(), (char *) 0) == -1){
80              perror("exec cannot be executed");
81          }
82          _exit(EXIT_FAILURE);
83      }
84  }
```

Código fuente 5.5: Función `testLaunch()` creada para ejecutar casos de test

Como puede apreciarse en la figura de código 5.5, en la línea 36 se activa un cronómetro con la función `time` del estándar de C++, y se consulta en la línea 50. Si se obtiene un valor que no ha excedido el tiempo máximo, entonces se espera una iteración y se vuelve a consultar en la siguiente. Si por el contrario el tiempo se ha excedido, se emplea la llamada al sistema `kill` para enviar la señal `SIGKILL` a todos los procesos que com-

parten el mismo identificador de grupo. El proceso `unit_tests` y todos los subprocesos hijos de este proceso comparten el mismo identificador de grupo gracias a la llamada a la función `setpgid()` ubicada en la línea 37 de la figura de código 5.5, donde se fuerza a dichos procesos a heredar el identificador de grupo del proceso `unit_tests`.

Con respecto al valor del timeout, se definieron tres valores distintos en función del tamaño del caso de test (*small*, *medium* y *large*). No obstante, se determinó que sería interesante conceder al usuario la posibilidad de establecer a través de un nuevo parámetro el timeout deseado, lo que puede verse en el código ubicado entre las líneas 9 y 25.

Con la implementación de la función `testLaunch()` y la sustitución de la llamada a la función `system` por la nueva función `testLaunch()`, la plataforma de verificación de rCUDA era por fin completamente autónoma, lo que sería de extrema utilidad para las ejecuciones orientadas a probar versiones de rCUDA con características en fases de desarrollo tempranas.

5.1.8. Cancelación de la ejecución actual

Esta ampliación supuso otro gran cambio en el núcleo de la plataforma de verificación (esto es, programa `unit_tests`). En esta ocasión, se recibía la solicitud de los usuarios de poder cancelar manualmente una ejecución en progreso del programa de verificación implementado en este TFM. Esto no sería un problema si dicha ejecución contuviera un solo proceso, ya que el usuario podría enviar la señal `SIGINT` a través de la combinación de teclas `Ctrl+C` y, si el proceso manejaba la señal correctamente, finalizar la ejecución del mismo.

No obstante, recuérdese que la plataforma de verificación funciona con dos capas de software y, por tanto, con un mínimo de dos procesos. Además, el uso de la función `exec` para ejecutar programas en una shell implica que se generen procesos del intérprete de dicha shell. A todo esto, si la ejecución del caso de test implica ejecutar un intérprete de Python, el número total de procesos hijos se incrementa todavía más. Así pues, la implementación de esta mejora implicaba que el proceso `autotests` tuviera conocimiento explícito de todos los identificadores de proceso que descendían de *él mismo*, con objeto de poder *enviarles* la señal correspondiente.

Para ello, y en primer lugar, se optó por trabajar con la señal `SIGINT` sobre el identificador de grupo en lugar de sobre el identificador de proceso. La razón es que gracias a la implementación de la mejora del timeout, los procesos descendientes de `autotests` compartían el mismo identificador de grupo y, por lo tanto, sería suficiente con enviar la señal a ese identificador. El problema radicaba en el hecho de que el identificador de grupo era un dato que poseía el programa `unit_tests`, y no `autotests`. Además, no podía emplearse el mecanismo de herencia entre procesos porque la información la poseía el proceso hijo, no el padre. ¿Cómo conseguir de forma eficiente que `autotests` tuviera acceso a información propia de un proceso hijo?

Tras realizar un pequeño estudio de las opciones disponibles, se decidió emplear un mecanismo de comunicación entre procesos. De entre las posibilidades para ello, finalmente se optó por emplear la librería Interprocess [58] de Boost, cuyo objetivo es, precisamente, simplificar el uso de mecanismos de sincronización y comunicación entre procesos.

Así pues, en primer lugar se modificó la función `testLaunch()` del código del programa `unit_tests` para que generase una región de memoria compartida y almacenase en la misma el identificador de grupo de todos los procesos que descendían de él mismo. La figura de código 5.6 muestra los cambios efectuados sobre la función `testLaunch()`.

```
boost::interprocess::permissions perm;
perm.set_unrestricted();
boost::interprocess::shared_memory_object::remove(unit_tests_constants::
    ↪  shm_ccpgid.c_str());
boost::interprocess::managed_shared_memory
    ↪  managed_shm_ccpgid(boost::interprocess::create_only,
    ↪  unit_tests_constants::shm_ccpgid.c_str(), 1024, 0, perm);
managed_shm_ccpgid.construct<pid_t>("CCPGID")(cpgid);
```

Código fuente 5.6: Uso de Boost.Interprocess para compartir una región de memoria entre procesos

Como se aprecia en la figura de código 5.6, se genera una nueva región de memoria compartida y se almacena en ella el identificador de grupo. Así, el programa `autotests` lanza, en el código asociado al proceso padre, un thread que se encarga de ejecutar una función cuyo objetivo es acceder a la región de memoria compartida por `unit_tests` y enviar la señal `SIGKILL` a todos los procesos relacionados con la ejecución actual. La figura de código 5.7 presenta la generación del thread por parte de `autotests`, y la figura de código 5.8 muestra la activación de este thread a través de variables condición del estándar de C++.

```
std::thread thread_sigint(thread_sigint_handler, ppid, cpgid);
thread_sigint.detach();
```

Código fuente 5.7: Creación del thread `thread_sigint`

```
static volatile sig_atomic_t sigint_signal_status = 0;
std::mutex m;
std::condition_variable cv;

/*Handler for signal SIGINT in autotests parent*/
void autotests_sigint_handler(int signal) {
    assert(signal == SIGINT);
    sigint_signal_status = signal;
    std::cout << std::endl << Color::yellow << "Signal SIGINT received, killing
        ↪  everything..." << Color::def << std::endl;
    cv.notify_one();
}

/*Handler for thread_sigint*/
void thread_sigint_handler(const pid_t ppid, const pid_t cpgid) {
    pid_t ccpgid;
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk);

    if (sigint_signal_status == SIGINT) {
        try {
            boost::interprocess::managed_shared_memory
                ↪  managed_shm_ccpgid(boost::interprocess::open_only,
                ↪  autotests_constants::shm_ccpgid.c_str());
            std::pair<pid_t*, std::size_t> p = managed_shm_ccpgid.find<pid_t>("CCPGID");

            if (p.first) {
                ccpgid = *p.first;
                kill(-ccpgid, SIGKILL);
                kill(-cpgid, SIGKILL);
                kill(ppid, SIGKILL);
            }

```

```
31              boost::interprocess::shared_memory_object::remove(
                    ↪  autotests_constants::shm_ccpgid.c_str());
32          }
33          catch (const std::exception &ex) {
34              std::cout << "managed_shm_ccpgid exception: " << ex.what() << std::endl;
35          }
36      }
37  }
```

Código fuente 5.8: Activación del thread `thread_sigint` y acceso a la región de memoria compartida

Así pues, tras la implementación de esta mejora, la plataforma de verificación ya estaba preparada para permitir a sus usuarios la cancelación total e instantánea de cualquier ejecución en activo, todo ello sin dejar rastro alguno de dicha ejecución en ninguno de los nodos implicados.

5.1.9. Lista de casos de test fallidos

En esta ocasión, los usuarios del programa de verificación de rCUDA solicitaron la posibilidad de ver la lista de los casos de test fallidos tras una ejecución. Esta característica ya se encontraba implementada a través del propio framework de Boost.Test. Sin embargo, la forma de presentar esta información podría no ser la más apropiada para localizar de forma rápida los casos de test fallidos, pues añadía mucha más información de la que a priori podría ser necesaria para los usuarios. Así pues, la solicitud consistía, simplemente, en implementar un mecanismo que mostrase por pantalla una lista lo más sencilla y escueta posible de los casos de test que habían fallado.

Para acometer esta nueva solicitud, debía considerarse el hecho de que el programa de interfaz de usuario `autotests`, el encargado de enviar la información a la salida estándar, no tenía acceso directo a los casos de test en sí mismos, pues estos estaban definidos en el programa de motor de verificación `unit_tests`. Además existía la misma limitación que en el caso de la mejora de la cancelación del caso de test en ejecución, esto es, no podía usarse el mecanismo de herencia entre procesos porque la comunicación requerida era en sentido ascendente. Estudiando las posibilidades para llevar a cabo la tarea, se decidió, de nuevo, emplear la libería Boost.Interprocess [58] para crear una región de memoria compartida desde `unit_tests` y que `autotests` pudiera acceder a los datos alojados en esa región de memoria.

Sin embargo, en este caso el procedimiento fue más complicado. En el caso de la cancelación del caso de test en ejecución, el dato a compartir era de tipo `pid_t`, que es un entero con signo, esto es, un tipo primitivo. Esto implicaba que el procedimiento de creación de la región de memoria compartida sería razonablemente sencillo. Sin embargo, en este caso se pretendía alojar en la región de memoria compartida un array de strings del estádar de C++, lo que complicaba sobremanera el diseño al no existir en Boost.Interprocess un método estándar para compartir tipos de datos extructurados.

Así pues, tras realizar un estudio para descubrir la mejor forma de abordar el problema, se descubrió que podría llevarse a cabo la solución definiendo nuevos tipos de contenedores basados en Boost.Interprocess que empleaban *allocators* alojados en memoria compartida. Usando estos contenedores en lugar de los del estándar de C++ se conseguía alojar estructuras de datos complejas en regiones de memoria compartidas.

Por lo tanto, la primera tarea fue implementar, dentro del código asociado a cada caso de test definido en `unit_tests`, una sentencia para comprobar si el caso de test había resultado en fallo o éxito. Si se daba el primer caso, entonces se llamaba a una función

llamada `registerFailedTests()` que admitía por parámetro un string del estándar con el nombre del caso de test fallido. La figura de código 5.9 muestra las líneas que se le añadieron a cada caso de test.

```
1  if (!boost::unit_test::results_collector.results(
       ↪  boost::unit_test::framework::current_test_case().p_id).passed()) {
2       registerFailedTests(boost::unit_test::framework::current_test_case().p_name);
3  }
```

Código fuente 5.9: Llamada a la función `registerFailedTests()`

La figura 5.10 muestra la función `registerFailedTests()` donde se generan regiones de memoria compartidas para listar la totalidad de los casos de test fallidos.

En las líneas 9 a 12 puede apreciarse la generación de nuevos tipos de datos basados en contenedores de Boost.Interprocess destinados a almacenar la lista de casos de test fallidos que posteriormente recuperará el programa `autotests`.

```
1  void registerFailedTests(std::string failed_test_name) {
2      //Setting share memory regions permissions
3      boost::interprocess::permissions perm;
4      perm.set_unrestricted();
5
6      //Sharing unit_tests_constants::failed_tests_list
7      boost::interprocess::managed_shared_memory
          ↪  managed_shm_failed_tests_list(boost::interprocess::open_or_create,
          ↪  unit_tests_constants::shm_failed_tests_list.c_str(), 1024000, 0, perm);
8
9      typedef boost::interprocess::allocator<char,
          ↪  boost::interprocess::managed_shared_memory::segment_manager> CharAllocator;
10     typedef boost::interprocess::basic_string<char, std::char_traits<char>,
          ↪  CharAllocator> myShmString;
11     typedef boost::interprocess::allocator<myShmString,
          ↪  boost::interprocess::managed_shared_memory::segment_manager> StringAllocator;
12     typedef boost::interprocess::vector<myShmString, StringAllocator> myShmStringVector;
13
14     CharAllocator charallocator(managed_shm_failed_tests_list.get_segment_manager());
15     StringAllocator stringallocator(managed_shm_failed_tests_list.get_segment_manager());
16
17     myShmString new_failed_test_name(failed_test_name.begin(), failed_test_name.end(),
          ↪  charallocator);
18
19     static myShmStringVector *failed_test_name_vector =
          ↪  managed_shm_failed_tests_list.construct<myShmStringVector>(
          ↪  "failed_test_name_vector")(stringallocator);
20
21     failed_test_name_vector->push_back(new_failed_test_name);
22 }
```

Código fuente 5.10: Función `registerFailedTests()`

Así, cuando `unit_tests` ha generado la región de memoria compartida y la ha llenado con la lista de casos de test fallidos, `autotests` ya está en disposición de acceder a ese recurso en el momento en que `unit_tests` finaliza su ejecución y devuelve el control al padre. La figura de código 5.11 muestra el código que ejecuta el padre para acceder a la región de memoria compartida cuando el hijo le devuelve el control.

```
1  if (existsSharedMemoryRegion(autotests_constants::shm_failed_tests_list)) {
2    boost::interprocess::managed_shared_memory
       ↪  managed_shm_failed_tests_list(boost::interprocess::open_only,
       ↪  autotests_constants::shm_failed_tests_list.c_str());
3
4    typedef boost::interprocess::allocator<char,
       ↪  boost::interprocess::managed_shared_memory::segment_manager> CharAllocator;
5    typedef boost::interprocess::basic_string<char, std::char_traits<char>, CharAllocator>
       ↪  myShmString;
6    typedef boost::interprocess::allocator<myShmString,
       ↪  boost::interprocess::managed_shared_memory::segment_manager> StringAllocator;
7    typedef boost::interprocess::vector<myShmString, StringAllocator> myShmStringVector;
8
9    myShmStringVector *failed_test_name_vector =
       ↪  managed_shm_failed_tests_list.find<myShmStringVector>(
       ↪  "failed_test_name_vector").first;
10
11   std::cout << Color::red << "\nList of failed test cases: " << Color::def << std::endl
       ↪  << std::endl;
12
13   for (unsigned int i = 0; i < failed_test_name_vector->size(); i++) {
14     std::string short_failed_test_name(failed_test_name_vector->at(i).begin(),
         ↪  failed_test_name_vector->at(i).end());
15   std::cout << Color::red << short_failed_test_name << Color::def << std::endl;
16   }
17
18   boost::interprocess::shared_memory_object::remove(
       ↪  autotests_constants::shm_failed_tests_list.c_str());
19 }
```

Código fuente 5.11: `autotests` accediendo a la región de memoria compartida creada por `unit_tests`

Tal y como puede apreciarse en la figura de código 5.11, la línea 2 permite que el programa acceda en modo `open_only` a la región de memoria creada y compartida por el proceso `unit_tests`, y la línea 9 realiza la búsqueda en sí misma de la estructura de datos compartida, en este caso identificada con el nombre de `failed_test_name_vector`. A partir de este momento, los nuevos contenedores que constituyen el vector `failed_test_name_vector` están alojados en memoria compartida y ya puede listarse el contenido del mismo de la forma habitual, lo que se lleva a cabo a partir de la línea 13.

5.1.10. Fichero de registro de casos de test fallidos

Una solicitud muy vinculada a la solicitud inmediatamente anterior fue la de que `autotests` generase, en la ubicación destinada a almacenar los registros de la ejecución actual, un nuevo fichero en el que se ubicara el nombre de cada test fallido y la salida de la aplicación para cada test fallido, junto con la información extra que pudiera ser relevante.

Dado que en la mejora anterior se había implementado un sistema para compartir la lista de casos de test fallidos en una ejecución dada, la implementación de esta mejora fue relativamente sencilla, pues gran parte del trabajo ya estaba hecho. Así pues, únicamente fue necesario añadir una llamada a la función creada ad-hoc llamada `createFailedTestsFile()` en el momento de recorrer el vector que contenía la lista de casos de test fallidos, pasándole como primer y único argumento el nombre del test que había fallado. La implementación de dicha función se muestra en la figura de código 5.12.

```
void createFailedTestsFile(std::string failed_test_name) {
    std::string test_file, failed_tests_file, messageApplication = "", line, timestamp =
       ↪  getEnvVar("TIMESTAMP");
    std::ifstream infile;
    std::ofstream outfile;

    messageApplication += "\n\n++++++++++++++++++++++++++++++++++++++++ test " +
       ↪  failed_test_name + " ++++++++++++++++++++++++++++++++++++++++\n";

    test_file = autotests_constants::path_tests_logs_user_hostname + "/" + timestamp +
       ↪  "/" + failed_test_name;
    failed_tests_file = autotests_constants::path_tests_logs_user_hostname + "/" +
       ↪  timestamp + "/_failedTestCases";

    infile.open(test_file, std::ios_base::in);

    if (infile.is_open()) {
        outfile.open(failed_tests_file, std::ios_base::app);

        if (outfile.is_open()) {
            outfile << Color::red << messageApplication << Color::def;
            while (std::getline(infile, line)) {
                outfile << line << std::endl;
            }
            infile.close();
        }
        else {
            std::cout << "Autotests function createFailedTestsFile() error -> Unable to
               ↪  open " << failed_tests_file << std::endl;
        }
        outfile.close();
    }
    else {
        std::cout << "Autotests function createFailedTestsFile() error -> Unable to open
           ↪  " << test_file << std::endl;
    }
}
```

Código fuente 5.12: Implementación de la función `createFailedTestsFile()` en el programa `autotests`

5.1.11. Retoques para pruebas de carga

En esta sección se detallan las últimas modificaciones realizadas sobre el programa original (además de las anteriores), modificaciones destinadas a hacerlo más versátil y práctico con objeto de utilizarlo en pruebas de carga. Estas pruebas se concibieron como ejecuciones del programa en horario en que el clúster estuviera libre de otras ejecuciones, generalmente horario nocturno.

Como primera y más importante prueba de carga se consideró la ejecución del programa `autotests` para probar rCUDA en sus escenarios más relevantes. Estos escenarios consistían en combinaciones específicas del hardware del clúster y los servidores de rCUDA para las que se esperaba un funcionamiento correcto. Un ejemplo de escenario podría ser usar usar la red TCP junto con 2 GPUs remotas y servidor de rCUDA compartido, mientras que otro escenario distinto podría ser usar la red InfiniBand, con 2 GPUs remotas y estando estas GPUs en dos servidores rCUDA distintos. Por lo tanto, la idea era considerar la ejecución de estos escenarios para cada una de las copias del código fuente de rCUDA de cada desarrollador. De esta forma, cada una de las copias era probada tantas veces como escenarios se definirían.

Además, dado que el programa realizado en este Trabajo Fin de Máster contiene una gran cantidad de casos de test cuyo tiempo de ejecución no es aceptable en el día a día (debido a la naturaleza de las aplicaciones de *High Performance Computing* y *Deep Learning* con tiempos de ejecución generalmente muy elevados), se concibieron las pruebas de carga de tal forma que se ejecutaría a diario una parte del total de casos de test y se analizarían los resultados de la ejecución en la siguiente jornada. Para ello, se requirió realizar ciertas modificaciones en el programa original que se detallan a continuación.

En primer lugar, la primera versión del programa estaba limitada a una ejecución simultánea por nodo y usuario. Esto implicaba que un mismo usuario no tenía permitido realizar dos o más ejecuciones concurrentes del programa desde el mismo nodo cliente. Esta restricción era debida a que en la versión original, los datos de la ejecución anterior eran movidos a un directorio de histórico, por lo que en caso de haber dos ejecuciones concurrentes, los datos de la ejecución que comenzara antes se moverían y el programa tendría errores de funcionamiento.

Así pues, la primera modificación consistió en otorgarle al programa la capacidad de ejecutar varias instancias desde un mismo nodo cliente y por parte del mismo usuario. Ello se consiguió modificando el programa para que usara un enlace simbólico llamado `CURRENT` apuntando siempre a la última ejecución finalizada, en lugar de emplear un directorio `CURRENT` y otro `PAST`. Además, se requirió modificar la parte del código que gestionaba los directorios y ficheros de cada ejecución, proporcionándole la capacidad de funcionar correctamente incluso en un entorno con múltiples ejecuciones concurrentes. Estas dos modificaciones le otorgaron al programa la capacidad de ser multiproceso, de tal forma que pudieran ser lanzadas múltiples instancias del programa por parte del mismo usuario, algo que posteriormente sería necesario pues las pruebas de carga se realizarían desde una única cuenta de usuario creada para tal efecto.

Además de la ampliación anterior, se consideró que la revisión de los resultados de las pruebas de carga de la jornada anterior podría resultar poco confortable desde el punto de vista del revisor dado que deberían consultarse tantos directorios de registro como ejecuciones totales se habían producido. Por lo tanto, el paradigma de ejecución del programa `autotests` cambiaba por completo. El programa original estaba diseñado desde el punto de vista de un usuario real que lanzaba ejecuciones en tiempo real, pero en esta ampliación se solicitaba que el programa también fuera capaz de ejecutar una batería de pruebas de forma desatendida y que la revisión de los resultados no resultara especialmente tediosa.

Por lo tanto, se consideró que la mejor opción era añadir una nueva funcionalidad al núcleo del programa que simplemente proporcionase, por la salida estándar y para cada caso de test fallido, las n últimas líneas de la salida de la aplicación ejecutada en ese caso de test en concreto. De esta forma, el revisor que se encargase de comprobar los resultados de la ejecución de la jornada anterior podría, con un simple vistazo a la salida del programa, y sin necesidad de acceder al sistema de ficheros, localizar el error simplemente viendo el contenido de las n últimas líneas. Esta funcionalidad la proporcionaría el parámetro `--ext_info` (*extended information*). Nótese que este comportamiento no era en absoluto excluyente con el comportamiento anterior. Es decir, los ficheros de registro que generaba el programa seguirían funcionando como siempre, pero en este caso, el revisor podría no tener la necesidad de consultarlos, ya que existiría una alta probabilidad de localizar la causa del fallo de la aplicación en las n últimas líneas de la aplicación que había fallado. Y en caso de que no fuera así, también se dotó al programa de la posibilidad de que el usuario introdujera manualmente por parámetro el número de líneas que deseaba mostrar por pantalla, a través del parámetro `--ext_info_num_lines`. La figura de código 5.13 muestra la función `showNLastLines()` que implementa esta nueva funcionalidad.

```
1  void showNLastLines(std::string failed_test_name, const unsigned int num_lines_to_show) {
2      std::string line;
3      std::string test_file =
           ↪  autotests_constants::path_tmp_autotests_user_hostname_timestamp + "/" +
           ↪  failed_test_name;
4      std::ifstream infile;
5      unsigned long long int lines_no = 0, start_line, end_line;
6      std::vector<std::string> current_failed_test_case_output_vector;
7
8      infile.open(test_file);
9
10     if (infile.is_open()) {
11         while (std::getline(infile, line)) {
12             lines_no++;
13             current_failed_test_case_output_vector.push_back(line);
14         }
15
16         end_line = lines_no;
17
18         if (num_lines_to_show < lines_no) {
19             start_line = end_line - num_lines_to_show;
20         }
21         else {
22             start_line = 3;
23         }
24
25         for (unsigned int i = start_line; i <
               ↪  current_failed_test_case_output_vector.size(); i++) {
26             std::cout << current_failed_test_case_output_vector.at(i) << std::endl;
27         }
28
29         std::cout << std::endl << std::endl;
30
31         infile.close();
32     }
33     else {
34         std::cout << "Autotests function showNLastLines() error -> Unable to open " <<
               ↪  test_file << std::endl;
35     }
36 }
```

Código fuente 5.13: Implementación de la función `showNLastLines()` en el programa `autotests`

5.2 Pruebas de carga

Con objeto de poner en marcha las pruebas de carga para el middleware rCUDA descritas con detalle en la sección 5.1.11, se estudiaron las opciones disponibles y se decidió que la forma más adecuada de llevarlo a cabo era implementar un shell script empleando el lenguaje interpretado Bash. La asignación de recursos se llevaría a cabo empleando una característica de rCUDA todavía en desarrollo consistente en un planificador de GPUs virtualizadas. El script se programaría mediante una tarea periódica y ejecutaría los casos de test que por su duración no son habitualmente ejecutados en el desarrollo del día a día.

Así pues, el script se encargaría, entre otras cosas, de gestionar las siguientes tareas:

1. Localizar y procesar la lista de usuarios dados de alta en el equipo de desarrollo de rCUDA
2. De éstos, procesar aquellos para los cuales habría que verificar su copia local de rCUDA
3. Localizar la lista de nodos disponibles en el clúster
4. De éstos, procesar y separar aquellos que disponen de GPU de aquellos que no, con objeto de poner en marcha servidores rCUDA
5. Lanzar el demonio del planificador interno de rCUDA con objeto de procurar recursos (nodos y GPU) cuando el shell script se los solicite
6. Lanzar tantos servidores rCUDA como nodos disponibles y usuarios haya que verificar (por cada usuario se lanza su versión particular del demonio rCUDA)
7. Para cada usuario objeto de verificación, asignar un nodo para el cliente de rCUDA de entre los disponibles
8. Lanzar el programa desarrollado en este TFM con objeto de verificar la copia local de rCUDA de cada usuario en los escenarios previamente definidos[3]
9. Copiar los ficheros de registro resultantes de la ejecución a una ubicación específica para ser consultados con mayor comodidad
10. Lanzar el programa desarrollado en este TFM con los restantes casos de test que no se han ejecutado durante la ejecución de los escenarios con la versión de rCUDA alojada en el repositorio compartido. Estos casos de test son aquellos que no están incluidos en el *sanity check*
11. Una vez finalizada la ejecución o expirado el tiempo máximo disponible (lo que suceda antes), se elimina todo rastro de ejecuciones de la memoria de los nodos implicados

5.3 Nuevos casos de test

A medida que los desarrolladores de rCUDA proporcionaban un inestimable feedback con objeto de añadir todas las nuevas funcionalidades que se han comentado en la sección 5.1, y simultáneamente con el desarrollo de dichas mejoras, se le iba proporcionando a la plataforma de verificación soporte para un conjunto cada vez mayor de aplicaciones.

Además, era fundamental dotar a la plataforma de soporte para los frameworks de Deep Learning más importantes del mercado, como era el caso de TensorFlow y Caffe, ya que los esfuerzos por hacer a rCUDA compatible con dichos frameworks se habían intensificado debido a la creciente popularidad de los mismos.

Así pues, y durante todo ese período de tiempo desde la versión inicial hasta la versión que está actualmente en producción, se fueron añadiendo más unidades de test para las aplicaciones y librerías `CuBLAS` [35], `Caffe` [23], `TensorFlow` [24], `GPU-LIBSVM` [52] y `nvGRAPH` [53], además de una nueva rama dentro del árbol de casos de test dedicada a casos de test sintéticos que iba generando directamente el equipo de desarrollo de rCUDA.

[3] Recuérdense estos escenarios como combinaciones específicas del hardware del clúster y los servidores de rCUDA.

Cabe destacar que la programación de los casos de test para los frameworks de Deep Learning resultó mucho más compleja que para el resto de casos, en especial los de Caffe. Si bien tanto este último como TensorFlow ponen a disposición del público diversos *samples* que incluyen problemas de Deep Learning conocidos, como son los casos de ImageNet [59], Cifar10 [60] y Mnist [61], la mayoría de estos *samples* están preparados para ser ejecutados por un solo usuario y de forma manual.

En el caso de TensorFlow, los *samples* consisten en scripts escritos en Python que descargan ciertos datos, los preparan, los usan para entrenar un modelo de red neuronal y prueban dicho modelo con unos datos ya clasificados. La ejecución de estos *samples* es sencilla, pues solo es necesario llamar al intérprete de Python y que éste se encargue de todo, con la excepción de los casos en los que hay conflicos con la ejecución multi-usuario, donde el programador debe modificar el script para evitar estos errores.

Sin embargo, para el caso de los *samples* de Caffe, estos vienen de serie implementados en forma de diversos shell scripts que deben ejecutarse secuencialmente y en orden y no son en absoluto compatibles con la ejecución multiusuario, ya que en este caso se producen múltiples errores debidos principalmente a problemas de permisos. Así pues, para añadir estos *samples* en la plataforma de verificación fue necesario generar desde cero nuevos shell scripts que corrigieran todos los problemas ocasionados por la ejecución multiusuario. Además, se añade la complicación de que estos shell scripts generados necesitarán ser reescritos para las futuras versiones de Caffe que sean introducidas en la plataforma de verificación.

5.4 Sanity Check

Uno de los enfoques más interesantes a la hora de utilizar la plataforma de verificación implementada en este trabajo era la posibilidad de emplearlo como un *Sanity Check*, esto es, una prueba o conjunto de pruebas efectuadas sobre un software con objeto de evaluar de forma rápida si se comporta como se espera. Por lo tanto, la idea consistió en definir un *Sanity Check* determinado con objeto de verificar de forma rápida el estado actual de una versión en particular de rCUDA.

Así pues, se definió un subconjunto del total de casos de test disponibles en el programa cuya ejecución debía realizarse y satisfacerse de forma obligatoria antes de efectuar un *commit* al repositorio de software común. De esta forma se garantizaba que las modificaciones efectuadas por el desarrollador en cuestión satisfacían los requisitos mínimos de funcionalidad. Esto era extremadamente útil desde el punto de vista de la calidad final del software, dado que permitía corroborar que los cambios efectuados no habían introducido colateralmente nuevos fallos.

Para implementar este *Sanity Check* se hizo uso de los *decorators* del framework Boost.-Test. En ese caso, se empleó el *decorator* `label` con nombre `SANITY_CHECK`, y se definió esta etiqueta en todos aquellos casos de test que debían formar parte de este *Sanity Check*. A partir de este momento, los desarrolladores de rCUDA tenían la posibilidad de realizar esta ejecución en el instante inmediatamente anterior a la subida de sus modificaciones al repositorio común, para cerciorarse, de esta forma, de que sus cambios introducidos no habían provocado nuevos errores desconocidos. La figura de código 5.14 muestra un caso de test con el *decorator* `label` definido como `SANITY_CHECK`.

```
BOOST_AUTO_TEST_CASE(tensorflow_mnist_small, * label("SANITY_CHECK")) {
    std::string final_complete_cmd_tensorflow_mnist_small =
       ↪  appendForwardingToCommand(complete_cmd_tensorflow_mnist_small,
       ↪  boost::unit_test::framework::current_test_case().p_name);
    std::string temporal_command = complete_cmd_tensorflow_mnist_small;
    tic(boost::unit_test::framework::current_test_case().p_name, temporal_command);
    BOOST_CHECK_EQUAL(testLaunch(boost::unit_test::framework::current_test_case().p_name,
       ↪  final_complete_cmd_tensorflow_mnist_small, test_size), success_value);
    toc(boost::unit_test::framework::current_test_case().p_name, temporal_command);
    storeTemporallyTestCasesOutput(
       ↪  boost::unit_test::framework::current_test_case().p_name);
    appendExecutionInfo(boost::unit_test::framework::current_test_case().p_name,
       ↪  complete_cmd_tensorflow_mnist_small);

    if (!boost::unit_test::results_collector.results(
       ↪  boost::unit_test::framework::current_test_case().p_id).passed()) {
        registerFailedTests(boost::unit_test::framework::current_test_case().p_name);
    }
}
```

Código fuente 5.14: Uso del *decorator* `label` con nombre `SANITY_CHECK` en la definición de un caso de test que ejecuta el *sample* Mnist del framework de Deep Learning TensorFlow

Al finalizar todas las modificaciones descritas a lo largo del capítulo 5, el programa diseñado e implementado en este Trabajo Fin de Máster alcanzó un número total de líneas de código de 21.490, distribuidas entre 1.717 líneas para el programa `autotests` y 19.773 líneas para `unit_tests`.

5.5 Descripción de la nueva interfaz y ejemplos de uso

En esta sección se describen algunos detalles sobre la interfaz de usuario con que cuenta la versión final del programa implementado en este Trabajo Fin de Máster. Nótese que para una descripción más exhaustiva de la interfaz, quizás el lector prefiera emplear el manual de usuario que se adjunta en este trabajo como apéndice A.

En primer lugar, el parámetro más importante de la aplicación, y sin el cual la misma aborta de forma intencionada, es el parámetro `rcuda_lib`. A través de este parámetro, el usuario pone a disposición de la aplicación su copia de los ficheros binarios de rCUDA con objeto de que el programa la utilice para ejecutar todos los casos de test solicitados. Así pues, el uso de este parámetro sería como el mostrado en el ejemplo 5.15.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
```

Ejemplo 5.15: Ejecución de `autotests` con el parámetro `rcuda_lib`

En este caso se ejecutarían todos los casos de test del programa, por lo que el tiempo de ejecución sería muy elevado. Si el usuario quisiera, sin embargo, seleccionar un subconjunto de casos de test, podría llevarlo a cabo con el parámetro `run_test`, que forma parte del framework Boost.Test y que ya se introdujo en la sección 3.5. El ejemplo 5.16 muestra una ejecución de `autotests` y su resultado especificando mediante el parámetro `run_test` que se desea ejecutar la suite de casos de test `cuda_samples_utilities` ubicada a su vez en la suite `cuda_samples`.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
   ↪  --run_test=cuda_samples/cuda_samples_utilities

Running 4 test cases...
Time elapsed for test deviceQuery: 0h 0m 0s 19ms
Time elapsed for test deviceQueryDrv: 0h 0m 0s 15ms
Time elapsed for test bandwidthTest: 0h 0m 1s 773ms
Time elapsed for test p2pBandwidthLatencyTest: 0h 0m 0s 339ms

*** No errors detected

Total elapsed time: 0h 0m 2s 203ms
```

Ejemplo 5.16: Ejecución de `autotests` con el parámetro `run_test`

Para localizar la lista de los casos de test y su posición exacta en el árbol de unidades de test, el usuario puede emplear el parámetro `list_content` que proporciona el framework Boost.Test, tal y como se muestra en el ejemplo 5.17.

```
$ autotests --list_content

cuda_samples*
    cuda_samples_simple*
        asyncAPI*
        cdpSimplePrint*
        cdpSimpleQuicksort*
        clock*
        cppIntegration*
        cppOverload*
        cudaOpenMP*
        fp16ScalarProduct*
        inlinePTX*
        matrixMul*
        matrixMulCUBLAS*
        matrixMulDrv*

      [...]

        systemWideAtomics*
        template_cuda*
        UnifiedMemoryStreams*
        vectorAdd*
        vectorAddDrv*
    cuda_samples_utilities*
        deviceQuery*
        deviceQueryDrv*
        bandwidthTest*

        [...]
```

Ejemplo 5.17: Ejecución de `autotests` con el parámetro `list_content`

Igualmente que en el caso anterior, el usuario puede listar el conjunto de etiquetas que han sido definidas en el programa a través del *decorator* `label`. El ejemplo 5.18 muestra dicha ejecución.

```
$ autotests --list_label
Available labels:
  BARRACUDA
  CAFFE
  CUBLAS
  CUDASW
  CUDA_MEME
  CUDA_SAMPLES
  CUDA_SAMPLES_WORKING
  GPU_BLAST
  GPU_LIBSVM
  GROMACS
  HPC_WORKING
  LAMMPS
  MAGMA
  NAMD
  NVGRAPH
  SANITY_CHECK
  TENSORFLOW
```

Ejemplo 5.18: Ejecución de `autotests` con el parámetro `list_label`

Si el usuario selecciona la versión *verbose* de una ejecución, el resultado sería el mostrado en la ejecución 5.19, donde se muestra la ejecución del caso de test `bandwidthtest`.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
   ↪  --run_test=cuda_samples/cuda_samples_utilities/bandwidthtest -v

Timestamp of this execution: D20180523_T173729

Running 1 test case...

Test bandwidthTest has begun at 17:37:29 on 23-05-2018
Test bandwidthTest has finished at 17:37:31 on 23-05-2018
Executed command: /usr/bin/rCUDA/TESTS/APPS/CUDA/8.0/samples/1_Utilities/
   ↪  bandwidthTest/bandwidthTest --device=all
Time elapsed for test bandwidthTest: 0h 0m 1s 805ms


*** No errors detected

Total elapsed time: 0h 0m 1s 837ms
```

Ejemplo 5.19: Ejecución la versión *verbose* de `autotests`

Tal y como se ha comentado en la subsección 5.4 relativa al Sanity Check, la existencia de esta etiqueta cobra una vital importancia en el desarrollo diario de rCUDA. En primer lugar, porque los usuarios tienen la posibilidad de ejecutar durante sus desarrollos una prueba automatizada que en poco menos de treinta minutos les confirma que los cambios efectuados sobre su copia del código cumplen con la calidad requerida. Y en segundo lugar, porque esta ejecución del Sanity Check manual no está completa: es necesario verificar cada copia del código fuente de rCUDA en todos sus escenarios. Esto, que resulta a todas luces inviable realizarlo de forma manual debido al tiempo de ejecución necesario, se ha incluído en las pruebas de carga nocturnas de tal forma que para cada usuario y para cada escenario, se ejecute el Sanity Check de forma desatendida.

Así pues, la ejecución del Sanity Check sobre una copia dada de rCUDA se realizaría como se muestra en el ejemplo 5.20. Puede comprobarse observando su salida que el

contenido del Sanity Check es ciertamente extenso, incluyendo en el momento de escribir estas líneas nada menos que 331 casos de test.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/ --run_test=@SANITY_CHECK

Running 331 test cases...
Time elapsed for test asyncAPI: 0h 0m 0s 371ms
Time elapsed for test cdpSimplePrint: 0h 0m 0s 271ms
Time elapsed for test cdpSimpleQuicksort: 0h 0m 0s 246ms
Time elapsed for test clock: 0h 0m 0s 191ms
Time elapsed for test cppIntegration: 0h 0m 0s 194ms
Time elapsed for test cppOverload: 0h 0m 0s 179ms
Time elapsed for test cudaOpenMP: 0h 0m 0s 184ms
Test fp16ScalarProduct disabled in runtime due to minimum compute capability not
   ↪  fulfilled
Time elapsed for test inlinePTX: 0h 0m 0s 185ms
Time elapsed for test matrixMul: 0h 0m 0s 332ms
Time elapsed for test matrixMulCUBLAS: 0h 0m 1s 286ms

[...]

Time elapsed for test tensorflow_cifar10_small: 0h 0m 18s 262ms
Time elapsed for test tensorflow_cifar10_multigpu_small: 0h 0m 16s 428ms
Time elapsed for test tensorflow_mnist_small: 0h 1m 8s 77ms
Time elapsed for test tensorflow_benchmark_small: 0h 1m 59s 432ms
Time elapsed for test tensorflow_inception_small: 0h 6m 0s 554ms
Time elapsed for test tensorflow_imagenet_small: 0h 0m 10s 737ms
Time elapsed for test caffe_mnist_small: 0h 0m 24s 882ms
Time elapsed for test caffe_cifar10_small: 0h 0m 43s 764ms
Time elapsed for test caffe_siamese_network_small: 0h 0m 49s 668ms

*** No errors detected

Total elapsed time: 0h 27m 37s 837ms
```

Ejemplo 5.20: Ejecución de ejecución del Sanity Check

Por otra parte, estos últimos años han estado marcados por la popularización masiva de las técnicas de Deep Learning y, por lo tanto, los usuarios de rCUDA han solicitado a sus desarrolladores proporcionar soporte para hacer a rCUDA compatible con los frameworks de Deep Learning más populares del mercado, como TensorFlow [24] y Caffe [23]. Dado que internamente estos frameworks trabajan de formas muy distintas a como lo hacen las aplicaciones del género HPC (principalmente haciendo uso de una gran cantidad de hilos de CPU), se experimentó la necesidad de modificar más del 80 % del código fuente de rCUDA para este cometido.

Así pues, y dado que este desarrollo llevó meses, surgió la necesidad de implementar en el programa `autotests` casos de test que ejecutaran estos frameworks de forma totalmente automática y desatendida, tal y como se ha explicado en la sección 5.3. El ejemplo 5.21 muestra una ejecución de `autotests` indicando a través del parámetro `run_test` que se desea ejecutar la suite completa de casos de test de TensorFlow junto con la suite completa de casos de test de Caffe (ambas en sus versiones *small*).

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
   ↪  --run_test=DL/DL_small/TENSORFLOW_small,CAFFE_small

Running 12 test cases...
Time elapsed for test tensorflow_multigpu_basics_small: 0h 0m 55s 439ms
Time elapsed for test tensorflow_alexnet_small: 0h 0m 11s 701ms
Time elapsed for test tensorflow_cifar10_small: 0h 1m 18s 565ms
Time elapsed for test tensorflow_cifar10_multigpu_small: 0h 1m 30s 228ms
```

```
Time elapsed for test tensorflow_mnist_small: 0h 1m 9s 476ms
Time elapsed for test tensorflow_benchmark_small: 0h 1m 59s 800ms
Time elapsed for test tensorflow_inception_small: 0h 5m 52s 672ms
Time elapsed for test tensorflow_imagenet_small: 0h 0m 17s 3ms
Time elapsed for test caffe_mnist_small: 0h 0m 29s 370ms
Time elapsed for test caffe_cifar10_small: 0h 0m 44s 18ms
Time elapsed for test caffe_siamese_network_small: 0h 0m 53s 407ms
Time elapsed for test caffe_imagenet_small: 0h 3m 22s 343ms

*** No errors detected

Total elapsed time: 0h 18m 44s 493ms
```

Ejemplo 5.21: Ejecución de todos los casos de test de TensorFlow y Caffe en sus versiones *small*

Si en la ejecución se producen fallos en algún caso de test, el resultado será el mostrado en el ejemplo 5.22, donde se ha empleado una suite de test específicamente diseñada para desarrollar y comprobar el correcto funcionamiento del programa de verificación. Esta suite está programada para que proporcione resultados de test conforme a lo que convenga en esa situación en concreto. Tal y como puede verse en el ejemplo, de cuatro casos de test ejecutados, dos de ellos han fallado. Esta información se muestra, en primer lugar, en tiempo real mientras se están ejecutando dichos casos de test y, en segundo lugar, cuando ya ha finalizado su ejecución mediante una lista de casos de test fallidos.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
   ↪  --run_test=tests_for_autotests

Time elapsed for test prueba_true1: 0h 0m 0s 27ms
Time elapsed for test prueba_true2: 0h 0m 0s 21ms
src/UNIT_TESTS/tests_for_test_autotests.cpp(52): error: in
   ↪  "tests_for_test_autotests/prueba_false1": check
   ↪  testLaunch(boost::unit_test::framework::current_test_case().p_name,
   ↪  final_complete_cmd_bandwidthTest_error, test_size) == success_value has failed [1
   ↪  != 0]
Time elapsed for test prueba_false1: 0h 0m 0s 20ms
src/UNIT_TESTS/tests_for_test_autotests.cpp(65): error: in
   ↪  "tests_for_test_autotests/prueba_false2": check
   ↪  testLaunch(boost::unit_test::framework::current_test_case().p_name,
   ↪  final_complete_cmd_bandwidthTest_error, test_size) == success_value has failed [1
   ↪  != 0]
Time elapsed for test prueba_false2: 0h 0m 0s 16ms

*** 2 failures are detected in the test module "RCUDA_TESTS"

List of failed test cases:

prueba_false1
prueba_false2

Total elapsed time: 0h 0m 0s 210ms
```

Ejemplo 5.22: Ejecución de `autotests` cuando se producen fallos en casos de test

Y si a la ejecución anterior se le añade el parámetro para obtener por pantalla las `n` últimas líneas de cada caso de test fallido, se obtiene lo mostrado en el ejemplo 5.23. Puede apreciarse en dicho ejemplo que a través de las líneas mostradas se ha podido localizar el error sin necesidad de acudir a los ficheros de registro ubicado en el sistema de ficheros compartido.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
   ↪  --run_test=tests_for_autotests --ext_info

Running 4 test cases...
Time elapsed for test prueba_true1: 0h 0m 0s 28ms
Time elapsed for test prueba_true2: 0h 0m 0s 20ms
src/UNIT_TESTS/tests_for_test_autotests.cpp(52): error: in
   ↪  "tests_for_test_autotests/prueba_false1": check
   ↪  testLaunch(boost::unit_test::framework::current_test_case().p_name,
   ↪  final_complete_cmd_bandwidthTest_error, test_size) == success_value has failed [1
   ↪  != 0]
Time elapsed for test prueba_false1: 0h 0m 0s 21ms
src/UNIT_TESTS/tests_for_test_autotests.cpp(65): error: in
   ↪  "tests_for_test_autotests/prueba_false2": check
   ↪  testLaunch(boost::unit_test::framework::current_test_case().p_name,
   ↪  final_complete_cmd_bandwidthTest_error, test_size) == success_value has failed [1
   ↪  != 0]
Time elapsed for test prueba_false2: 0h 0m 0s 20ms

*** 2 failures are detected in the test module "RCUDA_TESTS"

List of failed test cases:

prueba_false1

[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla K20m
Invalid mode - valid modes are quick, range, or shmoo
See --help for more information
Result = FAIL

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when
   ↪  GPU Boost is enabled.


prueba_false2

[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla K20m
Invalid mode - valid modes are quick, range, or shmoo
See --help for more information
Result = FAIL

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when
   ↪  GPU Boost is enabled.


Total elapsed time: 0h 0m 0s 172ms
```

Ejemplo 5.23: Ejecución de `autotests` cuando se producen fallos en casos de test ejecutándolo con el parámetro `ext_info` (*extended_information)*

Además de para servir de interfaz entre el usuario y el motor de testing, la capa de software `autotests` sirve, como ya se ha comentado, para proveer de una interfaz sencilla y homogénea para las aplicaciones a las que da soporte. Por ejemplo, muchas aplicaciones del género HPC admiten como información de entrada una serie de parámetros que en muchas ocasiones producen el mismo comportamiento en dichas aplicaciones pero sin embargo, cada una de ellas tiene una forma distinta de recibir esa información. El progra-

ma `autotests` aúna todos estos parámetros de esas aplicaciones en un solo parámetro, de tal forma que si el usuario utiliza ese parámetro, éste es *traducido* en tiempo de ejecución a la sintaxis original de cada aplicación. Así, por ejemplo, el parámetro `num_threads_omp` establece el número de hilos de CPU con que se ejecutarán las aplicaciones `CUDA-MEME`, `GPU-Blast`, `NAMD` y `GROMACS`, a pesar de que cada una de ellas tiene una sintaxis distinta para este parámetro. Igualmente, el parámetro de `autotests` `num_gpus` modifica el número de GPUs a utilizar por las aplicaciones `Cuda-SW++`, `MAGMA` y `TensorFlow`.

CAPÍTULO 6
Conclusiones

Como se ha visto a lo largo de esta memoria, en este Trabajo Fin de Máster se ha diseñado e implementado una plataforma de verificación automática para el middleware de virtualización remota de GPUs rCUDA empleando el framework de verificación Boost.Test. Con esta plataforma, los desarrolladores de rCUDA disponen de una potente herramienta que les permite automatizar la ejecución masiva de casos de test que les proporciona, tanto en tiempo real como en diferido, la información necesaria para acotar los posibles errores de funcionamiento. Además, las características propias del framework Boost.Test les proporciona mucha flexibilidad para seleccionar con precisión qué unidades de test deben ser ejecutadas en un momento dado. Si a esto se le añade la posibilidad de ejecutar pruebas de carga con objeto de probar el resto de escenarios cuya ejecución no es factible manualmente en el día a día, el espectro de funcionalidades probadas de rCUDA es extremadamente amplio.

La utilidad de esta herramienta les es tal, que los desarrolladores han cambiado por completo el paradigma mediante el cual proceden a lanzar y probar aplicaciones: mientras que antes localizaban, descargaban y empleaban los manuales de usuario de cada una de las aplicaciones integradas en la plataforma, con la existencia de ésta se limitan a listar las unidades de test disponibles y proceder a ejecutarlas en las combinaciones que más les conviene según sus necesidades actuales. Si por el contrario, únicamente necesitan verificar que sus últimas modificaciones sobre el código fuente de rCUDA satisfacen los requisitos mínimos de calidad, solamente deben ejecutar el programa empleando la etiqueta `SANITY_CHECK` y se ejecutará un número muy elevado de casos de test que les proporcionará la tranquilidad para subir sin preocupaciones sus modificaciones al repositorio compartido. En esencia, con el programa diseñado en este TFM se ha incrementado notoriamente el número de casos de test que los desarrolladores del middleware rCUDA pueden ejecutar por unidad de tiempo.

El programa diseñado e implementado en este TFM cuenta, en la versión final que se encuentra actualmente en producción, con 21.490 líneas de código que le otorgan toda su funcionalidad, incluyendo 67 suites de test únicas y 1.058 casos de test, y cuyo tiempo de desarrollo ha sido, desde la fase de análisis de requisitos hasta la puesta en marcha de la versión final, de aproximadamente 10 meses. A todo esto, el proyecto se mantiene actualmente vivo pues no deja de recibir cambios y mejoras, además de nuevos casos de test que le otorgan a la plataforma soporte para más aplicaciones. Esta versión final no podría haber existido sin la inestimable colaboración de todo el equipo de desarrollo de rCUDA, pues su predisposición para proponer mejoras fue crucial en el desarrollo de la versión que está siendo utilizada actualmente, cuya funcionalidad y manejabilidad en el día a día es extremadamente superior a las de la primera versión.

Por otra parte, el espectro de conocimientos adquiridos por el redactor de este trabajo sobre frameworks de verificación en general, y sobre Boost.Test en particular, ha sido significativamente amplio, teniendo en cuenta, además, que no se contaba con ningún tipo de experiencia previa en este campo. Así pues, la aplicación de éste o cualquier otro framework de verificación en cualquier proyecto de software futuro resultará mucho más ágil y eficiente. Además, colateralmente se ha adquirido experiencia en otros campos de la Ingeniería Informática, tales como el uso de aceleradores de cómputo basados en GPUs, usados en multitud de centros de datos a lo largo de todo el mundo, sin olvidar que esta nueva experiencia adquirida incluye la adquisición adicional de conocimientos en la programación, gestión y administración de clústeres de altas prestaciones y en la instalación y ejecución de aplicaciones usadas típicamente en dichos clústeres. Por todo ello, tras la realización de este Trabajo Fin de Máster, la empleabilidad del alumno se ha incrementado notablemente, siendo mucho más atractiva para empresas y organizaciones que requieren de ingenieros con tales conocimientos.

Finalmente, se ha descubierto que existen unas pocas aplicaciones a las que rCUDA da soporte pero no son integrables en la versión actual de la plataforma de verificación debido a que no siguen determinados estándares POSIX en cuyo funcionamiento se basa el motor de verificación implementado en este trabajo. Un claro ejemplo son las aplicaciones que no devuelven al proceso que las lanzan (típicamente una shell de comandos) un valor de cero cuando la ejecución se ha efectuado con éxito. Así pues, una posible línea de trabajo futuro consiste en implementar una segunda vía en el motor de verificación que tenga en cuenta las limitaciones de este tipo de aplicaciones. Además, la versión inicial de la plataforma de verificación implementada en este TFM se desarrolló para CUDA 8.0, esto es, la versión de CUDA para la cual rCUDA se encontraba proporcionando soporte en esos momentos. Sin embargo, desde entonces NVIDIA ha liberado las versiones de CUDA 9.0, 9.1 y 9.2, por lo que, con objeto de hacer posible la verificación de aplicaciones compiladas para tales versiones de CUDA, se hace necesario introducir en la plataforma de verificación el soporte para dichas aplicaciones, algo que ya se encuentra actualmente en desarrollo.

Bibliografía

[1] Wikipedia.org. *HPC*. [En línea]. Disponible en https://es.wikipedia.org/wiki/Computaci%C3%B3n_de_alto_rendimiento [Última consulta realizada el 28 de abril de 2018].

[2] Wikipedia.org. *Interfaz de paso de mensajes*. [En línea]. Disponible en https://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes [Última consulta realizada el 25 de abril de 2018].

[3] Nvidia Corporation. *Procesamiento paralelo con CUDA*. [En línea]. Disponible en http://www.nvidia.es/object/cuda-parallel-computing-es.html [Última consulta realizada el 25 de abril de 2018].

[4] Wikipedia.org. *OpenCL*. [En línea]. Disponible en https://es.wikipedia.org/wiki/OpenCL [Última consulta realizada el 25 de abril de 2018].

[5] OpenACC Developers. OpenACC. [En línea]. Disponible en https://www.openacc.org/ [Última consulta realizada el 25 de abril de 2018].

[6] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and remote GPUs perform similar with EDR 100G InfniBand. *Proceedings of the Industrial Track of the 16th International Middleware Conference, Middleware Industry 2015, Vancouver, BC, Canada, December 7-11, 2015*, páginas 4:1-4:7, 2015.

[7] Carlos Reaño and Federico Silla. A performance comparison of CUDA remote GPU virtualization frameworks. *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, páginas 488-489, 2015.

[8] Federico Silla, Sergio Iserte, Carlos Reaño, Javier Pradres. *On the benefits of the Remote GPU Virtualization Mechanism: the rCUDA Case* [En línea]. Disponible en https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4072 [Última consulta realizada el 26 de mayo de 2018].

[9] Sergio Iserte, Javier Prades, Carlos Reaño, Federico Silla. *Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm* [En línea]. Disponible en https://ieeexplore.ieee.org/document/7515675/ [Última consulta realizada el 28 de mayo de 2018].

[10] Linux KVM project. KVM. [En línea]. Disponible en https://www.linux-kvm.org/page/Main_Page [Última consulta realizada el 16 de junio de 2018].

[11] Universitat Politècnica de València. Departamento de Informática de Sistemas y Computadores. [En línea]. Disponible en http://www.upv.es/entidades/DISCA/index-es.html

[12] Universitat Politècnica de València. Grupo de Arquitecturas Paralelas. [En línea]. Disponible en http://www.gap.upv.es/

[13] Boost Developers. Boost.Test Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/index.html [Última consulta realizada el 25 de mayo de 2018].

[14] Boost Developers. Boost: C++ Libraries. [En línea]. Disponible en https://www.boost.org/ [Última consulta realizada el 25 de mayo de 2018].

[15] Wikipedia.org. *Infiniband*. [En línea]. Disponible en https://es.wikipedia.org/wiki/Macrodatos [Última consulta realizada el 24 de abril de 2018].

[16] Wikipedia.org. *Macrodatos*. [En línea]. Disponible en https://es.wikipedia.org/wiki/InfiniBand [Última consulta realizada el 24 de abril de 2018].

[17] Yamazaki I, Dong T, Solc R, Tomov S, Dongarra J, Schulthess T. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*. [Última consulta realizada el 24 de abril de 2018].

[18] Yuancheng Luo D. Canny edge detection on NVIDIA CUDA. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on, IEEE, 2008*. [Última consulta realizada el 24 de abril de 2018].

[19] Surkov V. Parallel option pricing with fourier space time-stepping method on graphics processing units. *Parallel Computing 2010*. [Última consulta realizada el 24 de abril de 2018].

[20] Agarwal PK, Hampton S, Poznanovic J, Ramanthan A, Alam SR, Crozier PS. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience 2013*. [Última consulta realizada el 24 de abril de 2018].

[21] Slurm Developers. Slurm. [En línea]. Disponible en https://slurm.schedmd.com/ [Última consulta realizada el 28 de abril de 2018].

[22] Wikipedia.org. *GPGPU*. [En línea]. Disponible en https://es.wikipedia.org/wiki/GPGPU [Última consulta realizada el 28 de abril de 2018].

[23] Caffe Developers. Caffe. [En línea]. Disponible en http://caffe.berkeleyvision.org/ [Última consulta realizada el 24 de mayo de 2018].

[24] TensorFlow Developers. TensorFlow: an open source machine learning framework for everyone [En línea]. Disponible en https://www.tensorflow.org/ [Última consulta realizada el 24 de mayo de 2018].

[25] Nvidia Corporation. cuSOLVER: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/cusolver/index.html [Última consulta realizada el 28 de mayo de 2018].

[26] Nvidia Corporation. cuBLASXT: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasXt-api [Última consulta realizada el 28 de mayo de 2018].

[27] Tyng-Yeu Liang, Yu-Wei Chang. *GridCuda: A Grid-Enabled CUDA Programming Toolkit*. [En línea]. Disponible en https://ieeexplore.ieee.org/document/5763452/ [Última consulta realizada el 27 de abril de 2018].

[28] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura. *DS-CUDA: A middleware to use many GPUs in the Cloud Environment.* [En línea]. Disponible en https://ieeexplore.ieee.org/document/6495928/ [Última consulta realizada el 27 de abril de 2018].

[29] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, Giuseppe Coviello. *A GPGPU Transparent Virtualization Component for High Performance Computing Clouds.* [En línea]. Disponible en https://link.springer.com/chapter/10.1007/978-3-642-15277-1_37 [Última consulta realizada el 27 de abril de 2018].

[30] Lin Shi, Hao Chen, Jianhua Sun. *vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines.* [En línea]. Disponible en https://ieeexplore.ieee.org/document/5928326/ [Última consulta realizada el 27 de abril de 2018].

[31] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, Parthasarathy Ranganathan. *GVim: GPU-accelerated Virtual Machines*. [En línea]. Disponible en https://tolia.org/files/pubs/hpcvirt2009.pdf [Última consulta realizada el 28 de abril de 2018].

[32] Nvidia Corporation. CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/ [Última consulta realizada el 26 de mayo de 2018].

[33] Alexander M. Merritt, Vishankha Gupta, Abhishek Verma, Ada Gavrilovska, Karsten Schwan. *Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies* [En línea]. Disponible en https://www.cc.gatech.edu/home/ada/papers/vtdc11.pdf [Última consulta realizada el 27 de abril de 2018].

[34] Nvidia Corporation. cuFFT: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/cufft/index.html [Última consulta realizada el 5 de mayo de 2018].

[35] Nvidia Corporation. cuBLAS: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/cublas/index.html [Última consulta realizada el 5 de mayo de 2018].

[36] Nvidia Corporation. cuSPARSE: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/cusparse/index.html [Última consulta realizada el 5 de mayo de 2018].

[37] Mellanox Technologies. *Mellanox OFED for Linux. User Manual.* [En línea]. Disponible en http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4.0.pdf [Última consulta realizada el 7 de mayo de 2018].

[38] CppUnit Developers. CppUnit - C++ port of JUnit. [En línea]. Disponible en https://sourceforge.net/projects/cppunit/ [Última consulta realizada el 2 de mayo de 2018].

[39] Nano Cpp Unit Developers. Nano Cpp Unit. [En línea]. Disponible en http://wiki.c2.com/?NanoCppUnit [Última consulta realizada el 2 de mayo de 2018].

[40] Unit++ Developers. The Unit++ Testing Framework. [En línea]. Disponible en http://unitpp.sourceforge.net/ [Última consulta realizada el 2 de mayo de 2018].

[41] CxxTest Developers. CxxTest. [En línea]. Disponible en https://cxxtest.com/ [Última consulta realizada el 2 de mayo de 2018].

[42] GoogleT Test Developers. Github: GoogleTest. [En línea]. Disponible en https://github.com/google/googletest [Última consulta realizada el 2 de mayo de 2018].

[43] Wikipedia.org. *JUnit*. [En línea]. Disponible en https://es.wikipedia.org/wiki/JUnit [Última consulta realizada el 18 de junio de 2018].

[44] BarraCUDA Developers, University of Cambridge. The BarraCUDA Project. [En línea]. Disponible en http://seqbarracuda.sourceforge.net/ [Última consulta realizada el 25 de abril de 2018].

[45] CUDA-MEME/mCUDA-MEME Developers. CUDA-MEME/mCUDA-MEME - Ultrafast Motif Discovery Using GPU Computing. [En línea]. Disponible en https://cuda-meme.sourceforge.io/homepage.htm#latest [Última consulta realizada el 27 de mayo de 2018].

[46] CUDASW++ Developers. CUDASW++ - GPU accelerated Smith Waterman algorithm. [En línea]. Disponible en http://cudasw.sourceforge.net/homepage.htm#latest [Última consulta realizada el 27 de mayo de 2018].

[47] GPU-Blast Developers. GPU-Blast, The GPU Basic Local Alignment Search Tool. [En línea]. Disponible en http://archimedes.cheme.cmu.edu/?q=gpublast [Última consulta realizada el 27 de mayo de 2018].

[48] Gromacs Developers. Gromacs. Fast. Flexible. Free. [En línea]. Disponible en http://www.gromacs.org/ [Última consulta realizada el 27 de mayo de 2018].

[49] Lammps Developers. Lammps Documentation. [En línea]. Disponible en http://lammps.sandia.gov/doc/Manual.html [Última consulta realizada el 27 de mayo de 2018].

[50] Magma Developers. Magma: Matrix Algebra on GPU and Multicore Architectures. [En línea]. Disponible en http://icl.cs.utk.edu/magma/ [Última consulta realizada el 27 de mayo de 2018].

[51] NAMD Developers. NAMD: Scalable Molecular Dynamics. [En línea]. Disponible en http://www.ks.uiuc.edu/Research/namd/ [Última consulta realizada el 27 de mayo de 2018].

[52] GPU-LIBSVM Developers. GPU-LIBSVM: LIBSVM Accelerated with GPU using the CUDA Framework. [En línea]. Disponible en http://mklab.iti.gr/project/GPU-LIBSVM [Última consulta realizada el 27 de mayo de 2018].

[53] Nvidia Corporation. nvGRAPH: CUDA Toolkit Documentation. [En línea]. Disponible en https://docs.nvidia.com/cuda/nvgraph/index.html [Última consulta realizada el 27 de mayo de 2018].

[54] Boost Developers. Boost.Program_options Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/doc/html/program_options.html [Última consulta realizada el 27 de mayo de 2018].

[55] Boost Developers. Boost.Date_Time Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/doc/html/date_time.html [Última consulta realizada el 27 de mayo de 2018].

[56] Boost Developers. Boost.Chrono Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/doc/html/chrono.html [Última consulta realizada el 27 de mayo de 2018].

[57] Boost Developers. Boost.Filesystem Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/libs/filesystem/doc/index.htm [Última consulta realizada el 27 de mayo de 2018].

[58] Boost Developers. Boost.Interprocess Library. [En línea]. Disponible en https://www.boost.org/doc/libs/1_63_0/doc/html/interprocess.html [Última consulta realizada el 27 de mayo de 2018].

[59] Stanford University, Princeton University. Imagenet. [En línea]. Disponible en http://www.image-net.org/ [Última consulta realizada el 24 de junio de 2018].

[60] Alex Krizhevsky. The Cifar10 dataset. [En línea]. Disponible en https://www.cs.toronto.edu/~kriz/cifar.html [Última consulta realizada el 24 de junio de 2018].

[61] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. The Mnist Database of handwritten digits. [En línea]. Disponible en http://yann.lecun.com/exdb/mnist/ [Última consulta realizada el 24 de junio de 2018].

APÉNDICE A
Manual de usuario

En las siguientes páginas se incluye el manual de usuario de la herramienta de verificación desarrollada en este Trabajo Fin de Máster.

Manual de usuario para la aplicación de Tests Unitarios de rCUDA

rCUDA Team

4 de julio de 2018

Resumen

Este documento consiste en un manual de usuario para utilizar la aplicación de tests unitarios para rCUDA. El objetivo es que el usuario conozca la forma de ejecución de la misma, así como los casos de test que es posible ejecutar.

Índice

1. Ejecución

1.1. Preparación del entorno

Para que este programa funcione correctamente es necesario que el usuario establezca manualmente en su variable de entorno `LD_LIBRARY_PATH` la ruta a las librerías de rCUDA, de tal forma que el programa sea capaz de obtener correctamente toda la información que necesita de las GPUs lógicas disponibles a través de las correspondientes llamadas a la API de CUDA.

El programa principal contiene un fichero ejecutable llamado `autotests`. Por lo tanto, se sugiere al usuario la inclusión del directorio donde está ubicado este ejecutable en la variable de entorno `PATH` para que la ejecución del programa resulte más cómoda. Además, nótese que algunas aplicaciones incluídas en `autotests` generan ficheros de log como resultado de sus ejecuciones, y estos ficheros suelen almacenarse en el mismo directorio desde el que se lanza la aplicación. Al estar protegido contra escritura el directorio donde se ubica el ejecutable autotests, estas aplicaciones podrían fallar. Así pues, se sugiere al usuario la ejecución de `autotests` desde un directorio ubicado en su `home` en el que resulte factible que una aplicación pueda generar y almacenar debidamente tantos ficheros de log como necesite.

1.2. Parámetro `--help`

Es posible ver las opciones que ofrece este programa mediante el parámetro `--help` o `-h`:

```
$ autotests --help

Allowed options:
  -h [ --help ]            Shows help message.
  -v [ --verbose ]         Verbose version
  --rcuda_lib arg          Path to CUDA or rCUDA lib path. Mandatory argument
  --list_content           Lists the tests units, their organization in the
                           test tree, their enabled/disabled state, etc
  --list_labels            Lists the labels defined in the application
  --build_info             Instructs the framework to display library build
                           information prior to show execution information
  --run_test arg           Tests to run with UNIT_TESTS_rCUDA
  --log_level arg          Log level. Default is error. Others are all,
                           test_suite, message, warning, cpp_exception,
                           system_error, fatal_error and nothing
  --random                 Run the tests cases in random order.
  --log_sink arg           Name of the file which will be used to store the log
  --report_sink arg        Name of the file which will be used to store the
                           report
  --report_level arg       Level of verbosity of the testing result report.
                           Default is confirm. Others are no, short and
                           detailed
  --log_format arg         Allows the user to set the framework log format to
                           one of the formats supplied. Default is HRF. Others
                           are XML and JUNIT
  --report_format arg      Allows the user to set the framework report format
                           to one of the formats supplied. Default is HRF.
                           Other is XML
  --output_format arg      Combines an effect of report_format and log_format
                           parameter. This parameter does not have a default
                           value. Options are HRF and XML
  --gpu_num arg            Number of the GPU to use in CUDA-MEME
  --num_threads_omp arg    Number of threads to use in CUDA-MEME, GPU-BLAST,
```

```
                          NAMD and GROMACS
  --num_gpus arg          Number of GPUs to use in CUDASW, MAGMA and
                          TENSORFLOW on samples MultiGPU basics, Cifar10
                          multiGPU, Benchmark and Inception
  --ntmpi arg             Number of threads MPI for GROMACS
  --nsteps arg            Number of steps to run for GROMACS
  --num_procs arg         Number of MPI processes for LAMMPS
  --tf_version arg        TENSORFLOW version, default is 1.2.0. Other options
                          are 0.12.1, 1.0.0, 1.0.1, 1.1.0, 1.2.0, 1.2.1,
                          1.3.0, 1.4.0 and 1.5.0
  --batch_size arg        Batch size for TENSORFLOW Alexnet, Benchmark,
                          Cifar10, Cifar10 multiGPU and Inception samples
  --num_batches arg       Number of batches for TENSORFLOW Alexnet and
                          Benchmark samples
  --max_steps arg         Max steps for TENSORFLOW Cifar10, Cifar10 multiGPU
                          and Inception samples
  --comp arg              Computation for TENSORFLOW MultiGPU Basics sample
  --model arg             Model to use in TENSORFLOW Benchmark sample,
                          available models are resnet50, inception3, vgg16 and
                          alexnet
  --use_nccl arg          Use or not NCCL in TENSORFLOW Benchmark sample, true
                          or false
  --timeout arg           Timeout in minutes to kill a running test
  --caffe_gpus arg        Indexes of GPUs separated by commas to use in
                          CAFFE, default is all
  --ext_info              Extended failed test information. This option will
                          show some more information in standard output for
                          each failed test case
  --num_lines_ext_info arg Amount of lines to show with the ext_info option.
                          Default is 20
```

1.3. Parámetro `--verbose`

El parámetro `--verbose` o `-v` permite al usuario aumentar la cantidad de información proporcionada por la salida estándar durante la ejecución del programa. Así, con la opción verbose el usuario obtiene la fecha y la hora de inicio y de fin de cada test ejecutado, el comando que se ha ejecutado en cada caso de test y el tiempo empleado en la ejecución de cada caso de test.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_utilities/deviceQuery -v

Running 1 test case...

Test deviceQuery has begun at 10:30:05 on 20-04-2018
Test deviceQuery has finished at 10:30:05 on 20-04-2018
Executed command:
  /usr/bin/rCUDA/TESTS/APPS/CUDA/8.0/samples/1_Utilities/deviceQuery/
  deviceQuery
Time elapsed for test deviceQuery: 0h 0m 0s 27ms


*** No errors detected
Total elapsed time: 0h 0m 0s 97ms
```

1.4. Parámetro `--rcuda_lib`

El parámetro `--rcuda_lib` es obligatorio, con el que el usuario deberá especificar la ruta al directorio `lib` de CUDA o rCUDA:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
```

Con el ejemplo anterior, se producirá una ejecución del programa `autotests` con la totalidad de los casos de tests, lo que llevará mucho tiempo.

1.5. Parámetro `--list_content`

El parámetro `--list_content` permite al usuario visualizar la totalidad de los tests unitarios, su organización jerárquica y su estado activado o desactivado. Por defecto están todos activados, por lo que todos se muestran con un asterisco (*) al final del nombre:

```
$ autotests --list_content

cuda_samples*
    cuda_samples_simple*
        asyncAPI*
        cdpSimplePrint*
        cdpSimpleQuicksort*
        clock*
        cppIntegration*
        cppOverload*
        cudaOpenMP*
        inlinePTX*
        matrixMul*
        matrixMulCUBLAS*
    cuda_samples_utilities*
        deviceQuery*
        deviceQueryDrv*
        bandwidthTest*
        p2pBandwidthLatencyTest*
HPC*
    HPC_large*
        BARRACUDA_large*
            barracuda_dataset3_test1*
            barracuda_dataset2_test1*
            ...
```

1.6. Parámetro `--list_labels`

El parámetro `--list_labels` permite al usuario visualizar la totalidad de las etiquetas que puede utilizar en la ejecución del programa (véase sección 1.8 para más información acerca de las etiquetas):

```
$ autotests --list_labels

Available labels:
  BARRACUDA
  CAFFE
  CUBLAS
  CUDASW
```

```
CUDA_MEME
CUDA_SAMPLES
CUDA_SAMPLES_WORKING
GPU_BLAST
GPU_LIBSVM
GROMACS
HPC_WORKING
LAMMPS
MAGMA
NAMD
NVGRAPH
SANITY_CHECK
TENSORFLOW
```

1.7. Parámetro `--build_info`

El parámetro `--build_info` muestra al inicio de una ejecución del programa datos tales como la plataforma usada, el compilador y la versión de la librería de Boost. Este parámetro no afecta en modo alguno al resultado de la ejecución del programa.

1.8. Parámetro `--run_test`

Para seleccionar manualmente los casos de test que deben ejecutarse, debe usarse el parámetro `--run_test`:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small
```

De esta manera, se ejecutará únicamente la suite de casos de tests `HPC_small`, que a su vez forma parte de la suite `HPC`.

Es posible ejecutar dos suites de tests de forma simultánea si pertenecen a la misma suite de tests. En el siguiente ejemplo, los casos de tests `barracuda_dataset1_test1` y `barracuda_dataset1_test2` forman parte de la suite de tests `BARRACUDA_small`, por lo que pueden ejecutarse así:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/BARRACUDA_small/barracuda_dataset1_test1,
  barracuda_dataset1_test2
```

Si los casos de tests no forman parte de la misma suite, es posible ejecutarlos simultáneamente empleando el carácter dos puntos (:):

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/BARRACUDA_small/barracuda_dataset1_test1:cuda_samples
```

Además, es posible deshabilitar un test en tiempo de ejecución mediante el operador cierre de exclamación (!). No obstante, la inclusión de este operador en un terminal confunde al intérprete ya que el operador (!) se usa para volver atrás en el historial de comandos. Con objeto de que funcione dicho operador en este programa, es necesario deshabilitar el histórico de comandos en la terminal en la que se está ejecutando el programa, y esto es posible mediante la orden `set +H` (únicamente es necesario deshabilitarlo una vez por cada terminal). En el siguiente ejemplo, se ejecutará la suite de tests `cuda_samples` omitiendo el test `clock`:

```
$ set +H
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples:!cuda_samples/cuda_samples_simple/clock
```

Por otra parte, es posible utilizar etiquetas de forma que el usuario pueda realizar ejecuciones de todos los casos de tests pertenecientes a una única aplicación, empleando para ello el carácter arroba (@):

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/ --run_test=@NAMD
```

También es posible combinar ambos modos de ejecución, es decir, ejecutar una suite de tests simultáneamente con un conjunto de tests definidos mediante una etiqueta:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_simple:@NVGRAPH
```

1.9. Parámetro `--log_level`

El parámetro `--log-level` permite al usuario seleccionar la forma en que desea visualizar la ejecución de los casos de tests. Las opciones son las siguientes:

- `all`: proporciona todos los mensajes
- `success`: lo mismo que `all`
- `test_suite`: muestra los mensajes relativos a las suites de tests
- `warning`: muestra warnings emitidos por el usuario desde el código fuente
- `error`: muestra todos los mensajes de error. Opción por defecto
- `cpp_exception`: proporciona las excepciones de C++ no tratadas
- `system_error`: muestra errores no fatales originados por el sistema operativo, tales como timeouts o excepciones de coma flotante
- `fatal_error`: muestra errores fatales del sistema operativo o del usuario, por ejemplo violaciones de acceso a memoria
- `nothing`: no muestra nada

Por ejemplo, la siguiente ejecución mostrará mensajes relativos a las suites de tests a medida que la ejecución pasa por todas ellas:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/GROMACS_small --log_level=test_suite

Running 2 test cases...
Entering test module "RCUDA_TESTS"
src/UNIT_TESTS/hpc_barracuda_large.cpp(19): Entering test suite "HPC"
src/UNIT_TESTS/hpc_barracuda_small.cpp(22): Entering test suite "HPC_small"
src/UNIT_TESTS/hpc_gromacs_small.cpp(23): Entering test suite "GROMACS_small"
src/UNIT_TESTS/hpc_gromacs_small.cpp(25): Entering test case
  "GROMACS_gpu_sample_small1"
Time elapsed for test GROMACS_gpu_sample_small1: 0h 0m 2s 357ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(25): Leaving test case
  "GROMACS_gpu_sample_small1"; testing time: 1620ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(39): Entering test case
  "GROMACS_gpu_nvml_sample_small1"
```

```
Time elapsed for test GROMACS_gpu_nvml_sample_small1: 0h 0m 3s 784ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(39): Leaving test case
  "GROMACS_gpu_nvml_sample_small1"; testing time: 2560ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(23): Leaving test suite "GROMACS_small";
  testing time: 4179999us
src/UNIT_TESTS/hpc_barracuda_small.cpp(22): Leaving test suite "HPC_small";
  testing time: 4179999us
src/UNIT_TESTS/hpc_barracuda_large.cpp(19): Leaving test suite "HPC"; testing
  time: 4179999us
Leaving test module "RCUDA_TESTS"; testing time: 4179999us

*** No errors detected

Total elapsed time: 0h 0m 6s 234ms
```

1.10. Parámetro `--random`

El parámetro `--random` permite que los casos de tests sean ejecutados en un orden distinto a como han sido declarados en el código fuente:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/GROMACS_small --random

Running 2 test cases...
Time elapsed for test GROMACS_gpu_nvml_sample_small: 4491 ms
Time elapsed for test GROMACS_gpu_sample_small: 4552 ms

*** No errors detected
```

1.11. Parámetro `--log_sink`

El parámetro `--log_sink` permite al usuario establecer un fichero para almacenar la información del log. No debe confundirse la información del log con la salida de la aplicación propiamente dicha. La información del log es simplemente la salida que proporciona Boost al establecer el parámetro `--log_level`. En caso de no proporcionarse este parámetro, se establecerá la opción `--log_level=error` por defecto:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/GROMACS_small --log_sink=./log --log_level=test_suite

Running 2 test cases...
Time elapsed for test GROMACS_gpu_nvml_sample_small: 4491 ms
Time elapsed for test GROMACS_gpu_sample_small: 4552 ms

*** No errors detected

$ cat ./log

Running 2 test cases...
Entering test module "RCUDA_TESTS"
src/UNIT_TESTS/hpc_barracuda_large.cpp(19): Entering test suite "HPC"
src/UNIT_TESTS/hpc_barracuda_small.cpp(22): Entering test suite "HPC_small"
src/UNIT_TESTS/hpc_gromacs_small.cpp(23): Entering test suite "GROMACS_small"
```

```
src/UNIT_TESTS/hpc_gromacs_small.cpp(25): Entering test case
  "GROMACS_gpu_sample_small1"
src/UNIT_TESTS/hpc_gromacs_small.cpp(25): Leaving test case
  "GROMACS_gpu_sample_small1"; testing time: 1480ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(39): Entering test case
  "GROMACS_gpu_nvml_sample_small1"
src/UNIT_TESTS/hpc_gromacs_small.cpp(39): Leaving test case
  "GROMACS_gpu_nvml_sample_small1"; testing time: 1480ms
src/UNIT_TESTS/hpc_gromacs_small.cpp(23): Leaving test suite "GROMACS_small";
  testing time: 2960ms
src/UNIT_TESTS/hpc_barracuda_small.cpp(22): Leaving test suite "HPC_small";
  testing time: 2960ms
src/UNIT_TESTS/hpc_barracuda_large.cpp(19): Leaving test suite "HPC"; testing
  time: 2960ms
Leaving test module "RCUDA_TESTS"; testing time: 2960ms
```

1.12. Parámetro `--report_sink`

El parámetro `--report_sink` es similar al anterior, solo que en lugar de cambiar la salida para el log de la ejecución, esta vez se cambia la salida para el informe final, esto es, la lista de casos de tests que se han ejecutado con éxito.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=HPC/HPC_small/GROMACS_small --report_sink=./report

Running 2 test cases...
Time elapsed for test GROMACS_gpu_sample_small: 3905 ms
Time elapsed for test GROMACS_gpu_nvml_sample_small: 4272 ms

$ cat ./report

*** No errors detected
```

1.13. Parámetro `--report_level`

El parámetro `--report_level` permite al usuario establecer el nivel de verbosidad del informe final de la ejecución del programa. Las opciones disponibles son `confirm` (opción por defecto), `no`, `short` y `detailed`:

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_utilities --report_level=detailed

Running 4 test cases...
Time elapsed for test deviceQuery: 29 ms
Time elapsed for test deviceQueryDrv: 23 ms
Time elapsed for test bandwidthTest: 2 s
Time elapsed for test p2pBandwidthLatencyTest: 539 ms

Test module "RCUDA_TESTS" has passed with:
  4 test cases out of 4 passed
  4 assertions out of 4 passed

  Test suite "cuda_samples" has passed with:
```

```
    4 test cases out of 4 passed
    4 assertions out of 4 passed

    Test suite "cuda_samples/cuda_samples_utilities" has passed with:
      4 test cases out of 4 passed
      4 assertions out of 4 passed

      Test case "cuda_samples/cuda_samples_utilities/deviceQuery" has passed
        with:
        1 assertion out of 1 passed

      Test case "cuda_samples/cuda_samples_utilities/deviceQueryDrv" has passed
        with:
        1 assertion out of 1 passed

      Test case "cuda_samples/cuda_samples_utilities/bandwidthTest" has passed
        with:
        1 assertion out of 1 passed

      Test case "cuda_samples/cuda_samples_utilities/p2pBandwidthLatencyTest"
        has passed with:
        1 assertion out of 1 passed


Total elapsed time: 2 s
```

1.14. Parámetro `--log_format`

El parámetro `--log_format` permite al usuario establecer el formato del log del programa. Las opciones disponibles son `HRF` (opción por defecto), `XML` y `JUNIT`. HRF consiste en el "human readable format", mientras que XML y JUNIT facilitan la automatización del procesado de la salida.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_utilities --log_format=XML

<TestLog>Time elapsed for test deviceQuery: 21 ms
Time elapsed for test deviceQueryDrv: 18 ms
Time elapsed for test bandwidthTest: 1 s
Time elapsed for test p2pBandwidthLatencyTest: 502 ms
</TestLog>
*** No errors detected

Total elapsed time: 2 s
```

1.15. Parámetro `--report_format`

El parámetro `--report_format` permite al usuario establecer el formato del informe final del programa. Las opciones disponibles son `HRF` (opción por defecto) y `XML`.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_utilities --report_format=XML
```

```
Running 4 test cases...
Time elapsed for test deviceQuery: 26 ms
Time elapsed for test deviceQueryDrv: 20 ms
Time elapsed for test bandwidthTest: 1 s
Time elapsed for test p2pBandwidthLatencyTest: 471 ms
<TestResult><TestSuite name="RCUDA_TESTS" result="passed" assertions_passed="4"
  assertions_failed="0" warnings_failed="0" expected_failures="0"
  test_cases_passed="4" test_cases_passed_with_warnings="0"
  test_cases_failed="0" test_cases_skipped="0"
  test_cases_aborted="0"></TestSuite></TestResult>
Total elapsed time: 2 s
```

1.16. Parámetro `--output_format`

El parámetro `--output_format` combina las opciones `--log_format` y `--report_format`. Este parámetro no tiene un valor por defecto. Las opciones disponibles son `HRF` y `XML`.

```
$ autotests --rcuda_lib=/home/usuario/rcuda/sources/rCUDA/lib/
  --run_test=cuda_samples/cuda_samples_utilities --output_format=XML

<TestLog>Time elapsed for test deviceQuery: 29 ms
Time elapsed for test deviceQueryDrv: 22 ms
Time elapsed for test bandwidthTest: 1 s
Time elapsed for test p2pBandwidthLatencyTest: 505 ms
</TestLog><TestResult><TestSuite name="RCUDA_TESTS" result="passed"
  assertions_passed="4" assertions_failed="0" warnings_failed="0"
  expected_failures="0" test_cases_passed="4"
  test_cases_passed_with_warnings="0" test_cases_failed="0"
  test_cases_skipped="0" test_cases_aborted="0"></TestSuite></TestResult>
Total elapsed time: 2 s
```

1.17. Parámetro `--ext_info`

El parámetro `--ext_info` proporciona las últimas líneas de la salida de cada caso de test fallido por la salida estándar junto al nombre completo del caso de test. Por defecto, el número de líneas mostrado es 20.

1.18. Parámetro `--ext_info_num_lines`

El parámetro `--ext_info_num_lines` sobreescribe el número de líneas mostradas al nuevo valor proporcionado con este parámetro.

1.19. Parámetros específicos de aplicaciones

El programa de tests admite una serie de parámetros para controlar la ejecución de las aplicaciones a las que da soporte. A continuación se detallan estos parámetros:

- `gpu_num`: especifica el identificador de la GPU a usar en la aplicación CUDA-MEME
- `num_threads_omp`: especifica el número de threads a usar en las aplicaciones CUDA-MEME (parámetro num_threads en la aplicación), GPU-BLAST (parámetro num_threads en la aplicación) y GROMACS (parámetro ntomp en la aplicación). Si el usuario no establece este parámetro, se ejecutarán dos casos base, el primero con un valor de 1 y el segundo con un valor de 12.

- `num_gpus`: especifica el número de GPUs a usar en las aplicaciones CUDA-SW (parámetro num_gpus), MAGMA (parámetro ngpu) y TensorFlow en sus samples MultiGPU basics, Cifar10 multiGPU, Benchmark e Inception. En los casos de CUDA-SW y MAGMA, si el usuario no establece este parámetro, el programa calculará el número de GPUs virtuales disponibles en el cliente y ejecutará un caso de test para cada combinación de GPUs. En el caso de TensorFlow, se ejecutarán los samples con el máximo número de GPUs disponibles.
- `ntmpi`: especifica el número de threads MPI de GROMACS.
- `nsteps`: especifica el número de steps a ejecutar en GROMACS.
- `num_procs`: especifica el número de procesos MPI en LAMMPS (parámetro np). Si el usuario no establece este parámetro, se ejecutarán dos casos base, el primero con un valor de 1 y el segundo con un valor de 12.
- `tf_version`: especifica la versión de TensorFlow que el usuario desea ejecutar. Las opciones son 0.12.1, 1.0.0, 1.0.1, 1.1.0, 1.2.0, 1.2.1, 1.3.0, 1.4.0 y 1.5.0. La versión de CUDNN requerida con cada versión de TensorFlow es automáticamente establecida en la variable de entorno `LD_LIBRARY_PATH`.
- `batch_size`: tamaño de lote (cantidad de imágenes a procesar por iteración) para los samples de TensorFlow Alexnet, Benchmark, Cifar10, Cifar10 multiGPU e Inception.
- `num_batches`: Número de lotes a ejecutar en los samples de TensorFlow Alexnet y Benchmark.
- `max_steps`: número de iteraciones a ejecutar en los samples de TensorFlow Cifar10, Cifar10 multiGPU e Inception.
- `comp`: tamaño de problema en el sample de TensorFlow multiGPU Basics.
- `model`: modelo a utilizar en el sample de TensorFlow Benchmark. Las opciones disponibles son `resnet50`, `inception3`, `vgg16` y `alexnet`.
- `use_nccl`: parámetro para controlar si el usuario desea utilizar la librería `NCCL` en el sample de TensorFlow Benchmark. Las opciones disponibles son `True` y `False`.
- `timeout`: parámetro para indicar el tiempo de ejecución máximo en minutos permitido para cada caso de test. Si no se establece este parámetro, el programa define un timeout por defecto para cada caso de test en función del tamaño del mismo.
- `caffe_gpus`: parámetro para controlar las GPUs a usar en CAFFE. Se trata de una lista de índices de GPUs separados por comas. Para usar todas las GPUs disponibles, se puede usar 'all'.

2. Verificación de resultados

2.1. Tiempos de ejecución

El programa `autotests` calcula, para cada caso de test ejecutado, el tiempo de ejecución del mismo. Este tiempo es proporcionado, en primer lugar, por la salida estándar, y en segundo lugar, en el fichero de registro ubicado en el directorio correspondiente, que se detallará en la siguiente sección.

2.2. Ficheros de registro actuales

El programa `autotests` redirige la salida estándar y de error de cada caso de test a un fichero de registro ubicado en la ruta `/usr/bin/rCUDA/TESTS/LOGS/username/node/timestamp`, donde:

- `username`: nombre del usuario que realiza la ejecución

- `node`: nombre del nodo en el que se realiza la ejecución de la aplicación
- `timestamp`: cadena de texto de la forma D20171026_T125748, donde la primera parte consiste en la fecha y la segunda parte consiste en la hora de la ejecución del test.

Para facilitar la tarea de consultar los ficheros de registro, en cada nueva ejecución se generará un enlace simbólico de nombre `CURRENT` apuntando a esa ejecución, por lo que dicho enlace simbólico siempre apuntará a la última ejecución que haya finalizado.

2.3. Ejecución concurrente

El programa `autotests` permite la ejecución de múltiples instancias concurrentes desde el mismo nodo y por parte del mismo usuario. En caso de que dos instancias de `autotests` se lancen simultáneamente y, por lo tanto, el timestamp de esas ejecuciones se comparta, el programa renombrará los directorios de registro de tal forma que no se produzcan condiciones de carrera asociadas al almacenamiento de los ficheros de registro.

2.4. Contenido de los ficheros de registro

Como ya se ha comentado anteriormente, el programa genera un fichero de registro por cada caso de test ejecutado. Cada fichero de registro se divide en dos partes. La primera consiste en la salida de la aplicación que se ha ejecutado, y la segunda consiste en información relevante acerca de la ejecución realizada, como lo es el nombre de host, la variable de entorno del enlazador dinámico, variables de entorno de rCUDA y tiempo de ejecución del caso de test en cuestión:

```
$ cat CURRENT/D20171026_T125748/NAMD_sample1

[...]

Info: 435 ANGLES
Info: 446 DIHEDRAL
Info: 45 IMPROPER
Info: 0 CROSSTERM
Info: 83 VDW
Info: 6 VDW_PAIRS
Info: 0 NBTHOLE_PAIRS
============== End of aplication output ==============



============== Execution Information ==============

HOSTNAME=node02

LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64/...

RCUDA_DEVICE_COUNT=2

RCUDA_DEVICE_0=node03:0
RCUDA_DEVICE_1=node03:1

RCUDA_NETWORK=IB

RCUDAIBDEVNO=2

RCUDAIBPORTNO=1
```

```
Time elapsed for test NAMD_sample1: 575 ms

============== End of execution Information ==============
```

2.5. Fichero de registro global

Para cada ejecución que realiza el usuario, se genera en la ruta donde se almacenan los ficheros de registro para esa ejecución en concreto un nuevo fichero llamado `_globalResults`, que contiene el resultado global de esa ejecución. El objetivo de este fichero es almacenar el estado de todas las ejecuciones, de forma que pueda consultarse fácilmente si los casos de test ejecutandos anteriormente resultaron en éxito o fracaso.

2.6. Fichero de registro de casos de test fallidos

En cada ejecución en la que se produzcan tests fallidos, se genera en la ruta donde se almacenan los ficheros de registro un nuevo fichero llamado `_failedTestCases` que contiene una lista de todos los casos de test que han fallado y su salida estándar, además de cierta información relevante para el usuario como por ejemplo las variables de entorno asociadas a esa ejecución.

Anexos

A. Casos de test disponibles

Los casos de tests del programa se encuentran organizados en forma de árboles jerárquicos, y tal y como se ha comentado anteriormente en este documento, estos casos de tests pueden consultarse a través del parámetro `--list_content`:

```
$ autotests --list_content
```

B. Etiquetas disponibles

Las etiquetas que se encuentran disponibles pueden consultarse con el parámetro `--list_labels`:

```
$ autotests --list_labels

Available labels:
  BARRACUDA
  CAFFE
  CUBLAS
  CUDASW
  CUDA_MEME
  CUDA_SAMPLES
  CUDA_SAMPLES_WORKING
  GPU_BLAST
  GPU_LIBSVM
  GROMACS
  HPC_WORKING
  LAMMPS
  MAGMA
  NAMD
  NVGRAPH
  SANITY_CHECK
  TENSORFLOW
```

APÉNDICE B

Makefile de compilación de los programas `autotests` y `unit_tests`

```
include ./Makefile.variables

all: $(AUTOTESTS_PROJECT) $(UNIT_TESTS_PROJECT)

$(AUTOTESTS_PROJECT): $(AUTOTESTS_OBJECTS) $(AUTOTESTS_CUDA_OBJECTS)
  $(CXX) $(AUTOTESTS_LDFLAGS) $(AUTOTESTS_OBJECTS) $(AUTOTESTS_CUDA_OBJECTS)
    ↪  $(AUTOTESTS_LDLIBS) -o $(BIN_DIR)/$@

$(UNIT_TESTS_PROJECT): $(UNIT_TESTS_OBJECTS)
  $(CXX) $(UNIT_TESTS_LDFLAGS) $(UNIT_TESTS_OBJECTS) $(UNIT_TESTS_LDLIBS) -o
    ↪  $(BIN_DIR)/$@

$(AUTOTESTS_OBJ_DIR)/%.o: $(AUTOTESTS_SRC_DIR)/%.cpp $(AUTOTESTS_HEADERS)
  $(CXX) $(AUTOTESTS_CXXFLAGS) $(AUTOTESTS_LDLIBS) $< -o $@

$(UNIT_TESTS_OBJ_DIR)/%.o: $(UNIT_TESTS_SRC_DIR)/%.cpp $(UNIT_TESTS_HEADERS)
  $(CXX) $(UNIT_TESTS_CXXFLAGS) $(UNIT_TESTS_LDLIBS) $< -o $@

$(AUTOTESTS_OBJ_DIR)/%.o: $(AUTOTESTS_SRC_DIR)/%.cu $(AUTOTESTS_HEADERS)
  $(NVCC) $(AUTOTESTS_NVCCFLAGS) $(AUTOTESTS_LDLIBS) $< -o $@

install:
ifeq ($(PRODUCTION), 1)
  @cp -f $(BIN_DIR)/$(AUTOTESTS_PROJECT) $(BIN_DIR)/$(UNIT_TESTS_PROJECT)
    ↪  $(UNIT_TESTS_PATH_BIN)
else
  @cp -f $(BIN_DIR)/$(AUTOTESTS_PROJECT) $(BIN_DIR)/$(UNIT_TESTS_PROJECT)
    ↪  $(UNIT_TESTS_PATH_DEVEL_BIN)
endif

uninstall:
ifeq ($(PRODUCTION), 1)
  @rm -f $(UNIT_TESTS_PATH_BIN)/$(AUTOTESTS_PROJECT)
    ↪  $(UNIT_TESTS_PATH_BIN)/$(UNIT_TESTS_PROJECT)
else
  @rm -f $(UNIT_TESTS_PATH_DEVEL_BIN)/$(AUTOTESTS_PROJECT)
    ↪  $(UNIT_TESTS_PATH_DEVEL_BIN)/$(UNIT_TESTS_PROJECT)
endif

clean:
  @rm -f $(BIN_DIR)/$(AUTOTESTS_PROJECT) $(BIN_DIR)/$(UNIT_TESTS_PROJECT)
    ↪  $(AUTOTESTS_OBJ_DIR)/*.o $(UNIT_TESTS_OBJ_DIR)/*.o
```

Código fuente B.1: Makefile para compilar los programas `autotests` y `unit_tests`

```
BOOST_VERSION := 1.63.0
DEFAULT_CUDA_VERSION = 8.0

ifdef CUDAVERSION
        CUDA_VERSION := $(CUDAVERSION)
        AUTOTESTS_CXXFLAGS := -DCUDAVERSION=$(CUDAVERSION)
else
        CUDA_VERSION := $(DEFAULT_CUDA_VERSION)
        AUTOTESTS_CXXFLAGS := -DCUDAVERSION=$(DEFAULT_CUDA_VERSION)
endif

BOOST_PATH := /usr/lib/boost/$(BOOST_VERSION)
BOOST_INC := $(BOOST_PATH)/include
BOOST_LIB := $(BOOST_PATH)/lib
CUDA_PATH := /usr/local/cuda-$(CUDA_VERSION)
CUDA_INC := $(CUDA_PATH)/include
CUDA_LIB := $(CUDA_PATH)/lib64
UNIT_TESTS_PATH_DEVEL := /home/andiaro/devel
UNIT_TESTS_PATH_DEVEL_BIN := $(UNIT_TESTS_PATH_DEVEL)/bin
UNIT_TESTS_PATH := /usr/bin/rCUDA/TESTS
UNIT_TESTS_PATH_BIN := $(UNIT_TESTS_PATH)/bin
CXXSTD := -std=c++11
CXX := g++
NVCC := nvcc
AUTOTESTS_PROJECT := autotests_CUDA$(CUDA_VERSION)
UNIT_TESTS_PROJECT := .unit_tests_rcuda_CUDA$(CUDA_VERSION)
AUTOTESTS_OBJ_DIR := obj/AUTOTESTS
UNIT_TESTS_OBJ_DIR := obj/UNIT_TESTS
AUTOTESTS_SRC_DIR := src/AUTOTESTS
UNIT_TESTS_SRC_DIR := src/UNIT_TESTS
AUTOTESTS_INC_DIR := inc/AUTOTESTS
UNIT_TESTS_INC_DIR := inc/UNIT_TESTS
BIN_DIR := bin
AUTOTESTS_CXXFLAGS += -Wall -c -I $(BOOST_INC) -I $(AUTOTESTS_INC_DIR) $(CXXSTD)

ifeq ($(PRODUCTION), 1)
        AUTOTESTS_CXXFLAGS += -DPRODUCTION
endif

UNIT_TESTS_CXXFLAGS += -Wall -c -I $(BOOST_INC) -I $(UNIT_TESTS_INC_DIR) $(CXXSTD)
AUTOTESTS_NVCCFLAGS := -c -Wno-deprecated-gpu-targets -cudart=shared -I $(CUDA_INC)
AUTOTESTS_LDFLAGS := -Wall $(CXXSTD) -Wl,-rpath=$(BOOST_LIB)
UNIT_TESTS_LDFLAGS := -Wall $(CXXSTD)
AUTOTESTS_LDLIBS := -L $(BOOST_LIB) -lboost_program_options -lboost_chrono -lboost_system
   ↪  -lboost_date_time -lboost_filesystem -lrt -L $(CUDA_LIB) -lcudart -lcuda
UNIT_TESTS_LDLIBS := -L $(BOOST_LIB) -lboost_unit_test_framework -lboost_chrono
   ↪  -lboost_system -lboost_date_time -lrt
AUTOTESTS_HEADERS := $(shell find $(AUTOTESTS_INC_DIR) -name '*.hpp')
UNIT_TESTS_HEADERS := $(shell find $(UNIT_TESTS_INC_DIR) -name '*.hpp')
AUTOTESTS_SOURCES := $(shell find $(AUTOTESTS_SRC_DIR) -name '*.cpp')
UNIT_TESTS_SOURCES := $(shell find $(UNIT_TESTS_SRC_DIR) -name '*.cpp')
AUTOTESTS_CUDA_SOURCES := $(shell find $(AUTOTESTS_SRC_DIR) -name '*.cu')
AUTOTESTS_OBJECTS := $(patsubst $(AUTOTESTS_SRC_DIR)/%.cpp, $(AUTOTESTS_OBJ_DIR)/%.o,
   ↪  $(AUTOTESTS_SOURCES))
UNIT_TESTS_OBJECTS := $(patsubst $(UNIT_TESTS_SRC_DIR)/%.cpp, $(UNIT_TESTS_OBJ_DIR)/%.o,
   ↪  $(UNIT_TESTS_SOURCES))
AUTOTESTS_CUDA_OBJECTS := $(patsubst $(AUTOTESTS_SRC_DIR)/%.cu, $(AUTOTESTS_OBJ_DIR)/%.o,
   ↪  $(AUTOTESTS_CUDA_SOURCES))
```

Código fuente B.2: Fichero `makefile.variables` que contiene las variables usadas por el fichero `makefile`