UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Análisis del impacto de rCUDA en las prestaciones de mCUDA-MEME y HOOMD-Blue

Proyecto Final de Carrera
Ingeniería Técnica Informática de Sistemas

Autor: Carlos Baiget
Directores: Federico Silla y Carlos Reaño

27 de septiembre de 2013

Palabras clave

rCUDA, virtualización de GPUs, CUDA, HPC, computación paralela.

Resumen

La presencia de Unidades de Proceso Gráfico (Graphics Processing Units, GPUs) en las instalaciones de Computación de Alto Rendimiento (High Performance Computing, HPC) es una opción cada vez más extendida por la mejora del rendimiento que proporcionan a la hora de realizar ciertos cálculos intensivos de manera repetitiva. Entre los inconvenientes que supone incluir estas tarjetas aceleradoras se encuentran los económicos, debidos principalmente a la necesidad de que cada nodo del cluster incorpore una de estas tarjetas, y a un alto consumo energético que es independiente de si se encuentran actualmente en uso o no. Teniendo en cuenta además que dichas tarjetas nunca son utilizadas al 100 % de manera continuada en ningún nodo del cluster, una solución que permita compartir entre los diferentes nodos del clúster los recursos instalados en uno de ellos (básicamente, una o varias GPUs) maximizaría su uso y reduciría los costes anteriormente mencionados.

El middleware rCUDA desarrollado por el Departamento de Informática de Sistemas y Computadores de la Universidad Politécnica de Valencia, en colaboración con el grupo High Performance Computing and Architecture, del Departamento de Ingeniería y Ciencia de Computadores de la Universidad Jaume I de Castellón, persigue este objetivo, mediante la virtualización de las GPU existentes en alguno de los nodos remotos carentes de ellas.

De esta forma incluso es innecesaria la modificación de las aplicaciones de cálculo que requieran servicios de cómputo por parte de GPUs, y pudiendo funcionar como si realmente los nodos que no tienen GPU dispusieran de dichas unidades instaladas localmente.

Mientras que los beneficios en términos económicos que se pueden obtener mediante la virtualización de GPUs pueden ser fácilmente estimados en base a una menor proporción entre número total de GPUs y número total de nodos del clúster, es menos inmediato anticipar la penalización en términos de tiempos de ejecución que puede suponer esta capa añadida de sofware, debida a latencias o cuellos de botella introducidos por los otros componentes del sistema requeridos por rCUDA para funcionar, como son la pila de protocolos utilizados para el traslado de información entre nodos a través de la red o la propia tecnología de red utilizada.

Para recrear un escenario real de virtualización de GPUs y tratar de medir estos impactos, hemos recurrido a dos aplicaciones de cálculo intensivo utilizando GPUs y que son usadas en entornos reales de producción, con el fin de contrastar su rendimiento en distintas configuraciones de virtualización de GPUs contra la correspondiente instalación en la que estos programas funcionarían utilizando las GPUs de manera local (que es la forma tradicional en la que se usan las GPUs). Por un lado se busca someter al middleware rCUDA a una carga de trabajo próxima a la esperada en un entorno de trabajo real y, por otro, el obtener algunos indicios sobre la evolución del rendimiento conforme varía el tamaño de los cálculos a resolver.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Debido al alto coste computacional de las actuales aplicaciones de cálculo intensivo, muchos científicos contemplan las Unidades de Proceso Gráfico (Graphics Processing Units, GPUs) como un medio eficiente de reducir los tiempos de ejecución de sus aplicaciones. Las GPUs de alta gama incluyen una extraordinaria cantidad de pequeñas unidades de cómputo así como acceso a su propia memoria de gran ancho de banda. No es de extrañar, por tanto, que todas las aplicaciones que realizan una gran cantidad de operaciones aritméticas por cada uno de sus datos puedan beneficiarse del gran potencial de estas aceleradoras hardware.

En las aplicaciones aceleradas por GPU, el alto rendimiento se consigue normalmente por medio de extraer las porciones computacionalmente intensivas, para ser ejecutadas en dichos dispositivos. Afortunadamente, durante los últimos años han habido varios intentos orientados a explotar el gran paralelismo de las GPUs, dando lugar a mejoras notables en la programabilidad de estos entornos híbridos CPU-GPU. Actualmente los programadores disponen de librerías y entornos, como CUDA u OpenCL entre otros, que se encargan de este proceso de separación. Como resultado, el uso de GPUs para computación de propósito general ha acelerado la adopción de estos dispositivos en áreas muy diversas.

Además, la tecnología GPU tiene como objetivo principal el mercado de los videojuegos, de grandes volúmenes de fabricación y por tanto con una relación coste/prestaciones muy favorable. El resultado neto es que las GPUs están siendo adoptadas como una manera efectiva de reducir el tiempo total de ejecución de diversas aplicaciones y por tanto una opción amplia y consolidada para la aplicación de la computación de alto rendimiento (High Performance Computing, HPC) en diferentes ciencias.

La estrategia habitual en las instalaciones de HPC para aprovechar el uso de GPUs consiste en incluir una o más de estas aceleradoras en cada nodo del clúster. Aunque esta configuración es atractiva desde el punto de vista del rendimiento final, no es eficiente a la hora de considerar el consumo energético, dado que una única GPU puede llegar a consumir el 25 % de la energía total requerida por un único nodo de un clúster HPC. Asimismo, en este tipo de sistemas, es bastante improbable que todas las GPUs del clúster sean usadas el 100 % del tiempo, ya que muy pocas aplicaciones muestran tal grado extremo de paralelismo de datos. En cualquier caso, incluso las GPUs inactivas consumen grandes cantidades de energía. En suma, el añadir una GPU a todos los nodos de un clúster de HPC está lejos del espíritu de la 'computación verde', siendo más bien una solución altamente ineficiente desde el punto de vista energético, a pesar de que las GPUs, cuando están usándose, presentan una alta eficiencia energética.

Por otra parte, la reducción de la cantidad de aceleradoras presentes en un clúster, de manera que su utilización se vea incrementada, es un solución menos costosa y más atractiva que además reduciría tanto el recibo del suministro eléctrico como el coste total de propiedad (Total Cost of Ownership, TCO) y el impacto medioambiental de la llamada Computación de Propósito General usando GPUs (GPGPU por sus siglas en inglés), debido a un menor consumo energético.

Sin embargo, una configuración en la cual sólo un número limitado de nodos en el clúster incluyen GPUs presenta algunas dificultades, ya que requiere un planificador global que distribuya las tareas a los nodos

con GPUs atendiendo a sus necesidades de aceleración, haciendo así esta nueva configuración más eficiente pero también más difícil de administrar de manera efectiva. Además, esta configuración no abordaría realmente la baja utilización de las GPUs a menos que el planificador global pudiera compartirlas entre varias aplicaciones, un detalle que incrementaría notablemente la complejidad de tales planificadores.

Una mejor solución para tratar con una configuración de clúster que tenga menos GPUs que nodos es la virtualización de las GPUs. De hecho, la virtualización de hardware se ha convertido en una solución comúnmente adoptada para mejorar el Coste Total de Propiedad (TCO) ya que reduce los costes de adquisición, mantenimiento, administración, espacio y de suministro energético de las instalaciones HPC y de proceso de datos. Con la virtualización de GPUs, éstas se instalan únicamente en algunos de los nodos, para ser compartidas posteriormente a lo largo del clúster. De esta manera, los nodos que incluyen GPUs se pueden considerar como servidores de aceleración que proporcionan servicios GPGPU al resto del clúster. Así, se evitan las dificultades de planificación mencionadas anteriormente, ya que las tareas pueden ser dirigidas a cualquier nodo independientemente de sus necesidades de hardware mientras que al mismo tiempo las aceleradoras son compartidas entre aplicaciones, incrementando por tanto la utilización de las GPUs.

Esta solución puede mejorarse aun más a base de hacer los planificadores más inteligentes de manera que los servidores de GPUs pasen a un estado de bajo consumo mientras no sean requeridas sus capacidades de aceleración, incrementando así su eficiencia energética notablemente. Más aun, en lugar de instalar el mismo número de GPUs en cada servidor de aceleración, estos podrían incluir diferentes cantidades de aceleradoras, de manera que algún tipo de granularidad estuviese disponible en los algoritmos planificadores para ajustar mejor los recursos y la energía consumida a la carga de trabajo presente en un momento dado en el sistema.

La mejora de los planificadores globales para dotarlos de la capacidad de migrar tareas de GPU entre nodos se traduciría en una arquitectura coherente con el paradigma de computación verde, dado que la energía consumida en un determinado momento por las aceleradoras sería la mínima requerida para la carga de trabajo asumida por el sistema.

Con el fin de hacer posible esta innovadora propuesta de mejora energética en instalaciones de computación de alto rendimiento, el Grupo de Arquitecturas Paralelas del Departamento de Sistemas Informáticos y Computadores de la Universidad Politécnica de Valencia , en colaboración con el grupo High Performance Computing and Architecture, del Departamento de Ingeniería y Ciencia de Computadores de la Universidad Jaume I de Castellón, ha desarrollado recientemente el entorno rCUDA.

La tecnología rCUDA es un middleware que emplea una arquitectura cliente-servidor. El cliente rCUDA se ejecuta en cada nodo del clúster, mientras que el servidor rCUDA se ejecuta únicamente en aquellos nodos equipados con GPU(s). El software cliente se presenta como una verdadera GPU a las aplicaciones, aunque en la práctica es solamente la interfaz de una GPU virtual. En esta configuración rCUDA, cuando una aplicación solicita servicios de aceleración, contactará con el software cliente, que remitirá la petición de aceleración a un nodo del clúster que disponga de la GPU real. Allí el servidor rCUDA procesará la petición y recurrirá a la GPU real para realizar la acción deseada. Tras ser completada la operación, el servidor rCUDA enviará de vuelta los resultados correspondientes, los cuales serán entregados a la aplicación que los solicitó por medio del cliente rCUDA. La aplicación no podrá discernir si está empleando una GPU virtualizada en lugar de una real.

Los prototipos iniciales de rCUDA demostraron que tal funcionalidad era factible, aunque también revelaron algunas cuestiones referidas al rendimiento. Dado que rCUDA hace de intermediario entre la memoria principal de los nodos cliente y la memoria de la GPU remota en los nodos servidores, necesariamente ha de existir una merma del rendimiento debida

a las características de la tecnología de red que interconecte los nodos. Esta merma es dependiente, principalmente, de las características de la red que comunica el cliente y el servidor rCUDA. Los estudios realizados hasta ahora indican que cuando se usan redes de altas prestaciones como InfiniBand QDR o FDR, la sobrecarga de rCUDA puede llegar a ser insignificante.

El presente proyecto final de carrera tiene por objetivo perfilar el impacto sobre el rendimiento que es atribuible al uso de rCUDA junto a la tecnología de red Ethernet y el protocolo TCP/IP para transportar los datos. Para ello se compararán dos aplicaciones científicas que hacen uso intensivo de la GPU al funcionar de manera nativa con CUDA, y en diferido con rCUDA. Ambas aplicaciones son maduras y se encuentran en un nivel de desarrollo apto para sistemas reales en producción. Para cada aplicación se evaluarán distintos volúmenes de datos y, en el caso de rCUDA, comparando el uso de la interfaz loopback proporcionada por el sistema operativo anfitrión con la correspondiente a la interfaz física que interconecta los nodos con el objetivo de disociar las latencias debidas al flujo de los datos por la pila TCP/IP del sistema operativo a las debidas por la transmisión de los datos por la red. Mientras que una de las dos aplicaciones sólo hace uso de una única GPU, la otra puede emplear varias GPUs simultáneamente mediante el uso de la librería MPI. En este caso, el nodo servidor se equipará con varias GPUs, de manera que se pueda evaluar el rendimiento que muestra rCUDA al prestar servicio a múltiples peticiones remotas de manera concurrente. Nótese que el entorno empleado (red Ethernet y TCP/IP) es el peor escenario para rCUDA, y en el que se esperan peores prestaciones dado que la sobrecarga introducida por la red es la máxima que se puede esperar.

El resto del documento está estructurado de la siguiente manera: en el Capítulo 2 describimos la arquitectura CUDA así como una introducción a la tecnología rCUDA; en el Capítulo 3 , detallamos el proceso de instalación y comprobación del middleware rCUDA; en el Capítulo 4 se describe el proceso de instalación y uso del programa HOOMD-blue, utilizado para realizar las pruebas de rendimiento al virtualizar 1 GPU; en el Capítulo 5

se describe el proceso de instalación y uso del programa mCUDA-MEME, utilizado para realizar las pruebas de rendimiento al virtualizar varias GPUs; en el Capítulo 6 describimos las pruebas realizadas así como los resultados obtenidos en cada caso; para finalizar, en el Capítulo 7 comentamos las principales conclusiones de este estudio.

Capítulo 2

Antecedentes

Este capítulo proporciona una visión general de la arquitectura CUDA, así como una introducción a la tecnología rCUDA.

2.1. CUDA: Compute Unified Device Architecture

Hoy en día, debido a la creciente demanda de recursos computacionales requeridos por las GPUs, se han realizado importantes progresos en su desarrollo, consiguiendo dispositivos de gran potencia computacional y gran ancho de banda de memoria, como se muestra en las Figuras 2.1 y 2.2.

La Figura 2.1 muestra cómo en algunos casos es posible conseguir mejoras teóricas hasta 8 veces superiores con respecto a las prestaciones de las CPUs más potentes de 2010. Al mismo tiempo, la Figura 2.2 muestra que el ancho de banda a memoria teórico de una GPU es 6 veces superior al correspondiente a una CPU de 2010, aun cuando éstas últimas continuamente experimentan grandes mejoras para mejorar este aspecto. Aunque las gráficas muestran únicamente datos hasta 2010, esta tendencia ha continuado a lo largo de los últimos años. Por ejemplo, las últimas Tesla K20X, con 6GB de memoria RAM y 2688 núcleos computacionales, alcanzan 3.95 Teraflops y un ancho de banda a memoria de 250GB/s, mientras que las CPUs de Intel más

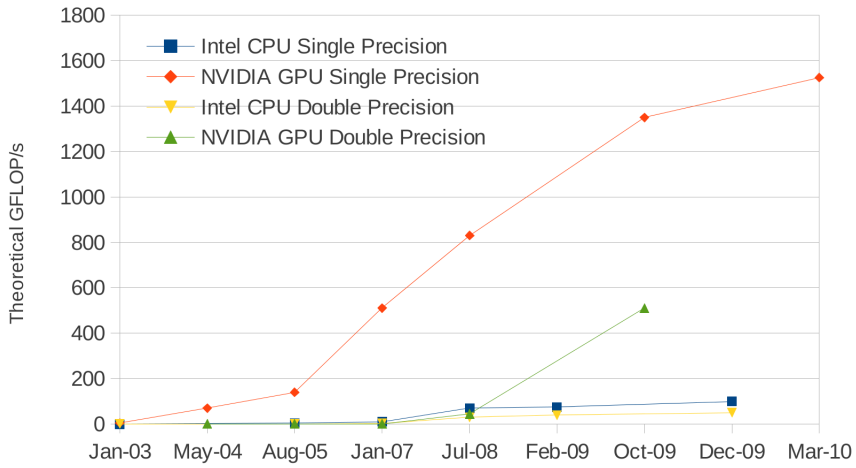

Figura 2.1: Operaciones en coma flotante por segundo de CPU y GPU.

modernas, las Xeon E7-8870 con 10 núcleos funcionando a 2.4 GHz, consiguen 384 Gigaflops y un ancho de banda de memoria de 102GB/s. Los avances conseguidos en el campo de las GPUs han permitido usarlas para mejorar la potencia de los clúster HPC. Con este fin, NVIDIA desarrolla desde 2006 una nueva tecnología llamada CUDA (Compute Unified Device Architecture) que aprovecha las capacidades de computación paralela de las GPUs NVIDIA para resolver muchos problemas complejos de computación de manera más eficiente que una CPU.

CUDA consiste en un nuevo modelo de programación así como una arquitectura de conjunto de instrucciones. También facilita un entorno software que permite a los desarrolladores usar C y otros lenguajes de programación de alto nivel. La Figura 2.3 muestra la jerarquía mediante la cual CUDA posibilita a las aplicaciones de usuario el acceso a la arquitectura por medio de estos lenguajes de programación de alto nivel. Otros lenguajes soportados son C++, Fortran, Java, Python, y el entorno .NET de Microsoft.

Los programas CUDA se compilan mediante el compilador nvcc, que analiza el código fuente en busca de fragmentos de código GPU y los compila de manera separada al código CPU.

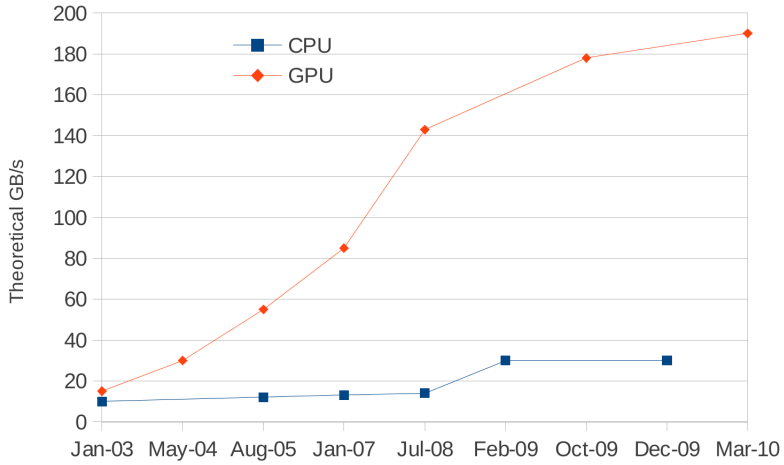

Figura 2.2: Ancho de banda de memoria de CPU y GPU

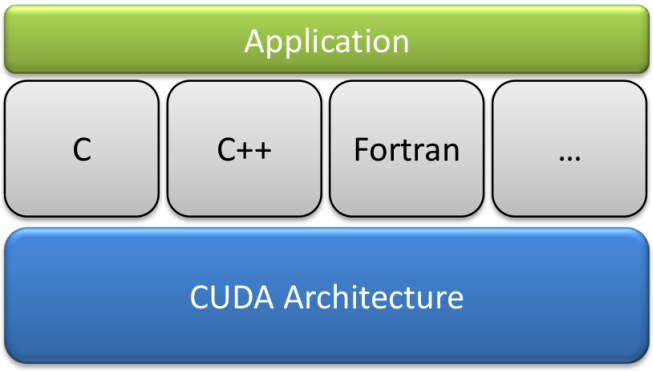

Figura 2.3: Interfaz de programación de la arquitectura rCUDA

2.2. rCUDA: CUDA remoto

Tal como se ha remarcado anteriormente, la estrategia empleada en las instalaciones HPC para sacar partido de las GPUs consiste en incluir una o varias aceleradoras por nodo del clúster. Pero ello presenta varios inconvenientes, principalmente:

- Alto consumo de energía.
- Baja utilización de cada GPU.
- Alto coste de adquisición.

rCUDA se desarrolló con el objetivo de superar estos inconvenientes. rCUDA es un middleware que otorga a las aplicaciones un acceso transparente a las GPUs instaladas en nodos remotos, de manera que éstas no pueden distinguir si están accediendo a un dispositivo externo. Este entorno está organizado siguiendo una arquitectura distribuida de cliente-servidor, como se muestra en la Figura 2.4

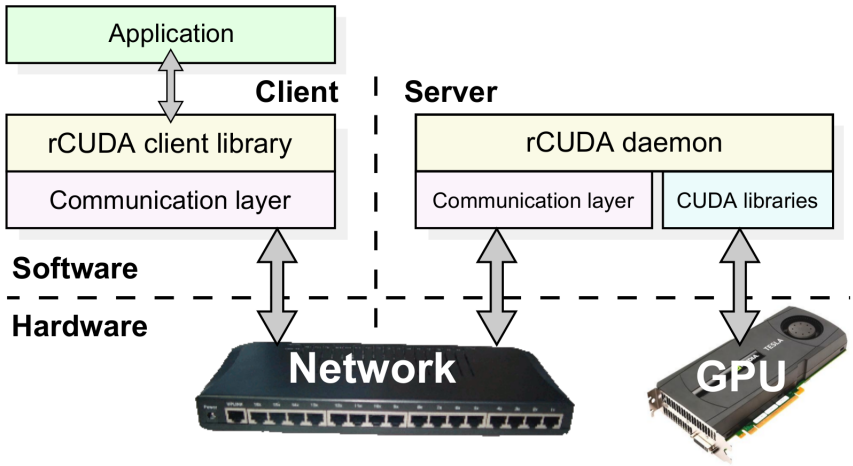

Figura 2.4: Esquema de la arquitectura rCUDA

El middleware cliente es accedido por la aplicación que requiere servicios GPGPU, ambas (el cliente middleware y la aplicación) ejecutándose en el mismo nodo del clúster, el cual no posee una GPU. El cliente rCUDA presenta a la aplicación una interfaz idéntica a la que presenta la API en tiempo de ejecución (Runtime API) el entorno NVIDIA CUDA normal. Tras la recepción de una solicitud de una aplicación, el middleware cliente la procesa y remite las peticiones correspondientes al middleware servidor rCUDA, el cual se ejecuta en un nodo remoto y en el que hay instaladas una o varias GPUs. A su vez, el servidor interpreta las solicitudes y realiza el procesamiento requerido accediendo a la GPU real de modo que ejecute la orden correspondiente. Una vez que la GPU ha completado la operación solicitada, los resultados son recuperados por el servidor rCUDA, que los devuelve al middleware cliente. Allí, la salida se transmite finalmente a la aplicación que los demandó originalmente. Hay que destacar que de este modo las GPUs se pueden compartir de manera concurrente entre varias aplicaciones solicitantes por medio del uso de diferentes procesos servidores rCUDA para soportar diferentes ejecuciones en contextos GPU independientes. Esta característica es la que permite conseguir una elevada utilización de las GPUs.

La comunicación entre los clientes rCUDA y los servidores GPU se lleva a cabo por medio de un protocolo a nivel de aplicación, este protocolo es interno a rCUDA y está adaptado para aprovechar la red disponible en el clúster. La Figura 2.5 muestra un ejemplo del protocolo implementado en el entorno rCUDA para una solicitud genérica. Este ejemplo ilustra cómo la solicitud de ejecución de un kernel se retransmite de cliente a servidor, así como el conjunto de datos utilizado como entrada. También se muestra la obtención del conjunto de datos de salida.

La versión más reciente del entorno rCUDA tiene por objeto los sistemas operativos basados en Linux, y soporta las mismas distribuciones Linux que NVIDIA CUDA, que son las versiones más recientes de Fedora, Redhat, Ubuntu, OpenSuse y Suse Server. Esta última versión de rCUDA soporta

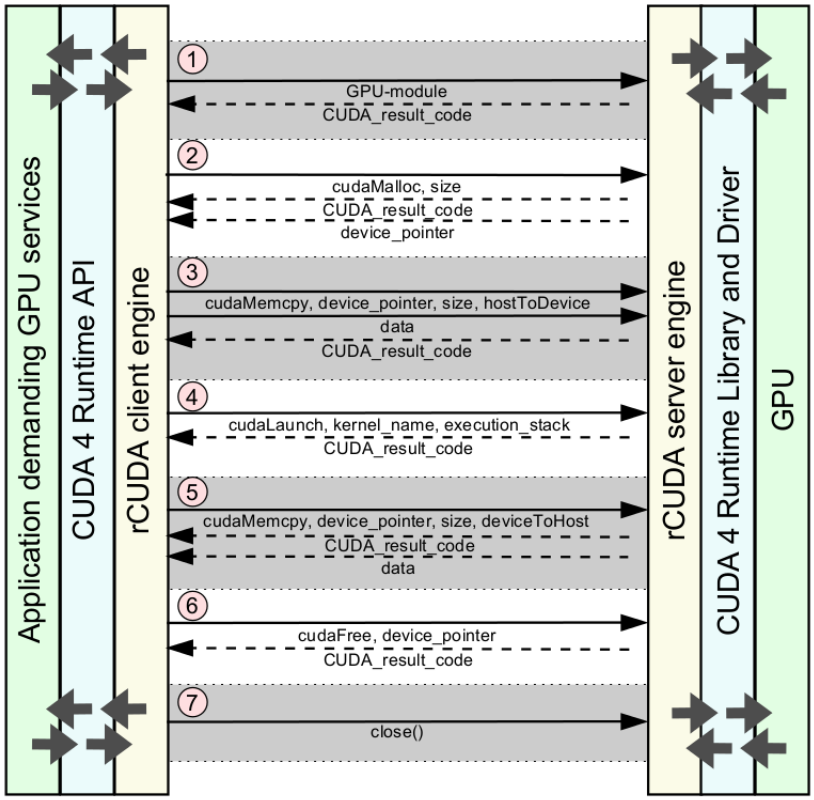

Figura 2.5: Ejemplo del protocolo de comunicación usado por rCUDA: (1) inicialización, (2) reserva de memoria en la GPU remota, (3) transferencia de los datos de entradas de la CPU a la GPU, (4) ejecución del kernel, (5) transferencia de los resultados de la GPU a la CPU, (6) liberación de la memoria, y (7) cierre del canal de comunicación y finalización del proceso servidor.

la versión 5 de la API en tiempo de ejecución de CUDA, exceptuando las capacidades CUDA relacionadas con los gráficos, ya que este tipo de características rara vez son de algún interés en entornos HPC. Sin embargo, sí que está planeado su soporte en futuras versiones, ya que podrían ser muy útiles en aplicaciones de computación en la nube, por ejemplo para virtualización de escritorios o juegos en la nube.

En general, es de esperar que el rendimiento alcanzado por rCUDA sea inferior al obtenido con CUDA original, debido a que con rCUDA la GPU está más distante de la aplicación que la invoca que en el caso equivalente en el que se emplea CUDA, y por lo tanto existe una sobrecarga añadida.

La Figura 2.6 muestra el ancho de banda efectivo obtenido en operaciones de copia de memoria a GPUs remotas por medio de distintas conexiones y módulos de comunicación.

Estos resultados, traducidos a la ejecución de una aplicación, llevan a un uso más o menos eficiente de la GPU remota, dependiendo de la red concreta que se use, y con sobrecargas despreciables al compararlas con la aceleración GPU local en el caso de las redes más rápidas.

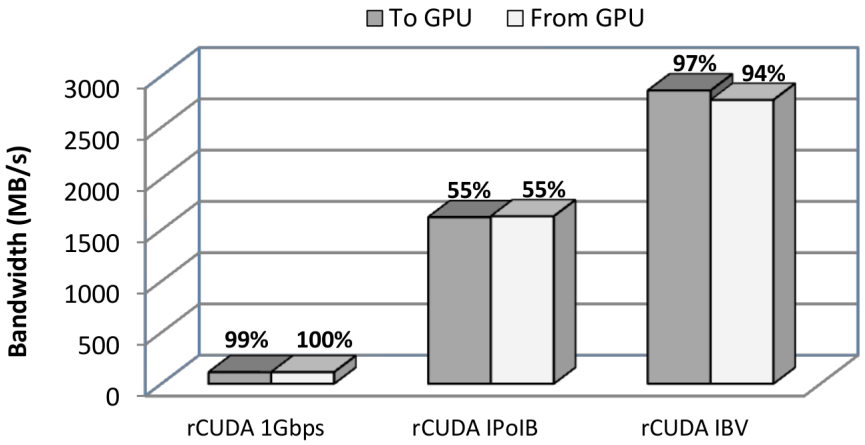

Figura 2.6: Ancho de banda entre una CPU y una GPU remota en distintos escenarios: tarjetas Nvidia GeForce 9800, red con Mellanox InfiniBand ConnectX-2 y 1Gbps Ethernet. La utilización de la red es próxima al 100 % al usar Gigabit Ethernet e InfiniBand, pero el uso de IP sobre InfiniBand introduce sobrecargas que hacen que su rendimiento disminuya prácticamente a la mitad.

La Figura 2.7 representa un ejemplo particular de multiplicación de matrices. En comparación con la computación tradicional con CPU, la figura muestra también que calcular el producto en una GPU remota empleando una red InfiniBand QDR es notablemente más rápido que realizar el mismo cálculo empleando 8 núcleos de CPU de propósito general en un nodo de computación usando una librería de computación de alto rendimiento altamente optimizada.

De la misma manera que ocurre en el ejemplo mostrado en la Figura 2.7, el rendimiento de las aplicaciones que usan rCUDA a menudo es notablemente mayor que el obtenido mediante cálculos generados por CPUs convencionales.

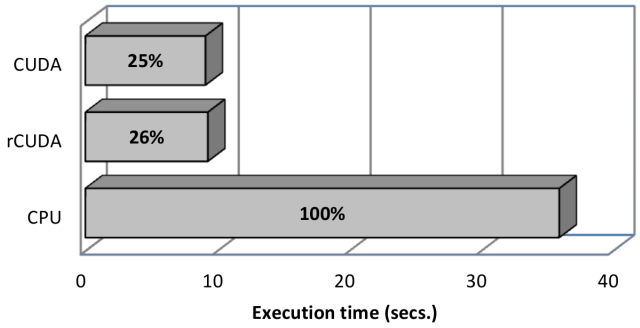

Figura 2.7: Tiempos de ejecución en segundos de un producto de matrices usando una Nvidia Tesla C2050 vs. cálculo con 2 CPU Intel Xeon E5520 Quad-Core usando GotoBlas 2. Matrices de 13.824x13.824 elementos en coma flotante de simple precisión. La realización de las operaciones usando una GPU requiere la cuarta parte del tiempo requerido por los 8 núcleos de CPU. El uso de rCUDA en una red InfiniBand ConnectX-2 introduce una pérdida de eficiencia de sólo un 1 % respecto a la utilización de CUDA en local.

Teniendo en cuenta la flexibilidad proporcionada por rCUDA, así como las reducciones de uso energético y costes de adquisición que permite, los beneficios de rCUDA sobrepasan los inconvenientes que genera la sobrecarga que introduce su uso.

Capítulo 3

Instalación de rCUDA

El entorno rCUDA permite el uso concurrente de dispositivos compatibles con CUDA de manera remota y transparente para las aplicaciones. En este capítulo se detalla la arquitectura de rCUDA y su instalación, así como un ejemplo de ejecución de un programa CUDA ejecutándose a través de rCUDA.

Esta plataforma de virtualización consta de un middleware cliente, que consiste en una librería de llamadas que sustituye a las librerías dinámicas CUDA proporcionadas por NVIDIA, y un middleware servidor, el cual se configura como un servicio de sistema en aquellos nodos que van a proveer servicios de aceleración GPGPU (los que tienen las GPUs instaladas en ellos).

rCUDA tiene una arquitectura cliente-servidor distribuida. Los clientes usan una librería que intercepta las llamadas a la API en tiempo de ejecución de CUDA (Runtime API) para permitir el acceso a dispositivos virtualizados, mientras que los nodos que albergan físicamente las aceleradoras gráficas ejecutan un servicio de sistema que atiende las solicitudes de ejecución recibidas. Para optimizar el intercambio de datos entre cliente y servidor, rCUDA emplea un protocolo de comunicación personalizado de nivel de aplicación .

El middleware ejecutado por parte de los clientes se distribuye en un fichero: "libcudart.so.5.0".

Estas librerías compartidas deben ubicarse en aquella máquina o máquinas que requieren acceso a servicios GPGPU remotos.

Es necesario además ajustar la variable de entorno LD_LIBRARY_PATH de forma acorde a la localización final de estos ficheros, típicamente:

/$HOME/rCUDA/framework/rCUDAl o /usr/local/cuda/lib64.

Con el fin de ejecutar aplicaciones correctamente usando la librería rCUDA, se requiere el establecimiento de las siguientes variables de entorno:

- RCUDA_DEVICE_COUNT: Indica el número de GPUs que son accesibles desde el nodo actual. Uso:

  ```
  RCUDA_DEVICE_COUNT=<número_de_GPUs>
  ```

  Por ejemplo, si el nodo puede acceder a dos GPUs:

  ```
  RCUDA_DEVICE_COUNT=2
  ```

- RCUDA_DEVICE_X: Indica dónde se encuentra la GPU X del nodo. Uso:

  ```
  RCUDA_DEVICE_X=<servidor[@<port>]>[:nº GPU]
  ```

  Por ejemplo, si las GPU 0 y 1 del nodo se encuentran en el servidor "192.168.0.1" (en el puerto rCUDA por defecto):

  ```
  RCUDA_DEVICE_0=192.168.0.1
  RCUDA_DEVICE_1=192.168.0.1:1
  ```

En los entornos MPI soportados (MVAPICH2 y OpenMPI 1.3+), la librería distribuirá las tareas MPI entre los diferentes servidores. El puerto utilizado por defecto es el 8308. Si se usa una conexión de red InfiniBand, hay que indicarlo estableciendo la variable "RCUDAPROTO" a IB.

De la misma manera que ocurre con el cliente, para la instalación del servidor es necesario establecer la variable de entorno "RCUDAPROTO" a IB si se está utilizando una conexión de red InfiniBand. El servicio rCUDA (rCUDAd) debe ejecutarse en la máquina o máquinas que ofrecen servicios GPGPU remotos.

El servicio ofrece las siguientes opciones en línea de comandos:

```
-d <device> : Selecciona el dispositivo (primer dispositivo en funcionamiento por defecto)
-i : No pasar a modo servicio de sistema. En su lugar, ejecutar en modo interactivo.
-l : Modo local usando sockets AF\_UNIX (TCP only).
-n <numero> : Número de servidores concurrentes admitidos. 0 indica ilimitados (por defecto).
-p <puerto> : Especifica el puerto en el que ha de escuchar (8308 por defecto).
-v : Modo de depuración.
-h : Imprime información de uso.
```

3.1. Ejecución de un programa CUDA con rCUDA

A continuación, se muestra a modo de ejemplo la ejecución del programa del SDK de CUDA "deviceQuery", el cual muestra información sobre las características de la tarjeta GPU disponible para las aplicaciones CUDA. Para ello se dispone de dos máquinas, llamadas rcu16 y rcu17 en el ejemplo. rcu16 incorpora cuatro tarjetas aceleradoras, mientras que rcu17 no dispone de ninguna. Por medio de la plataforma rCUDA, rcu16 exportará una de sus tarjetas y rcu17 hará uso de ella, de manera que el programa "deviceQuery" mostrará la información de dicha tarjeta tal y como si estuviera disponible de manera local, ya que el binario "deviceQuery" funciona sin ninguna modificación y obtiene esta información de manera transparente.

En primer lugar y para comparar el resultado tras la instalación de rCUDA, ejecutamos deviceQuery en rcu17. Lógicamente no funciona, pues ni siquiera tiene instalado el Runtime de CUDA con el que atender las llamadas que va a realizar deviceQuery para su funcionamiento:

```
[root@rcu17 ~]\# /root/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery/deviceQuery

/root/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery/deviceQuery: error while loading shared libraries:
libcudart.so.5.0: cannot open shared object file: No such file or directory
```

El error es el esperado en estas circunstancias, ya que no puede cargar las librerías de CUDA requeridas por la aplicación. Declaramos las variables de entorno utilizadas por el cliente de rCUDA para que hagan referencia a las librerías del cliente, y la máquina que exporta sus tarjetas:

```
[root@rcu17 ~]\# export LD_LIBRARY_PATH=/nas/colab/carbaior/rCUDA/framework/rCUDAl:\$LD_LIBRARY_PATH
[root@rcu17 ~]\# export PATH=\$PATH:/usr/local/cuda/bin
[root@rcu17 ~]\# export RCUDA_DEVICE_COUNT=1
[root@rcu17 ~]\# export RCUDA_DEVICE_0=192.168.12.116:0
```

En este momento, en rcu16 ejecutamos el servicio rCUDA. Puesto que se trata de un entorno de pruebas lo lanzamos con las opciones -i (interactivo), y -v (modo depuración, mostrará más mensajes de funcionamiento de los necesarios):

```
[root@rcu16 rCUDAd]\# ./rCUDAd -iv
rCUDAd v4.0.1
Copyright 2009-2013 Copyright 2009-2013 UNIVERSITAT POLITECNICA DE VALENCIA AND UNIVERSITAT JAUME I DE CASTELLO
All rights reserved.
rCUDAd[2275]: Using rCUDAcommTCP.so communications library.
rCUDAd[2275]: Server daemon succesfully started.
rCUDAd[2277]: Trying device 0...
rCUDAd[2277]: CUDA initialized on device 0.
```

Vemos que el servicio se ejecuta con éxito, e inicializa CUDA en la primera GPU del sistema.

A continuación probamos en rcu17 la ejecución de "deviceQuery":

```
[root@rcu17 scripts_mcudameme]\# /root/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery/deviceQuery
/root/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 590"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1536 MBytes (1610285056 bytes)
  (16) Multiprocessors x ( 32) CUDA Cores/MP:    512 CUDA Cores
  GPU Clock rate:                                1225 MHz (1.23 GHz)
  Memory Clock rate:                             1710 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       No
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0,
CUDA Runtime Version = 5.0, NumDevs = 1, Device0 = GeForce GTX 590
```

Ha funcionado con éxito, y obtenemos el mismo resultado que si tuviéramos una GPU instalada localmente.

Por su parte, el servicio rCUDA en rcu16 ha generado una salida adicional como resultado de haber procesado las peticiones, y que incluye las funciones CUDA atendidas:

```
[root@rcu16 rCUDAd]\# ./rCUDAd -iv
rCUDAd v4.0.1
Copyright 2009-2013 Copyright 2009-2013 UNIVERSITAT POLITECNICA DE VALENCIA AND
UNIVERSITAT JAUME I DE CASTELLO. All rights reserved.
rCUDAd[2275]: Using rCUDAcommTCP.so communications library.
rCUDAd[2275]: Server daemon succesfully started.
rCUDAd[2277]: Trying device 0...
rCUDAd[2277]: CUDA initialized on device 0.
rCUDAd[2277]: Connection established with 192.168.12.117.
rCUDAd[2279]: Trying device 1...
rCUDAd[2277]: 'cudaGetDeviceProperties' received.
rCUDAd[2277]: 'cudaDriverGetVersion' received.
rCUDAd[2277]: 'cudaGetDeviceProperties' received.
rCUDAd[2277]: Remote application finished.
rCUDAd[2279]: CUDA initialized on device 1.
```

Tras finalizar las tareas, el servicio queda a la espera de nuevas solicitudes.

Capítulo 4

HOOMD-blue: Simulación de dinámica de partículas

En este capítulo se describe el programa usado para la evaluación del rendimiento de rCUDA respecto a CUDA al usar una única GPU. También se detalla su instalación y la comprobación de su funcionamiento con un ejemplo.

4.1. Descripción

Para el análisis del impacto en el rendimiento de rCUDA en comparación al uso de una GPU local, se ha evaluado el tiempo de ejecución así como la medida de pasos de simulación conseguidos por una aplicación real de dinámica de partículas que puede emplear una GPU para acelerar sus cálculos.

HOOMD-blue es acrónimo de Highly Optimized Object-oriented Many-particle Dynamics – Blue Edition, y es esencialmente un simulador que recrea las interacciones que sufren las partículas en distintas configuraciones y circunstancias predefinidas en un contenedor tridimensional. La simulación de cada modelo se parametriza por el número de partículas, la naturaleza de sus interacciones y un factor de densidad que caracteriza el volumen en el

que transcurre la simulación. Una gran cantidad de partículas se corresponde con una mayor reserva de memoria, y el análisis de las fuerzas resultantes en cada instante de la simulación conlleva más cálculos por lo que los pasos de simulación conseguidos por unidad de tiempo disminuye.

El paquete HOOMD-blue proporciona dos benchmarks mediante los cuales es posible comparar distintos sistemas a la hora de realizar las mismas simulaciones. Estos benchmarks consisten en dos ficheros de configuración de la simulación en dos contextos distintos de dinámica de partículas: el comportamiento de un modelo de fluido Lennard-Jones y el de un sistema de polímeros (cada uno de los cuales está compuesto de 27 partículas) en un solvente implícito.

Ambos benchmarks realizan una simulación de 50.000 pasos y calculan el rendimiento medio en número de iteraciones por segundo.

Por el interés que supone para la medida del rendimiento de rCUDA el movimiento de datos entre cliente y servidor, se prepararon cinco variantes de cada benchmark cada una de las cuales doblaba el volumen de la simulación hasta emplear toda la memoria disponible en el sistema.

Así, se repitieron las pruebas para 64K, 128K, 256K, 512K y 1024K partículas en tres contextos distintos: una configuración de control consistiendo en la ejecución de las pruebas usando CUDA de manera local en un nodo de clúster HPC, una configuración usando rCUDA por medio de la interfaz loopback del sistema operativo, de manera que tanto el cliente como el servidor rCUDA funcionaban en el mismo nodo, y por último una configuración en la que el servidor y el cliente rCUDA se ejecutaban en nodos distintos pero con las mismas características de memoria y CPU, y conectadas a través de Ethernet.

4.2. Instalación de HOOMD-blue

Los requerimientos de sistema de HOOMD-blue son:

- SO basado en Linux (recomendado) o Mac OS X.
- Procesador de 64 bits con juego de instrucciones x86_64.
- Opcional: GPU con capacidad CUDA.
- Al menos la misma cantidad de RAM que memoria de vídeo de la GPU.

Al ser una aplicación Open Source, optamos por la compilación a partir de las fuentes en lugar de usar binarios precompilados. Los requisitos de software previos a la compilación son:

- Python 2.3 o superior.
- Boost: librería estándar C++ portable
- Compilador de GNU gcc.
- CMake.
- CUDA SDK.
- Programa de control de versiones Git.

Una vez instaladas las dependencias, se puede proceder a compilar los fuentes.

En primer lugar hay que obtener una copia local del código fuente:

```
$ mkdir hoomd
$ cd hoomd
$ git clone https://codeblue.umich.edu/git/hoomd-blue codeblue
```

Posteriormente compilar e instalar HOOMD-blue usando los makefiles:

```
$ cd hoomd
$ mkdir build
$ cd build
$ cmake ../code -DCMAKE_INSTALL_PREFIX=~/hoomd-install
$ make install -j4
```

La opción -j4 permite la compilación de varios ficheros en paralelo.

Por último actualizar la variable PATH

Editar el fichero /.bash_profile y añadir la ruta al binario al final del mismo:

```
export PATH=\$PATH:\${HOME}/hoomd-install/bin
```

4.3. Ejecución de HOOMD-blue

El programa HOOMD-blue acepta como entrada un fichero script que describe las simulaciones a realizar, así como un determinado número de opciones.

La sintaxis del comando es:

```
hoomd [fichero_script] [opciones]

Opciones:

ninguna opción
    hoomd se ejecutará en una GPU determinada automáticamente, o en las CPUs disponibles si no se encuentra
    ninguna GPU apta.

-h, --help
    muestra una descripción de todas las opciones de línea de comandos.

--mode={cpu | gpu}
    fuerza la ejecución de hoomd bien sobre cpu o sobre gpu.

--gpu={\#}
    especifica explícitamente la GPU en la que se ejecutará hoomd. Implica --mode=gpu.

--ncpu={\#}
    especifica el número de núcleos de CPU en los cuales se ejecutará hoomd. No implica --mode=cpu.
```

```
--ignore-display-gpu
    previene la ejecución de hoomd sobre la GPU que está vinculada a la pantalla de visualización.

--minimize-cpu-usage
    minimiza el uso de CPU por parte de hoomd cuando se ejecuta en una GPU.

--gpu_error_checking
    habilita la comprobación de errores tras cada llamada kernel de la GPU.

--notice-level=\#
    especifica el nivel de mensajes de notificación a mostrar.

--msg-file=filename
    especifica un fichero para escribir los mensajes (el fichero se sobreescribe).

--user
    opciones de usuario.
```

Para comprobar el buen funcionamiento de nuestra instalación de HOOMD-blue, lanzamos una prueba sencilla tal como se detalla en la guía rápida del usuario del programa en

http://codeblue.umich.edu/hoomd-blue/doc/page_quick_start.html.

Creamos el fichero de script test.hoomd propuesto, correspondiente a una simulación sencilla de un líquido Lennard-Jones:

```
from hoomd_script import *
# create 100 random particles of name A
%init.create_random(N=100, phi_p=0.01, name='A')

# specify Lennard-Jones interactions between particle pairs
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

# integrate at constant temperature
all = group.all();
integrate.mode_standard(dt=0.005)
integrate.nvt(group=all, T=1.2, tau=0.5)

# run 10,000 time steps
run(10e3)
```

Posteriormente usamos este fichero para realizar la simulación por medio de HOOMD-blue:

```
[root@rcu16 bin]\# ./hoomd test.hoomd
HOOMD-blue 0.11.3-unknown
Compiled: jue may  2 19:13:16 CEST 2013
Copyright 2008-2011 Ames Laboratory Iowa State University and the Regents of the University of Michigan
-----
All publications and presentations based on HOOMD-blue, including any reports
or published results obtained, in whole or in part, with HOOMD-blue, will
acknowledge its use according to the terms posted at the time of submission on:
http://codeblue.umich.edu/hoomd-blue/citations.html

At a minimum, this includes citations of:
* http://codeblue.umich.edu/hoomd-blue/
and:
* Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset - 'General
  Purpose Molecular Dynamics Fully Implemented on Graphics Processing
  Units', Journal of Computational Physics 227 (2008) 5342-5359
-----
test.hoomd:004  |  init.create_random(N=100, phi_p=0.01, name='A')
HOOMD-blue is running on the following GPU:
 [0]       GeForce GTX 590  16 SM_2.0 @ 1.23 GHz, 1535 MiB DRAM
notice(2): Group "all" created containing 100 particles
test.hoomd:007  |  lj = pair.lj(r_cut=3.0)
test.hoomd:008  |  lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
test.hoomd:011  |  all = group.all();
test.hoomd:012  |  integrate.mode_standard(dt=0.005)
test.hoomd:013  |  integrate.nvt(group=all, T=1.2, tau=0.5)
test.hoomd:016  |  run(10e3)
notice(2): -- Neighborlist exclusion statistics -- :
notice(2): Particles with 0 exclusions             : 100
notice(2): Neighbors excluded by diameter (slj)    : no
notice(2): Neighbors excluded when in the same body: no
** starting run **
Time 00:00:01 | Step 10000 / 10000 | TPS 9680.78 | ETA 00:00:00
Average TPS: 9680.49
---------
-- Neighborlist stats:
815 normal updates / 35 forced updates / 0 dangerous updates
n_neigh_min: 0 / n_neigh_max: 11 / n_neigh_avg: 3.76
shortest rebuild period: 3
** run complete **
```

La salida nos indica que el programa ha funcionado correctamente y ha realizado los cálculos con una de las GPUs instaladas en el sistema.

Capítulo 5

mCUDA-MEME: Análisis de secuencias de ADN

En este capítulo se describe el programa utilizado para evaluar el rendimiento de rCUDA respecto a CUDA a la hora de dar servicio con varias GPUs, así como su instalación. Dado que este programa utiliza la librería MPI para poder hacer uso de varias GPUs, se describe también la plataforma MPI utilizada y su instalación. Por último se muestra el resultado de una ejecución de prueba usando MPI con el fin de verificar que el sistema funciona correctamente.

5.1. Descripción

mCUDA-MEME es un derivado de la suite MEME, un conjunto de herramientas para análisis de secuencias de ADN basadas en patrones. Aprovecha la potencia de cálculo de las GPUs para búsqueda de patrones en secuencias de ADN o de proteínas. Además, proporciona una interfaz MPI, siendo capaz de sacar partido de varias GPUs simultáneamente ejecutando varias tareas de manera distribuida. Esta característica nos es útil para analizar el comportamiento de rCUDA al dar servicio a varias solicitudes remotas desde un único nodo que incorpora varias GPUs.

Al igual que ocurría en el caso de HOOMD-blue, la distribución del paquete MCUDA-meme incluye algunas configuraciones estándar con las que medir el rendimiento de distintos sistemas a la hora de resolver varios problemas de distintas magnitudes, en concreto la búsqueda de patrones de ADN entre 500, 1000 y 2000 secuencias predefinidas.

5.2. MVAPICH2: soporte MPI

Debido a que rCUDA virtualiza las tarjetas gráficas de los nodos servidores para ser usadas por varias aplicaciones cliente funcionando en distintos nodos de un clúster HPC, soporta también el dar servicio a un número indeterminado de solicitudes de manera concurrente. El soporte MPI de mCUDA-MEME permite usar múltiples tarjetas gráficas para la resolución de las tareas y es la característica que aprovechamos para medir el rendimiento de rCUDA a la hora de virtualizar varias tarjetas gráficas desde el mismo nodo del clúster.

Por cuestiones relativas a licencia de uso y rendimiento, hemos optado por la instalación de MVAPICH2, una variante de la MPICH para proporcionar el soporte MPI a nuestro sistema de pruebas.

Al igual que MPICH, MPVAPICH2 se ajusta al estándar MPI-3 pero además tiene soporte y está optimizado para las GPUs Nvidia usadas por CUDA y rCUDA, multihilo y tolerancia a fallos (Checkpoint-restart, Job-pause-migration-resume).

La versión utilizada para las pruebas ha sido la v1.9. La lista completa de características de esta implementación de MPI-3 se puede consultar en la página web de los desarrolladores en

http://mvapich.cse.ohio-state.edu/overview/mvapich2/features.shtml,

incluyendo el listado completo de optimizaciones relativas a la comunicación MPI con la memoria de las GPUs Nvidia.

La instalación se realiza a partir del fichero .tgz que contiene las fuentes.

Para no interferir con otros usuarios que puedan requerir otras librerías MPI, la instalación se realiza en un directorio de trabajo. Por otra parte, especificamos que el canal de comunicación entre procesos remotos será la red ethernet. Las opciones del comando configure para ambos requisitos son:

```
./configure --prefix $HOME/mpi_mvapich2 --with-device=ch3:nemesis
```

Seguidamente se puede proceder a la compilación e instalación con los comandos:

```
make && make install
```

Aunque las librerías ya están instaladas, hay que priorizar su uso sobre cualquier otra implementación MPI que pueda existir en el sistema, por lo que se antepone la ruta a estas librerías en las variables de entorno PATH y LD_LIBRARY_PATH que se establecen al inicio de cada sesión. La manera de conseguirlo es editando el fichero $HOME/.bash_profile y añadiendo las líneas:

```
export PATH=$HOME/mpi_mvapich2/bin:$PATH
export LD_LIBRARY_PATH=$HOME/mpi_mvapich2/lib:$LD_LIBRARY_PATH
```

5.3. Instalación de mCUDA-MEME

La compilación e instalación de mCUDA-MEME es sencilla a partir del archivo .tgz de la distribución. La única modificación necesaria consiste en especificar la capacidad ofrecida por la tarjeta GPU que está instalada en el sistema. En nuestro caso hacemos uso de la utilidad deviceQuery del SDK de CUDA para obtener este dato a partir del parámetro "CUDA Capability" y vemos que las tarjetas aceleradoras son del modelo GeForce GTX 590 y soportan la versión 2.0 de CUDA. En este caso es necesario modificar el fichero src/Makefile.mgpu e incluir la opción -arch sm_20 en la variable DEVICE_FLAGS. Dado que para la funcionalidad MPI usaremos librerias locales, también hay que incluir la ruta completa a ellas en este fichero, usando las variables de entorno MPI_CXXFLAGS y MPI_LIBS, en nuestro caso:

```
MPI_CXXFLAGS = -I$HOME/mpi_mvapich2/include
MPI_LIBS = -L$HOME/mpi_mvapich2/lib -lmpichcxx -lmpich -lmpl
```

Una vez realizadas estas modificaciones se usa el comando make para generar los binarios, indicando el fichero de configuración que ha de usar para generarlos. En nuestro caso y dado que vamos a requerir la funcionalidad MPI, el comando es:

```
make -f Makefile.mgpu
```

Lo que generará el ejecutable "mcuda-meme" en el directorio de trabajo.

5.4. Ejecución de mCUDA-MEME

El ejecutable "mcuda-meme" toma como entrada un fichero que contiene las secuencias de ADN a analizar, y un gran número de parámetros opcionales que se muestran al invocarlo sin ningún parámetro o con el parámetro -h:

```
[carbaior@rcu16 cuda-meme-3.0.13]\$ ./mcuda-meme -h

*********************************************************
Get environment variables MEME_ETC_DIR and MEME_BIN_DIR
Set ETC_DIR to ./etc
Set BIN_DIR to ./bin
*********************************************************
 USAGE:
mcuda-meme <dataset> [optional arguments]

<dataset>  file containing sequences in FASTA format
[-h] print this message
[-version] print the version
[-num_threads <num of threads>] specify the number of threads for EM step
[-not_sort] do not sort the intput sequences in length (default is FALSE)
[-o <output dir>] name of directory for output files
will not replace existing directory
[-oc <output dir>] name of directory for output files
will replace existing directory
[-text] output in text format (default is HTML)
[-dna] sequences use DNA alphabet
[-protein] sequences use protein alphabet
[-mod oops|zoops|anr] distribution of motifs
[-nmotifs <nmotifs>] maximum number of motifs to find
```

```
[-evt <ev>] stop if motif E-value greater than <evt>
[-nsites <sites>] number of sites for each motif
[-minsites <minsites>] minimum number of sites for each motif
[-maxsites <maxsites>] maximum number of sites for each motif
[-wnsites <wnsites>] weight on expected number of sites
[-w <w>] motif width
[-minw <minw>] minimum motif width
[-maxw <maxw>] maximum motif width
[-nomatrim] do not adjust motif width using multiple
alignment
[-wg <wg>] gap opening cost for multiple alignments
[-ws <ws>] gap extension cost for multiple alignments
[-noendgaps] do not count end gaps in multiple alignments
[-bfile <bfile>] name of background Markov model file
[-revcomp] allow sites on + or - DNA strands
[-pal] force palindromes (requires -dna)
[-maxiter <maxiter>] maximum EM iterations to run
[-distance <distance>] EM convergence criterion
[-psp <pspfile>] name of positional priors file
[-prior dirichlet|dmix|mega|megap|addone]
type of prior to use
[-b <b>] strength of the prior
[-plib <plib>] name of Dirichlet prior file
[-spfuzz <spfuzz>] fuzziness of sequence to theta mapping
[-spmap uni|pam] starting point seq to theta mapping type
[-cons <cons>] consensus sequence to start EM from
[-heapsize <hs>] size of heaps for widths where substring
search occurs
[-x_branch] perform x-branching
[-w_branch] perform width branching
[-allw] include all motif widths from min to max
[-bfactor <bf>] branching factor for branching search
[-maxsize <maxsize>] maximum dataset size in characters
[-nostatus] do not print progress reports to terminal
[-time <t>] quit before <t> CPU seconds consumed
[-sf <sf>] print <sf> as name of sequence file
[-V] verbose mode
```

Lo que indica que el programa funciona, y podemos pasar a probar la ejecución para la resolución de un problema concreto. Puesto que vamos a usarlo con el propósito de evaluar el rendimiento de un determinado sistema usando unos ficheros de prueba predefinidos, no haremos uso de la mayoría de opciones del programa. En nuestro caso, sólo es necesario especificar el tamaño máximo de los conjuntos de prueba con la opción -maxsize, e indicar

que el contenido del fichero de entrada está formado por secuencias de ADN con el parámetro -dna.

Para probar que el funcionamiento es correcto, lanzamos la ejecución con el fichero de benchmark que contiene 500.000 secuencias de ADN, usando la librería MPI para hacer uso de las 4 GPUs instaladas localmente en el nodo:

```
[root@rcu16 mcuda-meme]\# mpirun -np4 ./mcuda-meme -maxsize 500000 -dna nrsf_testcases/nrsf_500.fasta


********************************************************
Get environment variables MEME_ETC_DIR and MEME_BIN_DIR
Set ETC_DIR to /nas/colab/carbaior/mcuda-meme/cuda-meme-3.0.13/etc
Set BIN_DIR to /nas/colab/carbaior/mcuda-meme/cuda-meme-3.0.13/
********************************************************
Process 0 uses the 1-th GPU in host "rcu16"
Process 1 uses the 2-th GPU in host "rcu16"
Process 2 uses the 3-th GPU in host "rcu16"
Process 3 uses the 4-th GPU in host "rcu16"
enable openMP for EM step using 2 threads
The output directory 'meme_out' already exists.
Its contents will be overwritten.
node: 0 start_seq: 0 end_seq: 153
node: 1 start_seq: 154 end_seq: 277
node: 2 start_seq: 278 end_seq: 391
node: 3 start_seq: 392 end_seq: 499
Initializing the motif probability tables for 2 to 500 sites...
nsites = 500
Done initializing

seqs=   500, min= 307, max=  529, total=   226070
MemTotal:78669321 KB, NumProcsInHost: 4, MemThreshold:1440.48 MB
Reverse complement strand is NOT used
Using sequence-level parallelization
Loading sequences from CPU to GPU
host buffer size 4232000

****************************************
nmotifs: 1

motif=1
ic:0 n_nsites:499
motif length:8 n_nsites:499
Estimated GPU device memory consumption: 12.3984 MBytes
-----score computing runtime:7.67505
-----alignment processing time: 7.6839(s)
```

```
-----starting point computing runtime:7.73338
motif length:11 n_nsites:499
-----score computing runtime:8.21296
-----alignment processing time: 8.2233(s)
-----starting point computing runtime:8.2751
motif length:15 n_nsites:499
-----score computing runtime:9.34404
-----alignment processing time: 9.35603(s)
-----starting point computing runtime:9.41488
motif length:21 n_nsites:499
-----score computing runtime:11.3946
-----alignment processing time: 11.4133(s)
-----starting point computing runtime:11.4794
motif length:29 n_nsites:499
-----score computing runtime:13.3023
-----alignment processing time: 13.3106(s)
-----starting point computing runtime:13.3688
motif length:41 n_nsites:499
-----score computing runtime:15.6633
-----alignment processing time: 15.7239(s)
-----starting point computing runtime:15.7839
motif length:50 n_nsites:499
-----score computing runtime:17.6405
-----alignment processing time: 17.7215(s)
-----starting point computing runtime:17.7745
starting point search overall time: 89.0429 seconds
em: w=  50, psites= 128, iter=  40
----CUDA-MEME overall time: 118.522(s)
```

La salida indica que la ejecución se ha realizado sin problemas usando las 4 GPUs, así como el tiempo que ha requerido para completarse.

Capítulo 6

Evaluación

En este capítulo se muestran los resultados obtenidos al medir el rendimiento de rCUDA a la hora de virtualizar una o varias GPUs.

En el caso de una única GPU, se utiliza el programa HOOMD-blue con dos configuraciones distintas correspondientes a dos simulaciones de fluidos de distintas características. Para simplificar, dichas simulaciones se referirán como Benchmark A y Benchmark B.

En el caso de virtualización de varias GPU, se usa el programa mCUDA-MEME, ya que éste último permite el uso concurrente de varias tarjetas mediante el uso de MPI. Las pruebas consisten en la ejecución de mCUDA-MEME con hasta cuatro tareas simultáneas (procesos MPI) para la resolución de cada problema en un nodo sin ninguna GPU local, y desde una hasta cuatro GPUs virtuales ofrecidas por rCUDA. De nuevo se comparan los resultados obtenidos, con los obtenidos en unas pruebas de control realizadas usando GPUs de manera nativa usando CUDA, y también las realizadas en diferido usando rCUDA pero con el cliente y el servidor rCUDA funcionando en el mismo nodo, a través de la interfaz loopback del sistema operativo.

6.1. Descripción del entorno de pruebas

El entorno de pruebas consiste en dos nodos (llamados rcu16 y rcu17) con idénticas características hardware y software, a excepción de que uno de ellos (rcu16) incorpora cuatro GPUs, mientras que el otro (rcu17) no tiene instalada ninguna. Ambos equipos están conectados por una red Gigabit Ethernet.

Características comunes a ambos equipos:

- CPU: Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz (x8)
- RAM: 8GB
- OS: Open Suse 11.2 y Scientific Linux 6.1
- RED: Gigabit Ethernet
- HDD: 1TB

rcu16 además incorpora 4 GPUs GeForce GTX 590 (Compute Capability 2.0)

6.2. HOOMD-blue: Virtualización de una GPU

A continuación se muestran los resultados obtenidos al ejecutar HOOMD-blue en tres contextos distintos, usando dos simulaciones distintas y para varios tamaños de simulación cada una. Los tres contextos mencionados consisten en: uno de control usando el entorno CUDA local, otro usando rCUDA pero con el cliente y el servidor ejecutándose en el mismo nodo aprovechando la interfaz loopback del sistema operativo, y un último en el que el servidor y el cliente están en nodos distintos, pero con similares características de RAM y CPU. La intención es poder discriminar el impacto del rendimiento que se puede atribuir a cada uno de los elementos necesarios

al usar el entorno rCUDA, incluyendo la pila TCP/IP del sistema operativo, la tecnología de red empleada, o al propio middleware rCUDA.

Cada ejecución se repitió 10 veces, y los datos mostrados corresponden a la media de los valores obtenidos, los cuales muestran una desviación típica despreciable. Los benchmarks proporcionan un valor para comparar el rendimiento llamado TPS (Time steps Per Second), y además se muestran los tiempos de ejecución de cada prueba según los reporta el comando "time"de Linux, desglosados en tiempo real, de usuario, y de sistema.

6.2.1. Benchmark A

El Benchmark A se evaluó para varios tamaños de simulación, concretamente para 64K, 128K, 256K, 512K y 1024K partículas.

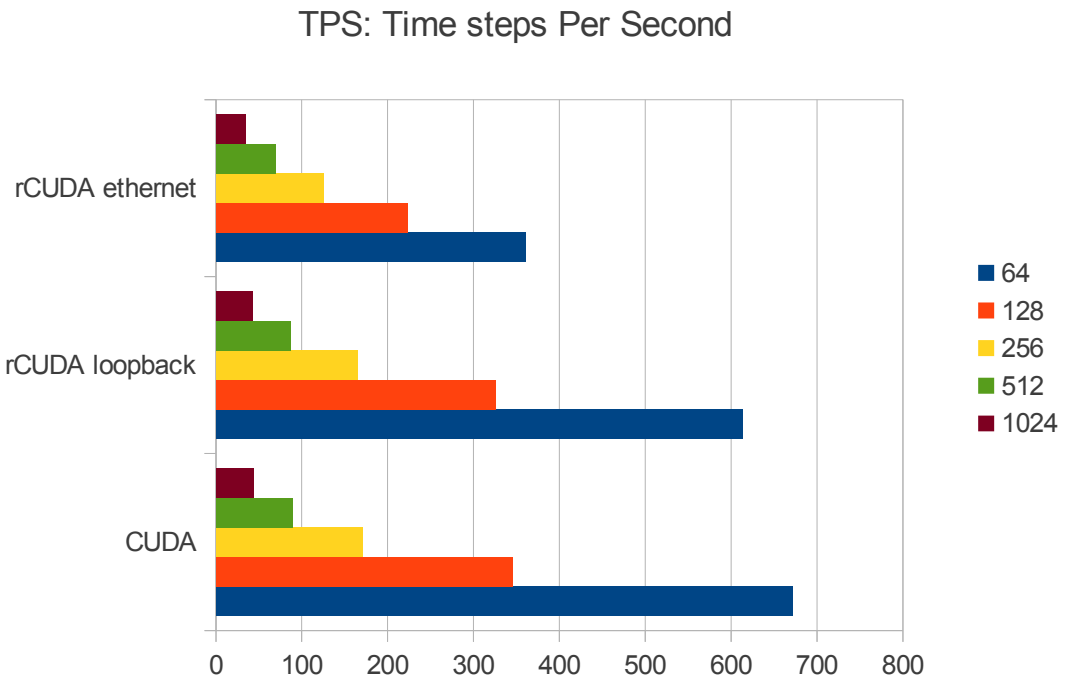

Figura 6.1: Benchmark A. Time steps Per Second.

Tabla 6.1: Benchmark A. TPS: Time steps Per Second.

Nº Partículas (x1000):	**64**	**128**	**256**	**512**	**1024**
CUDA	672 σ=0,8	345 σ=0,3	171 σ=0,2	89 σ=0,7	44 σ=0
rCUDA loopback	613 σ=2,8	326 σ=0,7	165 σ=0,1	87 σ=0,1	43 σ=0
rCUDA Ethernet	361 σ=0,7	223 σ=0,4	125 σ=0,2	69 σ=0,1	35 σ=0,1

Valores mayores de TPS indican un mayor rendimiento. Vemos que los valores obtenidos en las pruebas CUDA y rCUDA loopback son muy similares con variaciones menores al 10 % en el peor caso (prueba de 64K).

El uso de Ethernet introduce más variacion del rendimiento, siendo del 53 % en el peor caso (prueba de 64K) y del 80 % en el mejor (prueba de 1024K). En ambos casos el rendimiento de rCUDA aumenta conforme aumenta el tamaño del problema.

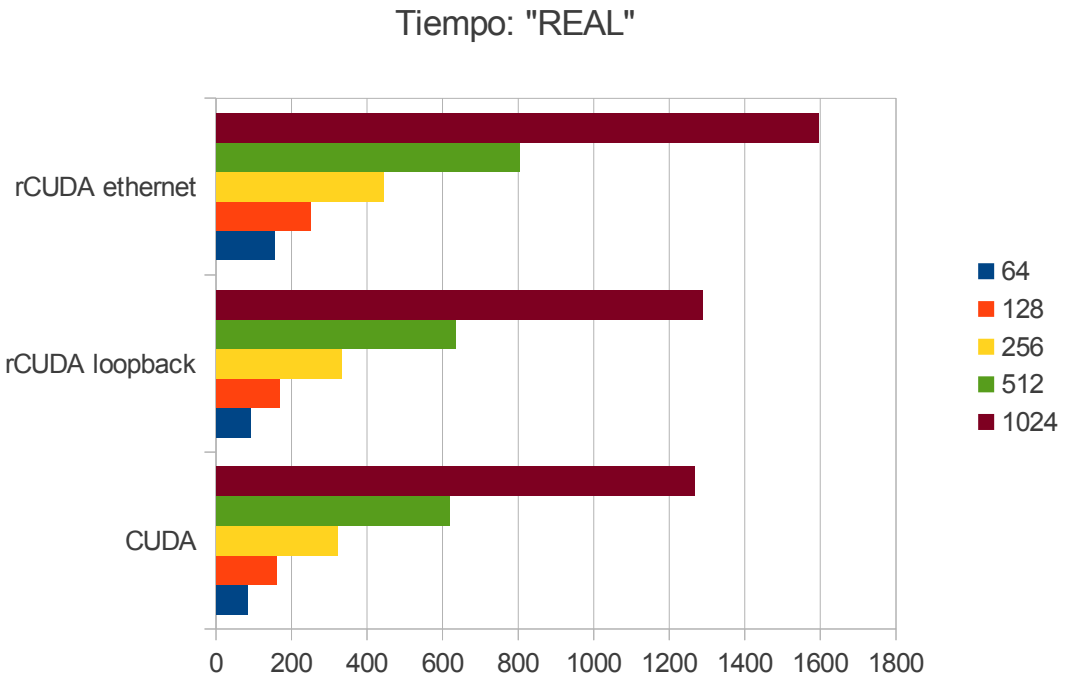

Figura 6.2: Benchmark A. Tiempo: "Real" (segundos).

Tabla 6.2: Benchmark A. Tiempo: "Real" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**	**1024**
CUDA	83 σ=0,1	161 σ=0,1	322 σ=0,3	618 σ=0,4	1267 σ=0,5
rCUDA loopback	91 σ=0,4	169 σ=0,4	333 σ=0,2	634 σ=0,7	1288 σ=1,1
rCUDA Ethernet	156 σ=0,3	250 σ=0,4	444 σ=0,6	803 σ=1,4	1595 σ=3,2

Tiempo total de ejecución. Los valores obtenidos de las pruebas con CUDA y rCUDA loopback están más próximos entre sí, lo que indica que la merma de rendimiento debida al uso rCUDA es muy baja. En la prueba de rCUDA Ethernet el retardo es más evidente pero mejora notablemente al aumentar el tamaño del problema.

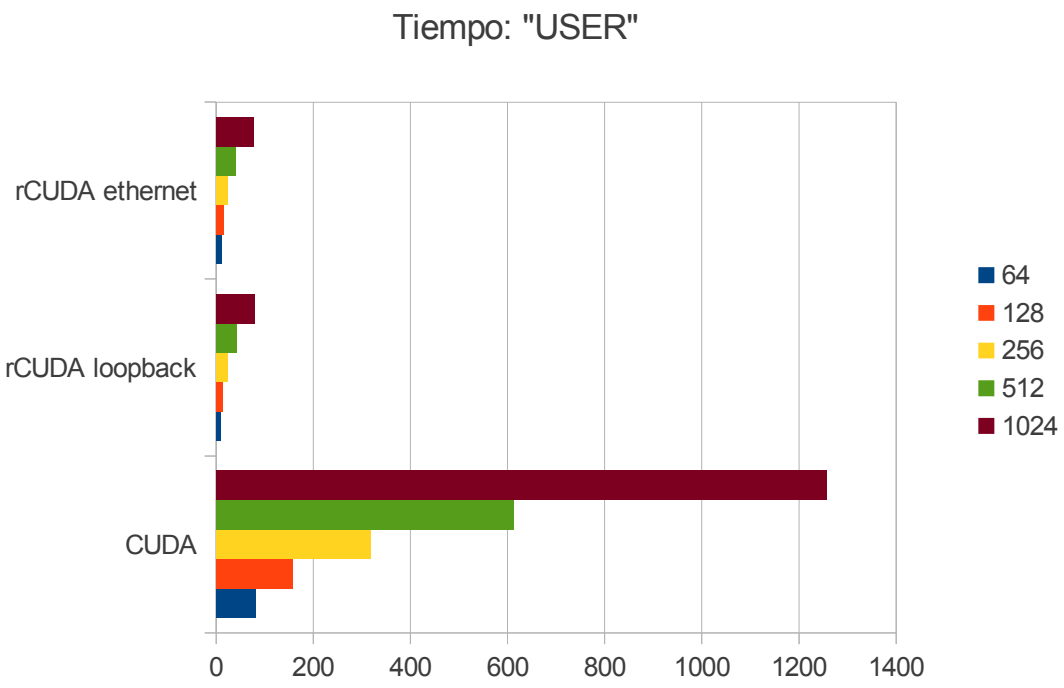

Figura 6.3: Benchmark A. Tiempo: "User" (segundos).

Tabla 6.3: Benchmark A. Tiempo: "User" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**	**1024**
CUDA	81	158	318	613	1258
rCUDA loopback	9	13	23	43	80
rCUDA Ethernet	11	15	23	40	77

Porción del tiempo de ejecución correspondiente a los procesos de usuario. En el caso de rCUDA Ethernet y rCUDA loopback es muy bajo debido a que la mayor parte de los procesos se ejecutan en remoto por medio de rCUDA.

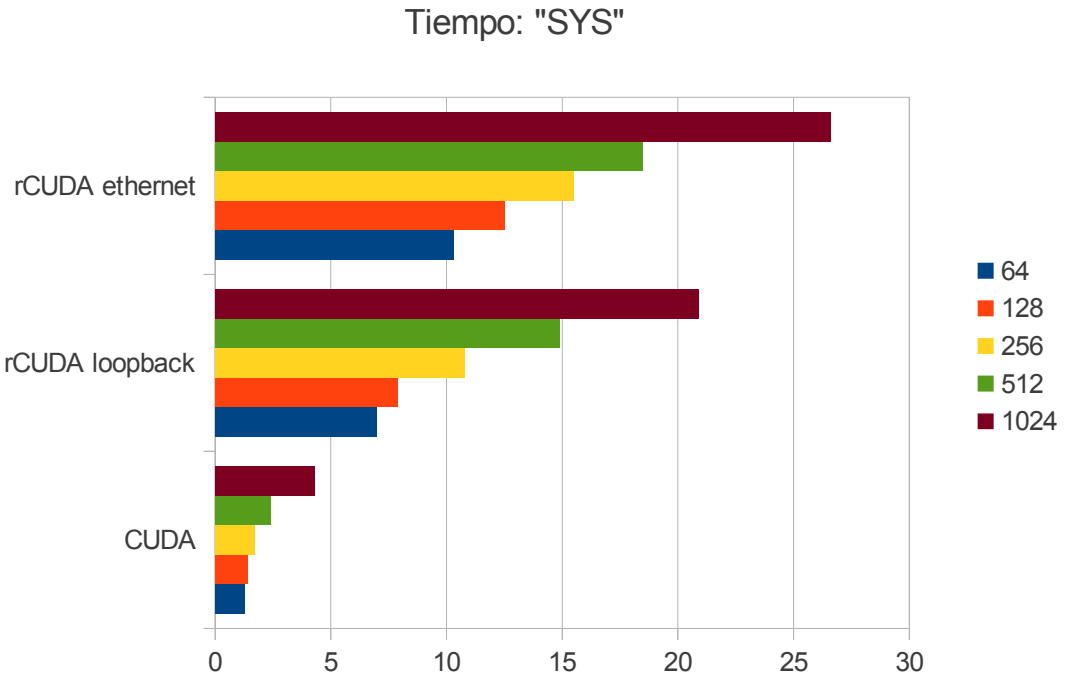

Figura 6.4: Benchmark A. Tiempo: "Sys" (segundos).

Tabla 6.4: Benchmark A. Tiempo: "Sys" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**	**1024**
CUDA	1,3	1,4	1,7	2,4	4,3
rCUDA loopback	7	7,9	10,8	14,9	20,9
rCUDA Ethernet	10,3	12,5	15,5	18,5	26,6

Porción del tiempo de ejecución correspondiente a los procesos de sistema. De nuevo los tiempos de rCUDA tienen un gran contraste con los tiempos de CUDA, dado que el middleware rCUDA se ejecuta en tiempo de sistema.

6.2.2. Benchmark B

El Benchmark B consume algo más de memoria y se evaluó para los tamaños de 64k, 128k, 256k, y 512k partículas, dado que el sistema no soportaba el tamaño de 1024k partículas.

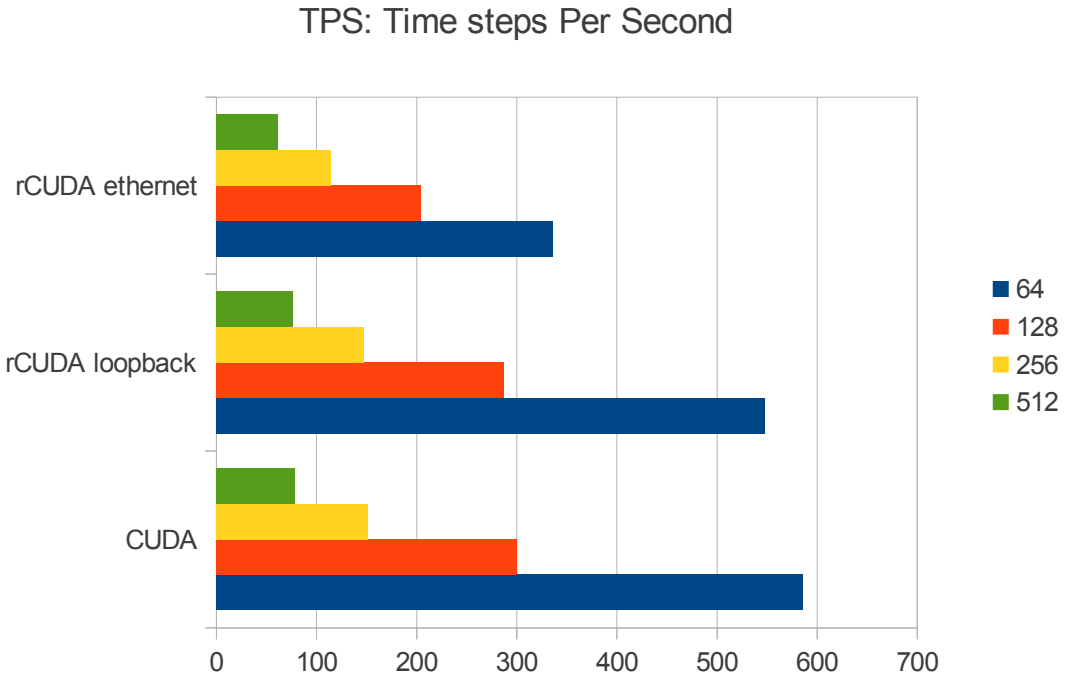

Figura 6.5: Benchmark B. Time steps Per Second.

Tabla 6.5: Benchmark B.TPS: Time steps Per Second.

Nº Partículas (x1000):	**64**	**128**	**256**	**512**
CUDA	586 σ=1,4	300 σ=0,4	151 σ=0,1	78 σ=0,1
rCUDA loopback	548 σ=2,9	287 σ=0,9	147 σ=0,2	76 σ=0,1
rCUDA Ethernet	336 σ=0,9	204 σ=0,4	114 σ=0,2	61 σ=0,2

Valores mayores de TPS indican un mayor rendimiento. Nuevamente los valores obtenidos en las pruebas CUDA y rCUDA loopback son parecidos, con variaciones de entre el 10 % en el peor caso y del 3 % en el mejor caso.

Como era de esperar la red Ethernet disminuye el rendimiento de manera apreciable. El rendimiento de rCUDA aumenta conforme aumenta el tamaño del problema.

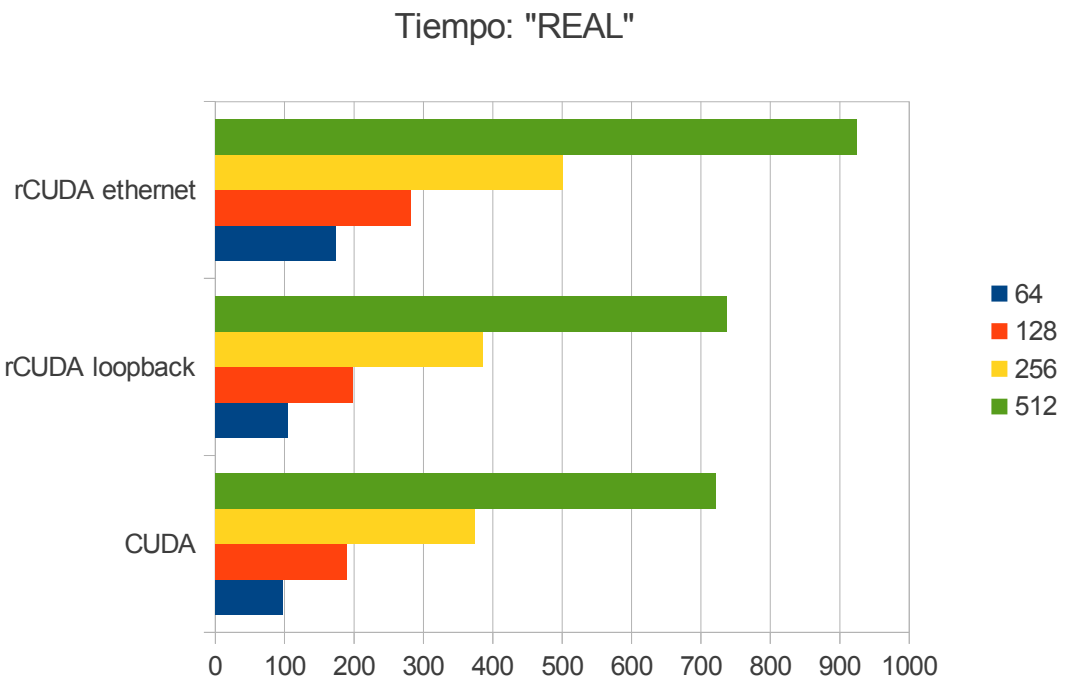

Figura 6.6: Benchmark B. Tiempo: "Real" (segundos).

Tabla 6.6: Benchmark B. Tiempo: "Real" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**
CUDA	98 σ=0,2	190 σ=0,3	374 σ=0,3	721 σ=0,5
rCUDA loopback	104 σ=0,5	198 σ=0,5	385 σ=0,6	738 σ=0,9
rCUDA Ethernet	174 σ=0,6	282 σ=0,5	501 σ=0,8	924 σ=1,4

Tiempo total de ejecución. Los tiempos requeridos para cada prueba son muy similares usando CUDA o usando rCUDA loopback, con pequeñas variaciones de entre el 6 % y el 3 % según el caso. De nuevo es apreciable el retardo introducido por el uso de Ethernet, así como la mejora del rendimiento al ir aumentando el tamaño de la simulación.

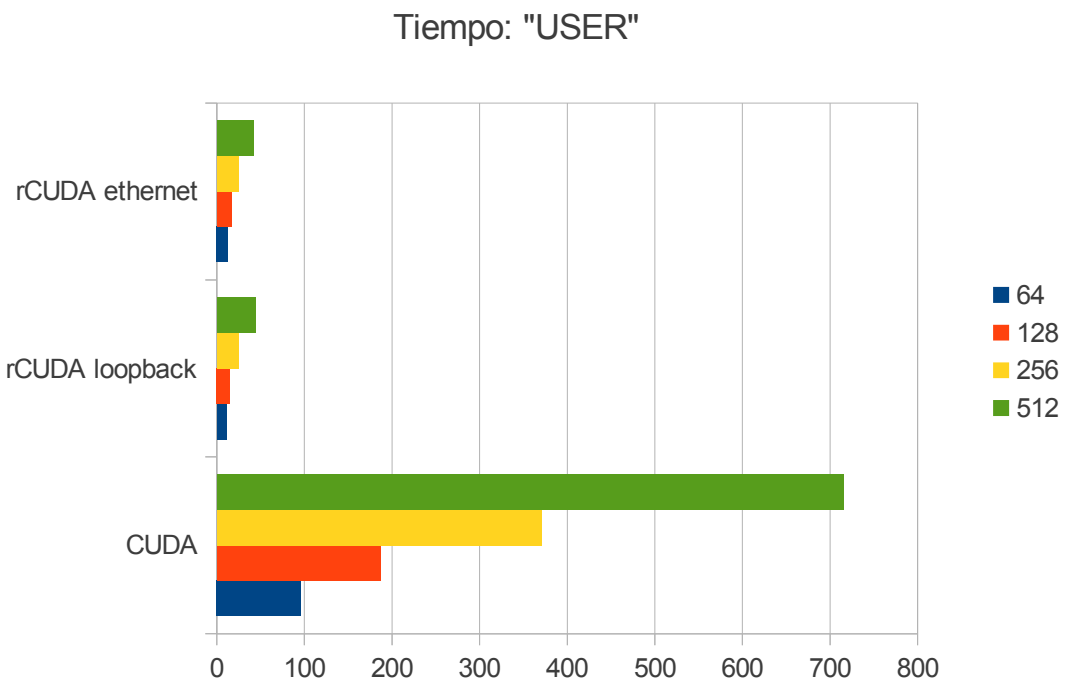

Figura 6.7: Benchmark B. Tiempo: "User" (segundos).

Tabla 6.7: Benchmark B. Tiempo: "User" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**
CUDA	96	187	371	715
rCUDA loopback	11	15	25	44
rCUDA Ethernet	13	17	25	42

Porción del tiempo de ejecución correspondiente a los procesos de usuario. En el caso de rCUDA Ethernet y rCUDA loopback es muy bajo debido a que la mayor parte de los procesos se ejecutan en remoto por medio de rCUDA.

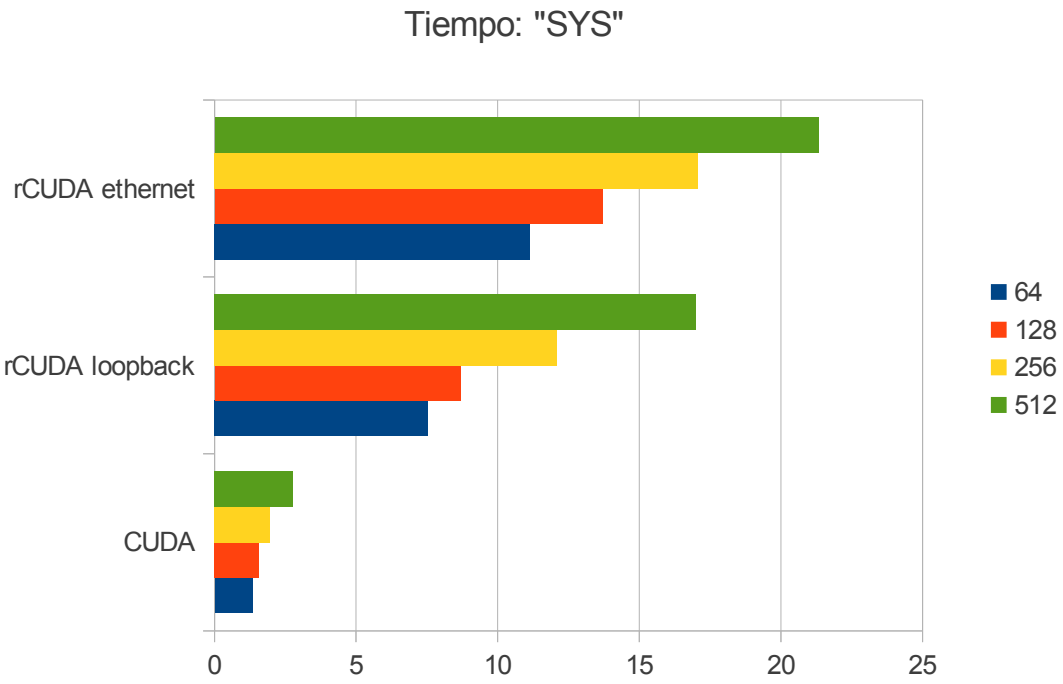

Figura 6.8: Benchmark B. Tiempo: "Sys" (segundos).

Tabla 6.8: Benchmark B. Tiempo: "Sys" (segundos).

Nº Partículas (x1000):	**64**	**128**	**256**	**512**
CUDA	1,3	1,6	2	2,8
rCUDA loopback	7,6	8,7	12,1	17
rCUDA Ethernet	11,1	13,7	17	21,3

Porción del tiempo de ejecución correspondiente a los procesos de sistema. Los tiempos de rCUDA tienen un gran contraste con los tiempos de CUDA, dado que el middleware rCUDA se ejecuta en tiempo de sistema.

A partir de los datos obtenidos, podemos calcular la penalización que supone el uso de rCUDA respecto a CUDA en los distintos casos. El incremento de tiempo para realizar las mismas simulaciones al usar rCUDA tanto con la interfaz loopback como a través de la red se muestra en las siguientes tablas en forma de porcentajes sobre el tiempo requerido al utilizar CUDA nativo:

Tabla 6.9: Benchmark A. Incremento de tiempo requerido por rCUDA (%)

Nº Partículas (x1000):	**64**	**128**	**256**	**512**	**1024**
rCUDA loopback	+10 %	+5 %	+3 %	+3 %	+2 %
rCUDA Ethernet	+88 %	+55 %	+38 %	+30 %	+26 %

Tabla 6.10: Benchmark B. Incremento de tiempo requerido por rCUDA (%)

Nº Partículas (x1000):	**64**	**128**	**256**	**512**
rCUDA loopback	+6 %	+4 %	+4 %	+2 %
rCUDA Ethernet	+78 %	+48 %	+34 %	+28 %

Las tablas nos muestran la tendencia que presenta el rendimiento de rCUDA según aumenta la carga de trabajo. En ambos benchmarks el rendimiento de rCUDA mejora de forma clara alcanzado un rendimiento equivalente al 98 % del obtenido al usar CUDA. Esta tendencia se mantiene también al usar Ethernet. Los peores rendimientos al usar Ethernet indican que la tecnología de red empleada puede tener más impacto en el rendimiento final obtenido que el introducido por el uso de rCUDA exclusivamente.

6.3. mCUDA-MEME: Virtualización de varias GPUs

Para el caso de virtualización de varias GPUs, hemos utilizado el programa mCUDA-MEME junto con MPI para utilizar hasta 4 tarjetas gráficas simultáneamente instaladas en un mismo nodo servidor. De la misma manera que se hizo con las pruebas usando HOOMD-blue, cada ejecución se repitió 10 veces para obtener los valores medios.

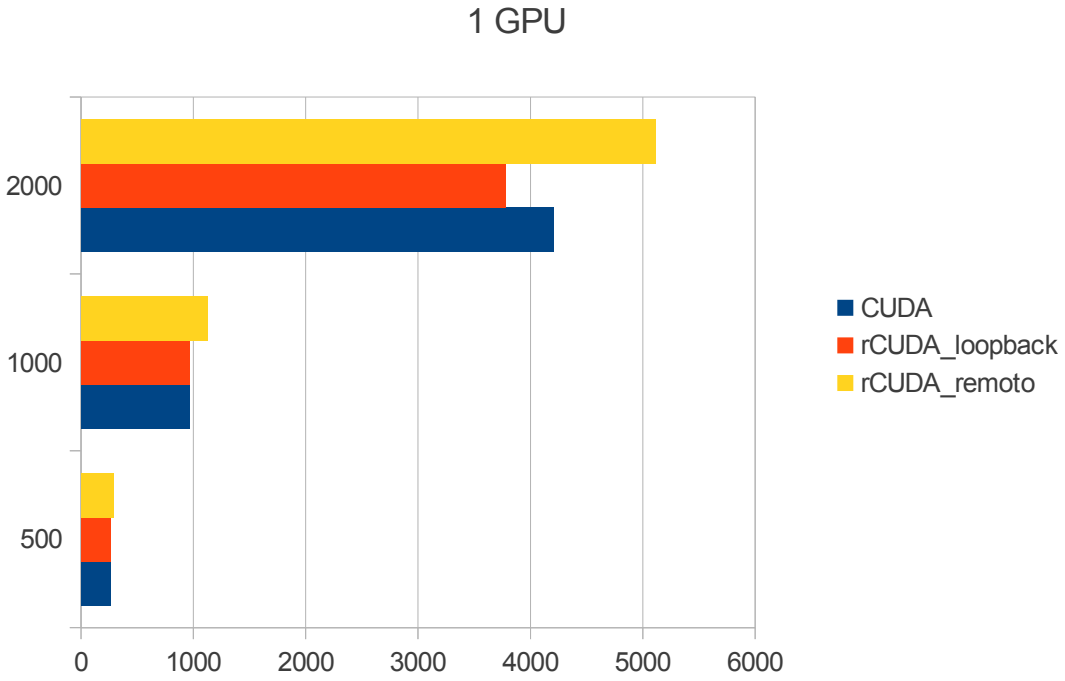

Figura 6.9: mCUDA-MEME usando 1 GPU (segundos).

Tabla 6.11: mCUDA-MEME usando 1 GPU (segundos).

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
CUDA	258 σ=2	970 σ=2	4202 σ=2
rCUDA loopback	264 σ=4	968 σ=8	3778 σ=14
rCUDA Ethernet	290 σ=1	1126 σ=3	5117 σ=1

Tiempo total de ejecución. Se observan mínimas variaciones entre los tiempos obtenidos por CUDA y rCUDA loopback, mientras que con rCUDA remoto usando Ethernet son superiores como es de esperar. En la prueba de 2000 secuencias, se observa que el tiempo requerido por rCUDA loopback es incluso inferior al requerido por CUDA. Las posibles causas de esta aparente anomalía son discutidas en el Capítulo 7.

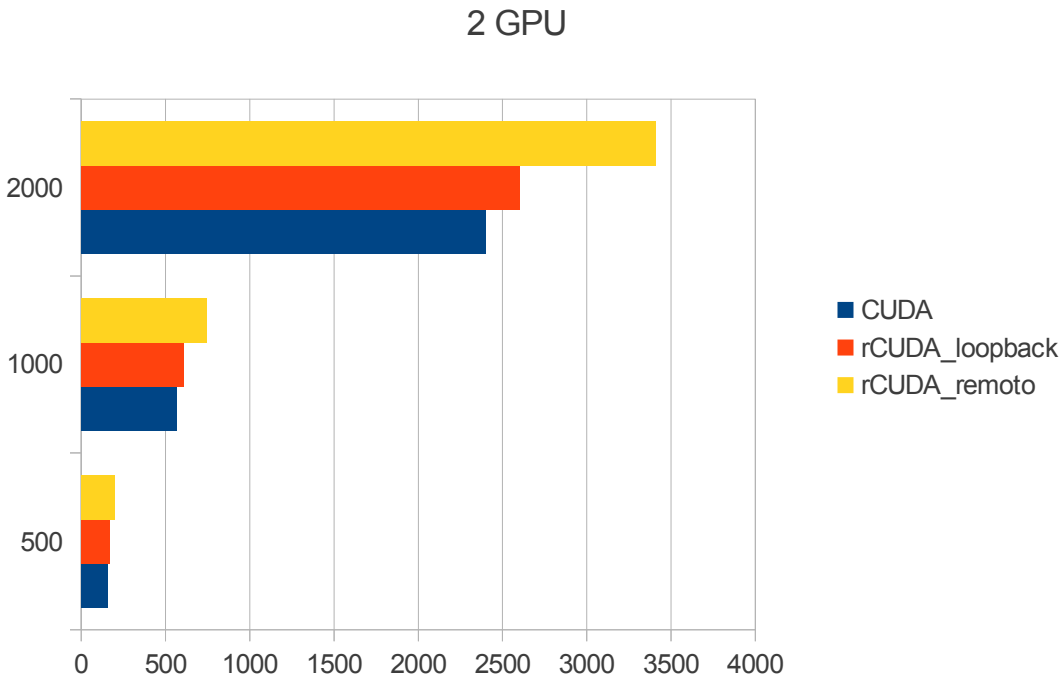

Figura 6.10: mCUDA-MEME usando 2 GPUs (segundos).

Tabla 6.12: mCUDA-MEME usando 2 GPUs (segundos).

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
CUDA	160 σ=2	567 σ=8	2401 σ=17
rCUDA loopback	171 σ=2	609 σ=4	2604 σ=8
rCUDA Ethernet	198 σ=1	745 σ=3	3409 σ=6

Tiempo total de ejecución. Los datos aparecen más escalonados en la gráfica con respecto a la prueba usando una única GPU, debido principalmente a que los tiempos obtenidos según se usa CUDA o rCUDA empiezan a distanciarse.

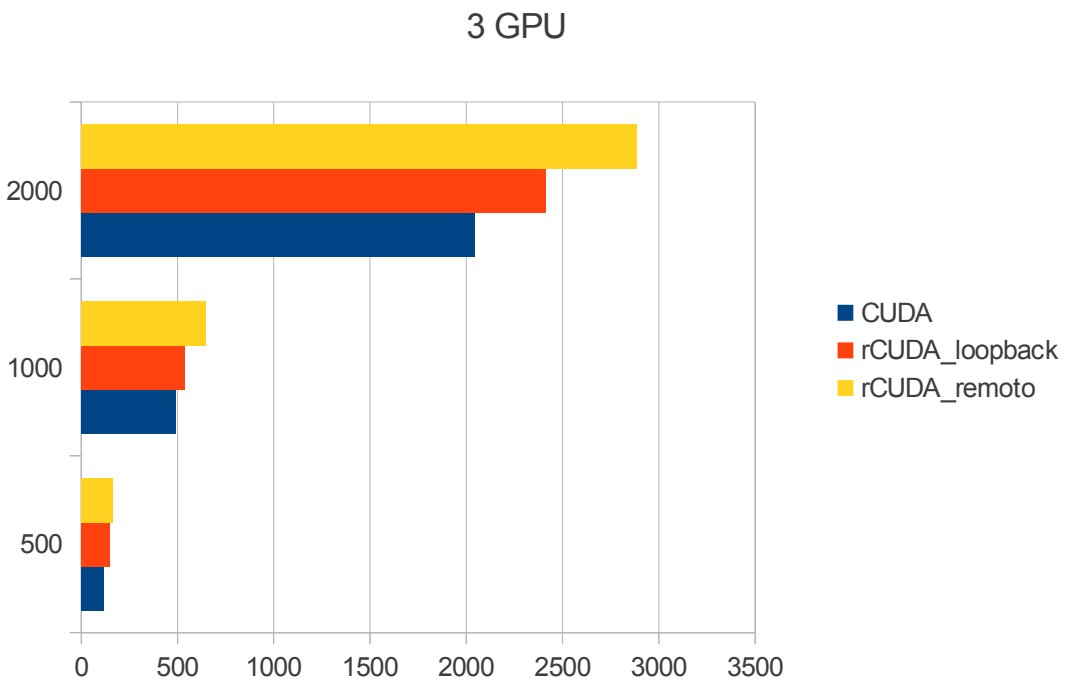

Figura 6.11: mCUDA-MEME usando 3 GPUs (segundos).

Tabla 6.13: mCUDA-MEME usando 3 GPUs (segundos).

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
CUDA	117 σ=1	490 σ=7	2045 σ=15
rCUDA loopback	146 σ=2	539 σ=0	2411 σ=7
rCUDA Ethernet	164 σ=1	647 σ=3	2885 σ=7

Tiempo total de ejecución. Se confirma la tendencia observada en la gráfica anterior, de que los tiempos requeridos por CUDA y rCUDA se diferencian más a medida que se usan más GPUs para realizar los cálculos.

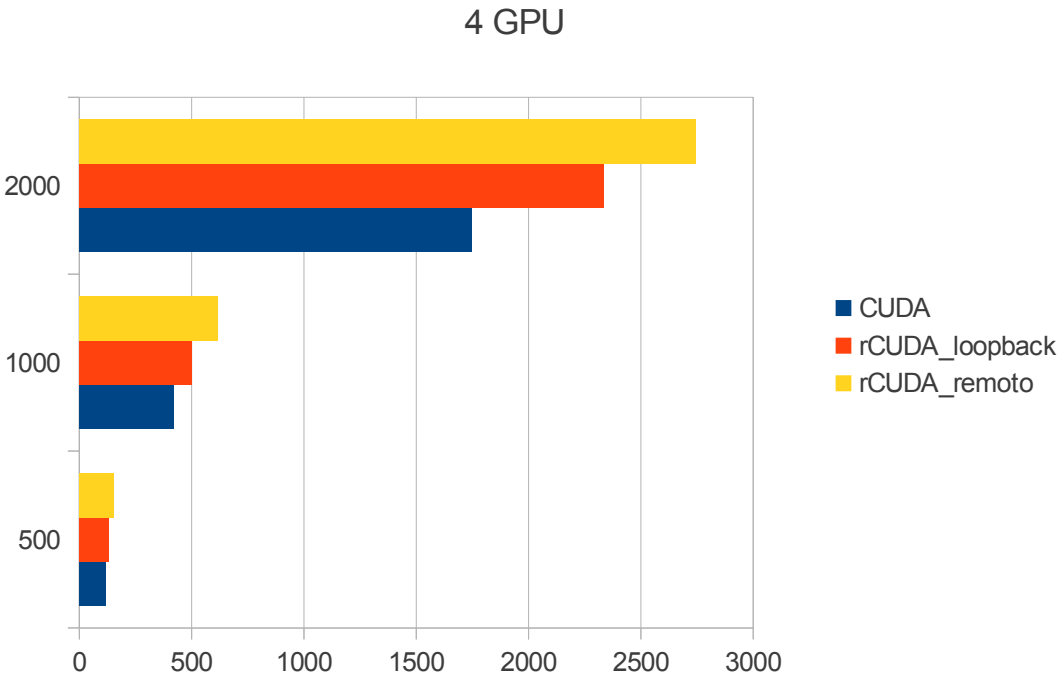

Figura 6.12: mCUDA-MEME usando 4 GPUs (segundos).

Tabla 6.14: mCUDA-MEME usando 4 GPUs (segundos).

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
CUDA	117 σ=1	420 σ=3	1749 σ=4
rCUDA loopback	130 σ=1	502 σ=2	2337 σ=5
rCUDA Ethernet	152 σ=3	615 σ=3	2744 σ=6

Tiempo total de ejecución. En el análisis de 2000 secuencias, los tiempos de rCUDA loopback y rCUDA Ethernet llegan a estar más próximos entre sí que los tiempos entre rCUDA loopback y CUDA como era habitual en las pruebas anteriores.

A continuación se muestran estos mismos resultados agrupados por los tamaños de cada benchmark para comparar el tiempo requerido en cada prueba en las distintas configuraciones de uso de GPUs:

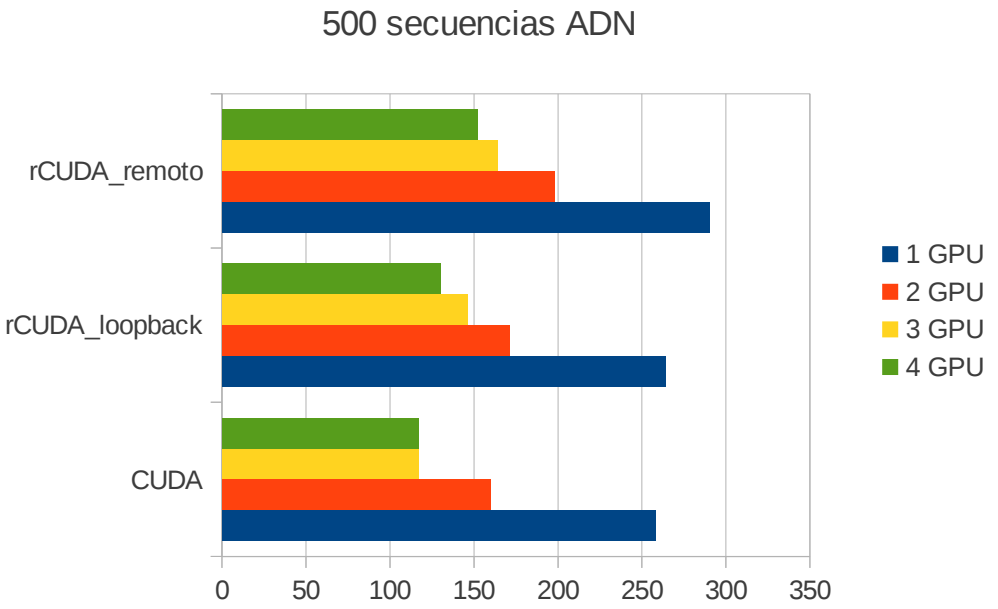

Figura 6.13: mCUDA-MEME usando de 1 a 4 GPUs para analizar 500 secuencias de ADN (segundos).

Tabla 6.15: mCUDA-MEME usando de 1 a 4 GPUs para analizar 500 secuencias de ADN (segundos).

500 secuencias ADN:	**1 GPU**	**2 GPU**	**3 GPU**	**4 GPU**
CUDA	258	160	117	117
rCUDA loopback	264	171	146	130
rCUDA Ethernet	290	198	164	152

El rendimiento por GPU disminuye rápidamente conforme participan más GPUs en los cálculos. En el caso de CUDA, no se obtiene ninguna ventaja por utilizar 4 GPUs en lugar de 3. Usando sólo una GPU, el rendimiento de rCUDA loopback es sólo ligeramente inferior al rendimiento de CUDA.

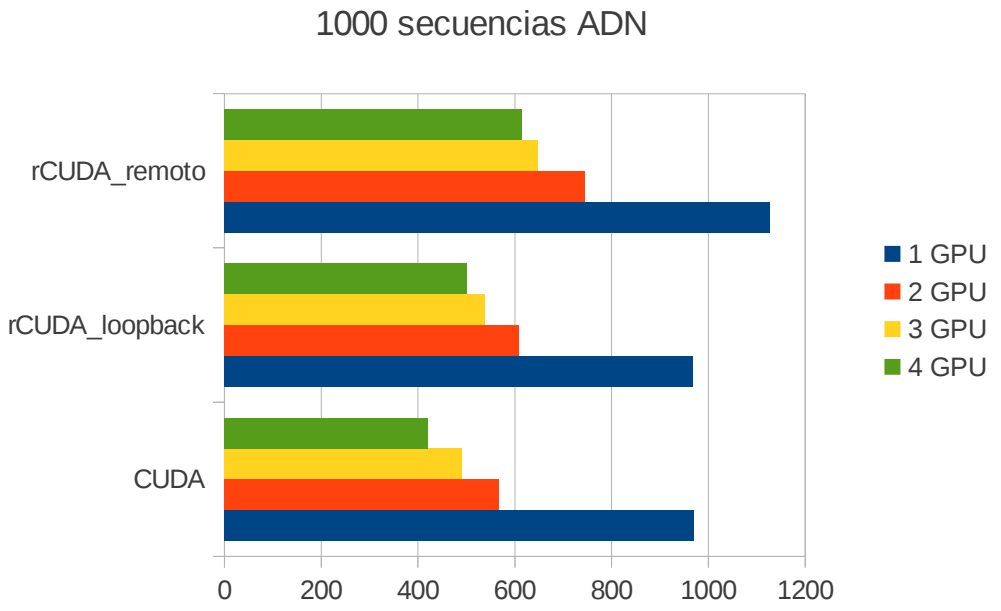

Figura 6.14: mCUDA-MEME usando de 1 a 4 GPUs para analizar 1000 secuencias de ADN (segundos).

Tabla 6.16: mCUDA-MEME usando de 1 a 4 GPUs para analizar 1000 secuencias de ADN (segundos).

1000 secuencias ADN:	**1 GPU**	**2 GPU**	**3 GPU**	**4 GPU**
CUDA	970	597	490	420
rCUDA loopback	968	609	539	502
rCUDA Ethernet	1126	745	647	615

Se acentúa la pérdida de rendimiento por GPU observada en el análisis de 500 secuencias. En el caso más favorable con CUDA, pasar de utilizar 2 a 3 GPUs sólo ofrece una mejora del 18 %, y pasar de utilizar 3 a 4 GPUs una mejora del 15 %. Usando sólo una GPU, el rendimiento de rCUDA loopback ha mejorado respecto al rendimiento de CUDA, siendo prácticamente idénticos.

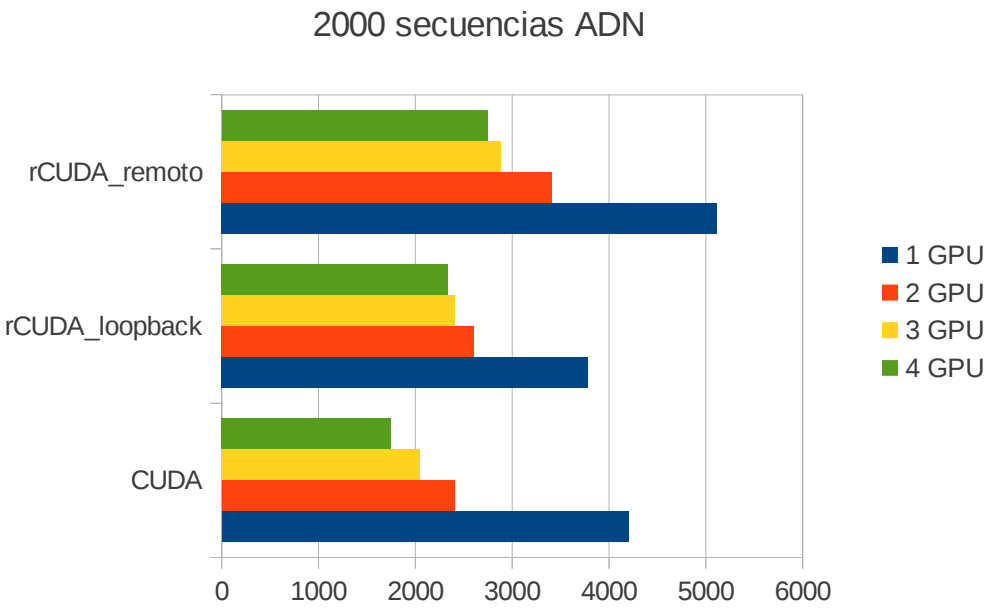

Figura 6.15: mCUDA-MEME usando de 1 a 4 GPUs para analizar 2000 secuencias de ADN (segundos).

Tabla 6.17: mCUDA-MEME usando de 1 a 4 GPUs para analizar 2000 secuencias de ADN (segundos).

2000 secuencias ADN:	**1 GPU**	**2 GPU**	**3 GPU**	**4 GPU**
CUDA	4202	2401	2045	1749
rCUDA loopback	3778	2604	2411	2337
rCUDA Ethernet	5117	3409	2885	2744

El rendimiento por GPU empeora en proporción al número de GPUs empleadas, y al tamaño de problema a resolver. Usando una sola GPU, el rendimiento de rCUDA es superior al de CUDA. En el Capítulo 7 se detallan algunas posibles explicaciones de este resultado.

En el caso de virtualización de varias GPUs, los incrementos en los tiempos de ejecución observados son:

Tabla 6.18: Incremento de tiempo usando rCUDA. Interfaz loopback.

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
1GPU	+2,3 %	-0,2 %	-10 %
2GPU	+0,7 %	+7 %	+8 %
3GPU	+25 %	+10 %	+18 %
4GPU	+11 %	+20 %	+34 %

Tabla 6.19: Incremento de tiempo usando rCUDA. Interfaz Ethernet.

Tamaño (nº secuencias ADN):	**500**	**1000**	**2000**
1GPU	+12 %	+16 %	+22 %
2GPU	+24 %	+31 %	+42 %
3GPU	+40 %	+32 %	+41 %
4GPU	+30 %	+46 %	+57 %

Los variaciones de los datos nos muestran que no hay una tendencia uniforme del rendimiento respecto al número de GPUs utilizadas para los cálculos, aunque por lo general el rendimiento diminuye conforme aumenta el número de GPUs.

Como última prueba de virtualización de varias GPUs, usamos rCUDA para virtualizar hasta 8 GPUs que se corresponden físicamente con una única GPU del nodo servidor. De esta manera, todos los procesos de mCUDA-MEME acaban usando en realidad la misma GPU, que es compartida gracias a rCUDA. En la ejecución con CUDA nativo, se usan secuencialmente de una a cuatro GPUs. En la ejecución con rCUDA usando la interfaz loopback, se usan secuencialmente de una a ocho GPUs virtuales, que están mapeadas a una única GPU física en el mismo nodo. En el caso de rCUDA remoto, con la GPU física en un nodo y las virtuales en otro, se usan hasta 8 GPUs virtuales mapeadas a la misma GPU real del nodo servidor.

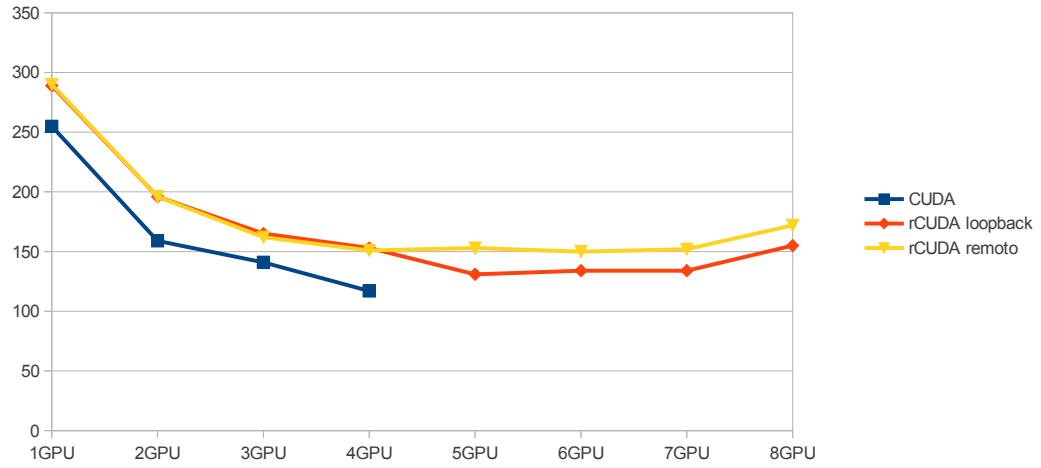

Figura 6.16: mCUDA-MEME: Uso de 1 a 8 GPUs virtuales con rCUDA (una GPU real) vs. Uso de 4 GPUs reales con CUDA (segundos).

Tabla 6.20: mCUDA-MEME: Uso de 1 a 8 GPUs virtuales con rCUDA (una GPU real) vs. Uso de 4 GPUs reales con CUDA (segundos).

Entorno:	**CUDA**	**rCUDA loopback**	**rCUDA remoto**
1 GPU	255	289	290
2 GPU	159	196	196
3 GPU	141	165	162
4 GPU	117	153	151
5 GPU	-	131	153
6 GPU	-	134	150
7 GPU	-	134	152
8 GPU	-	155	172

Capítulo 7

Conclusiones

En nuestras pruebas hemos tratado de medir la sobrecarga introducida por el middleware rCUDA usando dos aplicaciones científicas diseñadas para trabajar en entornos CUDA con tarjetas aceleradoras instaladas localmente en los nodos. Ambas aplicaciones han funcionado sin problemas y sin ninguna modificación expresa para adaptarlas al entorno rCUDA, lo que nos confirma que rCUDA consigue el objetivo de ser transparente de cara a las aplicaciones que requieren de aceleración GPU para funcionar. De los dos tipos de pruebas realizadas, virtualización de 1 GPU y virtualización de varias GPUs, hemos obtenido resultados aparentemente dispares.

Las pruebas realizadas con HOOMD-Blue para evaluar el rendimiento de la virtualización de un única GPU muestran claramente que la sobrecarga debida a rCUDA es muy pequeña incluso en los peores casos, cuando se realizan simulaciones pequeñas. Además, dicha sobrecarga disminuye uniformemente conforme aumenta el tamaño de las simulaciones lo cual ayuda a compensar el sobrecoste de realizar las transferencias a la GPU por la red cuando se utiliza rCUDA. Con una sobrecarga de sólo el 2 % en las simulaciones de mayor tamaño al usar la interfaz loopback del sistema operativo, todavía existe un margen de mejora mediante el uso de tecnologías de red y protocolos óptimos para la interconexión de los nodos en un clúster HPC.

En el caso de mCUDA-MEME para analizar el rendimiento de la virtualización de varias GPUs, nos encontramos con algunas particularidades que pasamos a comentar. En primer lugar, vemos que a la hora de usar una única GPU, el rendimiento que presenta rCUDA mejora con respecto a CUDA conforme aumentamos el tamaño del análisis, incluso superándolo en el análisis de 2000 secuencias. Este hecho puede parecer paradójico porque en última instancia todas los cálculos los realiza el Runtime de CUDA del nodo servidor, por lo que el uso de rCUDA no podría en principio mejorar este dato. La mejora del rendimiento apreciada se puede atribuir a los siguientes puntos:

- Inicialización del driver de CUDA: en rCUDA ha sido inicializado previamente por el demonio de rCUDA, por lo que el tiempo de inicialización es mucho más pequeño en rCUDA que en CUDA (donde se debe iniciar con cada ejecución). Si fuera esta la razón, el análisis de 500 secuencias debería ser más eficiente con rCUDA loopback que con CUDA, pero esto no sucede, por lo que sería improbable esta explicación.

- Puntos de sincronización: determinadas llamadas de sincronización (cudaDeviceSynchronize o cudaStreamSynchronize), son más eficientes en rCUDA para determinados casos.

- Polling interno de la red: el periodo de polling a la red beneficia a rCUDA para determinados casos.

Un estudio más profundo de este tema requeriría mucho tiempo y análisis de código, lo cual está fuera del alcance de este PFC.

Otra peculiaridad de los resultados de las pruebas con mCUDA-MEME es que el rendimiento por GPU disminuye a medida que participan más GPUs en el análisis de las secuencias. Esta circunstancia se da incluso cuando se utiliza CUDA localmente, por lo que debemos concluir que la apliación mCUDA-MEME presenta problemas de escalabilidad. Es de esperar entonces que el

comportamiento de rCUDA, tanto si se usa con la interfaz loopback del sistema operativo como cuando se usa la red ethernet, evolucione de forma similar y con el sobrecoste añadido de usar cada GPU de manera remota.

En la última prueba con mCUDA-MEME en la que hemos virtualizado más GPUs de las realmente existentes en el nodo hemos llegado a obtener tiempos de ejecución muy próximos entre rCUDA loopback y CUDA. En el caso de rCUDA loopback usando 5 GPUs virtuales correspondientes a una única GPU real y CUDA utilizando las 4 GPUs reales instaladas en el nodo, sólo hay una diferencia de alrededor del 10 % en el tiempo de ejecución. Esto nos indica que mCUDA-MEME no hace un uso eficiente de las GPUs disponibles o su rendimiento está limitado por otros factores. Además, sólo una parte de este 10 % es debida al uso de rCUDA, ya que existen otros condicionantes como por ejemplo el uso de un protocolo no optimizado para este uso como es TCP/IP.

Estos resultados por tanto no entran en conflicto con los obtenidos en las pruebas de virtualización de una única GPU, ya que nos indican que de manera general la sobrecarga introducida por rCUDA llega a ser despreciable cuando se realizan cálculos con mayores volúmenes de datos.

Bibliografía

[1] NVIDIA, "The NVIDIA CUDA API Reference Manual", NVIDIA, 2013.

[2] Carlos Reaño. CU2rCU: A CUDA-to-rCUDA Converter, Marzo 2013 http://hdl.handle.net/10251/27435

[3] HOOMD-Blue: http://codeblue.umich.edu/hoomd-blue

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units Journal of Computational Physics 227(10): 5342-5359, May 2008. http://dx.doi.org/10.1016/j.jcp.2008.01.047

[5] mCUDA-MEME: https://sites.google.com/site/yongchaosoftware/mcuda-meme

[6] Yongchao Liu, Bertil Schmidt, Weiguo Liu, Douglas L. Maskell: ÇUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units". Pattern Recognition Letters, 2010, 31(14): 2170 - 2177 http://www.hicomb.org/papers/HICOMB2011-03.pdf

[7] Yongchao Liu, Bertil Schmidt, Douglas L. Maskell: .ªn ultrafast scalable many-core motif discovery algorithm for multiple GPUs". 10th IEEE International Workshop on High Performance Computational Biology (HiCOMB 2011), 2011, 428-434 http://dx.doi.org/10.1016/j.patrec.2009.10.009

[8] A. J. Peña. "Virtualization of Accelerators in High Performance Clusters", Ph.D. thesis, University Jaume I of Castellón 2013