View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositori Institucional de la Universitat Jaume I

DEPARTAMENTO DE INGENIERÍA Y CIENCIA DE LOS COMPUTADORES
UNIVERSITAT JAUME I

SIU043
TRABAJO FIN DE MÁSTER

MÁSTER EN SISTEMAS INTELIGENTES
Curso 2013 / 2014

Evaluación de prestaciones mediante la aplicación HPL de clusters utilizando rCUDA

Adrián Castelló Gimeno

Dirigido por ***Rafael Mayo Gual***

Castellón, a 30 de julio de 2014

Resumen

A lo largo de este documento se describe el proyecto realizado en la asignatura SIU043-Trabajo Fin de Máster. Este trabajo se ha llevado a cabo en el grupo de investigación *High Performance Computing and Architectures* del *Departamento de Ingeniería y Ciencia de los Computadores* de la *Universitat Jaume I* bajo la supervisión de Rafael Mayo Gual.

El proyecto se ha centrado en la evaluación del rendimiento mediante el uso de la aplicación *Linpack Benchmark* del software *rCUDA*. Este software permite la ejecución de una aplicación CUDA en un nodo que no disponga de ninguna GPU instalada, utilizando mediante la red de interconexión una GPU instalada en otro nodo como si fuera local.

El objetivo de este trabajo es dotar a rCUDA de la funcionalidad necesaria para poder ejecutar este test y posteriormente analizar las prestaciones obtenidas. Estas prestaciones deben de ser comparadas con la ejecución de este mismo test sobre un nodo utilizando CUDA.

Palabras clave

GPU, CUDA, rCUDA, Linpack Benchmark, Virtualización, HPC

Agradecimientos

En primer lugar quiero agradecer a mi familia todo el apoyo mostrado durante toda mi vida, tanto en los buenos momentos como en las situaciones más difíciles. Además, quiero dar especialmente las gracias a mi querida Araceli ya que es la que más sufre mis trabajos.

También quiero dar las gracias a los miembros del grupo de investigación *High Performance Computing and Architectures* por hacerme sentir como en casa. Quiero agradecer a Rafael Mayo su dedicación y ayuda sin la cual este proyecto no hubiera sido posible. Además, agradecer a Carlos Reaño y Federico Silla de la Universitat Politècnica de València su apoyo e ideas durante este proyecto.

Un agradecimiento especial para mis compañeros del *Crazy Hector's Lab* con quien he compartido los últimos tres años.

Índice general

Índice general I

Índice de figuras III

Índice de cuadros V

1 Introducción 1
1.1. Motivación del proyecto 1
1.2. Objetivo del proyecto 2
1.3. Planificación Temporal 2

2 Entorno de trabajo 5
2.1. Linpack Benchmark 5
2.2. rCUDA 8

3 Modificaciones 11
3.1. Añadiendo funcionalidad 11
3.2. Resolviendo errores 16

4 Resultados 21
4.1. HPL-CUDA 21
4.2. HPL-Fermi 24

5 Conclusiones y trabajo futuro 35
5.1. Utilidad del proyecto 35
5.2. Dificultades 36
5.3. Conocimientos adquiridos 36
5.4. Trabajo futuro 36

Bibliografía 37

Índice de figuras

2.1. Arquitectura Cliente-Servidor de rCUDA 9

3.1. Transferencia de un bloque de datos de una matriz 18
3.2. Temporización de las posibles soluciones para el solapamiento de streams 20

4.1. GFlops obtenidos con distintos tamaños de bloque en CUDA y rCUDA 23
4.2. Máximos GFlops obtenidos con CUDA y rCUDA 23
4.3. GFlops obtenidos para 1 y 2 GPUs remotas 24
4.4. GFlops obtenidos utilizando CUDA y rCUDA en el HPL Fermi . . 25
4.5. Tratamiento de las peticiones dentro de streams en rCUDA 26
4.6. Orden de envío de las peticiones dentro de streams en rCUDA . . 26
4.7. GFlops obtenidos sobre una GPU con CUDA, rCUDA original y la nueva implementación de rCUDA. 29
4.8. Transferencias realizadas por el HPL. 31
4.9. Repercusión del tamaño de bloque en el tiempo de ejecución. . . . 32
4.10. GFlops obtenidos para 1, 2 y 4 GPUs locales. 32
4.11. GFlops obtenidos para 1 y 2 GPUs remotas. 33

Índice de cuadros

1.1. Planificación temporal. 3

4.1. GFlops para distintos tamaños de bloque para CUDA y rCUDA. . 22
4.2. GFlops obtenidos sobre una GPU con CUDA, rCUDA original y la nueva implementación de rCUDA. 28

Capítulo 1

Introducción

Índice

1.1. Motivación del proyecto **1**
1.2. Objetivo del proyecto **2**
1.3. Planificación Temporal **2**

En este capítulo se describe la motivación del proyecto y los objetivos del mismo.

1.1. Motivación del proyecto

En los últimos años se ha establecido la tendencia de crear supercomputadores no solamente por mejorar la capacidad de cómputo de los centros, sino también por la gran repercusión que se obtiene al aparecer en las conocidas listas Top500 [1] y Green500 [2].

La generación de estas listas se realiza a partir de los datos de rendimiento de la ejecución del test Linpack Benchmark (HPL) [3, 4]. En el caso de la lista Green500, se dividen los GFlops entre los Vatios de potencia consumidos. Para obtener mejores resultados, muchos de estos supercomputadores, incluyen aceleradores de cómputo como son NVIDIA [5] GPUs o Intel Xeon Phi [6]. Esta tendencia es más clara en la lista Green500 donde el top 10 de supercomputadores en Noviembre de 2013 utilizaban NVIDIA Tesla K20.

Debido a la importancia que posee esta aplicación en el campo de la computación de altas prestaciones (HPC del inglés High Performance Computing), se va a realizar un estudio de prestaciones utilizando como aceleradores GPUs remotas, y estos resultados se van a comparar con la ejecución sobre GPUs

locales para obtener el sobrecoste de la virtualización en entornos donde los recursos hardware son exprimidos al máximo.

Desde un punto de vista más útil para la investigación, se desea contruir supercomputadores para simular o predecir algunos de los retos más importantes para la humanidad como son por ejemplo: el cambio climático [7] o la simulación del cerebro humano [8]. Estos retos requieren de un gran poder computacional pero sin perder de vista el coste tanto de adquisición como de mantenimiento.

rCUDA [9, 10, 11, 12] es una tecnología que permite el uso de GPUs instaladas en un nodo remoto como si se encontraran físicamente instaladas de forma local. Además permite utilizar a un único nodo todas las GPUs del sistema, evitando así la restricción actual que solamente permite a un nodo utilizar las GPUs instaladas en ese mismo nodo para ejecutar una aplicación CUDA [13].

Actualmente existen 2 versiones del test *LINPACK*: el estándar [14], que acelera el código utilizando GPUs y el desarrollado por NVIDIA especialmente para sus tarjetas gráficas [15].

1.2. Objetivo del proyecto

El objetivo principal es la evaluación de las prestaciones de un cluster mediante la aplicación *LINPACK* (HPL) utilizando CUDA (software desarrollado por nVIDIA) y utilizando un software de virtualización de GPUs remotas como es rCUDA. Para alcanzar el objetivo principal, se han de realizar los siguientes pasos:

- Instalar HPL-cuda en un clúster
- Instalar HPL-nVIDIA-Fermi en un clúster
- Adaptar la tecnología rCUDA para poder ejecutar estos tests.
- Evaluar las prestaciones obtenidas.

1.3. Planificación Temporal

En el Máster de Sistemas Inteligentes de la Universitat Jaume I, se ha estipulado que la duración del proyecto final de máster sea de 300h. Este tiempo se ha distribuido tal y como se observa en la Tabla 1.1.

Tarea	Horas planificadas
Estudiar las aplicaciones HPL	10
Instalación de aplicaciones	20
Ejecución HPLs utilizando CUDA	60
Adaptar código rCUDA	120
Ejecución HPLs utilizando rCUDA	60
Redacción de memoria	30

Cuadro 1.1: Planificación temporal.

Capítulo 2

Entorno de trabajo

Índice

2.1. Linpack Benchmark 5
2.2. rCUDA 8

En esta sección se describe el *Linpack Benchmark (HPL)* y el software *rCUDA*.

2.1. Linpack Benchmark

El Linpack Benchmark (HPL del Inglés High-Performance Linpack Benchmark) es un software que resuelve un sistema lineal aleatorio de doble precisión aritmética (64 bits) en computadores de memoria distribuida. Se distingue por la libre implementación del test pudiendo ser optimizado para cada tipo de computador y/o arquitectura.

El HPL fue desarrollado en el Argone National Laboratory por Jack Dongarra en 1976, y es uno de los más usados en sistemas científicos y de ingeniería para el cálculo de prestaciones.

Las principales características del algoritmo utilizado por el HPL son:

- Distribución de datos cíclica de bloques de dos dimensiones.
- Variante derecha de la factorización LU con pivotamiento parcial.
- Factorización recursiva con pivotamiento y reenvío de columna.
- Distintas topologías virtuales de reenvíos.

- Algoritmo de reducción de reenvío para ancho de banda.

Los pasos en la ejecución del Linpack Benchmark son los que se observan en el Código 2.1:

Código Fuente 2.1: Pasos de la ejecución del HPL

```
/* Genera y particiona la matriz entre los nodos o */
/* procesos MPI */
MPI_Barrier(...); /* Todos los nodos empiezan a la vez. */
HPL_ptimer(...); /* Se inicia el tiempo. */
HPL_pdgesv(...); /* Se resuelve el sistema de ecuaciones. */
HPL_ptimer(...); /* Se para el tiempo. */
MPI_Reduce(...); /* Se obtiene el tiempo m\'aximo. */
/* Obtiene estadisticas sobre rendimiento (basandose en el
tiempo maximo)  y precision del resultado. */
/* ... */
```

La resolución del sistema de ecuaciones se realiza del siguiente modo:

1. Se crea un sistema de ecuaciones:

$$Ax = b; \quad A \in \Re^{nxn}; \quad x, b \in \Re^{n};$$

2. Primero se calcula la factorización LU con pivotamiento parcial de los coeficientes n y n+1 de la matriz [A, b]:

$$P_r[A, b] = [[LU], y]; \quad P_r, L, U \in \Re^{nxn}; \quad y \in \Re^{n};$$

3. Una vez el pivotamiento (representado por la permutación de la matriz $\mathrm{P_r}$) y la factorización inferior se aplican sobre b, la solución se obtiene en un paso resolviendo el sistema superior triangular:

$$Ux = y;$$

La matriz triangular inferior izquierda y el conjunto de pivotes no se devuelven en el resultado.

El HPL ofrece un programa de test y de temporización para cuantificar la precisión de la solución obtenida así como el tiempo de ejecución del problema. Obtener el máximo rendimiento del sistema depende de una gran variedad de factores. Sin embargo, con algunas restricciones por parte de la red de interconexión, el algoritmo descrito y su implementación son escalables en el sentido de que se mantiene constante su eficiencia con respecto al uso de la memoria por procesador.

El HPL necesita, para su ejecución, una versión de MPI [16] (del Inglés Message Passing Interface) para realizar la transferencia de datos entre computadores y una versión de BLAS [17] (del Inglés Basic Linear Algebra Subprograms) para realizar los cálculos.

Al realizar esencialmente cálculos con matrices es un test fácilmente paralelizable, y se puede utilizar para medir la eficiencia de sistemas multiprocesador. Por este motivo, es el test elegido para la composición de las listas Top500 y Green500.

Aunque es el resultado de este test el que regula la aparición en las listas de supercomputadores, ésto no implica que el test que ejecuta cada supercomputador sea idéntico ya que, el único interés radica en el resultado. Por este motivo existe gran variedad de tests HPL, cada uno con unas características, creados específicamente para un tipo de arquitecturas. Es por ello que actualmente existen versiones para supercomputadores tanto con GPUs como sin GPUs, con Intel Xeon Phi, etc...

Desde el punto de vista de las GPUs, hay dos versiones que utilizan CUDA: un test genérico y otro desarrollado y optimizado por NVIDIA para sus GPUs. También existen versiones del mencionado test para OpenCL [18]. En los siguientes puntos se explican las características de las vesiones que utilizan la tecnología de NVIDIA.

2.1.1. HPL-CUDA

La primera implementación del HPL a analizar es la que se puede descargar de forma gratuita desde el repositorio git `https://github.com/avidday/hpl-cuda`. Se trata de un HPL muy básico, sin ninguna optimización para ninguna arquitectura. Únicamente añade las funciones CUDA necesarias para que la ejecución del problema se realice en la memoria de la GPU.

Este HPL utiliza las siguientes bibliotecas:

- GoToBLAS [19]. Para el cómputo sobre la CPU.
- OpenMPI [20]. Para la comunicación entre procesos.
- CUDA. Para el cómputo del problema utilizando tarjetas gráficas.
- cuBLAS [21]. Biblioteca BLAS para GPUs.

El uso de este test servirá como toma de contacto tanto para la instalación como para la puesta en marcha debido a su simplicidad. Además servirá para comprobar la correcta comunicación entre computadores al ejecutarlo sobre rCUDA.

2.1.2. HPL-Fermi

El algoritmo de este HPL [22] ha sido optimizado por NVIDIA para que, al ejecutarse sobre las GPUs, obtenga el máximo rendimiento de estos aceleradores gráficos. La versión sobre la que se va a realizar el estudio es la 15. Esta versión está optimizada para la arquitectura Fermi de las GPUs.

Como ya se ha comentado, esta versión explota el paralelismo que las GPUs más recientes ofrecen para el cálculo y además de realizar los cálculos sobre la GPU utiliza las siguientes características de CUDA:

- Streams CUDA.
- Solapamiento de cálculo y envío de datos.
- Memoria no paginable.

Con el uso de streams, la ejecución consigue crear 4 flujos de ejecución donde no solamente se realizan las copias de los datos sino que también se realizan las ejecuciones de las funciones CUBLAS. El uso de las copias asíncronas sigue el mismo objetivo que los streams y no es otro que poder encolar todas las transferencias para que la GPU no se detenga entre cálculo de un bloque y cálculo del siguiente. Gracias a estos dos mecanismos, se consigue solapar envío y cálculo de datos para un mejor rendimeinto. La reserva de memoria no paginable permite un mayor ancho de banda en el acceso a los datos ya que la GPU accede a los datos almacenados en la memoria RAM. En caso de contar con una red de interconexión InfiniBand, se utiliza la tecnología GPUDirect que permite un ancho de banda mayor entre GPUs instaladas en distintos nodos.

Esta versión del HPL utiliza las siguientes bibliotecas:

- MKL [23]. Biblioteca de Intel para las funciones BLAS.
- OpenMPI. Para la comunicación entre procesos.
- CUDA. Para el cómputo del problema utilizando tarjetas gráficas.
- cuBLAS. Biblioteca BLAS para GPUs.

Esta versión del software se utiliza para probar rCUDA en un entorno con mucha carga de actividad (transferencias asíncronas y cálculo) y sirve para comprobar el verdadero rendimiento que un cluster con GPUs remotas puede ofrecer en comparación con un cluster que utiliza GPUs locales.

2.2. rCUDA

Como ya se ha comentado en la introducción, la tendencia en los clusters HPC es la de añadir uno o varios aceleradores gráficos en cada nodo a pesar de que esta acción conlleva algunas desventajas:

- Elevado consumo energético.
- Elevado coste de adquisición.
- Baja utilización de los dispositivos.

Con el propósito de minimizar estas desventajas se ha desarrollado rCUDA (remote CUDA). rCUDA es un software que permite el acceso transparente a una GPU que está instalada en un nodo remoto. Este acceso pasa desapercibido para la aplicación que se ejecuta en el nodo cliente. Este software se estructura con una arquitectura distribuida cliente-servidor tal y como se muestra en la Figura 2.1

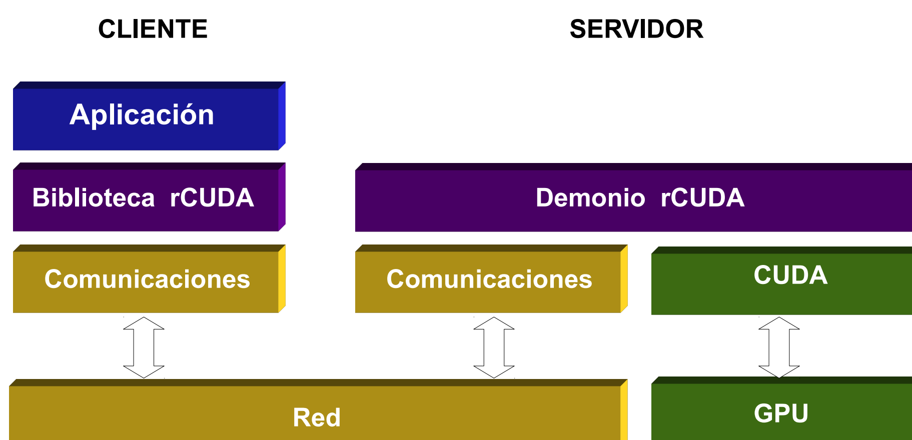

Figura 2.1: Arquitectura Cliente-Servidor de rCUDA

El software cliente (alojado en el nodo cliente) es llamado por la aplicación que demanda los servicios de una GPU. El cliente rCUDA ofrece la misma interfaz que la biblioteca CUDA Runtime API de NVIDIA [24] consiguiendo que el usuario no sea consciente de que está interactuando con rCUDA en lugar de una GPU local. El funcionamiento común de esta interacción es el siguiente:

- El software cliente recibe una petición CUDA de la aplicación.
- La petición es procesada y enviada al servidor.
- El servidor en el nodo remoto interpreta y ejecuta la petición sobre una GPU real.
- Cuando la GPU finaliza la ejecución, el servidor devuelve el resultado al cliente.
- El cliente devuelve el resultado a la aplicación en ejecución.

Para permitir el uso compartido de las GPUs entre varios procesos y que los eventos se ejecuten correctamente, rCUDA define distintos contextos independientes para cada aplicación. Esta característica permite alcanzar un gran porcentaje en cuanto a tiempo de utilización del dispositivo se refiere.

La comunicación entre los clientes rCUDA y los servidores (donde se encuentra la GPU real) se realiza a través de la capa de comunicaciones modular. rCUDA presenta una API propia que permite implementar un módulo de comunicaciones para cada tipo de red de interconexión. Actualmente, las redes de interconexión soportadas son Ethernet e InfiniBand.

La versión actual del software es rCUDA-4-1 que ofrece compatibilidad con CUDA 5.5 (Runtime API) a excepción de las funciones gráficas ya que son de poco interés en el campo del HPC además de bibliotecas como cuBLAS o cuFFT.

En general, las prestaciones esperadas con el uso de rCUDA serán menores al compararlas con el uso de una GPU local ya que la GPU a la que se accede está en un nodo remoto y en esta transferencia, el sobrecoste de la red de comunicaciones interviene en la pérdida de prestaciones.

Capítulo 3

Modificaciones

Índice

3.1. Añadiendo funcionalidad **11**
3.2. Resolviendo errores **16**

En esta sección se describen la funcionalidad añadida y modificaciones en el código de rCUDA para permitir la ejecución de los test HPL.

3.1. Añadiendo funcionalidad

Como se ha explicado en el capítulo anterior, rCUDA solamente soporta la biblioteca Runtime API de CUDA. Debido a que ambas versiones del HPL hacen uso de algunas funciones de la NVIDIA Driver API, ha sido necesario añadir código para permitir que estos tests se puedan ejecutar sobre rCUDA.

Gracias al código modular de rCUDA, añadir nueva funcionalidad es un proceso mecánico por lo que para todas las modificaciones se han seguido los mismos pasos:

- Añadir función en el código del cliente. En caso de que la función esté implementada por la Runtime API, se redirige a la función ya implementada.

- Crear estructura de datos propia para la función nueva.

- Añadir la función al código del servidor en caso de que fuera necesario.

La versión de rCUDA inicial (4.0.1) no incluía soporte para la versión Legacy de cuBLAS (solamente la versión v2 de cuBLAS era soportada). Debido a que el HPL utiliza la primera versión de cuBLAS, se ha utilizado una vieja biblioteca de rCUDA que daba soporte tanto a cuBLAS Legacy como a cuBLAS v2, pero solamente para simple precisión. Como el HPL utiliza la doble precisión, todas las funciones de cuBLAS utilizadas se han tenido que implementar en rCUDA. Para ello, el procedimiento será el mismo que en la Driver API.

Como no existe ningún módulo para las funciones de la Driver API, se han creado tanto los ficheros en la parte cliente como en la servidor y se ha modificado el Makefile para que cree la biblioteca dinámica libcuda.so

3.1.1. HPL-CUDA

Para esta versión solamente ha sido necesario añadir el soporte para la función *cuMemGetInfo* en lo que a la Driver API se refiere. Esta función obtiene la cantidad de memoria total y disponible de la GPU. Como esta función ya se encontraba en la biblioteca CUDA Runtime API (*cudaMemGetInfo*), se ha redirigido la llamada tal y como se muestra en en el Código 3.1.

Código Fuente 3.1: cuMemGetInfo

```
//Gets free and total memory
CUresult CUDAAPI cuMemGetInfo(size_t * free, size_t *   total){
    return (CUresult)cudaMemGetInfo(free, total);
}
```

En cuanto a las funciones de la biblioteca cuBLAS, tres funciones han sido añadidas:

- *cublasDscal*. Realiza el escalado de un vector y lo reescribe.
- *cublasDtrsm*. Resuelve un sistema lineal triangular.
- *cublasDgemm*. Realiza el producto de matrices.

Se trata de tres funciones asíncronas que pertenecen a la biblioteca de cuBLAS de doble precisión. Para añadir la funcionalidad al software rCUDA, se ha añadido la lógica necesaria tanto en la parte cliente como en la parte servidor. Además, se han añadido las estructuras de datos necesarias.

A continuación se muestra un ejemplo del código para la función *cublasDscal*. Para las otras dos funciones el procedimiento sería el mismo.

Primeramente, se han creado las estructuras de datos necesarias para cada función. El patrón es el que se muestra en el Código 3.2. Se crea un campo por cada parámetro de la llamada a la función.

Código Fuente 3.2: cublasDscal común

```
struct stCublasDscal {
  int                      n;
  double                   alpha;
  double                   *x;
  int                      incx;
};
```

El patrón que siguen las funciones añadidas en la parte del cliente es el mostrado en el Código 3.3. Donde los parámetros de la función original se añaden a la estructura de datos creada en el paso anterior y se envían a la función encargada de gestionar las llamadas asíncronas.

Código Fuente 3.3: cublasDscal cliente

```
void cublasDscal(int n, double alpha, double *x, int incx) {

    clientAsyncData asyncData;
    stCublasDscal *params = &asyncData.params.cublasDscal;
    map<pthread_t, rCUDAl>::iterator it = _getrCUDAlThread();

    asyncData.fid = f_cublasDscal;
    asyncData.size = sizeof (stCublasDscal);
    params->n = n;
    params->alpha = alpha;
    params->x = x;
    params->incx = incx;

    _pushAsync(&asyncData, it->second.streamCublas);

 }
```

En la parte del servidor, para cada función se sigue el patrón que se muestra en el Código 3.4. Una vez recibidos los parámetros de la llamada, se procede a ejecutarla sobre la GPU real.

Código Fuente 3.4: cublasDscal servidor

```
void CudaObject::rCUDAd_cublasDscal(void) {

    syslog(LOG_INFO, "'cublasDscal' received.");

    stCublasDscal *params =
                &commInterface->getFParams()->cublasDscal;
    cublasDscal(params->n, params->alpha,
                params->x,   params->incx);
}
```

3.1.2. HPL-Fermi

Para conseguir un mejor rendimiento cuando se ejecuta esta implementación del HPL, las llamadas a las funciones CUDA se realizan utilizando la Driver

API en lugar de la Runtime API. Esto se debe a que la Runtime API es una abstracción de la Driver API por lo que añade un pequeño sobrecoste en cada llamada a las funciones CUDA.

Como rCUDA no soporta las funciones de la Driver API, para poder ejecutar esta versión del HPL sobre rCUDA se ha añadido funcionalidad para las tres funciones siguientes:

- *cuCtxGetCurrent*. Obtiene, en caso de que ya exista, el contexto de ejecución en la GPU.
- *cuStreamCreate*. Crea un stream (o flujo de trabajo) en la GPU.
- *cuMemcpy2DAsync*. Realiza la transferencia asíncrona en bloques de dos dimensiones.

Para la primera función, se han añadido las lineas que se muestra en el Código 3.5.

Código Fuente 3.5: Código de cuCtxGetCurrent

```
/*--------------------- Client ------------------------*/

// Returns the CUDA context bound to the calling CPU thread.
CUresult CUDAAPI cuCtxGetCurrent(CUcontext * pctx){
   map<pthread_t, rCUDAl>::iterator it = _getrCUDAlThread();
   it->second.runtimeInitialized = true;
   _commonProtocol(f_cuCtxGetCurrent, 0, NULL,
                        pctx, sizeof (CUcontext));
   return (CUresult)_rCUDAError;
}

                        ...

/*--------------------- Server ------------------------*/

void CudaObject::rCUDAd_cuCtxGetCurrent(void) {
    CUcontext pctx;
    syslog(LOG_INFO, "'cuCtxGetCurrent' received.");
    cudaError = (cudaError_t)cuCtxGetCurrent(&pctx);
    sendDataBack(&pctx, sizeof(CUcontext));
}
```

En el caso de la función *cuStreamCreate*, al igual que en la función del apartado anterior *cuMemGetInfo*, esta función ya tiene su implementación en la biblioteca CUDA Runtime API por lo que se ha redirigido la llamada.

La función *cuMemcpy2DAsync* ha sido la que más modificaciones ha necesitado ya que, aunque esta llamada ya está soportada en la CUDA Runtime API, en este caso los parámetros se encapsulan en una estructura de datos propia para estas llamadas. Por esta razón, se ha tenido que desglosar esta

estructura en los parámetros necesarios para la llamada a la función *cudaMemcpy2DAsync*. El resultado es el mostrado en el Código 3.6. Aunque el HPL solo utiliza transferencias desde el Host al Device y viceversa de forma asíncrona, el código siguiente está adaptado para el resto de transferencias 2D tanto síncronas como asíncronas.

Código Fuente 3.6: Código de cuMemcpy2DAsync

```
// Asynchronous copies memory for 2D arrays.

CUresult CUDAAPI _cuMemcpy2DX (const CUDA_MEMCPY2D *pCopy,
                                CUstream stream=NULL){

    void * src=NULL;
    void * dst=NULL;
    cudaMemcpyKind kind=cudaMemcpyHostToHost;
    //From Host to...
    if(pCopy->srcMemoryType==CU_MEMORYTYPE_HOST){
        src=(void *)pCopy->srcHost;
        if(pCopy->dstMemoryType==CU_MEMORYTYPE_HOST){//..host
            kind= cudaMemcpyHostToHost;
            dst=pCopy->dstHost;
        }
        if(pCopy->dstMemoryType==CU_MEMORYTYPE_DEVICE){//..device
            kind= cudaMemcpyHostToDevice;
            dst=(void *)pCopy->dstDevice;
        }
    }
    //From Device to...
    else if(pCopy->srcMemoryType==CU_MEMORYTYPE_DEVICE){
        src=(void *)pCopy->srcDevice;
        if(pCopy->dstMemoryType==CU_MEMORYTYPE_HOST){//..host
            kind= cudaMemcpyDeviceToHost;
            dst=pCopy->dstHost;
        }
        if(pCopy->dstMemoryType==CU_MEMORYTYPE_DEVICE){//..device
            kind= cudaMemcpyDeviceToDevice;
            dst=(void *)pCopy->dstDevice;
        }
    }
    else if(pCopy->dstMemoryType==CU_MEMORYTYPE_ARRAY){//Array
        //from array 2 array (sync and async)
        return (CUresult)cudaMemcpy2DArrayToArray(
                (cudaArray_t)pCopy->dstArray,0,0,
                (cudaArray_const_t)pCopy->srcArray,
                0,0,pCopy->WidthInBytes,
                pCopy->Height,cudaMemcpyDefault);
    }
    else{
         //CU_MEMORYTYPE_UNIFIED
        _rCUDAError = cudaErrorNotSupported;
        return (CUresult) _rCUDAError;
```

```
    }
        //Synchronous copies
    if (stream==NULL){
        return (CUresult)cudaMemcpy2D(dst,pCopy->dstPitch,
            src,pCopy->srcPitch,
            pCopy->WidthInBytes,pCopy->Height,kind);
    }
        //Asynchronous copies
    else{
        return (CUresult)cudaMemcpy2DAsync(dst,pCopy->dstPitch,
            src,pCopy->srcPitch,
            pCopy->WidthInBytes,pCopy->Height,kind,stream);
    }
}


CUresult CUDAAPI cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy,
                                  CUstream stream){
    return _cuMemcpy2DX(pCopy,stream);
}
```

Esta versión, al igual que la anterior, también hace uso de algunas funciones de la biblioteca cuBLAS:

- *cublasDtrsm.*
- *cublasDgemm.*
- *cublasSetKernelStream.* Fija la ejecución de las funciones cuBLAS a un stream ya existente.

Las dos primeras funciones han sido comentadas en el punto anterior y la tercera ya estaba soportada en la biblioteca que daba soporte a cuBLAS simple precisión en rCUDA por lo que se ha aprovechado el trabajo realizado anteriormente.

3.2. Resolviendo errores

El Linpack Benchmark genera una gran cantidad cálculos y copias de datos. Estas cantidades de transferencias de datos generan un estrés importante sobre rCUDA, y por lo tanto son un buen banco de pruebas para el software. Más concretamente, nos va a permitir testear la parte de las copias asíncronas entre cliente y servidor así como el buen funcionamiento de los módulos de comunicaciones.

3.2.1. Módulo de memoria (función cudaMemcpy2D)

Durante la ejecución del HPL estándar se observó que, para algunos tamaños del problema, el resultado no era el esperado. Puesto que este HPL solamente

realiza copias de memoria y ejecuciones de funciones cuBLAS, en primer lugar se testearon por separado las funciones de cuBLAS con un test para cada función. El resultado de estos tests fue el esperado por lo que el problema se encontraba en las transferencias de memoria.

Una vez revisado el código referente a este tipo de copias en el servidor, se observó que funcionaba correctamente en transferencias donde el bloque de datos tuviera los mismos valores para anchura y altura o bien cuando el valor de altura fuera 1. En ambos casos, la transferencia se realiza de forma correcta pero fallaba cuando las dimensiones de la transferencia no eran iguales.

Este error se producía ya que el envío de datos en bloques rectangulares no es común en las implementaciones y todos los tests realizados hasta el momento realizaban transferencias de bloques cuadrados por lo que el error pasó desapercibido. En primer lugar se añadió la lógica necesaria para el tratamiento de este tipo de envíos y posteriormente se realizaron tests para verificar su correcto funcionamiento.

3.2.2. Módulo de InfiniBand (función recvChunk)

Este error apareció en la ejecución del HPL-Fermi sobre la biblioteca de comunicaciones de InfiniBand cuando se realizaban transferencias de bloques de las matrices principales tal y como se muestra en la Figura 3.1. Para este tipo de transferencias, en el servidor, se aprovecha la función setAsync que indica que la transferencia va a ser asíncrona para enviar la primera ráfaga de escrituras en memoria remota y posteriormente se llama a la función recvChunk que es la encargada de enviar el resto de las transferencias.

El problema estaba en que el número de envíos restantes no se actualizaba al iniciar la función recvChunk por lo que solamente se hacía la comprobación utilizando el valor del puntero fuente. Con esta comprobación, la función recvChunk se salía de rango del buffer a enviar y devolvía un error.

Para solucionar este problema, se añadió la actualización del número de envíos pendientes en el servidor con el Código 3.7.

Código Fuente 3.7: recvChunk

```
virtual size_t recvChunk(size_t count, size_t height,
                         char *asyncPtr, size_t pitch) {
        size_t m = min(count * height, ChunkSize);

        height=min(height, asyncRemainingRows); // HPL bug fixed
                        ...
```

3.2.3. Solapamiento de streams

Este es el último error encontrado hasta el momento en el uso de rCUDA y se producía al ejecutar el HPL-Fermi para tamaños medianos y grandes

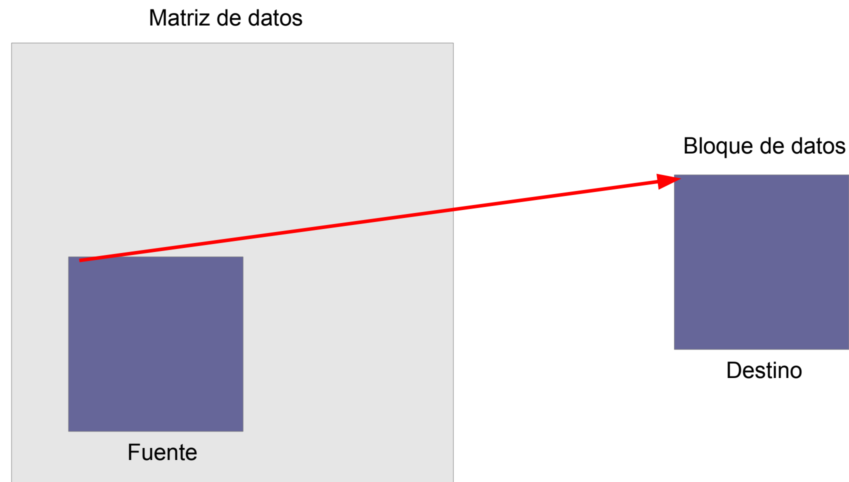

Figura 3.1: Transferencia de un bloque de datos de una matriz

(8192, 16384 y 32768). Para estos valores la ejecución se quedaba congelada debido a que, a pesar de que el servidor finalizara todas las transferencias correctamente, el cliente no era capaz de procesarlas en el orden adecuado. Esta implementación del HPL, realiza una gran cantidad de transferencias (160 para tamaños de problema de 4092, 6000 para tamaños de problema de 16384, etc.) y esto conlleva al solapamiento en la recepción de transferencias en el software cliente.

El procedimiento que se sigue para controlar estos envíos y recepciones es el siguiente: cuando se recibe una función asíncrona en el cliente, éste incrementa un contador para el stream correspondiente y cuando finaliza la ejecución de esta función, el servidor envía al cliente el stream al que pertenece para poder decrementar el número de peticiones pendientes. Mientras el contador de un stream sea superior a 0, ese estream no podrá encolar más petciones.

El problema residía en que, al realizarse varias escrituras remotas desde el servidor, el valor que leía el cliente correspondiente al stream podía no estar actualizado. Esta situación, no se había dado anteriormente puesto que ninguna de las aplicaciones testeadas realizaba tal cantidad de transferencias.

Para solucionar este error, se ha añadido al servidor el Código 3.8 a la fun-

ción sendDataBackAsync que se asegura de que si la transferencia a realizar es el último chunk (porción del envío) de una transferencia más grande, se espera a que el cliente confirme de que ha atendido la petición anterior (Código 3.9) en la función getRemoteRes. Esta comprobación se realiza mediante lecturas a memoria remota (RDMA).

Código Fuente 3.8: sendataBackAsync

```
if(last){
        send_wr_dba[1].next = NULL;
        send_wr_dba[1].sg_list = &sge_dba[1];
        send_wr_dba[1].num_sge = 1;

        send_wr_dba[1].opcode = IBV_WR_RDMA_READ;
        send_wr_dba[1].send_flags = IBV_SEND_SIGNALED;
        send_wr_dba[1].wr.rdma.remote_addr = rmsg.addr;
        send_wr_dba[1].wr.rdma.rkey = rmsg.rkey;

//Wait until the client reads the previous last transfer chunk
        do{
                ibv_post_send(qp, &send_wr_dba[1], &bad_send_wr);
                waitCompletion<SRV > (s_comp_chan);
                ctr=0;
        }while(asyncStream != (cudaStream_t)0xFF);
}
```

Código Fuente 3.9: getRemoteRes

```

virtual int getRemoteRes(int t, cudaStream_t *stream) {
        if (ibv_post_recv(qp, &recv_wr_res, &bad_recv_wr)) {
                abort();
        }
        int r = waitCompletion<CLI > (r_comp_chan, t);
        if (r == SUCCESS) {
                asyncStreamPrev = asyncStream;
                if(stream != NULL) *stream = asyncStreamPrev;
        } else {
            abort();
        }
        asyncStream=(cudaStream_t)0xFF;
        return r;
}
```

Aunque finalmente se adoptó por esta solución, se implementó otra alternativa para la resolución de este problema utilizando una doble confirmación entre cliente y servidor. Luego se realizó un estudio entre ambas para verificar cual de ellas era más efectiva. En la Figura 3.2 se muestran los tiempos obtenidos para cada solución.

Las barras azules muestran el tiempo que consume la implementación original en la ejecución de la función SendDataBackAsync. En rojo se muestran

los tiempos de la solución adoptada. Como se observa en la gráfica, en la mayoría de casos introduce un ligero sobrecoste ya que se realiza una lectura remota. Los puntos donde hay una gran diferencia entre ambas es debido a que se realizan sincronizaciones en el código y en ese punto, el cliente se encuentra en fase de sincronización con la aplicación CUDA y el servidor se encuentra esperando a que el cliente le confirme la lectura. Los valores en amarillo representan la solución descartada. Finalmente se decidió por la lectura remota ya que sobrecargaba menos el sistema cuando se acumulaban grandes cantidades de transferencias.

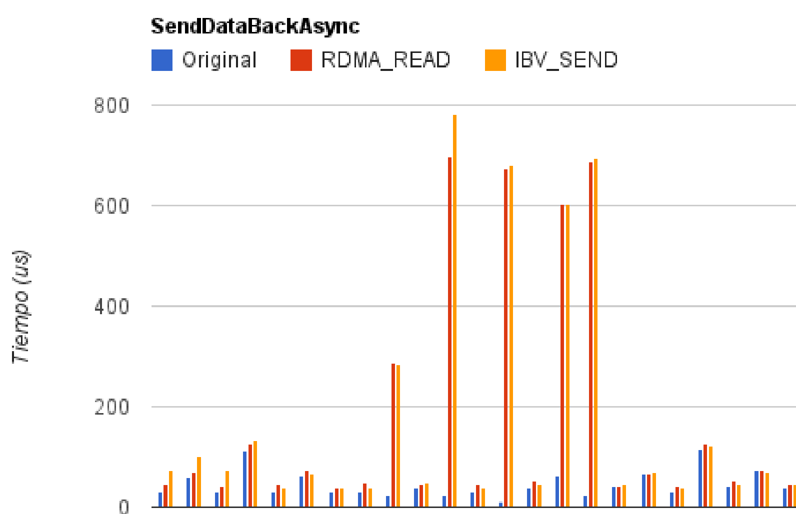

Figura 3.2: Temporización de las posibles soluciones para el solapamiento de streams

Capítulo 4

Resultados

Índice

4.1. HPL-CUDA **21**
4.2. HPL-Fermi **24**

En esta sección se muestran y comentan los resultados obtenidos al lanzar ambas versiones del HPL sobre rCUDA y se comparan con los obtenidos de forma local utilizando CUDA.

4.1. HPL-CUDA

4.1.1. Entorno

Este primer test se ha ejecutado en nodos del cluster tintorrum. Más concretamente utilizando la cola CUDA que está formada por: 9 nodos con dos procesadores quad-core Intel Xeon E5520 (con un total de 8 cores @ 2.27 GHz) y una GPU Tesla C2050. Los nodos están conectados mediante una red QDR de InfiniBand (Mellanox MTS3600 switch).

4.1.2. Resultados

Para la ejecución de este test, se utilizará un solo nodo con una GPU para obtener el rendimiento con CUDA y un nodo y la GPU de un nodo remoto para la obtención del rendimiento al utilizar rCUDA. Posteriormente, se ejecutará el mismo test con 2 GPUs remotas para comprobar el escalado de la aplicación y si el resultado es satisfactorio se volverá a ejecutar con 4 y 8 GPUs remotas.

Para empezar con las pruebas se ejecutaron varias instancias del test modificando el tamaño del bloque de datos a enviar tanto para CUDA como para rCUDA para elegir el tamaño que mejor rendimiento ofrece. Como se observa tanto en la Tabla 4.1 y en la Figura 4.1, a mayor tamaño de bloque, mejor rendimiento se obtiene. Este resultado se debe a que con un mayor tamaño de bloque, se realizan un menor número de transferencias por lo que las comunicaciones (que son el punto débil) quedan enmascaradas por el poder computacional de la GPU.

	Tamaño de bloque					
Tamaño	512		768		1024	
del problema	CUDA	rCUDA	CUDA	rCUDA	CUDA	rCUDA
4096	37,3	44,1	41,7	40,5	46,7	36,3
8192	75,4	71,0	76,2	72,9	74,6	72,3
16384	91,7	86,3	101	96,7	105	100
32768	104	98,8	118	113	129	126
65536	124	110	152	138	171	158

Cuadro 4.1: GFlops para distintos tamaños de bloque para CUDA y rCUDA.

Los resultados comentados anteriormente nos sirven para obtener el sobrecoste al introducir rCUDA en el sistema teniendo en cuenta la red de interconexión. Si nos centramos solamente el el tamaño de bloque de 1024, el sobrecoste introducido se encuentra entre el 23 % para el tamaño de problema de 4096 al 3,2 % para 32768. El resto de valores para este sobrecoste es de 4, 5 y 8 % (8192, 16384 y 65536 respectivamente).

Si nos centramos ahora la Figura 4.2, donde se muestra solamente la ejecución del HPL sobre una GPU local y una GPU remota, se puede observar como los resultados son prácticamente los mismos tanto al ejecutar rCUDA como CUDA (columnas para tamaño de bloque 1024 en la Tabla 4.1).

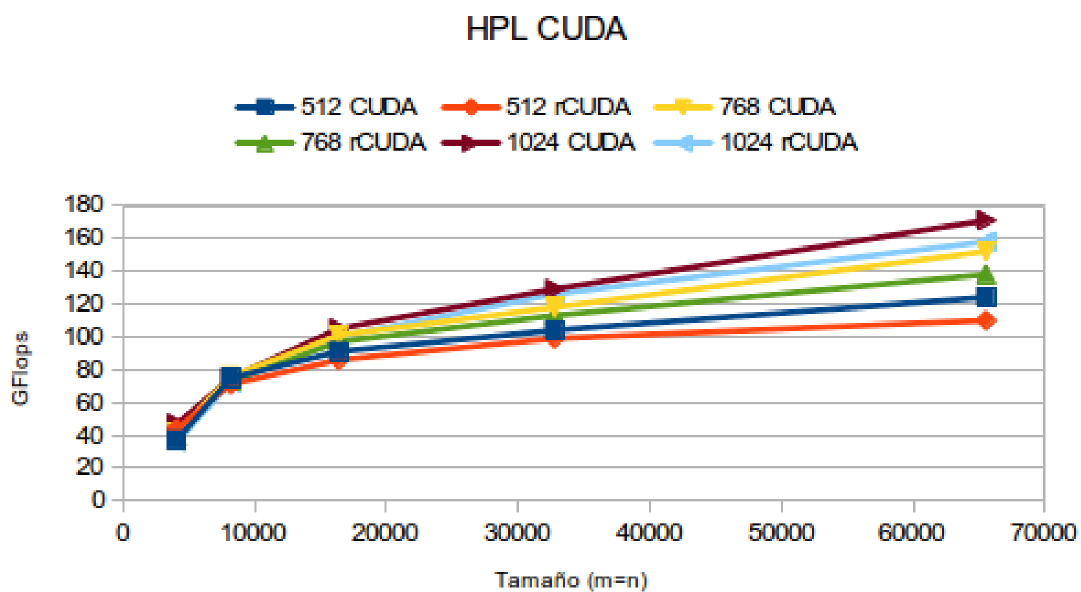

Figura 4.1: GFlops obtenidos con distintos tamaños de bloque en CUDA y rCUDA

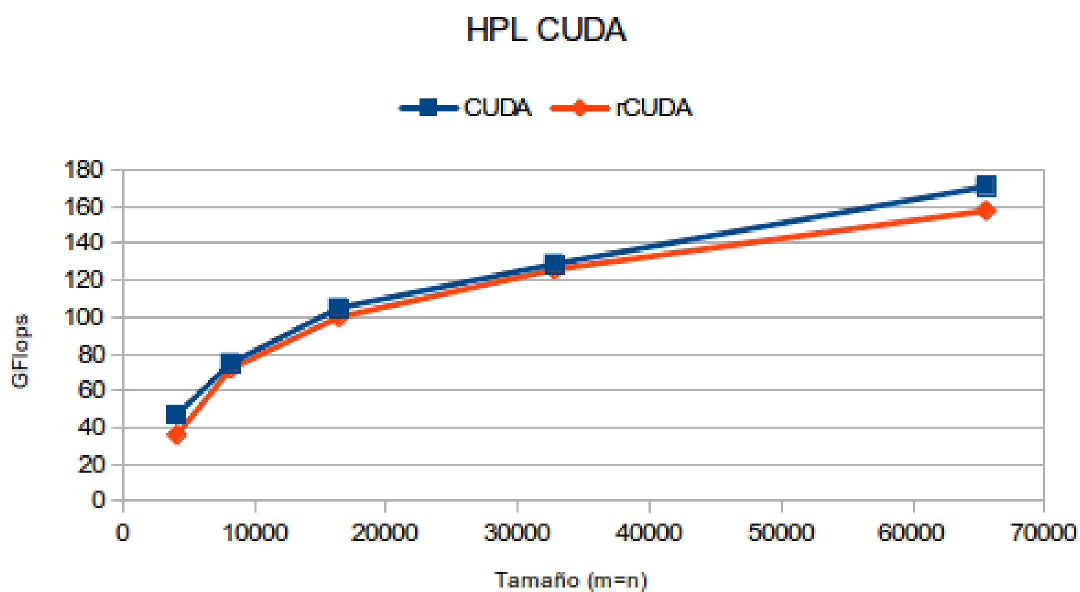

Figura 4.2: Máximos GFlops obtenidos con CUDA y rCUDA

Estos resultados nos confirman que para esta versión del HPL el sobrecoste al utilizar una GPU remota es muy bajo y es debido a que todas las funciones CUDA utilizadas son síncronas y en ese entorno rCUDA está bien diseñado.

4.1.3. Escalado

Para finalizar con esta versión, se va a ejecutar utilizando 2 GPUs remotas (cada una instalada en un nodo distinto al nodo cliente) para comprobar el escalado que ofrece esta aplicación.

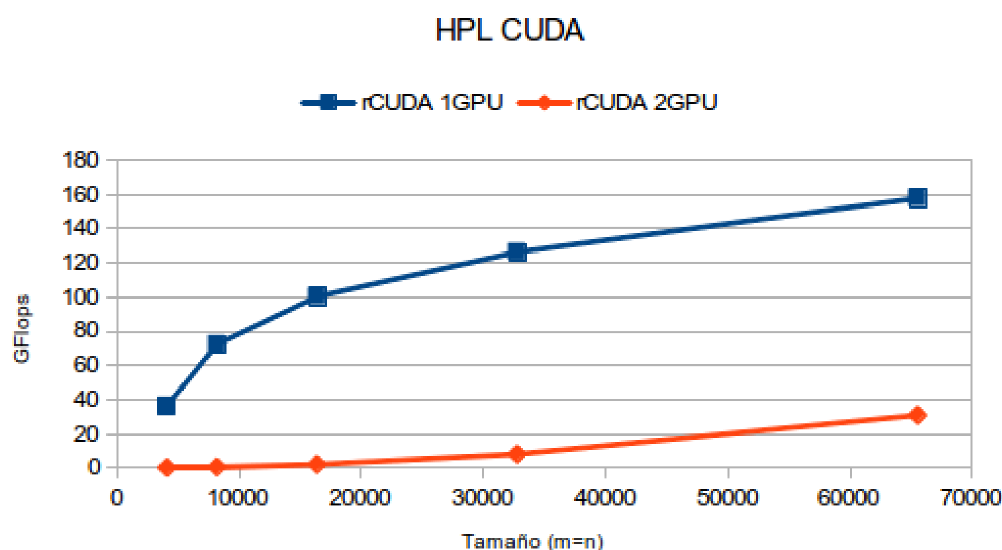

Figura 4.3: GFlops obtenidos para 1 y 2 GPUs remotas

Como se observa en la Figura 4.3, esta aplicación no escala cuando se incrementa el números de GPUs por nodo y es debido a que esta implementación no divide el problema entre los dispositivos. No utiliza la segunda GPU y el trabajo pasa a los procesadores que son mucho más lentos y por eso la caída de prestaciones. Por este motivo, no se ve necesario seguir con este estudio.

4.2. HPL-Fermi

4.2.1. Entorno

Este test, debido a que necesita unas mejores prestaciones, se va ejecutar en el cluster mlxc de la Universitat Poliècnica de València. Este cluster esta compuesto por 17 nodos con procesadores Intel Xeon E5-2620 (con un total de

24 cores @ 2.10GHz) y GPUs NVIDIA Tesla K20. Los nodos están conectados mediante una red FDR de InfiniBand

4.2.2. Resultados

Al igual que el test anterior, se empezará comparando la ejecución del HPL sobre el entorno CUDA con la ejecución sobre el entorno rCUDA y posteriormente se comprobará el escalado de la aplicación.

Los resultados al ejecutar el test sobre CUDA y rCUDA se muestran en la Figura 4.4. Claramente se observa la pérdida de prestaciones al ejecutar el HPL sobre rCUDA llegando a 526 GFlops (58 %) de diferencia para el tamaño de problema 54272.

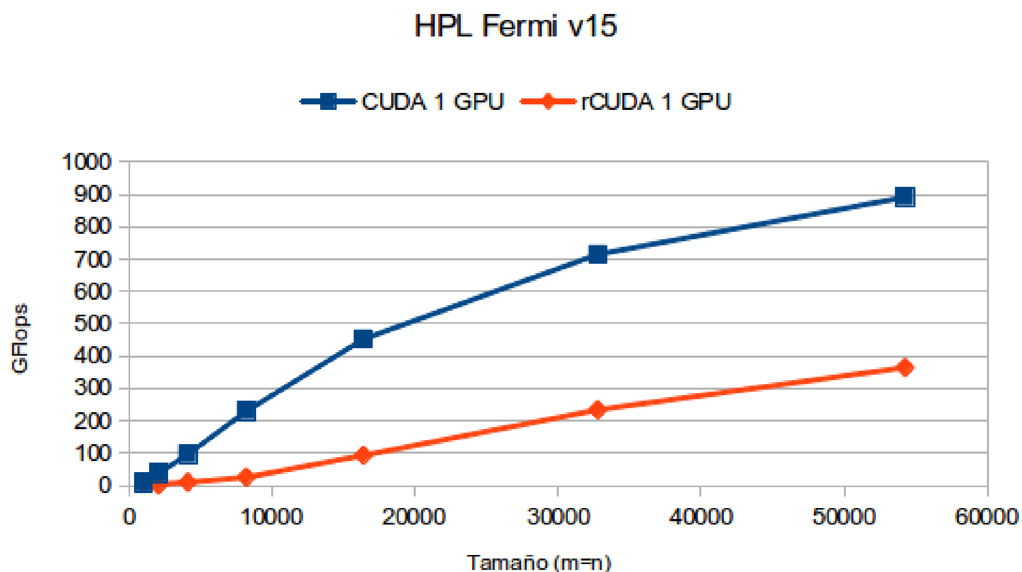

Figura 4.4: GFlops obtenidos utilizando CUDA y rCUDA en el HPL Fermi

Llegados a este punto se decide investigar el porqué de la pérdida de prestaciones. Como se ha comentado en el Capítulo 2, esta versión se centra en 2 puntos básicos:

- Gestión de streams.
- Copias de memoria asíncronas.

Gestión de streams

La actual gestión que rCUDA realiza con los streams, es la de servir las peticiones en un sistema round-robin para los streams y para cada uno de

ellos, las peticiones son servidas en el mismo orden de llegada tal y como se muestra en la Figura 4.5.

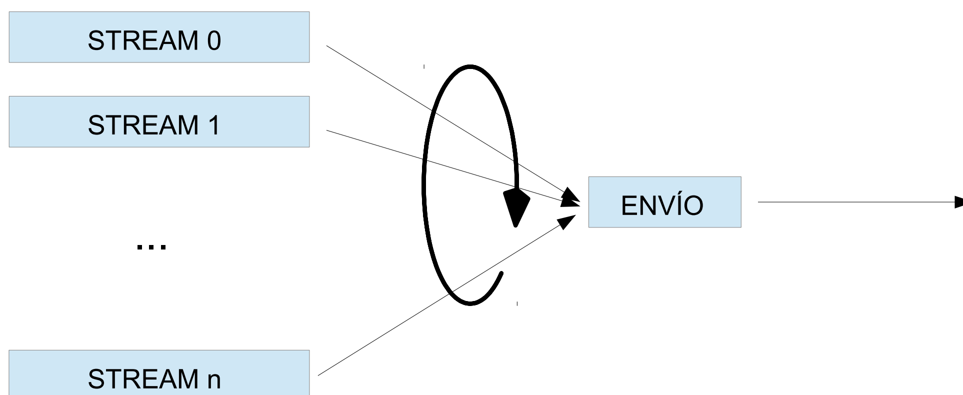

Figura 4.5: Tratamiento de las peticiones dentro de streams en rCUDA

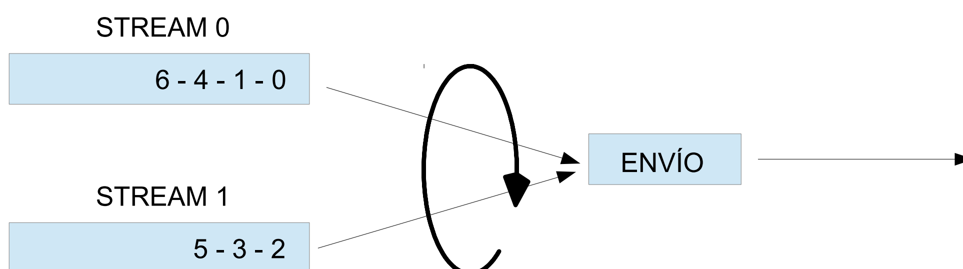

Figura 4.6: Orden de envío de las peticiones dentro de streams en rCUDA

Con este algoritmo, primero se atienden las peticiones del stream 0, a continuación las del stream 1 y así sucesivamente. Este algoritmo presenta dos inconvenientes que solamente se dan en el caso de que se produzca un uso masivo de los streams.

- Mientras un stream esta siendo atendido (se están enviando sus peticiones al servidor) no se puede añadir ninguna petición nueva a este stream por lo que se produce un bloqueo en el cliente. En el caso del HPL Fermi, se crean 4 streams y se van utilizando de forma cíclica, por lo que el envío de peticiones de funciones CUDA se bloquea para cada bloque de la matriz que va a ser calculado. Por este motivo, las prestaciones que ofrece rCUDA son pobres.

- El orden en que se envían las peticiones puede no ser el mismo en que se atienden. Supongamos la situación de la Figura 4.6 donde el stream 0 y el 1 tienen peticiones pendientes. El orden de llegada de las peticiones ha sido el que aparece dentro de los streams. En ese caso, las peticiones se enviarían al servidor en el orden 0, 1, 4, 6, 2, 3 y 5, es decir, primero todas las peticiones del stream 0 y posteriormente las del stream 1. Aunque esta situación no altera el resultado final, a la GPU del servidor remoto no le llegan las peticiones en el mismo orden que las pide el cliente. Este hecho puede afectar a las prestaciones de una aplicación ya que el uso de streams indica un trabajo de optimización del código que no se respeta en rCUDA.

Para modificar este comportamiento se ha añadido una nueva estructura de datos (cola) que almacenará el orden en que se deben atender las peticiones. Con cada petición asíncrona, se añadirá en la cola una nueva entrada correspondiente al stream al que va asiociado dicha petición. En el momento en que se vayan a atender las peticiones, se seguirá el orden indicado por el primer elemento de la cola. La función que se encarga de la gestión de los streams resultante es la que se muestra en el Código 4.1. Con este nuevo algoritmo, las peticiones que aparecen en la Figura 4.6 se atenderán con el orden correcto.

Código Fuente 4.1: asyncRoutine

```
void * _asyncRoutine(void * args) {

    stRoutinesArgs * rArgs = (stRoutinesArgs *) args;
    pthread_t threadId = rArgs->threadId;
    int device = rArgs->device;

    while (_findrCUDAlThreads(threadId) == _endrCUDAlThreads()) {
        // _asyncRoutine is executed in an independent thread
        // which is created by the constructor of rCUDAl. This
        // loop is intended to avoid this function start doing
        // stuff before rCUDAl constructor ends.
        pthread_yield();
    }

    map<pthread_t, rCUDAl>::iterator it =
                        _getrCUDAlThread(threadId, device);
```

```
    while (true) {
        pthread_mutex_lock(&it->second.ar[device]->aMutex);

        if (it->second.ar[device]->sentOrder.empty()) {
            it->second.ar[device]->async = false;
            pthread_cond_signal(
                        &it->second.ar[device]->aCondFree);
            while (!it->second.ar[device]->async) {
                pthread_cond_wait(
                        &it->second.ar[device]->aCondBusy,
                         &it->second.ar[device]->aMutex);
            }
        }
        pthread_mutex_unlock(&it->second.ar[device]->aMutex);

        while (!it->second.ar[device]->sentOrder.empty()) {
            cudaStream_t currentSt =
                        it->second.ar[device]->sentOrder.front();
            if(it->second.serviceAlrm(device, currentSt)) {
               pthread_mutex_lock(
                &it->second.ar[device]->asyncMap[currentSt].mut);
               it->second.ar[device]->sentOrder.pop();
               pthread_mutex_unlock(
                &it->second.ar[device]->asyncMap[currentSt].mut);
            }
         }
    }
        return NULL;
}
```

Una vez realizadas las modificaciones, se ejecuta de nuevo el test y los resultados son los que se muestran en la Tabla 4.2 y en la Figura 4.7.

Tamaño	CUDA	rCUDA	new rCUDA
1024	7,83	5,3	5,25
2048	37,2	1,34	22,3
4096	95,9	9,51	55,2
8192	230	24,8	131
16384	453	92,6	240
32768	714	233	412
54272	891	365	511

Cuadro 4.2: GFlops obtenidos sobre una GPU con CUDA, rCUDA original y la nueva implementación de rCUDA.

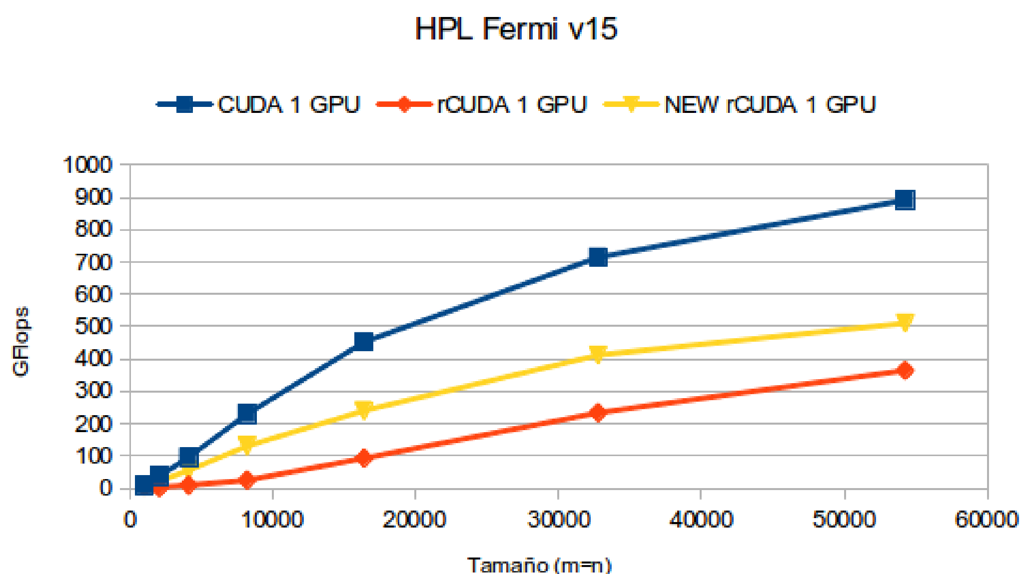

Figura 4.7: GFlops obtenidos sobre una GPU con CUDA, rCUDA original y la nueva implementación de rCUDA.

A la vista de estos datos, se observa que las prestaciones que rCUDA ofrece aun se encuentran muy alejadas de las que se obtienen con CUDA. Sin embargo, con esta nueva implementación, se obtiene una mejora de hasta 178 GFlops para el tamaño de problema 32768. Esta mejora se debe a que con este nuevo algoritmo, el cliente no se bloquea esperando a que todas las peticiones de un stream sean atendidas para poder añadir nuevas peticiones. Evitando este tiempo de bloqueo se consiguen reducir los tiempos de ejecución.

Copias de memoria asíncronas

El segundo punto sobre el que se puede actuar es en las transferencias de datos. Estas posibles modificaciones de código no son triviales y quedan al margen del propósito de este estudio de prestaciones. Estas posibles modificaciones implicarían modificar más del 50 % del código del módulo de comunicaciones de InfiniBand. Además, se debería rediseñar el protocolo de comunicaciones entre cliente y servidor de rCUDA.

En primer lugar se ha obtenido el tiempo que consumen las llamadas a transferencias de datos. Se ha elegido el tamaño de problema 4096 y el tamaño de bloque 1024. Para esta configuración se realizan 163 transferencias. Los tiempos de CUDA y rCUDA para estas llamadas se muestran en la Figura 4.8. En esta gráfica el eje X representa el instante en el que se realiza la transferencia dentro del tiempo total de ejecución del test y el eje Y representa

la duración de la llamada a la función de transferencia.

A pesar de que en esta gráfica se observan varias transferencias que son mucho más lentas con rCUDA que con CUDA, hay que añadir que son del orden de milisegundos por lo que no pueden afectar al bajo rendimiento de rCUDA. También hay que añadir que, como son llamadas asíncronas, no se está midiendo el tiempo total de la transferencia sino solamente el tiempo de la llamada de la biblioteca Runtime API.

Para finalizar el estudio de este bajo rendimiento se ha incrementado el tamaño de las transferencias de datos al utilizar rCUDA. Se ha ejecutado el HPL ampliando el tamaño de bloque para comprobar que al realizar un menor número de transferencias, las prestaciones se incrementan. Se han utilizado tamaños de bloque de 1024, 2048 y 4096 elementos.

Como se observa en la Figura 4.9, al duplicar el tamaño de la transferencia, se ha obtenido una mejora cercana a los 100 GFlops para el tamaño más grande del problema. Este hecho indica que el elevado número de transferencias y su tamaño, influyen negativamente en las prestaciones que se obtienen al ejecutar este test sobre rCUDA.

4.2.3. Escalado

Par finalizar, se va a ejecutar el test sobre una máquina con 4 GPUs locales para comprobar el escalado de ésta.

Como se observa en la Figura 4.10, la aplicación no escala correctamente cuando se utiliza más de una GPU por nodo con CUDA. Solamente obtiene mejores resultados cuando se utilizan dos GPUs locales para el tamaño máximo del problema. Cuando se utilizan 4 GPUs locales, los resultados obtenidos son inferiores a lo esperado. Esto se debe a que la configuración óptima para este test es un conjunto de nodos y en cada uno de ellos una sola GPU.

Para finalizar, en la Figura 4.11, se observa, como era de esperar, que tampoco escala correctamente al utilizar rCUDA y además, el rendimiento al utilizar 2 GPUs remotas es siempre inferior que al utilizar una GPU remota.

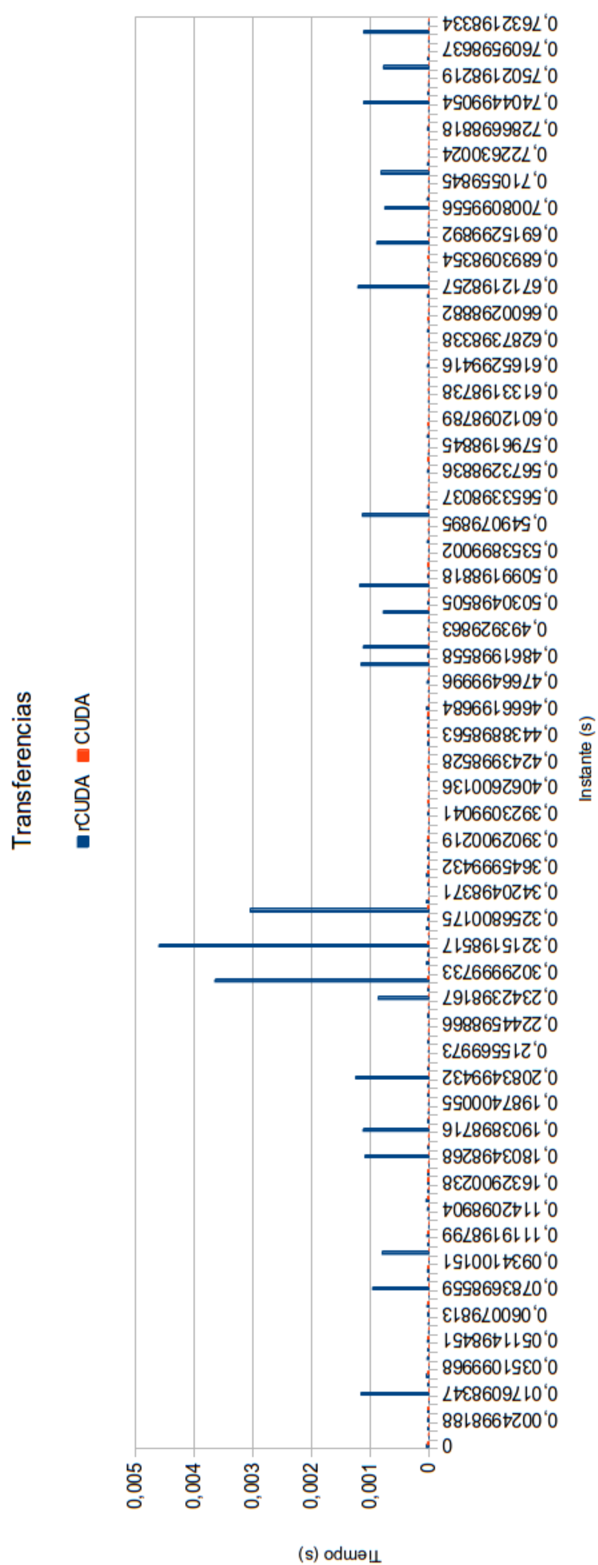

Figura 4.8: Transferencias realizadas por el HPL.

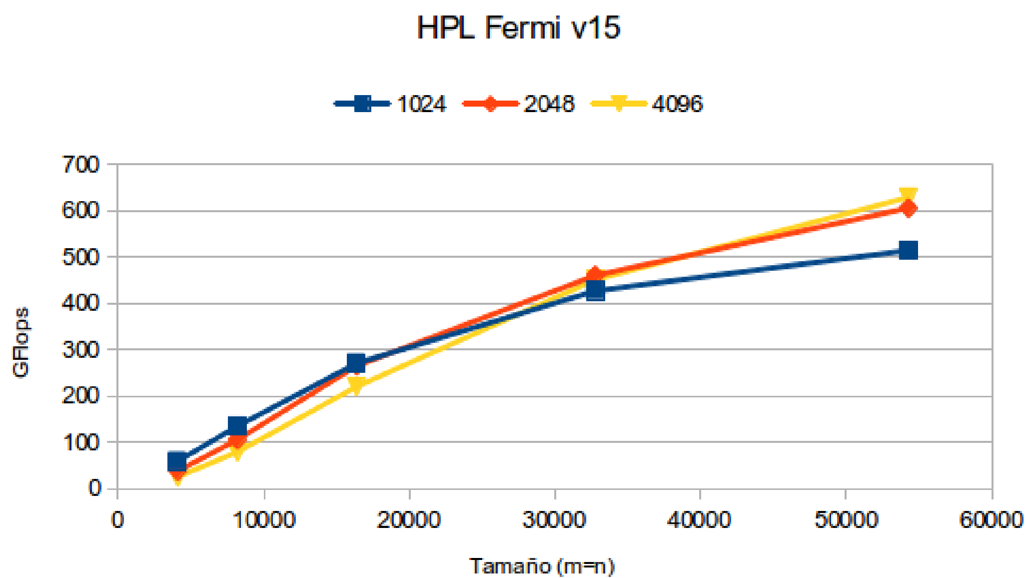

Figura 4.9: Repercusión del tamaño de bloque en el tiempo de ejecución.

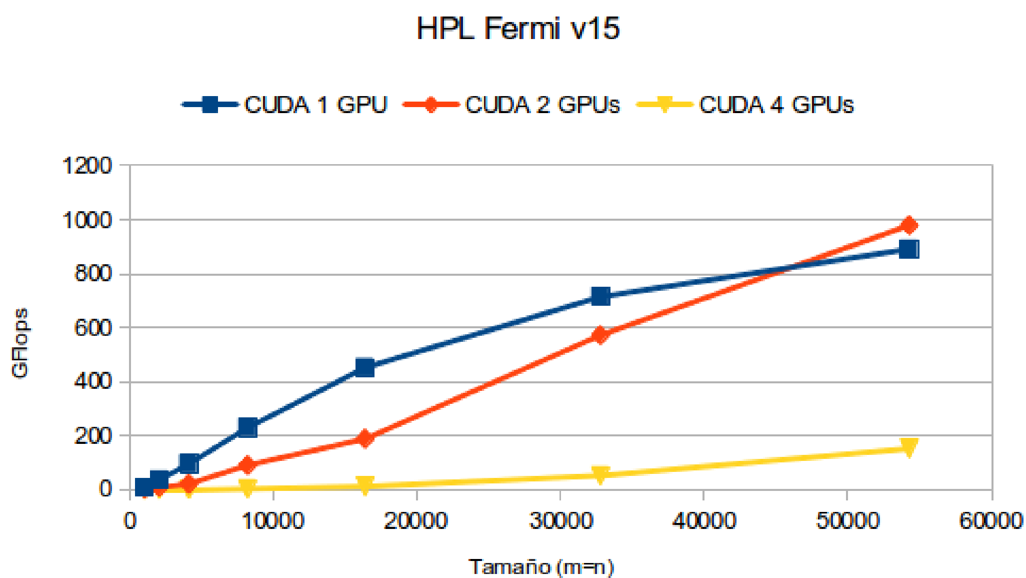

Figura 4.10: GFlops obtenidos para 1, 2 y 4 GPUs locales.

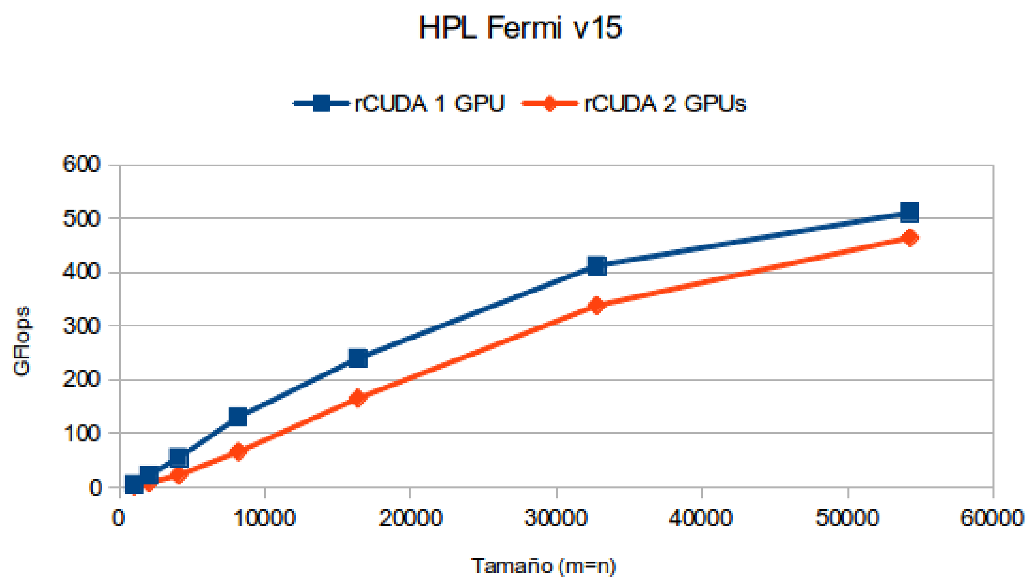

Figura 4.11: GFlops obtenidos para 1 y 2 GPUs remotas.

Capítulo 5

Conclusiones y trabajo futuro

Índice

5.1. Utilidad del proyecto **35**
5.2. Dificultades **36**
5.3. Conocimientos adquiridos **36**
5.4. Trabajo futuro **36**

En este capítulo se presentan las conclusiones del proyecto, así como sus futuras ampliaciones.

5.1. Utilidad del proyecto

El trabajo realizado se ha llevado a cabo con el objetivo de evaluar el rendimiento obtenido al ejecutar el test Linpack Benchmark sobre un cluster utilizando virtualización de GPUs. Este estudio se ha centrado en dos implementaciones del nombrado test que utilizan GPUs para su aceleración.

Además, este test ha servido para detectar puntos débiles en el software rCUDA. Algunos de estos puntos se han mejorado obteniendo un incremento en las prestaciones. Sin embargo, el apartado de las transferencias de datos asíncronas se ha dejado como trabajo futuro ya que quedaba fuera de este estudio.

5.2. Dificultades

La primera dificultad en este estudio ha sido la instalación de las versiones del test en distintas máquinas para probar su funcionamiento, ya que cada una tiene su propia configuración tanto de bibliotecas como de directorios y no se podía automatizar.

La principal dificultad del trabajo ha sido la resolución de los errores encontrados a lo largo de las ejecuciones. La solución de éstos ha representado la mayor parte del tiempo del proyecto. De todos los explicados en el Capítulo 3, el más complicado ha sido el de la función recvChunk ya que se encontraba en dos capas distintas, el código del servidor y el código de la capa de comunicaciones.

Una vez solucionados los errores y el cambio en la gestión de los streams, la evaluación de prestaciones se ha podido realizar de forma rápida y sencilla puesto que el mismo test indica que las ejecuciones y los resultados obtenidos son satisfactorios.

5.3. Conocimientos adquiridos

La realización del trabajo ha sido muy satisfactoria. Se han adquirido conocimientos nuevos en distintos campos como son: debug, técnicas de parametrización de funciones, automatización de tareas, modificación de algoritmos, etc. Pero principalmente se ha adquirido conocimiento sobre la tecnología InfiniBand, no solamente desde el punto de vista de la programación sino también desde el punto de vista del hardware de los componentes.

Además, este estudio ha servido para aprender a interpretar los resultados obtenidos y poder verificar que son coherentes con la teoría que existe detrás de cada prueba.

5.4. Trabajo futuro

Aunque el trabajo se da por concluido, quedan algunos pasos que se podrían realizar en un futuro. Desde el punto de vista del Linpack Benchmark, en su presentación se ha explicado que el código puede ser modificado para cada arquitectura por lo que se podría implementar una versión que obtuviese el máximo rendimiento al software rCUDA.

Desde el punto de vista del software rCUDA, se puede modificar el código de la capa de comunicaciones de InfiniBand para mejorar las transferencias que se realizan cuando se llama a funciones asíncronas que, como se ha comentado en el Capítulo 4, es el punto débil de esta tecnología. Además, una vez iniciada la labor de la implementación del soporte para la Driver API de NVIDIA y probado que puede coexistir con la Runtime API, se podría dar soporte para que rCUDA ofreciera una compatibilidad total con CUDA.

Bibliografía

[1] "The Top500 list," Nov.,2013. Available at `http://www.top500.org`.

[2] "The Green500 list," Nov., 2013. Available at `http://www.green500.org`.

[3] "Linpack Benchmark," May, 2014. Available at `http://www.netlib.org/benchmark/hpl/`.

[4] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience," *Concurrency and Computation: Practice and Experience*, vol. 15, p. 2003, 2003.

[5] "NVIDIA," May, 2014. Available at `http://www.nvidia.com`.

[6] "Intel Xeon Phi," May, 2014. Available at `http://http://www.intel.com/`.

[7] I. P. O. C. Change, "Climate change 2007: The physical science basis," *Agenda*, vol. 6, no. 07, p. 333, 2007.

[8] H. Markram, "The human brain project," *Scientific American*, vol. 306, no. 6, pp. 50–55, 2012.

[9] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla, "An efficient implementation of GPU virtualization in high performance clusters," *Euro-Par 2009, Parallel Processing – Workshops*, vol. 6043, pp. 385–394, 2010.

[10] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "Performance of CUDA virtualized remote GPUs in high performance clusters," in *Proceedings of the 2011 International Conference on Parallel Processing (ICPP 2011)*, pp. 365–374, Sept. 2011.

[11] J. Duato, A. J. Peña, F. Silla, J. C. Fernández, R. Mayo, and E. S. Quintana-Ortí, "Enabling CUDA acceleration within virtual machines using rCUDA," in *Proceedings of the 2011 International Conference on High Performance Computins (HiPC 2011)*, pp. 1–10, Dec. 2011.

[12] C. Reaño, R. Mayo, E. S. Quintana-Ortí, F. Silla, J. Duato, and A. J. Peña, "Influence of Infiniband FDR on the performance of remote GPU virtualization," in *IEEE Cluster 2013*, pp. 1–8, 2013.

[13] "NVIDIA CUDA," May, 2014. Available at `http://www.nvidia.com/object/cuda_home_new.html`.

[14] "Linpack Benchmark para cuda," May, 2014. Available at `https://github.com/avidday/hpl-cuda`.

[15] "Linpack Benchmark," May, 2014. Available at `http://www.netlib.org/benchmark/hpl/`.

[16] "Message Passing Interface," May, 2014. Available at `https://computing.llnl.gov/tutorials/mpi/`.

[17] "Basic Linear Algebra Subprograms," May, 2014. Available at `http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms`.

[18] J. Lee, "Snucl and an mpi+opencl implementation of hpl on heterogeneous cpu/gpu clusters," in *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?*, ATIP '12, (Singapore, Singapore), pp. 31:1–31:28, A*STAR Computational Resource Centre, 2012.

[19] "GoToBlas," May, 2014. Available at `http://c2.com/cgi/wiki?GotoBlas`.

[20] "OpenMPI," May, 2014. Available at `http://www.open-mpi.org/`.

[21] "CUDA CUBLAS User Guide," 2014. NVIDIA.

[22] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), pp. 46–51, ACM, 2009.

[23] "MKL," 2014. Available at `https://software.intel.com`.

[24] "CUDA Runtime API Guide," 2014. NVIDIA.