Benchmarking de arquitecturas de hardware modernas para HPC

Proyecto de grado

Danilo Espino

Tutores:
Pablo Ezzatti
Martín Pedemonte

27 de diciembre de 2016

Resumen

En este trabajo se estudia el uso de arquitecturas de hardware que incluyan GPUs para acelerar los benchmarks más difundidos en el campo de la computación de alto desempeño, es decir *High Performance LINPACK* (HPL), *High Performance Conjugate Gradients* (HPCG) y el que se usa para elaborar el ranking Graph500.

Específicamente, se estudia la instalación y configuración de variantes de los benchmarks HPL y HPCG capaces de explotar el poder de cómputo ofrecido por plataformas de hardware masivamente paralelas. Además, considerando que Graph500 no dispone de una versión oficial que permite explotar las GPUs se realiza una revisión del estado del arte, se identifican los desarrollos más prometedores y se extienden los mismos para transformarlos en herramientas comparables con la versión del benchmark en CPU. Por último, se evalúan experimentalmente las diferentes variantes de los benchmarks y se estudian e interpretan los resultados obtenidos.

Palabras clave: *HPC, LINPACK, HPL, HPCG, Graph500, GPGPU, CUDA, BFS.*

Índice general

Capítulo 1

Introducción

En los últimos años las arquitecturas de hardware utilizadas para computación de alto desempeño (HPC, del inglés *High Performance Computing*) han sufrido grandes cambios. Principalmente en la última década ha proliferado el uso de arquitecturas de hardware híbridas, especialmente aquellas que combinan procesadores multi-core con aceleradores de hardware. Entre estos últimos se destacan las tarjetas gráficas (GPUs, del inglés *Graphic Processing Units*) y los procesadores Intel Xeon Phi. Este hecho se puede constatar fácilmente en la evolución que sufre la lista TOP500 dedicada a enumerar y describir las 500 computadoras más potentes del mundo (esta lista se publica en forma semestral) [50].

Adicionalmente, otro aspecto que ha cobrado importancia a la hora de evaluar las plataformas de hardware y/o algoritmos es el consumo energético de los dispositivos. Este hecho está asociado tanto a los grandes costos económicos (directos e indirectos) que implica la energía consumida por las plataformas de hardware de gran porte, como al concepto de autonomía relativo a los dispositivos que utilizan baterías. Esta tendencia ha motivado la confección de la lista Green500 [49] que ordena las 500 computadoras del TOP500 de acuerdo a su eficiencia energética, es decir según la cantidad de *flops* (operaciones de punto flotante) – en general GFLOPS o miles de millones de *flops* – por watt consumido por el equipo.

Para evaluar la potencia de las distintas plataformas de hardware se suelen utilizar benchmarks especialmente diseñados para ello. Un benchmark se puede definir como una colección de programas o rutinas que resumen un comportamiento representativo de una gran cantidad de herramientas (informáticas en nuestro caso). El benchmark más difundido en el campo de HPC es el LINPACK [36] que entre otras cosas es el utilizado para conformar la lista TOP500 (e indirectamente, la lista Green500). El benchmark LINPACK se basa en operaciones de álgebra densa, fundamentalmente la factorización LU de matrices [15]. Si bien este tipo de problemas son de gran difusión y aplicación en el ámbito de la computación científica, presentan algunas características particulares, como los accesos regulares y alineados a memoria, que los vuelve poco representativos de aplicaciones que involucran el manejo de grandes volúmenes de datos y/o el uso de grafos. Considerando entonces que, en los últimos años ha crecido de forma importante el abordaje de problemas que involucran la utilización de operaciones de álgebra dispersa (grafos) y que en este tipo de problemas los accesos a memoria son completamente distintos a los involucrados en la resolución de problemas de álgebra densa se están realizando diferentes esfuerzos por plasmar estos requerimientos en nuevos benchmarks. En este sentido, en los últimos años se han presentado, entre otros, los benchmarks HPCG (*High Performance Conjugate Gradients*) [27] y el Graph500 [17].

Por este motivo, el estudio y optimización de los diferentes benchmarks propuestos constituye un tema de investigación de gran actualidad. El avance en el conocimiento del uso de los benchmarks permite por un lado entender qué plataforma de hardware es mejor para qué clase de problemas y, por otro lado, estudiar el uso de diferentes técnicas que ayuden a abatir las restricciones que presentan las plataformas para las distintas clases de problemas.

A partir de lo expresado anteriormente, los objetivos principales del proyecto son: resumir el estado del arte en cuanto a las arquitecturas de hardware modernas para HPC, identificar los distintos benchmarks utilizados por la comunidad y estudiar sus principales características, aplicar los benchmarks a las arquitecturas de hardware que incluyan procesadores masivamente paralelos disponibles en nuestro entorno y evaluar e interpretar los resultados obtenidos.

El resto del documento se estructura de la siguiente forma. En el Capítulo 2 se describen algunas de las infraestructuras de hardware más difundidas para computación de alto desempeño. Los principales benchmarks utilizados para evaluar las diferentes arquitecturas se describen en el Capítulo 3. También se ofrece una breve introducción a los conceptos matemáticos necesarios para entender los benchmarks. Luego, en el Capítulo 4, se detalla la forma de instanciación de los benchmarks descritos anteriormente (esto incluye el cómo fueron utilizados o características del código que se desarrollaron para ellos) y los resultados de la evaluación experimental. Finalmente, en el Capítulo 5, se resumen las conclusiones y trabajo futuro que puede extender este proyecto.

Capítulo 2

Infraestructuras para HPC

En este capítulo se describen algunas de las principales plataformas de hardware para HPC, con especial énfasis en aquellas que se van a utilizar en el proyecto. Se hará referencia a las plataformas a partir de los atributos de un sistema visibles para un desarrollador o, en otras palabras, se estudiarán aquellos atributos que tienen impacto directo en la ejecución lógica de un programa. Ejemplos de estos atributos son el conjunto de instrucciones, número de bits utilizados para la representación de diferentes tipos de datos, jerarquía de memoria y técnicas de asignación de ella.

2.1. Estructura de un computador

Como referencia para secciones subsiguientes se definirán algunas de las componentes básicas de la estructura interna de un computador (dichos componentes se pueden ver en la Figura 2.1). Esta introducción está fuertemente basada en [47].

En particular, un computador se constituye de cuatro componentes estructurales fundamentales:

- **Unidad de procesamiento central (CPU)**

  Se encarga de controlar la operación de un computador y ejecuta sus funciones de procesamiento de datos. Este procesamiento se realiza en un ciclo de obtención, ejecución y almacenamiento de datos. Asimismo administra sus recursos y orquesta el desempeño de sus partes funcionales en respuesta a las instrucciones brindadas por el usuario.

- **Memoria principal**

  Cumple la función de almacenar datos. El computador debe almacenar datos inclusive si se están procesando en el mismo instante. Esto hace que, al menos, haya una función de almacenamiento de datos temporal. También debe existir un almacenamiento a largo plazo.

- **Entrada/salida**

  Realiza el movimiento de datos entre el computador y el exterior. El ambiente operacional del computador consta de dispositivos que sirven como fuentes de origen o destino de datos. Cuando los datos se mueven desde la memoria principal a dispositivos que están directamente conectados al computador el proceso se denomina de entrada/salida.

- **Interconexión del sistema**

  Mecanismo de comunicación entre la CPU, la memoria principal y la entrada/salida. El ejemplo común es utilizar un bus de sistema, que consiste en una serie de cables a las que el resto de las componentes se conectan.

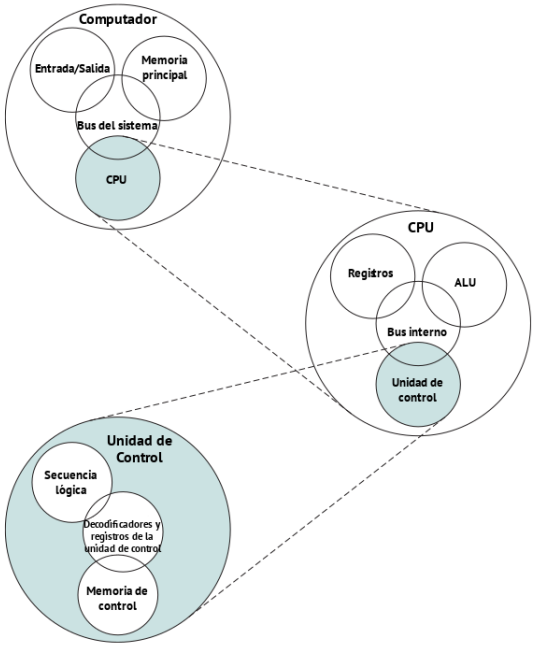

Figura 2.1: Estructura a alto nivel de un computador. Adaptado de [47].

Una de las componentes más compleja de esta arquitectura es la CPU, donde se pueden identificar, entre otras:

- **Unidad de control**: controla la operación de la CPU.
- **Unidad aritmético/lógica (ALU, del inglés Arithmetic and Logic Unit)**: desarrolla las funciones de procesamiento de datos.
- **Registros**: provee de almacenamiento interno a la CPU.
- **Interconexión de la CPU**: permite la comunicación entre la unidad de control, la ALU y los registros.

La primera máquina diseñada con estas componentes estructurales fue la llamada "computadora IAS" dado que fue diseñada en el Princeton Institute for Advanced Studies en el año 1952 [14]. Un esquema de la misma se puede apreciar en la Figura 2.2. El equipo de diseñadores estaba integrado, entre otros, por John von Neumann. Por ese motivo, y en adelante, las máquinas que cumplieran esta estructura fueron conocidas como máquinas von Neumann.

Con el advenimiento de los transistores, que reemplazaron a las máquinas de tubos de vacío, se logró la introducción de unidades lógicas más complejas y con mayor disponibilidad de memoria. Esto último también permitió la provisión de sistemas de software a dichas unidades. Además se incorporaron nuevas características a los computadores como son: el uso de canales de datos (oficiando de módulos independientes con procesador y set de instrucciones independientes), y los multiplexores cuya función es brindar acceso

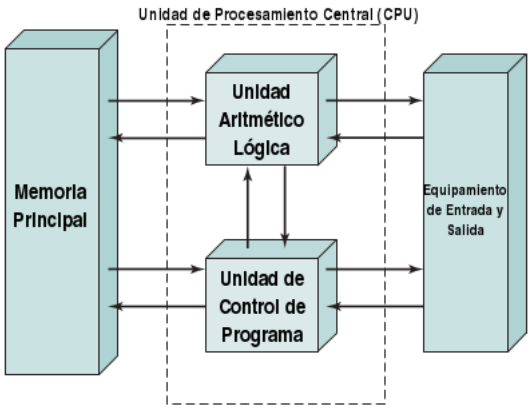

Figura 2.2: Estructura del computador IAS. Adaptado de [47].

a la memoria desde la CPU y los canales de datos de forma que trabajen de manera independiente.

2.1.1. Procesamiento de instrucciones

Como se dijo anteriormente, la CPU se encarga de procesar datos. Entre los datos que es capaz de procesar se encuentran las instrucciones que un programador pretende que la computadora ejecute. Típicamente, las instrucciones que se ejecutan en la CPU indican el nombre de una operación así como la ubicación de operandos (referenciando a registros) y el resultado. El procesamiento de dichas instrucciones se hace, por ejemplo, de la siguiente manera:

1. **Obtención de la instrucción**: se carga la siguiente instrucción a ejecutar en el procesador.

2. **Decodificación de la instrucción**: el procesador inspecciona la instrucción para determinar la operación y los operandos.

3. **Obtención de memoria**: se traen los datos desde memoria hacia un registro en caso que fuera necesario.

4. **Ejecución**: la operación es ejecutada, leyendo datos desde los registros y escribiendo también en un registro.

5. **Devolución**: para operaciones de almacenamiento, el contenido del registro se vuelve a escribir en memoria.

2.2. Arquitecturas single-core

Se puede considerar que el advenimiento de las arquitecturas single-core se inicia con la era de la microelectrónica y los circuitos integrados. Estos últimos aprovechan el hecho que transistores, resistencias y conductores pueden ser fabricados a partir de un semiconductor como el silicio. En particular, muchos transistores pueden ser fabricados a partir de una pastilla de silicio de tamaño pequeño y pueden ser conectados para formar circuitos.

Año a año la cantidad de componentes que han podido ser empacados en el mismo chip fue en aumento. Este aumento de densidad ha sido reflejado en la *ley de Moore*, enunciada por Gordon Moore [32] en 1965, que indica que los diseñadores serían capaces de introducir el doble de transistores en la siguiente generación de procesadores cada vez. En particular, Moore sostenía en un primer momento que este aumento se daría año a año. Más adelante, en 1975 [33], reformula este enunciado aumentando el mencionado período a dos años.

Inicialmente, la forma en que se mejoraba la performance de las computadoras era aumentando la frecuencia de procesador. Otras de las formas de mejorar el rendimiento de las máquinas fue explotando el paralelismo de grano fino y el uso de cachés de memoria. Las primeras arquitecturas en aprovechar esto fueron las RISC (sigla en inglés para *Reduced Instruction Set Computer*), a principios de los años 1980. Algunas de las técnicas utilizadas son:

- **Pipelining**: es una técnica de paralelismo que implica ejecutar más de una instrucción al mismo tiempo. Se aprovecha del paralelismo que existe entre las acciones que se necesitan para ejecutar una instrucción.

- **Cachés de memoria**: son memorias relativamente pequeñas pero rápidas que se ubican antes, en la jerarquía, que una más grande y lenta, dotadas de lógica que permite acceder a esta última de forma eficiente. El objetivo de las cachés es almacenar datos accedidos recientemente y acelerar los accesos subsecuentes a los mismos datos.

A lo largo del tiempo la industria ha hecho que el enunciado de Moore se transformara en ley, pero este aumento en performance implicó forzar los límites en los diseños al máximo.

Asimismo existía un desbalance entre la evolución de los chips de memoria y los microprocesadores. Particularmente se había estancado el avance en las velocidades de transferencia de datos entre memoria y procesador. Si la memoria no puede soportar la velocidad de pedidos del procesador, este último queda en un estado de espera perdiéndose valioso tiempo de procesamiento, estableciendo otra limitante en el desempeño.

En otro sentido, una de las formas de incrementar el rendimiento de los procesadores de un único núcleo fue la introducción del paralelismo a nivel de instrucciones [47]. Como su nombre lo indica, esta práctica permite ejecutar varias instrucciones al mismo tiempo. Existen dos acercamientos posibles: a nivel de hardware (enfoque dinámico) o a nivel de software (enfoque estático, en tiempo de compilación). El primero de ellos se aplica principalmente en servidores y máquinas de escritorio y el segundo en dispositivos móviles motivado por el ahorro energético.

Otra de las formas existentes fue a través del paralelismo a nivel de datos. En particular utilizando la técnica SIMD (del inglés *Single Instruction - Multiple Data*), que como su nombre lo indica implica ejecutar una instrucción sobre un set de datos. Esto es útil cuando varios datos deben ser modificados con la misma instrucción con un valor (por ejemplo, en aplicaciones multimedia cuando se debe modificar el contraste y se debe incrementar el valor de los distintos colores en una unidad).

Un problema adicional en el desarrollo de las plataformas de hardware ha sido el consumo energético y la disipación de calor. Este último punto en particular ha limitado el incremento de la frecuencia de reloj y los niveles de actividades productivas que pueden ser desarrolladas en cada ciclo de reloj con un único núcleo.

Así como ha evolucionado el rendimiento a nivel de memoria y de consumo energético también lo ha hecho el tamaño de buses de datos. El crecimiento de los mismos ha permitido realizar operaciones de números cada vez más anchos en tiempos cada vez menores, ya que la cantidad de bits que podía manejar el procesador al mismo tiempo también se incrementaba. Uno de los puntos altos del avance en este aspecto fue el advenimiento de la arquitectura x86 que agrupa los procesadores compatibles con el juego de instrucciones del procesador Intel 8086, primer procesador diseñado para aplicaciones de propósito general. Las características principales de esta arquitectura y de las instrucciones que maneja son que tienen un caché o cola de instrucciones que preprocesa algunas instrucciones antes de ser ejecutadas. El tamaño de las mismas ha evolucionado desde los 8 bits en su primer generación (Intel 8080) hasta los 64 bits introducidos en las series Intel Core 2.

Considerando que varios sistemas de mayor escala habían hecho buen uso de múltiples chips de CPU para generar multiprocesadores simétricos (SMP), entonces la integración de múltiples núcleos era el paso más natural para aumentar el rendimiento.

2.3. Arquitecturas multi-core

Como su nombre lo indica, las arquitecturas multi-core involucran tres aspectos fundamentales [47]:

- Se trata de múltiples núcleos computacionales.
- Existe una forma a través de la cual los núcleos se comunican.
- Los núcleos de los procesadores deben comunicarse con el "exterior".

El primer aspecto, aunque parezca trivial, hace que se deba considerar varios otros como ser si el procesador debe ser homogéneo o exponer cierta heterogeneidad a nivel de instrucciones y performance. Por ejemplo, una arquitectura homogénea con memoria global compartida permite realizar programación paralela de forma más sencilla que una arquitectura heterogénea, donde los núcleos no tienen el mismo set de instrucciones.

El avance de estas tecnologías tiene el objetivo de mantener las velocidades de ejecución de programas secuenciales. Un ejemplo actual son los microprocesadores Intel Core i7 que cuenta con 4 núcleos, cada uno de los cuales es un procesador no sincronizado implementando el set completo de instrucciones x86, soportando hyperthreading con dos hilos de ejecución en hardware, diseñados para maximizar la velocidad de ejecución de programas secuenciales.

Las ventajas sobre las arquitecturas de un único núcleo o de multiprocesadores se ven en la reducción de los costos de comunicación y sincronización.

2.4. Intel Atom

Se denomina de esta forma a la línea de procesadores de 32 y 64 bits lanzados por Intel en el año 2008 con foco en el consumo energético limitado. En particular, el objetivo de estos procesadores era su utilización en dispositivos móviles y máquinas ultra-portátiles ya que implican un consumo energético acotado [51].

Las principales características de estos procesadores son:

- Compatibilidad total con el set de instrucciones x86, permitiendo ejecutar sistemas operativos y aplicaciones para PC.
- Tener una caché L1 divididas en 32KB para instrucciones y 24KB para datos y una o más cachés L2 de 512KB.

Los primeros procesadores de esta línea (que datan del año 2008), cuyo nombre código era *Silverthorne*, contaban con un solo núcleo. En la actualidad (2016) hay versiones con 8 núcleos y velocidades de hasta 2.4GHz.

2.5. ARM

La arquitectura ARM (acrónimo de *Advanced RISC Machines*) refiere a una arquitectura que evolucionó a partir de los principios de diseño RISC y es usado especialmente en sistemas embebidos (una combinación de software y hardware diseñados para desarrollar una función específica). Los chips ARM se caracterizan por su pequeño tamaño y ultra bajos requerimientos energéticos. Por estos motivos es que se utilizan principalmente en dispositivos móviles como teléfonos o tablets. Por ejemplo, estos chips fueron utilizados en las primeras generaciones de iPads y iPhones [16].

La empresa ARM Holdings fue la encargada de desarrollar el primer procesador comercial RISC de estas características, el Acorn RISC Machine (ARM). Su primer versión fue lanzada en el año 1985 y fue utilizado inicialmente para investigación interna y desarrollo. La segunda versión, denominada ARM2, fue lanzada un año más tarde y fue utilizada comercialmente.

2.6. Graphics Processing Units (GPUs)

Una unidad de procesamiento gráfico es, desde un punto de vista estrictamente computacional, un procesador masivamente paralelo con un ancho de transferencia de memoria grande. La misma está organizada jerárquicamente en memorias de procesador locales pequeñas, memoria compartida ligeramente más grandes para grupos de procesadores vectoriales y memorias globales de tamaños cercanos a los de las memorias disponibles en las CPUs actuales. A modo ilustrativo, en la Figura 2.3 se puede ver la organización de memoria en las tarjetas de la familia Kepler de la empresa Nvidia.

En 2012 se calculaba que la diferencia en rendimiento a nivel de cálculo de operaciones de punto flotante entre CPUs y GPUs comparadas era favorable a estas últimas en un orden de 10 veces [26]. La justificación para semejante diferencia radica en la filosofía de diseño detrás de cada una de estas arquitecturas de procesadores. La primera está optimizada para la ejecución de código secuencial. Por eso hace uso de lógica de control sofisticado para permitir que se ejecuten en paralelo instrucciones de un único hilo o por fuera de su orden secuencial mientras parezca que se ejecuta secuencialmente. Resulta más importante para estas arquitecturas que están provistas con grandes caché de memoria para reducir latencia de acceso a instrucciones y datos de aplicaciones complejas. Notar que ninguna de estas dos características contribuyen a la velocidad de cálculo pico.

El ancho de transferencia de memoria en los chips CPU ha sido, en las diferentes generaciones, aproximadamente 6 veces menores que en el caso de los chips gráficos. Como la velocidad de muchas aplicaciones se encuentra limitada por la tasa de transferencia

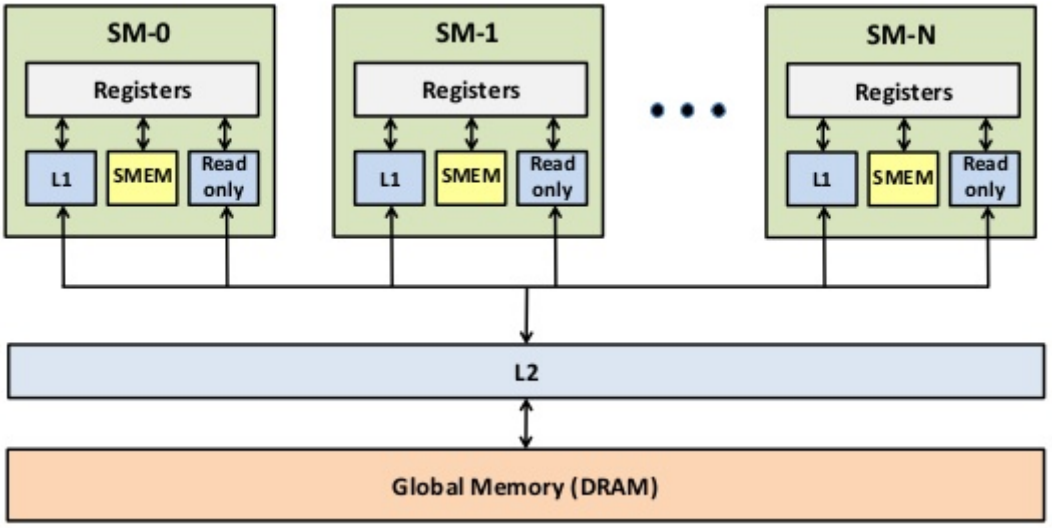

Figura 2.3: Jerarquía de memoria de las tarjetas gráficas de la familia Kepler. Adaptado de [31]

de datos entre la memoria y el procesador, hace que esta característica sea de suma importancia para el desarrollador.

La industria de las tarjetas gráficas, y por lo tanto de las GPUs, está principalmente impulsada por la industria de los videojuegos. Esto debido a la exigencia de los últimos por desarrollar un número alto de cálculos por cuadro de video en los juegos modernos. Para ello, la solución encontrada fue optimizar el diseño de los chips para el volumen de ejecución de un número masivo de hilos de ejecución. Estos diseños se centran en ahorrar área de chip y potencia permitiendo a los canales de memoria y a las operaciones aritméticas tener latencias grandes. Esto habilita a los diseñadores a tener más canales de memoria y, de esta manera, incrementar el volumen total de ejecución.

La arquitectura de las GPUs se organiza en un arreglo de multiprocesadores de flujo altamente paralelizado. Cada GPU se fabrica en la actualidad con varios gigabytes de DRAM gráfica de tasa de datos doble (GDDR del inglés *Graphic Double Data Rate*). Estas GDDR DRAMs difieren de las DRAMs del sistema en una CPU en que son esencialmente el marco de memoria buffer usado para gráficos. Para aplicaciones gráficas, como por ejemplo videojuegos, almacenan imágenes de video e información de textura para realizar renderizaciones 3D. En cambio, para los usos de computación de alta performance estudiados en este trabajo, funciona como memoria de muy alto ancho de transferencia fuera del chip, aunque con latencia un poco mayor que la típica para la memoria del sistema. Para aplicaciones masivamente paralelas, el mayor ancho de transferencia hace que las latencias sean mayores.

Se espera que las aplicaciones desarrolladas para este tipo de arquitecturas tengan un gran número de hilos paralelos. El hardware se aprovecha del número de hilos que pueden ser lanzados concurrentemente para encontrar trabajo para hacer cuando algunos de ellos están esperando por accesos a memoria de gran latencia u operaciones aritméticas. Se provee de cachés pequeños para ayudar a controlar los requerimientos de ancho de banda de estas aplicaciones de forma que los múltiples hilos que acceden a los mismos datos en memoria no necesitan ir todos a la DRAM. Este estilo de diseño es comúnmente conocido como diseño orientado al *throughput* dado que intenta maximizar el throughput de ejecución total de un gran número de hilos mientras que implica que los hilos individuales se ejecuten potencialmente en un tiempo más largo.

Ahora bien, uno de los motivos principales para utilizar plataformas del tipo de las GPU es el costo, ya que ofrecen un rendimiento excepcional (cantidad de GFLOPS por dólar invertido y/o por watt consumido). A su vez, el desarrollo de estas plataformas ha propiciado que el acceso a los recursos de las mismas por parte de los desarrolladores se haga de forma más sencilla. Mientras en el pasado era necesario desarrollar código específico basados en el conocimiento del pipeline gráfico, en la actualidad se puede acelerar el proceso mediante ambientes altamente amigables o utilizando procedimientos automatizados (por ejemplo, convertidores de código Matlab a núcleos CUDA).

2.6.1. CUDA

Además de ser la sigla en inglés de *Compute Unified Device Architecture*, es una plataforma informática y modelo de programación desarrollado por Nvidia. El objetivo de la misma es incrementar el desempeño computacional de un dispositivo a partir de los recursos de las GPUs de la empresa mencionada.

Permite aprovechar la potencia de cómputo de una GPU en aplicaciones de propósito general ofreciendo herramientas en tres niveles [54]:

- **Hardware:** habilita el paralelismo de la GPU para programación de propósito general a través de un número de multiprocesadores gemelos dotados de un conjunto de núcleos computacionales arropados por una jerarquía de memoria.

- **Firmware:** ofrece un driver para la programación GPGPU que es compatible con el utilizado para renderizar.

- **Software:** permite programar la GPU con mínimas pero potentes extensiones SIMD para lograr una ejecucion eficiente y escalable. Una de las componentes es un lenguaje que de forma general es el lenguaje C con mínimas extensiones para acceder a la GPU. También ofrece todo un ecosistema de herramientas para abordar problemáticas particulares (por ejemplo, cuBLAS [34] y cuSPARSE [35])

Una aplicación CUDA ejecuta un programa sobre uno o varios dispositivos (GPUs), que actúan como coprocesadores de un anfitrión o host (la CPU). Cada archivo con código fuente CUDA puede tener una mixtura de código tanto del anfitrión como del dispositivo. Las funciones o declaraciones de datos para el dispositivo se marcan con palabras clave CUDA. Típicamente son funciones que exhiben una gran cantidad de paralelismo de datos. Para compilar estos archivos se utiliza un compilador de CUDA desarrollado por Nvidia denominado NVCC (derivado de *Nvidia C Compiler*). El código del anfitrión es código ANSI C, es compilado en el anfitrión por su compilador de C/C++ y también ejecutado como un proceso CPU tradicional. Mientras tanto, el de o los dispositivos (esto último porque puede haber más de una GPU conectada a un anfitrión) está diferenciado del código del host con palabras clave CUDA para etiquetar las funciones que utilizan paralelismo de datos. Estas funciones se llaman *kernels*. Todo este proceso de compilación se puede ver en la Figura 2.4.

La ejecución de un programa CUDA comienza con una ejecución en el anfitrión (CPU). Cuando una función *kernel* es llamada, o lanzada, es ejecutada por un gran número de hilos en el dispositivo. Todos los hilos que son generados por un lanzamiento de *kernel* son llamadas colectivamente un *grid*. La Figura 2.5 muestra la ejecución de dos *grids* de hilos.

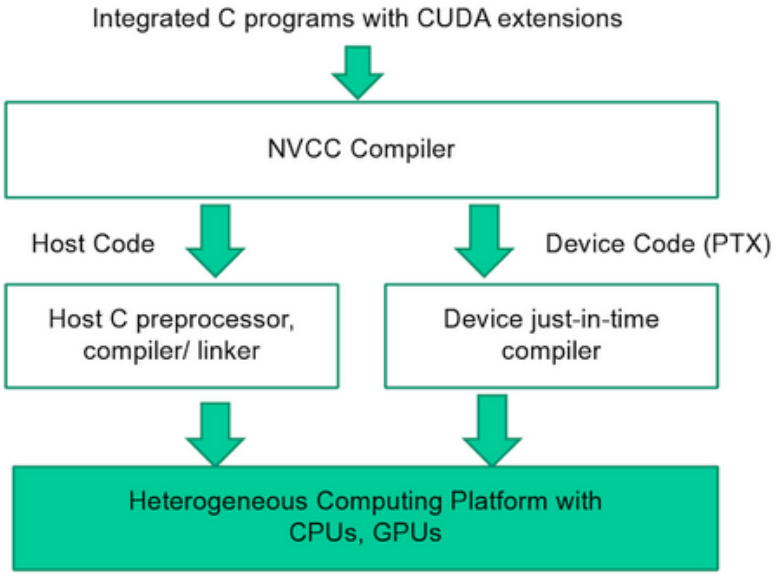

Figura 2.4: Proceso de compilación de un programa CUDA. Adaptado de [26].

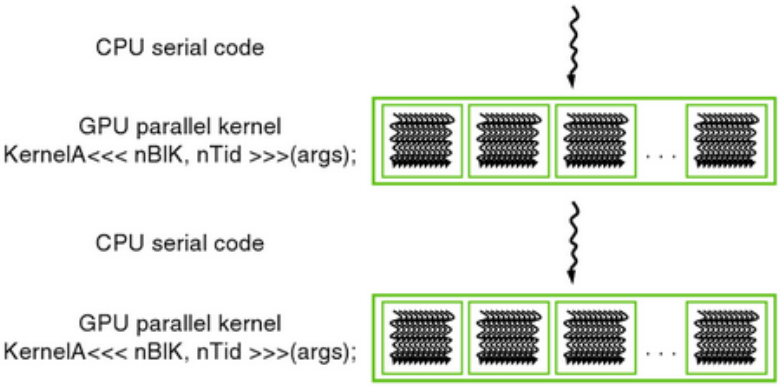

Figura 2.5: Ejecución de un programa CUDA. Adaptado de [26].

Lanzar un *kernel* típicamente implica generar un gran número de hilos para explotar el paralelismo de datos. Los programadores CUDA pueden asumir que estos hilos toman muy pocos ciclos de reloj para su generación y gestión gracias a un soporte de hardware eficiente. Esto en contraste con los hilos CPU tradicionales que toman cientos de ciclos de reloj para la generación y gestión.

2.6.2. OpenCL

Esta herramienta, cuyo nombre se deriva del inglés *Open Computing Language* [18], es una API de bajo nivel para computación heterogénea. En particular, en unidades centrales de procesamiento como en unidades de procesamiento gráfico.

A diferencia de CUDA, su utilización no está restringida a dispositivos fabricados por la empresa Nvidia. Es decir, el objetivo de OpenCL es definir un estándar abierto de programación paralela para los diversos procesadores (en especial masivamente paralelos) encontrados en múltiples dispositivos.

2.7. Intel Xeon Phi

Esta familia de procesadores fue gestada en la búsqueda por parte de la empresa Intel de alcanzar un acelerador de hardware con altas prestaciones en cuanto a las operaciones de punto flotante y, al mismo tiempo, reducir el consumo de energía de los procesadores de la familia Intel Xeon [42]. Fueron diseñados para explotar la arquitectura Intel de múltiples núcleos integrados (MIC del inglés *Many Integrated Core*), capaz de proveer desempeños de teraflops (trillones de operaciones matemáticas por segundo) de punto flotante de doble precisión.

El principio que se maneja es similar al de la utilización de las GPUs, es decir utilizar procesadores masivamente paralelos (varias decenas de procesadores) para incrementar el poder computacional.

Desde la arquitectura, además de utilizar técnicas ya vistas en este documento como ser el paralelismo a nivel de instrucción, pipelining, SIMD y multithreading, es importante resaltar dos cuestiones importantes de la misma: el enfoque utilizado para obtener paralelismo a nivel de los diferentes núcleos de procesador y la forma de interconexión de componentes.

Xeon Phi utiliza la arquitectura *manycore*. En comparación con las arquitecturas *multicore* que integran varios núcleos en un mismo chip, esta arquitectura se basa en la interconexión de núcleos más simples corriendo a frecuencias aún menores que las de los núcleos en *multicore*, pero utilizando un número elevado de núcleos.

La idea inicial de interconexión de estos núcleos fue que estuvieran conectados a través de un bus y que, a través de un controlador de memoria, accedan a la memoria principal. Luego de un tiempo se llegó a ver que esto sería un cuello de botella para extraer el mayor desempeño de aplicaciones paralelas. Esto hizo que la tecnología de interconexión utilizada para Intel Xeon Phi fuera una topología de tipo anillo bidireccional. De esta forma los núcleos se comunican a través de canales bidireccionales por los que acceden a datos y código que reside en memoria principal. Quien se encarga de distribuir estos pedidos e información entre la memoria principal y el anillo son controladores de memoria (pueden ser uno o más).

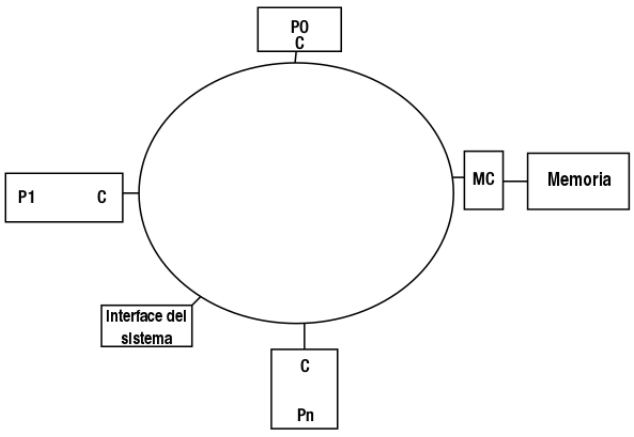

Figura 2.6: Arquitectura de procesadores *manycore* (Xeon Phi) conectados a través de un anillo. P_0-P_n son los núcleos, C es la caché y MC es el controlador de memoria. Adaptado de [42].

Finalmente, la comunicación con el dispositivo host generalmente se hace a través de slots PCI Express.

La arquitectura de estos dispositivos tiene, entre otros elementos a destacar, los siguientes componentes:

- **Núcleos de coprocesador**: están basados en núcleos P54c (Intel Pentium del año 1995) con modificaciones mayores (por ejemplo, las series 7100, 5100 y 3100 de 61, 60 y 57 núcleos respectivamente).
- **Interconexión de anillo**: la interconexión entre los núcleos y el resto de las componentes de los coprocesadores se establece según un anillo bi-direccional.
- **Controlador de memoria**: interface entre el anillo y la memoria GDDR.
- **Interface PCI Express**: esta es necesaria para conectar los coprocesadores a los buses PCI Express.

El primer coprocesador Xeon Phi se conoció con el nombre Knights Corner y fue anunciado en el año 2010 [43]. Entre sus características principales constaba de más de 50 núcleos de 64 bits (notar que en esa época era un número muy elevado de cores) que soportaban hasta 4 hilos de hardware en cada núcleo.

2.8. Arquitecturas híbridas (CPU + aceleradores)

Dado que múltiples áreas de investigación han requerido de alto poder de procesamiento, en los sistemas de HPC se busca explotar el poder de procesamiento de diversos recursos computacionales. En la actualidad, los procesadores utilizan varios núcleos con frecuencias reducidas para mejorar la performance. Las unidades de procesamiento gráfico (GPUs) han seguido una tendencia similar teniendo varios elementos de procesamiento en una sola pastilla de silicio. Los reportes indican que la performance de los núcleos computacionales intensivos se multiplican cuando son ejecutados en GPUs. Como consecuencia, el uso combinado de CPUs y GPUs en sistemas HPC se ha convertido en una elección popular al momento de diseñar este tipo de plataformas.

Capítulo 3

Benchmarks para computación científica

Como se dijo anteriormente, los benchmarks pueden ser definidos de manera informal, y en el contexto de la informática, como una colección de programas, rutinas o colecciones de datos que resumen un comportamiento representativo de una gran cantidad de ellos. En este proyecto, el foco está puesto en programas de computación científica. En este contexto, su objetivo es evaluar el desempeño computacional de diferentes arquitecturas de hardware y/o algoritmos.

En este capítulo se hace una descripción de los benchmarks más difundidos en la comunidad y, por lo tanto, que son objeto de estudio de este proyecto. Previo a ello se introducen algunos conceptos relacionados con algoritmia, álgebra lineal y grafos que servirán de base para las descripciones previamente mencionadas.

3.1. Base matemática utilizada por los benchmarks

En esta sección se introducen conceptos y algoritmos matemáticos utilizados por los tres benchmarks que se estudian en el presente proyecto de grado. En particular, y considerando los tres benchmarks estudiados, se agrupará esta sección en dos apartados: por un lado conceptos sobre álgebra lineal (densa y dispersa) y por otro lado sobre grafos. En algunos casos y con el fin de mantener el objetivo de no hacer pesada la lectura de este documento, se incluyen referencias externas para profundizar sobre los distintos conceptos.

3.1.1. Álgebra lineal

Se trata de la rama de la matemática que estudia principalmente los sistemas lineales de ecuaciones, matrices, vectores, espacios vectoriales y sus transformaciones lineales.

3.1.1.1. Álgebra lineal densa

El álgebra lineal densa, como lo dice su nombre, se centra en el estudio de operaciones vectoriales sobre estructuras matriciales completas. Tratado desde un punto de vista de las estructuras de datos se utilizan estructuras que almacenan todos los elementos de las matrices, o dicho de otra forma, las matrices que tienen "pocos"[1] coeficientes con valores en 0.

[1] No existe un criterio para determinar que es mucho o pocos ceros en una matriz. En general las matrices dispersas tienen menos de un 5 % de los elementos distintos de cero.

3.1.1.2. Resolución de sistemas lineales

Un sistema lineal se puede representar como un problema del estilo:

$$AX = B,$$

donde A, X y B son matrices dadas de tamaño $n \times n$, $n \times k$ y $n \times k$ respectivamente. X es una matriz conformada por k vectores incógnita y B es la solución de resolver cada uno de esos sistemas independientes.

Existen dos grandes tipos de estrategias para resolver estos sistemas: métodos directos e iterativos. Los primeros resuelven los problemas en un número específico de pasos, en función del tamaño del sistema a resolver, y se obtiene la solución exacta al problema, a menos de errores numéricos. Por otro lado, los métodos iterativos aproximan la solución al problema a partir de una sucesión generada iterativamente, convergiendo a una solución cuyo error satisface algún criterio prefijado. En general los sistemas lineales densos están fuertemente asociados a la utilización de métodos directos de resolución. Algunas referencias básicas donde profundizar sobre los métodos directos aplicados a matrices densas se pueden encontrar en [8] y [11].

Eliminación Gaussiana

Es un método directo de resolución de sistemas lineales que también se denomina escalerización y eliminación de Gauss. Consiste en transformar un sistema en uno triangular mediante operaciones elementales como ser intercambio de filas, multiplicar una fila por un escalar no nulo o sumar dos filas.

Estas operaciones que, si se nota el sistema lineal como $AX = B$, se hacen sobre la matriz A, deben ser replicadas sobre el vector B. La forma de hacer esto es multiplicando la matriz y el vector por una matriz M tal que, si se pretende por ejemplo "eliminar" el coeficiente no nulo a_{ij}, el valor $m_{ij} = \frac{a_{ij}}{a_{jj}}$. La matriz resultado tiene unos en su diagonal principal y los valores calculados mencionados en la columna i-esima.

Esto se repite para cada columna y el valor de a_{ii} en $A^{(i)} = MA^{(i-1)}$ se denomina pivote. Si uno de los pivotes es cero o cercano a este valor se realiza la técnica de pivoteo que implica intercambiar filas y/o columnas para que lo dejen de ser. Existen dos técnicas principales de pivoteo:

- **Pivoteo parcial:** se intercambian las filas k y r, siendo k la columna actual y r el mínimo entero para el cual:

$$|a_{rk}^{(k)}| = max_{k \leq i \leq n} |a_{ik}^{(k)}|$$

- **Pivoteo total:** se intercambian las filas k con r y columnas k con s tal que r y s son los mínimos enteros para los cuales:

$$|a_{rs}^{(k)}| = max_{k \leq i,j \leq n} |a_{ij}^{(k)}|$$

Descomposición LU

De forma general, se trata de una factorización de una matriz como el producto de dos matrices diagonales superior e inferior notadas como L y U respectivamente. Algunas particularidades son que:

- si la matriz A tiene uno o varios ceros en la diagonal principal es necesario multiplicar previamente por una o varias matrices de permutación (es decir, aplicar pivoteo).
- si la matriz A es invertible y admite una factorización LU, la misma es única.

La utilidad de esta factorización es que el problema de resolver un sistema lineal $AX = B$ se puede transformar en $PAX = PB$, llegando a resolver $LY = PB$ y $UX = Y$. Como se puede ver, la matriz P es la que realiza el pivoteo sobre la matriz A si expresamos el problema en términos de eliminación gaussiana.

En el Algoritmo 1 se puede ver el pseudocódigo de una implementación de la factorización LU.

Algoritmo 1 Pseudocódigo de Descomposición LU

```
1: procedure DescomposicionLU(A) ▷ A es la matriz a factorizar de tamaño n × n
2:     U = A
3:     L = I
4:     for i = {1 ... n − 1} do
5:         for j = {k + 1 ... n} do
6:             L(j, k) = U(j,k) / U(k,k)
7:             U(j, {k ... n}) = U(j, {k ... n}) − L(j, k)U(k, {k ... n})
8:         end for
9:     end for
10: end procedure
```

Rutina `DGEFA`

DGEFA (acrónimo del inglés *Double precision GEneral matrix FActor*) es una subrutina del paquete LINPACK. Como se dijo en la descripción del benchmark LINPACK, esta subrutina implementa una descomposición LU con pivoteo parcial de la matriz del sistema lineal a resolver. Esto implica que al momento de tratar el elemento de la diagonal (el pivot) en una fila se intercambian dos columnas k y r.

Esta rutina tiene $O(n^3)$ en operaciones de punto flotante (más específicamente $\frac{2}{3}n^3$), haciendo que en la ejecución del benchmark LINPACK sea la que consuma la mayor parte del tiempo.

Rutina `DGESL`

DGESL (acrónimo del inglés *Double precision GEneral matrix SoLve*) es una subrutina del paquete LINPACK. En esta biblioteca, la rutina utiliza el resultado de la descomposición que realiza DGEFA para resolver el sistema lineal. Esta rutina tiene $O(kn^2)$ operaciones de punto flotante, siendo k la cantidad de sistemas a resolver (HPL se basa en la resolución de un único sistema, i.e. $k = 1$). Por lo tanto se puede ver fácilmente que LINPACK invierte más tiempo en la rutina DGEFA que en DGESL.

3.1.1.3. Álgebra lineal dispersa

En contraposición al álgebra lineal densa, el área de estudio del álgebra lineal dispersa son las estructuras matriciales dispersas. Esto significa que dichas estructuras representan matrices con "muchos ceros".

Formatos de almacenamiento

Los problemas ingenieriles y, en particular, de cómputo intensivo como el que realizan los benchmarks, involucran el uso de matrices de tamaños cada vez más grandes. Por cuestiones de limitantes en el uso de memoria primero y aceleración en los cálculos luego, se evita almacenar los coeficientes con valores 0. Esto se sustenta en que en muchos problemas las matrices subyacentes presentan gran porcentaje de estos valores. A raíz de ello se han estudiado diversos formatos de almacenamiento alternativos a las matrices densas. Las bondades de cada formato dependen del contexto y las plataformas de hardware utilizadas.

A continuación se describen las dos grandes familias en las que se clasifican estos formatos.

Formatos estáticos

Estos formatos se utilizan cuando los valores no cambian o la estructura de la matriz es conocida. Se encuentran 5 formatos en este grupo:

- **Simple o elemental (representación por coordenadas)**

  En este formato solo se almacenan los valores no nulos y las coordenadas de fila y columna correspondientes a cada uno de esos valores. Las ventajas vienen por la intuitividad de esta representación, el uso de estructuras de igual tamaño y la indistinción si se pretende acceder a los valores por fila o columna[2]. Por otra parte, es una de las representaciones que usa más memoria y presenta cierta dificultad para acceder por fila o por columna a los valores. Por ejemplo, sea A una matriz tal que:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 & 4 & 0 \\ 0 & 5 & 0 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 9 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 6 & 0 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 \end{pmatrix},$$

  la representación en este formato sería en tres vectores fila f, columna c y dato d de la siguiente forma:

$$d = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$
$$f = \begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 5 & 5 & 5 & 6 & 6 & 6 & 7 \end{pmatrix}$$
$$c = \begin{pmatrix} 1 & 2 & 5 & 6 & 2 & 4 & 2 & 3 & 4 & 1 & 4 & 2 & 5 & 6 & 3 & 5 & 6 & 7 \end{pmatrix}$$

[2]Si no se especifica ningún orden al almacenar los términos.

- **Comprimido por fila o *Compressed Row Storage* (CRS)**

  Al igual que en formato simple, se mantienen tres vectores. En uno de estos vectores se almacenan todos los valores no nulos y en otro la coordenada de columna de esos valores. Sin embargo, para almacenar la coordenada de filas, se mantiene un vector de tamaño la cantidad de filas más uno en donde se almacena el índice donde comienza cada una de las filas en los vectores anteriores. Las ventajas en menor uso de memoria y facilidad de acceso a filas se contraponen al hecho de no utilizar estructuras equidimensionales, la dificultad e ineficiencia para acceder a columnas debido al direccionamiento indirecto. Para el caso de la matriz A utilizada en el formato anterior, su representación en CRS sería la siguiente:

$$d = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

$$f = \begin{pmatrix} 1 & 5 & 7 & 10 & 12 & 15 & 18 & 19 \end{pmatrix}$$

$$c = \begin{pmatrix} 1 & 2 & 5 & 6 & 2 & 4 & 2 & 3 & 4 & 1 & 4 & 2 & 5 & 6 & 3 & 5 & 6 & 7 \end{pmatrix}$$

- **Comprimido por columna o *Compressed Sparse Column* (CSC)**

  Se trata de un formato idéntico al anterior con la diferencia en que en lugar de almacenar los índices donde comienzan las filas, se hace lo propio con las columnas así como se almacenan las coordenadas de las filas de todos los valores no nulos. Las ventajas y desventajas son las mismas que el formato anterior.

- **Comprimido por bloque de filas o *Block Compressed Row Storage* (BCRS)**

  Es un formato que se utiliza cuando se pueden identificar bloques densos con cierto patrón en la matriz. La forma de almacenamiento es una matriz rectangular en la que se almacenan los bloques y dos arreglos en los que se guarda la pseudofila y pseudocolumna a la que pertenece cada bloque respectivamente. Cuando se habla de pseudofila y pseudocolumna son las filas y columnas que se forman a partir de los bloques; es decir, el bloque que contiene el elemento de la primer fila y columna de la matriz estará en la pseudofila 1 y pseudocolumna 1, el bloque que lo sigue a la derecha estará en la pseudofila 1, pseudocolumna 2 y así sucesivamente. La ventaja viene por el lado de la rapidez en los accesos a los bloques. La desventaja es que, como se dijo, solo es útil cuando se puede identificar bloques en las matrices.

- **Comprimido por diagonales o *Compressed Diagonal Storage* (CDS)**

  Este formato, a diferencia de los anteriores, solo utiliza una estructura de datos para almacenar los valores no nulos: una matriz rectangular en la que se guardan las diagonales. La ventaja principal es que solo es necesaria una estructura de datos para almacenar los valores pero, al igual que en el formato comprimido por bloque, no sirve para todos los casos ya que es necesario que los datos no nulos se encuentren en diagonales consecutivas a la diagonal principal. Existe una variante que almacena el *offset* de cada diagonal en el caso que sean pocas diagonales y estén dispersas en la matriz.

Formatos dinámicos

Se trata de formatos que utilizan estructuras dinámicas para almacenar los datos, como pueden ser vectores y/o listas enlazadas. Algunos formatos de esta familia se presentan a continuación:

- ***Linked List Row-Column Storage*** **(LLRCS)**

  Utiliza una multiestructura bidimensional con dos vectores de tamaño la cantidad de filas y columnas respectivamente. Para cada entrada se dispone de un puntero para recorrer el vector correspondiente. Los datos almacenados en cada entrada son el valor, la fila, la columna y dos punteros hacia el siguiente en la fila y siguiente en la columna.

- ***Linked List Row Storage*** **(LLRS)**

  En este caso se usa una estructura unidimensional en donde, con un arreglo de tamaño la cantidad de filas, cada entrada almacena una lista con todos los elementos de la fila correspondiente al índice de esa entrada.

- ***Linked List Column Storage*** **(LLCS)**

  La estructura es muy similar al caso anterior con la diferencia que el arreglo es de tamaño la cantidad de columnas y las entradas almacenan una lista de los valores no nulos de esa columna.

Solvers iterativos

Se trata de métodos de resolución, en nuestro caso particular, de sistemas lineales que aproximen su solución mediante una sucesión generada iterativamente. La forma de hacerlo es que la n-ésima aproximación de la solución se pueda derivar a partir de las aproximaciones anteriores. El error que tiene esta solución debe satisfacer algún criterio prefijado. En general los métodos de esta familia están asociados a la resolución de problemas dispersos.

Se puede agrupar estos solvers en dos grupos:

- **Métodos estacionarios**

  Reescribiendo el problema de resolver el sistema lineal en la forma:

  $$x^{k+1} = Mx^k + c,$$

  los métodos estacionarios son aquellos en los que la matriz M y c no dependen del paso de iteración.

- **Métodos no estacionarios**

  Son métodos que hacen uso de la información evaluada en cada iteración, permitiéndoles obtener la solución de modo dinámico. También son conocidos como métodos basados en subespacios de Krylov. El subespacio de Krylov de dimensión m asociado a la matriz A y al vector v se define como el subespacio definido por la base $\{v, Av, A^2v, \cdots, A^{m-1}v\}$; por una cobertura más detallada consultar [3].

Método del gradiente conjugado

Se trata de un método iterativo no estacionario que permite resolver sistemas lineales bajo la restricción que la matriz de coeficientes $A \in R^{n \times n}$ sea simétrica y definida positiva, o sea, simétrica y con valores propios positivos. Está basado en subespacios de Krylov, propiedad que lo hace un método no estacionario.

La idea básica consiste en construir una base de vectores ortogonales y utilizarla para realizar la búsqueda de la solución de forma más eficiente.

El algoritmo se puede describir de la siguiente forma:

- Los valores de entrada son el sistema lineal $Ax = b$ y un vector de búsqueda inicial u_0.
- Se calcula para iniciar el algoritmo $r_0 = b - Au_0$ y $p_0 = r_0$ denominados vectores residuo y dirección de descenso.
- Para los próximos $k+1$-ésimos pasos de iteración se calculan los siguientes valores hasta que se satisfaga un criterio de detención:

$$\beta_{k+1} = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$$

$$p_{k+1} = r_{k+1} + \beta_{k+1} p_k$$

$$\alpha_{k+1} = \frac{< r_k, p_k >}{< A p_k p_k >}$$

$$u_{k+1} = u_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k A p_k,$$

De forma de reducir la cantidad de iteraciones generalmente se utilizan precondicionadores. Esto implica reescribir el sistema lineal $Ax = b$ como $A'x = b'$, siendo A' y b' matriz y vector obtenidos a partir de la pre o posmultiplicación del sistema por una matriz M llamada precondicionador. En particular se reescribe el sistema como:

$$M^{-1}Ax = M^{-1}b.$$

El nuevo sistema debe cumplir con las siguientes propiedades: tener la misma solución y disminuir significativamente la cantidad de operaciones requeridas para resolver el sistema original. Para ello, típicamente aumentan la cantidad de operaciones por iteración pero disminuyen drásticamente la cantidad de iteraciones. Dicho en otras palabras, cumple con ciertas propiedades que hacen que la convergencia sea más rápida. La forma genérica de este método, conocido como método del gradiente conjugado precondicionado (PCG por su sigla en inglés), se puede apreciar en el Algoritmo 2.

Para profundizar en estos puntos se brindan las referencias [3] y [44].

Algoritmo 2 Forma genérica del Método del Gradiente Conjugado Precondicionado (PCG). Adaptado de [15].

1: **procedure** PCG(A,b,m,x_0) ▷ $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, M una matriz simétrica definida positiva (el precodicionador), u_0 un paso base tal que $Au_0 \approx b$
2: $k = 0$
3: $r_0 = b - Au_0$
4: **while** $r_k \neq 0$ **do**
5: Resolver $Mz_k = r_k$
6: $k = k + 1$
7: **if** $k = 1$ **then**
8: $p_1 = z_0$
9: **else**
10: $\beta_k = r_{k-1}^T z_{k-1} / r_{k-2}^T z_{k-2}$
11: $p_k = z_{k-1} + \beta_k p_{k-1}$
12: **end if**
13: $\alpha_k = r_{k-1}^T z_{k-1} / p_k^T A p_k$
14: $u_k = u_{k-1} + \alpha_k p_k$
15: $r_k = r_{k-1} - \alpha_k A p_k$
16: **end while**
17: $x = u_k$
18: **end procedure**

3.1.2. Grafos

Informalmente se puede definir un grafo como un conjunto de entidades, denominadas vértices o nodos, y enlaces, llamadas aristas o arcos, que relacionan a un subconjunto de las primeras. Más formalmente, un grafo G es una colección definida sobre un conjunto de vértices V y un conjunto de aristas A notadas como pares $(v, w) \in A$, donde los vértices $v, w \in V$ [22].

3.1.2.1. Recorrida en profundidad o *Depth First Search (DFS)*

Se trata de una recorrida de grafos en la que, a partir de un vértice, se va expandiendo la recorrida de todos los nodos del grafo en un camino. Cuando la recorrida por ese camino termina, vuelve y repite el proceso con todos y cada uno de los hermanos del nodo ya procesado. Tener en cuenta que si el grafo tiene más de una componente conexa se debe repetir el proceso desde otro vértice. Un pseudocódigo de esto se plantea en el Algoritmo 3.

3.1.2.2. Recorrida en amplitud o *Breadth First Search (BFS)*

El algoritmo BFS es una recorrida de grafos que parte de un vértice y recorre en primer término a todos sus vértices vecinos. Luego, para cada uno de los vecinos, se recorren sus nodos vecinos adyacentes y así sucesivamente hasta recorrer todo el árbol. En caso de incluir el grafo más de una componente conexa también se necesita repetir el prociedimiento para cada componente. Un pseudocódigo de esto se plantea en el Algoritmo 4.

Algoritmo 3 Pseudocódigo de DFS

```
1: procedure DFS(v)
2:     Marcar(v)
3:     Preprocesamiento
4:     for cada w adyacente a v do
5:         if w no marcado then
6:             DFS(w)
7:         end if
8:     end for
9:     Postprocesamiento
10: end procedure
11: procedure RecorridoDFS(G)                ▷ G = (V, A) es un grafo.
12:     for cada v ∈ V do
13:         Inicializar(v)                   ▷ Inicializa v como no marcado
14:     end for
15:     for cada v ∈ V do
16:         if v no marcado then
17:             DFS(v)
18:         end if
19:     end for
20: end procedure
```

Algoritmo 4 Pseudocódigo de BFS

```
1: procedure BFS(v)
2:     CrearCola(Q)
3:     Marcar(v)
4:     InsBack(Q,v)                          ▷ Encolar
5:     while la cola Q no esté vacía do
6:         u ← Primero(Q)
7:         Procesar(u)
8:         Q ← Resto(Q)
9:         for cada w adyacente a v do
10:            if w no marcado then
11:                Marcar(w)
12:                InsBack(Q,w)
13:            end if
14:        end for
15:    end while
16: end procedure
17: procedure RecorridoBFS(G)                ▷ G = (V, A) es un grafo.
18:    for cada v ∈ V do
19:        Inicializar(v)                    ▷ Inicializa v como no marcado
20:    end for
21:    for cada v ∈ V do
22:        if v no marcado then
23:            BFS(v)
24:        end if
25:    end for
26: end procedure
```

3.2. LINPACK

LINPACK [37] es un benchmark que permite evaluar el desempeño de las operaciones de punto flotante (también llamado coma flotante) de una computadora. La evaluación se realiza a partir de la ejecución de una rutina que resuelve un sistema de ecuaciones lineales denso. Esta herramienta se utiliza en la práctica para confeccionar la lista TOP500 [50] y GREEN500 [49].

Básicamente, la herramienta consiste en una implementación de una descomposición LU con pivoteo parcial para resolver un sistema lineal de ecuaciones para matrices densas de tamaño $n \times n$. La solución al sistema se obtiene por la resolución de los dos sistemas triangulares subyacentes [15].

Este benchmark ha variado a lo largo de los años al punto que en la actualidad el reporte de este benchmark tiene tres rutinas:

- **LINPACK Fortran n=100**

  Se calcula a partir de una matriz de orden 100 (matriz A de tamaño 100×100) utilizando el software LINPACK en FORTRAN.

  Lo que este benchmark mide es la performance de dos rutinas de la colección de software LINPACK: `DGEFA` y `DGESL`. La primera realiza una descomposición LU con pivoteo parcial *in-situ* y la segunda rutina usa esta descomposición para resolver el sistema de ecuaciones lineales (i.e. la resolución de dos sistemas triangulares). Como se dijo anteriormente, la mayor parte del tiempo transcurre ejecutando `DGEFA` dado que implica tareas de $O(n^3)$, mientras las tareas de `DGESL` son de $O(n^2)$.

  Las reglas básicas para usar esta variante son que el programa sea ejecutado sin cambios en el código [37], ni siquiera comentarios. El compilador, a través del uso de opciones de compilación, puede desarrollar optimización en tiempo de compilación. El usuario debe brindar una función de conteo de tiempo denominada `SECOND`. La función devuelve el tiempo de ejecución de CPU para el proceso.

- **LINPACK n=1000 (TPP, Best Effort)**

  A diferencia del anterior, esta variante se ejecuta sobre una matriz cuadrada de dimensión 1000. El objetivo de este benchmark está más relacionado con la posibilidad de especificar un solver para ecuaciones lineales implementado en cualquier lenguaje. Un requerimiento previo que tiene es que el método debe computar una solución y ésta debe retornar un resultado con la precisión definida inicialmente (es decir, con un valor de $|B - AX|$ acotado, siendo $AX = B$ el sistema lineal de ecuaciones a resolver).

  Las reglas básicas para la ejecución del benchmark permiten un reemplazo total por parte del usuario de la factorización LU y los pasos de resolución. La secuencia de invocaciones debe ser igual a la de las rutinas originales. El tamaño del problema, como ya se dijo es de dimensión 1000. La precisión de la solución debe satisfacer el siguiente límite:

  $$\frac{\|Ax - b\|}{\|A\| \, \|n\| \, n\epsilon} \leq O(1),$$

  donde ϵ es la precisión de la máquina en doble precisión (conocido también como ϵ_{mach}) y n el tamaño del problema. Esto implica que el cómputo debe ser realizado en aritmética de punto flotante de 64 bits.

- **High Performance LINPACK (HPL)**

  Esta herramienta, utilizada para confeccionar el reporte de TOP500 que se mencionó antes, pretende medir la mejor performance de una máquina resolviendo un sistema de ecuaciones. El tamaño del problema y el software a utilizar pueden ser elegidos para producir el mejor desempeño. Al igual que la rutina anterior, el cómputo se realiza (en general) en doble precisión.

  Para este benchmark las reglas básicas de ejecución permiten modificar por completo la factorización LU y los pasos de resolución. La precisión de la solución debe satisfacer el mismo límite que para el TPP.

El benchmark LINPACK utiliza el paquete LINPACK [28] y su predecesora LAPACK [55], que son colecciones de rutinas en FORTRAN para resolver sistemas de ecuaciones lineales. El software está basado en un enfoque descomposicional del álgebra lineal numérica. La idea general es que dado un problema que involucre una matriz, se factoriza o descompone la matriz en un producto de matrices más simples y estructuradas que pueden ser manipuladas fácilmente para resolver el problema original. El paquete tiene la capacidad de manejar varios tipos diferentes de matrices (generales, de banda, etc.) y de datos (doble precisión, complejos, etc.). LINPACK y LAPACK están construidos sobre la especificación BLAS [4] que también maneja este tipo de matrices (datos) y da soporte a las operaciones básicas de álgebra.

3.3. HPCG

HPCG (sigla en inglés de *High Performance Conjugate Gradient*) es un benchmark complementario a HPL. La motivación detrás del desarrollo de este benchmark, según el sitio oficial [27], es que los patrones de acceso a datos y a recursos que hace HPL no están haciendo que los diseños de sistemas computacionales vayan en una dirección beneficiosa para diversas aplicaciones. En especial, con aquellos que utilizan matrices en formatos dispersos o grafos de grandes dimensiones y que son los que permiten escalar en forma importante en la dimensión de los problemas.

El benchmark fue definido en el año 2013, ver [9, 10], como se dijo anteriormente con foco principal en cubrir aspectos de desempeño no considerados en el HPL. En particular, la herramienta genera una ecuación diferencial parcial discreta en el espacio tri-dimensional y computa iteraciones del solver iterativo para sistemas lineales simétricos y definidos positivos (SPD). El precondicionador utilizado en este caso es el Gauss-Siedel simétrico local.

El código de referencia de HPCG está implementado en C++ y consta de cuatro etapas:

1. **Configuración del problema**

   Se genera una matriz simétrica y definida positiva (SPD) usando el formato de filas dispersas comprimidas CSR (del inglés *Compressed Sparse Row*), un vector derecho b y un vector solución x inicial.

2. **Configuración del precondicionador**

   Se organizan inicialmente las estructuras de datos para el precondicionador de tipo Gauss-Siedel. La versión de referencia utiliza la representación CSR para la matriz triangular superior e inferior, cada una como una matriz separada.

3. **Configuración de la validación y verificación**

   Se computan pre y post condiciones e invariantes que ayudarán en la detección de anomalías durante la fase de iteración.

4. **Iteración**

   Se ejecutan m iteraciones, n veces, usando la misma semilla cada vez, donde m y n son lo suficientemente grandes para probar el tiempo de ejecución del sistema. Al hacer esto se puede comparar la correctitud de los resultados numéricos al final de cada m-ésima fase de iteración.

5. **Post procesamiento y reporte**

   Se reportan los siguientes datos:

   - Métricas de verificación y validación computacional.
   - Tiempos de ejecución.
   - Número de nodos, almacenamiento total, procesadores, aceleradores, precisión utilizada, versión de compilador, nivel de optimización, directivas de compilador utilizadas, conteo de flops, energía utilizada, efectos de caché, carga y almacenamiento, etc.

3.4. Graph500

Es un benchmark diseñado para evaluar uso de grandes volúmenes de datos no estructurados. Se hizo en contraposición a otros benchmarks como HPL que están basados en la realización intensiva de operaciones de punto flotante [17]. El benchmark del ranking Graph500 se basa en hacer un uso intensivo de los sistemas de comunicación interno tanto en lo que refiere al acceso a memoria en contextos centralizados como de transferencias de datos en los contextos distribuidos.

El problema que resuelve este benchmark está basado en un grafo no dirigido con pesos en las aristas. Consiste en tres fases (o núcleos): la construcción de un grafo en formato CSR, un núcleo BFS (del inglés *Breadth First Search*) y, finalmente, la validación transversal del BFS.

La ejecución de este benchmark consta de los siguientes pasos:

- **Generar una lista de aristas**

  Para una escala S y un factor de vértices dado, siendo S el logaritmo en base 2 del número de vértices y el factor de vértices la relación entre la cantidad de aristas y vértices, se genera una lista de aristas aleatorias para construir el grafo. Vale decir que la escala S define la cantidad de vértices N ya que $N = 2^S$.

- **Núcleo 1: construir un grafo**

  En este paso se acepta la lista de aristas y se crea un grafo válido. Esto requiere eliminar ciclos propios (aristas cuyo inicio y fin son el mismo vértice). El grafo resultado se devuelve en una estructura de datos conveniente como una matriz o lista de adyacencia.

- **Generación aleatoria de 64 vértices de salida**

  La especificación requiere que se definan 64 vértices distintos de salida para la ejecución de la búsqueda. El vértice debe producir una búsqueda no trivial, lo que implica que debe tener al menos una arista a otro vértice.

- **Núcleo 2: desarrollar una búsqueda BFS comenzando por cada uno de los vértices anteriores**

  El algoritmo general para el BFS se basa generalmente en el uso de colas. Una versión básica se describe a continuación:

  1. Agregar el vértice raíz a la cola (como último elemento).
  2. Desencolar el primer elemento de la cola, marcarlo como visitado y agregar a cada uno de sus *vecinos* no visitados a la cola.
  3. Repetir el paso previo hasta que la cola está vacía, indicando que todos los vértices conectados han sido visitados.

- **Validar que los resultados son correctos**

  Este paso busca verificar errores en el grafo descubierto efectuando las siguientes pruebas:

  1. El árbol BFS no contiene ciclos.
  2. Cada arista del árbol conecta vértices cuyos niveles BFS difieren exactamente en uno.
  3. Cada arista de la lista de entrada tiene vértices con niveles que difieren en a lo sumo uno o no están en el árbol BFS.
  4. El árbol BFS contiene a toda la componente conexa.
  5. Un vértice y su *padre* están unidos por una arista en el grafo original.

- **Salida**

  En la salida se debe incluir la siguiente información:

  - Escala.
  - Número de aristas.
  - Número de búsquedas BFS.
  - Tiempo de construcción del grafo.
  - Tiempos de búsqueda mínimo, máximo, promedio y desviación estándar.
  - Número de aristas recorridas mínimo, máximo, promedio y desviación estándar.
  - Aristas recorridas por segundo mínimo, máximo, promedio, desviación estándar y media armónica[3].

El benchmark usa la media armónica de aristas recorridas por segundo para determinar la posición de una computadora en el ranking.

El código base se puede bajar desde el sitio oficial del benchmark [17] y tiene una serie de directorios con diversas implementaciones:

- Basada en listas, secuencial.
- Basada en matriz de adyacencia utilizando formato disperso, en particular, comprimido por filas (CSR), secuencial.

[3]Dados $x_1, x_2, \ldots, x_n$, la media armónica de ellos es el inverso de la media aritmética de sus recíprocos. Es decir, si notamos este valor como H, $H = \frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\cdots+\frac{1}{x_n}}$

- Basada en matriz de adyacencia dispersa, comprimido por filas (CSR) para OpenMP.
- Basada en filas dispersas comprimidas (CSR) para Cray XMT[4].
- Utilizando MPI en listas (contiene una implementación de referencia a partir de la cual se puede convertir también en una estructura de filas dispersas comprimidas).

El benchmark permite que se hagan modificaciones para hacerlo más eficiente. En particular, se puede ejecutar el generador de datos del grafo utilizando técnicas de programación paralela, pero es necesario que los vértices sean nombrados globalmente y los datos generados no tengan localidad ni en memoria local ni globalmente entre procesadores. También da libertad en la forma de representación de la estructura del grafo que se utilizará a partir del Núcleo 1.

[4]Se trata de un supercomputador con multithreading masivo diseñado para atacar, entre otros problemas, el análisis y administración de patrones aleatorios de referencia a gran escala. [5]

Capítulo 4

Instanciación de los benchmarks

En este capítulo se describen aspectos fundamentales en lo que respecta a la configuración de los benchmarks estudiados en el proyecto cuando son aplicados en las distintas plataformas de hardware abordadas. Además, se resumen los resultados en cuanto a desempeño computacional que se han obtenido luego de su ejecución en las distintas plataformas.

En primera instancia se detallan las plataformas de hardware utilizadas para luego describir el trabajo realizado con cada benchmark.

4.1. Plataformas de hardware utilizadas

Para realizar la evaluación experimental se utilizaron tres plataformas de hardware diferentes, dos de las cuales incluyen GPUs y una que no lo hace. En la Tabla 4.1 se detallan las principales características de las plataformas empleadas tanto en lo que concierne al hardware como al software.

En lo que respecta a las GPUs utilizadas para la evaluación, se puede señalar que ambas son de alta gama, de la línea de cálculo científico (Tesla) y las dos pertenecen a la familia de arquitectura Kepler. Por otro lado, las CPUs utilizadas en la evaluación experimental presentan una mayor disparidad en sus características ya que claramente la CPU de BUENAVENTURA, un procesador Intel Core i3, es de gama baja, mientras que los otros dos procesadores considerados en la evaluación, son Intel Core i7, y ofrecen un poder de cómputo medio.

4.2. Bibliotecas utilizadas

Además de las bibliotecas oficiales correspondientes a cada una de las rutinas se ha experimentado con otras alternativas a las mismas. En esta sección se explica el origen de las herramientas y algunas características destacables de ellas.

4.2.1. BLAS

BLAS (sigla en inglés de *Basic Linear Algebra Subroutines*) es un paquete de rutinas que define una interfaz estándar y provee de una implementación de referencia para gran parte de las operaciones básicas de vector y matrices que son el estándar de facto para bibliotecas de álgebra lineal. Existen tres niveles de BLAS según el orden cronológico de definición y publicación así como del orden de operaciones de punto flotante y accesos a datos que realizan.

Platforma		CPU	GPU
R2D2	Procesador	Intel Core i7 - 3630QM	-
	Número de núcleos	4	-
	GHz.	2.4	-
	Memoria	16GB	-
	S.O.	Fedora 23	-
	Compilador	gcc 5.3.1-6	-
MOJIGATA	Procesador	Intel Core i7 - 4770	Tesla K40
	Número de núcleos	4	2880 núcleos CUDA
	GHz.	3.4	-
	Memoria	16GB	12GB
	S.O.	Fedora 21	-
	Compilador	gcc 4.9.2-6	CUDA 6.5
BUENAVENTURA	Procesador	Intel Core i3 - 3220	Tesla K20
	Número de núcleos	2	2496 núcleos CUDA
	GHz.	3.3	-
	Memoria	8GB	5GB
	S.O.	CentOS 6	-
	Compilador	gcc 4.4.7-16	CUDA 4.0

Tabla 4.1: Características de las plataformas de hardware utilizadas en la evaluación experimental.

Las principales características de cada nivel se describen a continuación:

- *Nivel 1:* Desarrollan operaciones escalares, vectoriales y vector-vector. Generalmente son operaciones de $O(n)$. Un ejemplo de operaciones de este nivel es la suma generalizada de vectores, llamada *axpy*, de la forma:

$$y \leftarrow \alpha x + y,$$

  siendo α un escalar y x e y vectores de iguales dimensiones.

- *Nivel 2:* Desarrollan operaciones matriz-vector. Típicamente son operaciones de $O(n^2)$. Ejemplificando operaciones de este nivel está la multiplicación de matriz general y vector, llamada *gemv*, de la forma:

$$y \leftarrow \alpha Ax + \beta y,$$

  siendo α y β escalares, A una matriz de tamaño $m \times n$ y x e y vectores.

- *Nivel 3:* Desarrollan operaciones matriz-matriz, como ser la multiplicación de matrices generales, llamada *gemm*, de la forma:

$$C \leftarrow \alpha AB + \beta C,$$

  siendo α y β escalares y A, B y C matrices de tamaño $n \times m$, $m \times n$ y $m \times m$ respectivamente. En la mayoría de los casos se trata de operaciones de $O(n^3)$.

Las implementaciones modernas de esta biblioteca proveen operaciones de los tres niveles.

4.2.1.1. Versión de referencia

La versión de referencia se encuentra disponible de forma gratuita en Netlib[1] [4] y está escrita en FORTRAN 77. La última actualización de esta implementación es de noviembre del año 2015.

4.2.1.2. OpenBLAS

Se trata de una implementación de la biblioteca BLAS altamente optimizada que se encuentra bajo la licencia BSD [40], que la hace de código abierto. Se basa en GotoBLAS, que es otra implementación desarrollada en el Texas Advanced Computing Center [38]. La motivación del desarrollo de esta biblioteca fue el fin del mantenimiento de GotoBLAS2 por parte del Texas Advanced Computing Center. En la actualidad, OpenBLAS es desarrollada principalmente por el Laboratorio de Software Paralelo y Ciencia Computacional del Instituto de Software de la Academia China de Ciencias.

Las optimizaciones de esta biblioteca están hechas principalmente en las operaciones de nivel 3.

4.2.2. LAPACK

LAPACK (siglas en inglés de *Linear Algebra PACKage*) es un paquete de software que provee rutinas para resolver sistemas lineales, mínimos cuadrados, descomposición en valores propios y valores singulares. A diferencia de LINPACK, que originalmente fue diseñado para aprovechar los procesadores vectoriales con memoria compartida, este paquete explota de manera eficiente la jerarquía de memoria en arquitecturas que incluyen memorias cachés. En particular, a partir de reorganizar los algoritmos resuelve el problema que tiene LINPACK de estar el mayor porcentaje del tiempo moviendo datos en lugar de realizando operaciones de punto flotante. Esta reorganización implica el uso de operaciones de bloques de matrices, como el producto de matrices, en sus iteraciones internas. Estas operaciones de bloque pueden ser optimizadas para cada arquitectura para aprovechar la jerarquía de memoria y así proveer una forma transportable de alcanzar alta eficiencia en diversas máquinas modernas. Por este motivo, esta implementación es la utilizada para el benchmark HPL.

La versión de referencia se encuentra de forma gratuita disponible a través de Netlib y está escrito en FORTRAN 90. La última actualización data de junio de 2016.

4.3. LINPACK

Se ha decidido utilizar la variante HPL de este benchmark principalmente por dos motivos. En primer lugar es la única que permite determinar el tamaño de los problemas a utilizar y, en segundo lugar, es el utilizado para confeccionar el ranking TOP500 de supercomputadoras.

4.3.1. Configuración inicial

Para utilizar este benchmark, se debe comenzar por descargar el código de esta implementación desde el sitio oficial [36]. Existe una variante estable de este benchmark que permite explotar el poder de cómputo de la GPU. Luego de ello, se debe generar un archivo `makefile` con nombre `Make.<arch>`, siendo `<arch>` un identificador arbitrario

[1] **Netlib** es un repositorio mantenido por AT&T, Laboratorios Bell, la Universidad de Tennessee y el Laboratorio Nacional de Oak Ridge.

de la arquitectura a evaluar. En él se debe indicar, entre otras, ubicaciones de directorios y comandos para ejecutar la biblioteca BLAS.

Luego de creado este archivo se ubica en la raíz del directorio de instalación del benchmark y se ejecuta el comando `make arch=<arch>`. Esto genera un ejecutable en el directorio `bin/<arch>` denominado `xhpl`.

Para ejecutar finalmente el benchmark en una plataforma de memoria compartida u otra de memoria distribuida, se debe utilizar el comando `mpirun` perteneciente a la biblioteca OpenMPI[2]. Tanto la documentación oficial del benchmark como la *man-page* de este comando indican que uno de los parámetros que se debe incluir es la cantidad de procesos del programa que se ejecutarán como aplicación MPI.

La ejecución anterior utiliza los parámetros que se indican en el archivo `HPL.dat`, que debe estar ubicado en el mismo directorio que el ejecutable `xhpl`. Este archivo consta de varias líneas y concentra los principales parámetros del benchmark. Los más importantes son:

- *Línea 5*: Especifica la cantidad de tamaños de problemas que se ejecutarán (no debe ser mayor a 20).
- *Línea 6*: Indica los tamaños de problemas que se quiere ejecutar separados por espacio. La cantidad de valores está limitada por lo ingresado en la línea 5.
- *Línea 7*: Especifica el número de tamaños de bloque a ser ejecutados (no debe ser mayor a 20).
- *Línea 8*: Indica los tamaños de bloque que se quiere ejecutar separados por espacio. La cantidad de valores está limitada por lo ingresado en la línea 7.
- *Línea 9*: Indica como deben ser mapeados los procesos MPI en los nodos de la plataforma a testear. Hay dos opciones posibles: por filas y por columnas.
- *Línea 10*: Especifica el número de grillas de proceso a ser ejecutadas (no debe ser mayor a 20).
- *Líneas 11 y 12*: Especifica el número de filas y columnas de cada grilla de procesos que se quiere ejecutar.

4.3.2. Ejecuciones en entornos CPU

La primera prueba realizada se focaliza en evaluar el desempeño de HPL en CPU y la incidencia en el desempeño del benchmark del uso de diferentes implementaciones de la especificación BLAS. En particular, se evalúa el desempeño del benchmark al utilizar OpenBLAS [58] y la implementación de referencia (disponible en Netlib [37]). En este sentido se evaluó el desempeño de HPL sobre sistemas con diferentes dimensiones (2000, 5000 y 10000), usando dos procesadores de capacidades dispares. Además, se estudió el impacto al variar otros parámetros como la definición de la grilla o el tamaño de bloque utilizado.

Las Tablas 4.2 y 4.3 presentan los GFLOPS (10^9 *flops*, i.e. operaciones de punto flotante) alcanzados por las distintas variantes del benchmark sobre los equipos R2D2 y MOJIGATA. La primera plataforma se eligió para hacer pruebas iniciales y la segunda porque cuenta con tarjeta gráfica y es útil para realizar la comparación directa con

[2]Es una biblioteca open source que implementa el estándar de pasaje de mensajes de alta performance MPI [39].

la siguiente prueba experimental. También conviene aclarar que se utilizaron solo dos tamaños de grilla y en consecuencia hasta 4 procesos para simplificar las pruebas y solo para usarlas como pruebas de concepto, pero es posible probar con mayor cantidad de procesos. Si bien se evaluó el benchmark con una larga lista de valores para el tamaño de bloque (nb), en las tablas se presentan únicamente los resultados para las tres mejores configuraciones de nb.

Dimensión del problema	Biblioteca BLAS	Tamaño de bloque (nb)	Tamaño de grilla	Rendimiento (en GFLOPS)
2000	OpenBLAS	64	1x1	6,0
			2x2	29,3
		128	1x1	4,6
			2x2	40,1
		256	1x1	6,1
			2x2	33,8
	BLAS	64	1x1	0,8
			2x2	9,3
		128	1x1	0,8
			2x2	8,4
		256	1x1	0,8
			2x2	6,2
5000	OpenBLAS	64	1x1	7,1
			2x2	60,4
		128	1x1	7,0
			2x2	58,3
		256	1x1	7,3
			2x2	51,2
	BLAS	64	1x1	0,8
			2x2	9,6
		128	1x1	0,8
			2x2	6,8
		256	1x1	0,8
			2x2	4,6
10000	OpenBLAS	64	1x1	7,3
			2x2	66,0
		128	1x1	7,2
			2x2	66,6
		256	1x1	7,4
			2x2	57,5
	BLAS	64	1x1	0,8
			2x2	6,7
		128	1x1	0,8
			2x2	4,6
		256	1x1	0,8
			2x2	4,2

Tabla 4.2: Desempeño (en GFLOPS) alcanzado por HPL empleando la CPU de R2D2. Se utilizó las bibliotecas BLAS y OpenBLAS, grillas de 1×1 y 2×2 y tamaño de bloque entre 32 y 512.

Dimensión del problema	Biblioteca BLAS	Tamaño de bloque (nb)	Tamaño de grilla	Rendimiento (en GFLOPS)
2000	OpenBLAS	128	1x1	17,8
			2x2	16,8
		256	1x1	20,6
			2x2	15,0
		512	1x1	17,1
			2x2	11,9
	BLAS	64	1x1	1,2
			2x2	6,9
		128	1x1	1,2
			2x2	6,5
		256	1x1	1,2
			2x2	5,5
5000	OpenBLAS	256	1x1	39,0
			2x2	46,4
		512	1x1	40,0
			2x2	39,4
		768	1x1	34,5
			2x2	40,0
	BLAS	128	1x1	1,2
			2x2	11,3
		256	1x1	1,1
			2x2	6,5
		512	1x1	1,1
			2x2	4,7
10000	OpenBLAS	128	1x1	49,9
			2x2	79,8
		256	1x1	53,2
			2x2	74,3
		512	1x1	52,1
			2x2	77,5
	BLAS	64	1x1	1,2
			2x2	0,1
		128	1x1	1,2
			2x2	6,4
		256	1x1	1,1
			2x2	5,0

Tabla 4.3: Desempeño (en GFLOPS) alcanzado por HPL empleando la CPU de mojigata. Se utilizó las bibliotecas BLAS y OpenBLAS, grillas de 1×1 y 2×2 y tamaño de bloque entre 32 y 512.

De los resultados experimentales obtenidos se puede ver que en todos los casos las ejecuciones del benchmark utilizando la biblioteca OpenBLAS son más eficientes, en tiempo de ejecución, en cuanto permiten computar una mayor cantidad de operaciones en un mismo tiempo que utilizando la versión de referencia de BLAS. Asímismo, se puede apreciar que a medida que aumenta la dimensión del caso tratado se alcanzan mayores ratios de desempeño. En cuanto al tamaño de bloque, no se puede determinar

a priori un tamaño de bloque óptimo para todos los casos. Todos estos resultados son coherentes con la bibliografía especializada en el tema (ver por ejemplo [2, 41]). Notar que OpenBLAS incluye diversas optimizaciones a nivel de uso de la jerarquía de memoria mientras que la versión de referencia sólo hace uso de técnicas de *loop unrolling* (además de las ofrecidas automáticamente por el compilador empleado). En cuanto al tamaño de bloque óptimo, la definición de este valor implica la optimización de diferentes balances (por ejemplo uso de memoria caché, tamaños amigables para la paralelización multihilo, etc.), entonces es razonable que el valor óptimo dependa del tamaño de problema abordado, la plataforma de ejecución y la bibiloteca BLAS utilizada. Por último, el tipo de algoritmo empleado por HPL (la factorización LU a bloques) implica que al aumentar la dimensión del problema tratado se esté aumentando la cantidad de operaciones de tipo BLAS3 (ricas en cómputos con bajos requerimientos de acceso a memoria) en detrimento de las operaciones BLAS1 y BLAS2 (limitadas por los accesos a memoria). Esta situación genera que a mayor dimensión de problema mayor desempeño (generalmente hasta cierta dimensión de saturación del procesador; el desempeño en este caso se espera limitado por el pico de desempeño del procesador).

4.3.3. Ejecuciones en entornos CPU + aceleradores

En segunda instancia se estudió el uso de arquitecturas híbridas que incluyan GPUs como aceleradores de hardware. En esta línea, la Tabla 4.4 incluye los desempeños alcanzado por HPL al utilizar la tarjeta gráfica de MOJIGATA para instancias de diferentes dimensiones (2000, 5000 y 10000). Nuevamente, la evaluación se realizó con diversas configuraciones pero se presenta únicamente la que alcanzó mejores desempeños (además es la sugerida por la herramienta).

Dimensión	GFLOPS
2000	48,4
5000	159,7
10000	321,2

Tabla 4.4: Desempeño (en GFLOPS) alcanzado por HPL utilizando la GPU de MOJIGATA. Se utilizaron grillas de 1×1 y tamaño de bloque 768.

Considerando los resultados resumidos en las Tablas 4.3 y 4.4 se puede decir que el uso de la GPU mejora el desempeño de HPL notoriamente. Adicionalmente, es importante destacar que el desempeño de la variante en GPU crece en mayor medida ante el aumento en la dimensión del problema que la versión en CPU. Este resultado es previsible si se consideran, además de los aspectos vertidos cuando se evaluó la variante en CPU, que la variante en GPU tiene que compensar los tiempos de transferencia entre ambos dispositivos. En este sentido, es importante notar que el algoritmo implica un orden cuadrático (en la dimensión de los sistemas lineales) de transferencias de datos y cúbico de cálculos.

Por otro lado, estos resultados muestran también que es más difícil acercarse a la velocidad pico de las GPUs que en las CPUs. En este caso, por ejemplo el desempeño pico (teórico) de la CPU de R2D2 en artimética de doble precisión es de 76,8 GFLOPS, mientras que el de la tarjeta gráfica de MOJIGATA (NVIDIA K40) alcanza los 1,49 TFLOPS (10^{12} *flops*) para la misma precisión. Si se compara con los desempeños alcanzados, 66,6 y 321,2 GFLOPS respectivamente para CPU y GPU, se puede concluir que el primer dispositivo supera el 85 % de su capacidad, y por otro lado, la GPU apenas alcanza un 21,5 %.

4.3.4. Valores de referencia del benchmark

A manera informativa se incluyen algunos resultados, disponibles en la literatura, del benchmark HPL ejecutado en CPU para otras plataformas de hardware. Notar que este tipo de benchmark generalmente se ejecuta sobre grandes plataformas de hardware de HPC, en nuestro caso el foco es la utilización de GPUs en conjunción con equipos de escritorio, por lo tanto se agregan algunos pocos resultados de ejecutar el HPL sobre equipamiento comparable. En particular se muestran en la Tabla 4.5 los resultados reportados sobre el equipo Myrinet [36], máquina de la Universidad de Tennessee (8 Duals Intel PIII 550 Mhz –512 Mb–), donde se ejecutó HPL con diferentes configuraciones de grilla y tamaños de problema y empleando la biblioteca BLAS ATLAS[3]. Es importante tener en cuenta que el pico de rendimiento teórico de cada uno de estos procesadores es 1,1 GFLOPS [24].

Dimensión	Grilla	GFLOPS
2000	2×4	1,8
	4×4	2,3
5000	2×4	2,3
	4×4	4,0
10000	2×4	2,5
	4×4	4,5

Tabla 4.5: Desempeño (en GFLOPS) alcanzado por HPL utilizando Myrinet. Extraído de [36].

Como se puede apreciar, los valores obtenidos en nuestro contexto de ejecución en CPU son comparables con los obtenidos en la máquina de la Universidad de Tennessee, siendo completamente superados ambos resultados al explotar el poder de cómputo de la GPU.

4.4. HPCG

El código del benchmark se descarga desde el sitio oficial y para su utilización se debe compilar el mismo utilizando el makefile correspondiente (incluido en la distribución). Las dependencias mínimas que tiene son MPI y el compilador de C++. Los directorios de instalación de estas dependencias deben ser especificados en el archivo `Makefile` utilizado por HPCG para generar el binario. Al igual que HPL, este benchmark dispone de una versión estable para explotar el poder de cómputo de las GPUs.

[3] **ATLAS** es una implementación de BLAS que brinda mayor rendimiento que la implementación base redefiniendo algunas de las operaciones básicas de la biblioteca. En especial, la biblioteca incluye una etapa de ajuste de los tamaños de bloque al ser instalada en cada sistema.

4.4.1. Configuración inicial

Pasada la parte de configuración se debe realizar la personalización del benchmark para, entre otras cosas, determinar el tamaño del problema y el tiempo máximo que se utilizará en la ejecución. Por defecto el archivo de personalización, llamado `hpcg.dat`, tiene un tamaño de problema de 104 puntos (en una dimensión) y el tiempo máximo de ejecución se encuentra fijado en 60 segundos. Según lo que se indica en el archivo de personalización estos parámetros son suficientes para hacer una ejecución de prueba. En el sitio oficial de este benchmark se indica que para que la ejecución sea considerada oficial debe demorar al menos 30 minutos.

4.4.2. Ejecuciones en entornos CPU

En el caso del HPCG se ejecutó la variante original sobre CPU. Al igual que el HPL se emplearon las plataformas R2D2 y MOJIGATA (es decir, procesadores con capacidades dispares). Para estudiar esta herramienta se utilizaron problemas de tres dimensiones: 88, 96 y 104. Recordar que la dimensión, en este caso, define la cantidad de puntos de discretización de la ecuación diferencial en una sóla dimensión, pero el problema resuelto se establece por un cubo (tridimensional), i.e. la dimensión 96 implica resolver sistemas lineales dispersos cuya matriz tiene un tamaño de 884736×884736.

La Tabla 4.6 resume los desempeños alcanzados por el benchmark en ambas plataformas. En todos los casos éstos son presentados en HPCG rating, que es un valor ponderado calculado a partir de las operaciones realizadas en la fase de iteración sobre el tiempo de ejecución (expresado en GFLOPS/s). En forma más específica, el indicador se define como una estimación de las operaciones realizadas sobre el tiempo que insumen las operaciones efectuadas en la fase de iteración de gradiente conjugado precondicionado. El tiempo de generación del problema y cualquier otra modificación efectuada para mejorar el rendimiento también se considera para este índice: la suma de dichos valores se divide entre 500 iteraciones (actuando como peso de amortización) y es agregado al tiempo de ejecución.

Dispositivo	Plataforma	Tamaño del cubo	HPCG Rating (GFLOPS/s)
CPU	R2D2	88	1,1
		96	1,2
		104	1,3
	MOJIGATA	88	1,3
		96	1,4
		104	1,4

Tabla 4.6: Desempeño (en GFLOPS/s) alcanzado por HPCG utilizando la CPU de R2D2 y MOJIGATA.

4.4.3. Ejecuciones en entornos CPU + aceleradores

Para utilizar la tarjeta gráfica como acelerador en este benchmark, se debe emplear el binario ya compilado que se encuentra disponible en el sitio oficial del benchmark [27]. El código que genera este binario no está público. Según la documentación que se encuentra disponible en el sitio, los requerimientos que se tienen para la ejecución es la versión de CUDA 6.5 y OpenMPI 1.6.5. También se necesitará que el driver instalado en

la plataforma sea compatible con esta versión de CUDA. Sin embargo, se pudo comprobar que estos requerimientos están incompletos. Mediante comunicación personal con uno de los desarrolladores (M. Fatica) de dicha implementación, se constató que también es necesario el uso de GPUs con capacidad computacional (CUDA compute capability[4]) por encima de 3.0.

En este contexto, se evaluó la variante de HPCG sobre la GPU de MOJIGATA con las mismas dimensiones de problema que para los experimentos realizados sobre los CPUs (88, 96 y 104). La Tabla 4.7 resume los desempeños alcanzados para los diferentes casos expresados en HPCG Rating.

Dispositivo	Plataforma	Tamaño del cubo	HPCG Rating (GFLOPS/s)
GPU	MOJIGATA	88	23,9
		96	22,0
		104	23,7

Tabla 4.7: Desempeño (en GFLOPS) alcanzado por HPCG utilizando la GPU de MOJIGATA.

La primera conclusión inmediata que se puede obtener de los resultados es que el uso de la GPU claramente mejora el desempeño del benchmark. Específicamente, se obtienen aceleraciones de hasta 22× y 18× cuando se comparan los tiempos de ejecución en GPU y las CPUs de R2D2 y MOJIGATA respectivamente. Otro resultado claro es la diferencia de rendimiento entre HPCG y HPL, los desempeños alcanzados en las diferentes plataformas muestran al menos un orden de diferencia. Este resultado está claramente alineado con la teoría, ya que en general los algoritmos de álgebra dispersa (HPCG) están limitados por los accesos a memoria, mientras que las operaciones de álgebra densa (cuando son ejecutadas con algoritmos a bloques, como por ejemplo en HPL) están limitadas por el cómputo. Este concepto, también puede explicar la no escalabilidad de las aceleraciones al utilizar las GPUs en el caso de HPCG. Ya que, la diferencia constante entre el desempeño de ambas plataformas (CPU y GPU) al crecer el tamaño del problema, se puede atribuir al mayor ancho de banda de acceso a memoria que ofrece la jerarquía de memoria de las GPUs. Entonces, en este caso un dato más significativo que el pico teórico de rendimiento en doble precisión es el ancho de banda de los dispositivos. Notar que las CPUs utilizadas ofrecen un ancho de banda de 25,6 GB/s, mientras que la GPU en cuestión asciende hasta los 288 GB/s, y estas diferencias en el ancho de banda son comparables a las obtenidas en los experimentos.

4.5. Graph500

El benchmark ofrece diversas implementaciones para diferentes tipos de plataformas. Para poder realizar una comparación razonable de este benchmark con LINPACK y HPCG, y considerando la forma de ejecución de los anteriores, se utilizó la implementación que se basa en MPI. En este caso no se dispone de una versión estable del benchmark para usar GPUs.

[4]La capacidad computacional (compute capability) de una tarjeta gráfica Nvidia permite saber las características y especificaciones que puede soportar la tarjeta.

Al ejecutar el `makefile` que se encuentra en el directorio de la implementación MPI, se generan cinco ejecutables distintos:

- `graph500_mpi_custom`: es solo un template de cómo utilizar las estructuras de datos generadas para implementar un BFS provisto por el usuario.
- `graph500_mpi_one_sided`: se basa en la representación de datos para utilizar *algunos artilugios para adaptarse a* las operaciones *one-sided* [46] (es decir, operaciones que involucren la transferencia de datos desde un proceso y no desde un proceso origen y otro destino como es habitual [23]) de MPI.
- `graph500_mpi_replicated`: versión de BFS sincronizada por niveles y utilizando dos colas.
- `graph500_mpi_replicated_csc`: igual que la versión anterior pero utiliza matrices de adyacencia bajo formatos dispersos, en particular utiliza el formato CSC (ver Sección 3.1.1.3) para generar el grafo inicial.
- `graph500_mpi_simple`: versión tradicional de BFS sincronizada por niveles y utilizando dos colas. Aprovecha el envío (y proceso) asíncrono de mensajes a lo largo del código para solapar comunicación y cómputo.

Como caso de estudio, se utilizó la implementación `graph500_mpi_simple`. Los ejecutables reciben dos parámetros: la escala del problema (logaritmo en base 2 de la cantidad de vértices; es un parámetro obligatorio) y el factor de las aristas (el cociente entre número de aristas y número de vértices; es un parámetro opcional que, por defecto, toma el valor 16).

4.5.1. Configuración inicial

Para este benchmark no se necesita hacer una configuración inicial más allá de verificar si se encuentra instalado MPI y un compilador de C++.

4.5.2. Ejecuciones en entornos CPU

Para el caso de Graph500, se ejecutó la variante oficial del benchmark sobre CPU extrayendo el tiempo de generación y construcción del grafo, el tiempo de ejecución de la operación y el tiempo de validación. En este benchmark se calcula un factor denominado *TEPS* (sigla en inglés para *traversed edges per second*) de las 64 búsquedas BFS correspondientes a los 64 vértices de salida que se crean, luego del Núcleo 1 (ver Sección 3.4). El factor refleja el cociente entre la cantidad de aristas borde y el tiempo de ejecución del Núcleo 2. De hecho, este factor es el empleado por la lista Graph500 para ordenar los equipos según su capacidad de cómputo. Por las razones antes mencionadas se utilizó los *TEPS* para resumir los resultados obtenidos al ejecutar en dos CPUs diferentes (R2D2 y BUENAVENTURA), resultados que se muestran en la Tabla 4.8. En estas plataformas se evalúa hasta escala 12 por restricciones de memoria de BUENAVENTURA.

Luego de las ejecuciones iniciales se realizaron pruebas más intensivas sobre este benchmark con grafos de escalas mayores. Dichos valores servirán como referencia para análisis posteriores. En la Figura 4.1 se puede ver el rendimiento para grafos de escalas entre 4 y 24 en la CPU de MOJIGATA. No se pudo verificar en escalas mayores debido a restricciones de memoria de la infraestructura utilizada.

Plataforma	Escala	*TEPS*
R2D2	4	$5{,}113\text{x}10^7$
	8	$6{,}941\text{x}10^7$
	12	$7{,}578\text{x}10^7$
BUENAVENTURA	4	$8.260\text{x}10^7$
	8	$6{,}674\text{x}10^7$
	12	$6{,}674\text{x}10^7$

Tabla 4.8: Desempeño (en *TEPS*) alcanzado por Graph500 utilizando la CPU de R2D2 y BUENAVENTURA.

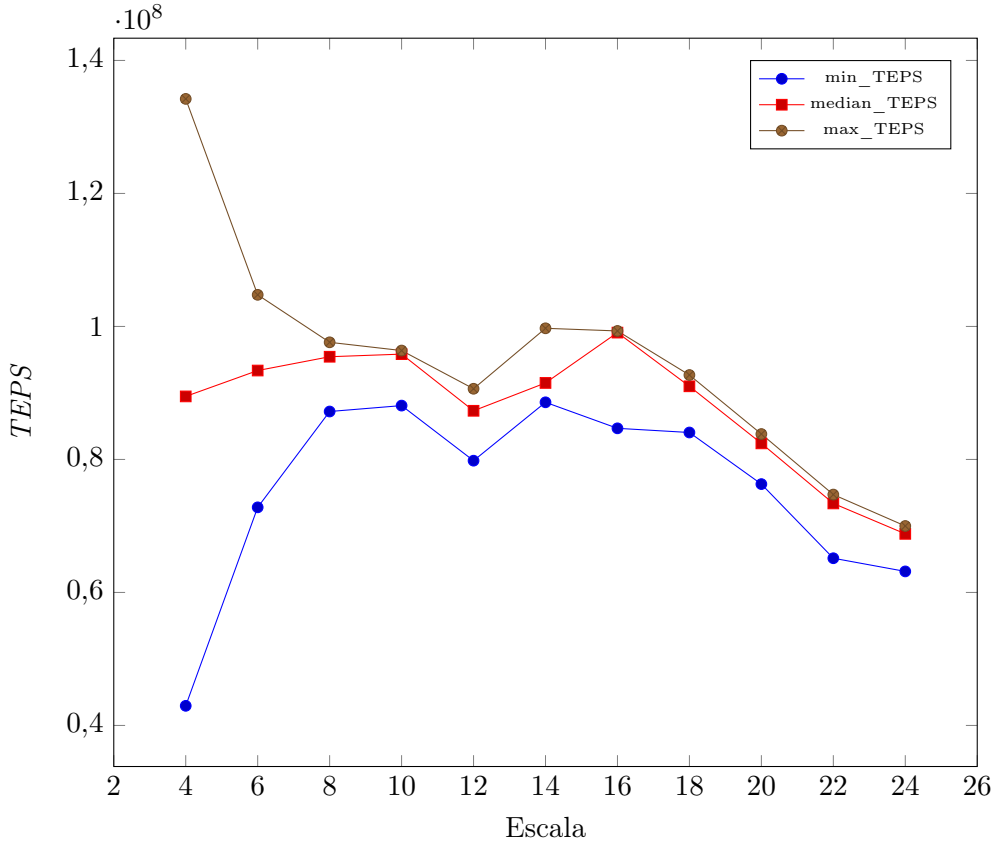

Figura 4.1: Desempeño (en *TEPS*) alcanzado por Graph500 para escalas entre 4 y 24 utilizando la CPU de MOJIGATA.

4.6. Graph500 para GPUs

En el caso del benchmark Graph500 no se dispone de una alternativa oficial que permita aprovechar el poder de cómputo de las GPUs. Por este motivo se decidió realizar un relevamiento de trabajos relacionados, buscando generar alternativas a partir de los mismos. En concreto, se buscó en la literatura esfuerzos sobre implementaciones del método BFS que utilicen GPUs para calcular al menos una parte del proceso. El objetivo de la revisión fue identificar propuestas candidatas a ser transformadas (completadas) para actuar como variantes del benchmark sobre arquitecturas masivamente paralelas. Durante el relevamiento realizado se encontraron diversos trabajos, entre otros [6], [19] [30], [48], [52] y [53]. Estos trabajos se analizaron, junto con las implementaciones que

ofrecen (en algunos casos). Por último, se completaron los desarrollos para las opciones identificadas y se evaluaron experimentalmente las versiones alcanzadas.

4.6.1. Aceleración del BFS en GPU

Un primer trabajo importante en este contexto es el realizado por Harish y Narayanan [19], cuyo objetivo es plantear algunos algoritmos para computar sobre grafos de grandes dimensiones usando CUDA, entre los que se encuentra la recorrida BFS. La representación del grafo se hace con una lista de adyacencias que se almacena en memoria como un arreglo de tamaño variable. Esto se justifica, según los autores, en el hecho que la representación en forma de lista puede no ser eficiente en las GPUs. Notar que se almacena en un único arreglo de índices de vértices, es decir, como una lista compacta. Por otro lado, en un arreglo de largo la cantidad de vértices, se almacenan los índices de la lista de adyacencias en donde comienza la lista de adyacencia del vértice que representa cada espacio del array.

Pasando al algoritmo propiamente dicho, el trabajo plantea una implementación en la que se asigna un hilo de ejecución por vértice del grafo. A nivel de CUDA, se determina una cantidad máxima de hilos por bloque y, en función de la cantidad de nodos y este máximo, se determina la cantidad de bloques que tendrá la ejecución. Asimismo, se almacena en la memoria compartida por todos los hilos de un bloque los nodos visitados, los nodos correspondientes al nivel que se está procesando y un arreglo con la mínima cantidad de aristas que hay desde cada nodo hasta el nodo origen. Harish y Narayanan realizan el análisis experimental en una máquina con procesador Intel Core 2 Duo conectado a una tarjeta Nvidia GeForce 8800GTX. Entre los datos recolectados en dicho trabajo se encuentra un gráfico que representa el tiempo que le insume procesar al algoritmo un grafo a medida que varía el grado por vértice (es decir, a medida que aumenta la cantidad de aristas por vértice). Estos resultados, al tener la cantidad de aristas procesadas en un tiempo determinado, son los empleados para comparar. Teniendo en cuenta que este trabajo se centraba en analizar el algoritmo BFS, los resultados están expresados en el tiempo insumido para la ejecución del algoritmo para diferentes escalas de grafo. Interpretando esto, en la unidad utilizada para evaluar el rendimiento en Graph500, se puede ver que los autores alcanzan desempeños entre aproximadamente 1×10^7 y 1×10^8 *TEPS* en los grafos que cuentan con menor y mayor cantidad de aristas respectivamente.

Otros trabajos estrechamente ligados son los de Ueno y Suzumura [52], [53], en los que se esboza una implementación del benchmark Graph500. Basado en los trabajos de Harish y Narayanan los autores desarrollan una implementación del BFS que genera menores costos de comunicación entre la memoria de la CPU y la de la GPU. Para ello se particiona la matriz de adyacencia utilizada para representar el grafo, ahí radica la principal diferencia con el trabajo de Harish y Narayanan. Las particiones de esta matriz se asignan a cada uno de los procesadores disponibles, haciendo que cada uno tenga una lista de vértices (denominada *NQ* por *next queue*) y una lista parcial de sus aristas. Al momento de realizar la recorrida BFS por niveles, los procesadores de la misma columna se envían la lista *NQ* entre ellos en una fase de comunicación denominada *expand*. Luego de recibir la lista de cada procesador, se une con la de ese procesador y se la envía a los vértices destino que se encontrarán en la misma fila. Esta última comunicación está en una fase llamada *fold*. La propuesta incluye una implementación que se puede descargar desde [29].

En otra línea de investigación se identifica el proyecto generado por Shirahata [45] que implementa aplicaciones de procesamiento de grafos que se ejecutan en GPUs. En

particular utiliza un generador de grafos basado en el algoritmo Kronecker (algoritmo empleado también en el benchmark Graph500) e implementa una serie de aplicaciones de ejemplo. Para las pruebas realizadas en nuestro contexto, y considerando que lo que hace Graph500 es encontrar componentes conexas, se utiliza la aplicación *Connected Components* (encuentra las componentes conexas) disponible en las aplicaciones de ejemplo. Para utilizar la herramienta de Shirahata de forma similar al benchmark, primero se debe generar la lista de vértices con el generador Kronecker que luego se utiliza como entrada para la aplicación mencionada antes. Es claro que los resultados no son directamente comparables con los obtenidos por el benchmark ya que se está realizando otra operación: en lugar de realizar una recorrida BFS hace dos recorridas DFS para encontrar las componentes conexas.

También es interesante considerar el artículo de Daga [6], que se centra en el estudio del costo de mover datos entre la memoria de la máquina y la GPU. Para ello se utilizó como caso de estudio la optimización del benchmark Graph500 debido a que, según se expresa en el artículo, cerca de la mitad del tiempo de ejecución del benchmark se invierte en movimiento de datos hacia la memoria de la GPU. Este trabajo está realizado sobre tarjetas gráficas AMD. El algoritmo usado para implementar BFS en esta implementación es `hybrid++` [7]. Se trata de una implementación que determina si utilizar un acercamiento top-down (se ejecuta mejor en CPU) o bottom-up (genera mayor paralelismo y es recomendable para plataformas tipo GPUs) a partir de una heurística en línea. Dicha heurística determina las características del grafo a partir de iteraciones previas del BFS para decidir el algoritmo apropiado a utilizar.

Por su parte, el trabajo de Merrill [30] analiza el algoritmo BFS de forma similar a la propuesta de Harish y Narayanan (es decir, sin estar directamente relacionado al benchmark Graph500 propiamente dicho). Este trabajo afirma que los algoritmos BFS para GPU que se han implementado son asintóticamente ineficientes y su rendimiento en grafos de diámetros no triviales es pobre (específicamente, las pruebas realizadas en este trabajo involucran grafos con componentes de diámetro máximo de 5 órdenes de magnitud). Lo que se plantea es una nueva implementación de BFS utilizando paralelismo de grano fino de forma de llegar a un orden asintótico óptimo lineal.

4.6.2. Extensiones propuestas

Del relevamiento realizado surge que las propuestas más cercanas a los objetivos (alcanzar una variante del benchmark acelerada por GPU) y que ofrecen implementaciones, al menos parciales, son los siguientes trabajos: [19], [45], [52] y [53]. Entonces, en este apartado se describen las modificaciones y extensiones realizadas sobre los esfuerzos mencionados.

En primer término, se evaluó la implementación del benchmark en GPU de Harish y Narayanan [19], notar que esta variante está fuertemente basada en el algoritmo disponible en [57]. Para transformar en comparables las implementaciones de esta propuesta y la del benchmark Graph500 se realizaron algunas pequeñas modificaciones que se presentan a continuación:

- Se agregó la repetición de la recorrida sobre el grafo en 64 oportunidades de la misma forma que lo hace Graph500.
- Se agregó la medición del tiempo que insume realizar la recorrida. Vale decir que el tiempo medido es exclusivamente el que se utiliza para recorrer las aristas de forma tal que ese insumo sirva para el siguiente punto.
- Se agregó el cálculo de *TEPS* para cada recorrida.

Asimismo para generar el grafo de entrada se hace uso del algoritmo Kronecker que se emplea en Graph500. Para ello, se utilizó la implementación incluida en el benchmark haciendo algunas adaptaciones como ser eliminar las aristas repetidas que genera el algoritmo. Notar que la implementación en GPU del benchmark utiliza como entrada una representación del grafo mediante una lista de adyacencias. Específicamente, la lista de adyacencias de cada nodo se almacena en un arreglo único de enteros. Para poder obtener los nodos correspondientes, en el archivo de entrada se indica, para cada nodo, cuantos nodos adyacentes tiene y en qué lugar del arreglo comienza su lista de adyacencia. En relación al BFS, el único dato que se necesita saber es el nodo desde el cual se iniciará la recorrida. Por simplicidad, se decidió que la recorrida siempre comenzara desde el nodo de índice más bajo (que tenga aristas).

En segundo lugar se evaluaron los trabajos de Ueno y Suzumura [52] y [53]. Como ya se mencionó se trata de un esbozo de implementación de Graph500 en GPU. Dicho trabajo presenta algunas características (que pueden ser vistas como falencias) que hacen que no cumpla con las definiciones de la implementación de Graph500. Mientras la implementación original de Graph500 hace 64 iteraciones de BFS, esta variante genera 15 iteraciones para realizar los cálculos de *TEPS*. La diferencia radica en que este trabajo está basado en la versión de Graph500 1.2, lanzada en julio de 2010, mientras la versión analizada en CPU es la 2.1.4, última estable disponible en el sitio oficial de Graph500, de mayo de 2011. También, como ya se dijo, otra de las diferencias está en la implementación de la recorrida BFS. Esta propuesta hace un particionamiento de dos niveles mientras que Graph500 paraleliza las recorridas a nivel de vértices. Esto significa que la variante de Ueno y Suzumura divide la matriz de adyacencias en submatrices y se paraleliza las recorridas a partir de ese subconjunto de aristas y vértices asociados.

Finalmente, se evaluó el trabajo de Shirahata [45]. En este caso, como se dijo anteriormente, el algoritmo que se utiliza para evaluar es el de encontrar componentes conexas. En particular, se realizan sucesivas recorridas DFS sobre el grafo buscando, en cada iteración, que la solución converja. En forma más detallada el procedimiento seguido hace una recorrida DFS inicial sobre el grafo en cuestión y una recorrida DFS sobre el grafo traspuesto (igual al grafo original pero con sus aristas invertidas). Si ambas recorridas coinciden, el algoritmo para. En caso contrario, se repite el algoritmo hasta que esto suceda. La recorrida DFS sobre el grafo original se hace con el ejecutable `ConCmpt` y la recorrida sobre el grafo traspuesto se hace con `ConCmpt3`.

4.6.3. Evaluación de los trabajos extendidos

A continuación, y luego de haber presentado los trabajos abordados y su extensión, se presentan los resultados experimentales de dichas implementaciones.

Manteniendo el orden propuesto en el apartado anterior, se comienza la evaluación con la propuesta de Harish y Narayanan. En esta línea, en la Figura 4.2 se puede observar el rendimiento del método medido en *TEPS* para grafos de diferentes escalas en la plataforma MOJIGATA. Vale decir que se probó hasta escala 18 ya que la transformación de los archivos de entrada para escalas más grandes insumía tiempos de ejecución demasiado prolongados (mayor a un día). Notar que en el caso de Graph500 para CPU el límite es mayor debido a que la entrada se almacenaba directamente en memoria y no existía tiempo de escritura a disco.

En esta misma línea, en la Figura 4.3 se pueden ver los resultados de las ejecuciones de la implementación de la propuesta de Ueno y Suzumura para diferentes tamaños de grafo, con escalas que varían entre 4 y 24. La plataforma de ejecución en este caso también es MOJIGATA.

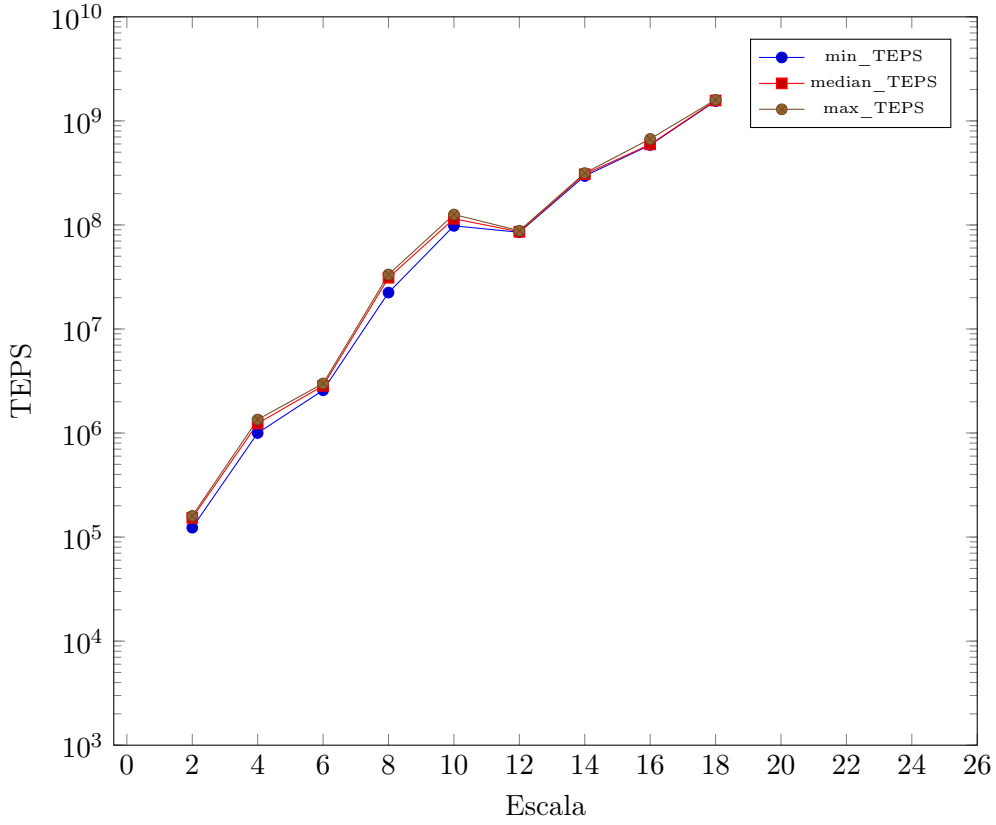

Figura 4.2: Desempeño (en *TEPS*) alcanzado por el método de Harish y Narayanan utilizando la GPU de MOJIGATA.

Analizando los resultados se puede ver como en ambas propuestas a mayor dimensión del grafo, mejor es el rendimiento en cantidad de aristas procesadas por segundo. Comparando los resultados obtenidos con los que se plantearon inicialmente en la Figura 4.1 para arquitecturas basadas en CPUs, recordar que en estas arquitecturas el rendimiento es aproximadamente estable (se encuentra en valores cercanos a 1×10^8 *TEPS*), se puede decir que las GPUs necesitan problemas de mayor dimensión para lograr aceleraciones. En este caso particular, la GPU es superada por la CPU en grafos de escala menores a 12, pero la GPU es sensiblemente mejor en escalas mayores, llegando a un pico de casi 6×10^9 *TEPS*. Esto se puede justificar por dos motivos: por un lado la mayor capacidad de paralelismo ofrecida por la gran cantidad de procesadores disponibles en las tarjetas gráficas necesita volúmenes importantes de datos para explotar esta característica del hardware, y por otro lado, al aumentar el trabajo disminuye el costo relativo de las transferencias entre la memoria de la CPU y de la GPU.

Pasando ahora a comparar los resultados de ambas propuestas se puede ver que los resultados alcanzados por el código propuesto por Harish y Narayanan son más altos en cuanto al desempeño y, en consecuencia, mejores que los de la propuesta de Ueno y Suzumura. Esto también puede estar relacionado a dos motivos: por un lado en el hecho que la primer propuesta involucra menor cantidad de accesos a memoria. Por otra parte, y referida a la recorrida propiamente dicha, mientras el planteo de Ueno y Suzumura. hace las 64 ejecuciones de BFS a partir de diferentes vértices, la de Harish y Narayanan hace las misma cantidad de ejecuciones pero a partir de un mismo vértice de inicio.

Por último, se evaluó el código de Shirahata. En la Tabla 4.9 se presentan los tiempos de ejecución y la cantidad de iteraciones para resolver problemas de diferentes dimen-

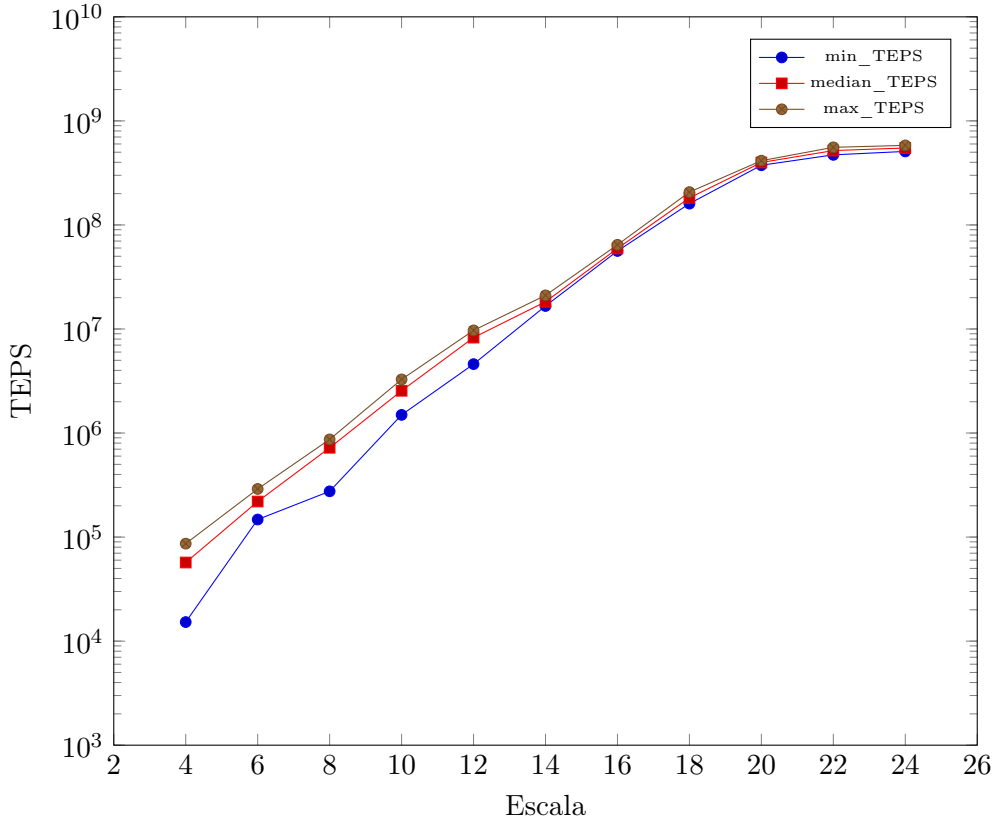

Figura 4.3: Desempeño (en *TEPS*) alcanzado por la variante de Ueno y Suzumura utilizando la GPU en MOJIGATA.

siones en la máquina BUENAVENTURA. Fue necesario utilizar esta plataforma debido a que la implementación tenía como requerimiento utilizar la versión de CUDA 4.0, versión que no estaba instalada en MOJIGATA y cuya instalación local generó algunos problemas al momento de la compilación de este código. En cada una de las iteraciones, y a partir de la lista de vértices generada por el algoritmo Kronecker, se crea una componente conexa con todos los vértices del grafo. Los tiempos insumidos por la ejecución de los mismos se encuentran discriminados en cada iteración del método. Vale reiterar que GraphGPU (el nombre de la propuesta de Shirahata) utiliza DFS para encontrar componentes conexas. En cambio, Graph500 utiliza BFS para calcular sus métricas. Considerando los resultados se puede ver que la cantidad de aristas recorridas por unidad de tiempo es muy bajo en comparación a las dos propuestas anteriores. Esto reafirma que una propuesta de evaluación de arquitecturas a partir de una recorrida DFS sería poco útil para comparar con los recorridos BFS.

Escala	Iters	Tiempo	TEPS
4	3	0,774 + 0,769	10,4
8	5	0,778 + 0,787	163,6
12	11	1,206 + 0,864	1978,7

Tabla 4.9: Tiempo de ejecución (en segundos) del trabajo de Shirahata utilizando la GPU de BUENAVENTURA. El primer valor representa el tiempo de ejecución de `ConCmpt3` y el segundo el de `ConCmpt`. En la última columna se hace el cálculo de *TEPS*.

Capítulo 5

Conclusiones y trabajo futuro

Considerando los objetivos planteados al inicio del proyecto, se puede decir que estos fueron razonablemente alcanzados. Específicamente, se identificaron los tres benchmarks principales utilizados por la comunidad de computación científica, es decir LINPACK (en su variante HPL), HPCG y Graph500. Estas herramientas fueron conceptualizadas no solamente en lo que pretenden evaluar sino también como en su forma de funcionamiento. Asimismo, las tres herramientas fueron instanciadas en las distintas plataformas de hardware a las que se tuvo acceso durante el proyecto, algunas de las cuales incluyen procesadores masivamente paralelos (GPUs).

Respecto a las instanciaciones, para el caso de los primeros dos benchmarks, se describió como realizar la configuración en las diferentes plataformas ya que esto no es una tarea trivial. Para Graph500 solo existe una alternativa oficial para analizar arquitecturas basadas en CPUs. En el caso de plataformas que incluyen GPUs se relevaron trabajos relativos al uso de GPUs para acelerar el método de recorrida de grafos BFS. Además, se extendieron las opciones más prometedoras para alcanzar un funcionamiento similar al ofrecido por Graph500 y estas variantes extendidas se evaluaron y compararon experimentalmente.

El análisis experimental del HPL en ambas plataformas (CPU y GPU) arrojó primero la importancia de la implementación de la biblioteca BLAS utilizada en CPU para obtener buenos resultados. Específicamente, OpenBLAS superó en todos los casos en rendimiento a BLAS en la implementación de referencia. En cuanto al uso de GPU para acelerar el benchmark, el desempeño mostró importantes mejoras (del orden de 10×). Además las mejoras escalan con la dimensión de los sistemas resueltos. Estos resultados están alineados con la teoría, en especial considerando que el benchmark está basado en operaciones de álgebra densa.

Por otro lado, el estudio del desempeño del HPCG permitió ver que al estar basado en operaciones de álgebra dispersa incluye métodos limitados por accesos a memoria. En contraposición al HPL que ofrece una mayor intensidad computacional. Esto explica que las diferencias obtenidas al utilizar la GPU son proporcionales a las diferencias de ancho de banda de las arquitecturas evaluadas.

Por último, la comparación de los esfuerzos relativos a utilizar GPU para el algoritmo BFS y la implementación en CPU de Graph500 muestra que las GPUs permiten acelerar el cómputo de la herramienta a partir de cierta dimensión del problema. Además, el mismo algoritmo, cambiando la forma en que se accede a los datos en memoria de GPU genera mejoras de hasta un orden en los tiempos de ejecución, es decir corrobora la mayor incidencia del uso de estructuras de datos adecuadas para resolver problemas en GPU.

En otro eje de evaluación, los benchmarks HPL y HPCG permitieron analizar y comparar el rendimiento de las arquitecturas abordadas a nivel de cómputo intensivo de datos en forma detallada, mientras que en el caso de Graph500 el estudio fue a nivel básico ya que no se dispone de una versión estable (oficial). En los tres casos se pudo verificar que el uso de arquitecturas híbridas mejora sensiblemente el rendimiento de los benchmarks tanto en lo referente a la capacidad de cómputo de operaciones aritméticas como de acceso a datos.

Por último, es importante destacar que resultados parciales de este proyecto fueron aceptados para su presentación y publicación como parte de los proceedings de la XLII Conferencia Latinoamericana de Informática (CLEI 2016). Específicamente se publicó el trabajo:

- Danilo Espino, Gerardo Ares, Martín Pedemonte, and Pablo Ezzatti. *Overview of HPC benchmarks in hybrid hardware platforms (CPUs+GPUs)*. In *Computing Conference (CLEI), 2016 XLII Latin American*, 2016.

El trabajo realizado permitió identificar diversos puntos de interés para extender la propuesta. Por limitantes de tiempo o definición de alcance muchos de ellos no fueron abordados y se plantean como trabajos futuros. Las principales líneas de trabajo futuro son:

- avanzar en definir una versión estable de Graph500 capaz de sacar partido de las GPUs, y al mismo tiempo completamente comparable con la distribución oficial de Graph500 (en CPU).
- extender la evaluación de estos benchmarks a otras arquitecturas (memoria distribuida, Xeon Phi y ARM por ejemplo), buscando analizar el comportamiento ante mayor/menor capacidad de cómputo, el efecto del manejo de datos o las implicancias del uso de la red.
- analizar otras cuestiones como ser el consumo energético. Específicamente, estudiando la relación de la cantidad de operaciones de punto flotante por watt consumido de los diferentes benchmarks y plataformas.

Capítulo 6

Bibliografía

[1] Jordan B. Angel, Amy M. Flores, Justine S. Heritage, Nathan C. Wardrip, Andrew M. Raim, Matthias K. Gobbert, Richard C. Murphy, and David J. Mountain. The Graph 500 Benchmark on a Medium-Size Distributed-Memory Cluster with High-Performance Interconnect, Accedido: Noviembre 2016. URL `http://userpages.umbc.edu/~gobbert/papers/Graph500ParallelComput.pdf`.

[2] Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Orti. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[3] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. SIAM, 1994.

[4] BLAS. BLAS: Basic Linear Algebra Subprograms, Accedido: Enero 2016. URL `http://www.netlib.org/blas/`.

[5] Cray. Introducing the Cray XMT Supercomputer. *White paper*. URL `http://www.cray.com/Assets/PDF/products/xmt/CrayXMTOverviewWhitepaper.pdf`.

[6] Mayank Daga. On the Acceleration of Graph500: Characterizing PCIe Overheads with Multi-GPUs. In *12th International Meeting on High Performance Computing for Computational Science*, 2016.

[7] Mayank Daga, Mark Nutter, and Mitesh Meswani. Efficient breadth-first search on a heterogeneous processor. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 373–382. IEEE, 2014.

[8] Timothy A Davis. *Direct methods for sparse linear systems*, volume 2. SIAM, 2006.

[9] Jack Dongarra and Michael A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. *Sandia Report, SAND2013-4744*, 312, 2013.

[10] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[11] Iain S Duff, Albert Maurice Erisman, and John Ker Reid. *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[12] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2014.

[13] Danilo Espino, Gerardo Ares, Martín Pedemonte, and Pablo Ezzatti. Overview of HPC benchmarks in hybrid hardware platforms (CPUs+GPUs). In *Computing Conference (CLEI), 2016 XLII Latin American*, 2016.

[14] Institute for Advanced Study. Electronic Computer Project, Accedido: Noviembre 2015. URL `https://www.ias.edu/people/vonneumann/ecp/`.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (4th Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 2013.

[16] John Goodacre. Technology Preview: The ARMv8 Architecture. *White paper*, 2011.

[17] Graph500. Graph500, Accedido: Diciembre 2016. URL `http://www.graph500.org`.

[18] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems, Accedido: Diciembre 2016. URL `https://www.khronos.org/opencl/`.

[19] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[21] Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors. *Readings in Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[22] Ellis Horowitz and Sartaj Sahni. *Fundamentals of computer algorithms*. Computer Science Press, 1978.

[23] Intel. MPI One-Sided Communication | Intel Software, Accedido: Diciembre 2016. URL `https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication`.

[24] Intel. Export Compliance Metrics for Intel Itanium and Pentium Processors, Accedido: Diciembre 2016. URL `http://www.intel.com/content/www/us/en/support/processors/000007250.html`.

[25] Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee. *Multicore Processors and Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[26] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

[27] Sandia National Laboratories. HPCG, Accedido: Noviembre 2015. URL `http://www.hpcg-benchmark.org`.

[28] Argone National Laboratory. LINPACK, Accedido: Octubre 2016. URL `http://www.netlib.org/linpack/`.

[29] Suzumura Laboratory. Graph500 Challenge, Accedido: Noviembre 2016. URL https://sites.google.com/site/tokyotechsuzumuralabeng/graph-500-challenge.

[30] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[31] Paulius Micikevicius. GPU performance analysis and optimization. In *GPU technology conference*, 2012.

[32] Gordon E Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35, 2006.

[33] Gordon E Moore et al. Progress in Digital Integrated Electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[34] NVIDIA. cuBLAS | NVIDIA Developer, Accedido: Diciembre 2016. URL https://developer.nvidia.com/cublas.

[35] NVIDIA. cuSPARSE | NVIDIA Developer, Accedido: Diciembre 2016. URL https://developer.nvidia.com/cusparse.

[36] Innovative Computing Laboratory University of Tennessee. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, Accedido: Noviembre 2015. URL http://www.netlib.org/benchmark/hpl/.

[37] Innovative Computing Laboratory University of Tennessee. HPL: A Portable Implementation of the High Performance Linpack Benchmark for Distributed-Memory Computers | LINPACK Implementation, Accedido: Noviembre 2015. URL http://www.netlib.org/benchmark/linpackd.

[38] University of Texas at Austin. gotoblas2 - Texas Advanced Computing Center, Accedido: Julio 2016. URL https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2.

[39] OpenMPI. Open MPI: Open Source High Performance Computing, Accedido: Diciembre 2016. URL http://www.open-mpi.org/.

[40] Linux Information Project. BSD license definition, Accedido: Diciembre 2016. URL http://www.linfo.org/bsdlicense.html.

[41] Gregorio Quintana-Ortí, Enrique S Quintana-Ortí, Ernie Chan, Robert A Van De Geijn, and Field G Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[42] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition, 2013.

[43] Intel Press Room. Intel Unveils New Product Plans for High-Performance Computing, Accedido: Diciembre 2016. URL http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm.

[44] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[45] Koichi Shirahata. GraphGPU, Accedido: Abril 2016. URL `https://github.com/koichi626/GraphGPU`.

[46] Marc Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[47] William Stallings. *Computer Organization and Architecture: Designing for Performance (9th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2013.

[48] Toyotaro Suzumura, Koji Ueno, Hitoshi Sato, Katsuki Fujisawa, and Satoshi Matsuoka. Performance characteristics of Graph500 on large-scale distributed environment. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 149–158. IEEE, 2011.

[49] TOP500.org. Green500 | TOP500 Supercomputer Site, Accedido: Diciembre 2016. URL `https://www.top500.org/green500/`.

[50] TOP500.org. TOP500 Supercomputers Site, Accedido: Diciembre 2016. URL `http://www.top500.org`.

[51] Jim Turley. Introduction to Intel Architecture: The Basics. *White paper*, 2014.

[52] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the Graph500 benchmark. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2012.

[53] Koji Ueno and Toyotaro Suzumura. Parallel Distributed Breadth First Search on GPU. In *20th Annual International Conference on High Performance Computing*, pages 314–323. IEEE, 2013.

[54] Manuel Ujaldón. Nuevas tendencias en GPU y su programación. In *Escuela de Informática, CACIC'16*, 2016. URL `http://www.ac.uma.es/~ujaldon/descargas/CUDA/CUDA.pdf`.

[55] Berkeley; Univ. of Colorado Denver; NAG Ltd. Univ. of Tennessee; Univ. of California. LAPACK, Accedido: Octubre 2016. URL `http://www.netlib.org/lapack/`.

[56] András Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[57] Ke Wang. Rodinia Packages, Accedido: Noviembre 2016. URL `http://www.cs.virginia.edu/~kw5na/lava/Rodinia/Packages/Current/rodinia_3.0/cuda/bfs/`.

[58] Zhang Xianyi. OpenBLAS: An optimized BLAS library, Accedido: Octubre 2016. URL `http://www.openblas.net`.

[59] David A Yuen, Long Wang, Xuebin Chi, Lennart Johnsson, Wei Ge, and Yaolin Shi. *GPU Solutions to Multi-Scale Problems in Science and Engineering*. Springer, 2013.